Equivalence-Based Reductions and checking for preorders

Research Thesis

Submitted in Partial Fulfillment of the
Requirements for the
Degree of Doctor of Philosophy

Doron Bustan

Submitted to the Senate of
the Technion - Israel Institute of Technology

Av, 5762 HAIFA June 2002

The Research Thesis Was Done Under
The Supervision of Orna Grumberg in the Faculty of Computer Science

THE GENEROUS FINANCIAL HELP OF INTEL ISRAEL AND GALILEO TECHNOLOGIES, IS GRATEFULLY ACKNOWLEDGED.

Acknowledgments

Looking at this Ph.D. thesis, one would think that it required a lot of research work. For me the real work was learning what a research is, and how to perform it. I cannot imagine myself doing it without the guidance of Orna Grumberg. I want to thank her for opening this world for me.

I also want to thank the sequential equivalence verification CAD group of Intel design technologies in Haifa and in particular Ziyad Hanna, Nir Naor, Alex Levin, Shlomit Ozer, Guy Wolfovitz and Ranan Fraer for giving me all the help I needed at Intel (and I needed a lot).

Contents

List of Figures

List of Tables

Abstract

Due to the fast development of the hardware and software industry, there is a growing need for formal verification tools and techniques. Two widely used formal verification methods are temporal logic model checking and sequential equivalence checking. Temporal logic model checking is a method for verifying finite-state systems with respect to propositional temporal logic specifications. In sequential equivalence checking, two sequential hardware designs are compared for language equivalence, meaning that for every sequence of inputs, the two designs produce the same sequence of outputs. Both model checking and equivalence checking are fully automatic. However, they both suffer from the *state explosion problem*, that is, their space requirements are high and limit their applicability to large systems.

Many approaches for overcoming the state explosion problem have been suggested; most of them use abstraction. An abstract model of a verified system is a model, which describes a system that is similar to the verified system but is simpler. An abstraction is conservative if the results of verifying the abstract model are true for the verified system. Thus, the abstract model preserves the verified properties of the verified system. There are two types of preservations: We say that an abstraction *strongly preserves* a set of verified properties, if for every property in the set, the system satisfies the property if and only if the abstract model satisfies it. Sometimes we relax our requirements such that for every verified property, if the abstract model satisfies the property then the system satisfies it as well. In this case the abstraction *weakly preserves* the verified properties.

Strong preservation with respect to a set of properties can also be seen as an equivalence relation: A system is equivalent to an abstract model if the set of properties that the system satisfies is equal to the set of properties that the abstract model satisfies. Similarly, weak preservation with respect to a set of verified properties can be seen as a preorder: An abstract model is greater than a system if the set of properties that the system satisfies contains the set of properties that the abstract model satisfies.

In this work we investigate different equivalence relations and preorders and their usage for abstraction. We present an algorithm that given a system constructs the smallest abstract model, which is simulation equivalent to the system. Next, we compare different preorders, which reflect a weak preservation of infinite behaviors. Finally, we present an algorithm which exploits the modular structure of the system to be verified, to improve the construction of an abstract model.

Notation and Abbreviations

ρ	—	A trace in a model
ρ^i	—	The i-th state of trace ρ
$[s]$	—	The equivalence class of the state s
$\|\|$	—	The composition operator
$\approx$	—	Approximately equal
$\ll$	—	Significantly smaller

Chapter 1

Introduction

Following the fast development of the hardware and software industry, there is a growing need for formal verification tools and techniques. Typically, a verification technique uses a model to describe the verified system, and the verification of the system is executed on the model. The model is represented as a graph, where the vertices of the graph represent all possible configurations of the system. These vertices are called states. The edges of the graph represent a possible change from one configuration to another, and are called transitions. It is common to associate the observable behavior (outputs) of a system in a specific configuration with states, and to associate internal events (inputs) with the transitions which they enable.

Two widely used verification methods are temporal logic model checking and sequential equivalence checking. Temporal logic model checking is a method for verifying finite-state systems with respect to propositional temporal logic specifications. There the problem of checking whether a system satisfies a temporal formula is reduced into checking a graph property in the model. The method is fully automatic and efficient in time. In sequential equivalence checking, two systems are required to have the same behavior. The systems are equivalent if for every sequence of input events, every possible behavior in one system is possible in the other. For equivalence checking, a model is constructed for each system, and the problem is reduced to trace matching in the graphs.

Although modeling systems seems to be very useful, this method has a major disadvantage, which is the size of the model. Since the model represents each configuration of the system, its size is exponential in the size of the system. This problem, called the *state explosion problem*, is seem to limit the use of such verification methods to small systems only. Many approaches for overcoming the state explosion problem have been suggested, including abstraction, partial order reduction, modular methods, and symmetry ([CGP99]). All are aimed at reducing the size of the model to which verification methods are applied, thus extending their applicability to larger systems. In this work we investigate some of these methods and improve them.

In basic techniques that reduce the size of the model, the verified model does not describe the system precisely but a simpler system, which is "close enough" to the verified system. Since the described system is simpler, the size of the verified model is smaller. Such a model is called an abstract model. When using this technique, a naive approach might give false results. The false results occur because the abstract model is not close enough to the concrete model and there exist some properties, which distinguish the concrete model from the abstract model.

Thus we would like to limit the reduction technique to "correct reductions". However the correctness of the reduction depends on the property that we wish to verify. For example, if the property depends on the number of different configurations in the system, then every model which is smaller than the precise model, will give a false result. However, if the property depends only on one variable x, then all states that agree on x can be collapsed to a single state, resulting in a smaller model.

Given a concrete model M and a property ψ, a model A is a *correct abstraction of* M with respect to ψ, if $M \models \psi \Leftrightarrow A \models \psi$. This definition can be extended to a specification language $\mathcal{L}$ as follow: A is a correct abstraction of M with respect to $\mathcal{L}$ if for every $\psi \in \mathcal{L}$, $M \models \psi \Leftrightarrow A \models \psi$. This definition can be further extended from a single pair of models to a reduction technique $\mathcal{T}$ which receives a concrete model M and constructs an abstract model $\mathcal{T}(M)$. We say that $\mathcal{T}$ *strongly preserves* the specification language $\mathcal{L}$, if for every model M, $\mathcal{T}(M)$ is a correct abstraction of M with respect to $\mathcal{L}$.

Sometimes we relax our requirements so that for every model M and property $\psi \in \mathcal{L}$, if $\mathcal{T}(M) \models \psi$ then $M \models \psi$. In this case we say that

$\mathcal{T}$ *weakly preserves* the specification language $\mathcal{L}$. However verifying an abstract model that weakly preserves the concrete model might give a false negative result.

Thus the first step in constructing a reduced model is to decide which properties need to be preserved and what type of preservation should be used. Then, a smaller abstract model, which preserves the properties, is constructed, and finally the abstract model is verified.

Our work is concerned with the second step, and includes methods for constructing abstract models which strongly/weakly preserve a desired specification language. These methods are evaluated by three basic criteria:

1. The difference in the sizes of the concrete and abstract models.
2. The specification languages they preserve.
3. The complexity of the method.

1.1 Using equivalence relations and preorders for reductions

In the previous section, we related the problem of verifying a system, to a graph problem. We would like to do the same for the construction of "correct abstract model". One way to do it is to use equivalence relations and preorders.

Strong preservation with respect to a specification language $\mathcal{L}$ can also be seen as an equivalence relation: Two models M and A are equivalent with respect to $\mathcal{L}$ ($M \equiv_L A$) if for every $\psi \in \mathcal{L}$, $M \models \psi \Leftrightarrow A \models \psi$. It is easy to see that $\equiv_L$ is an equivalence relation. Similarly, weak preservation with respect to a specification language $\mathcal{L}$ can be seen as a preorder: Model A is greater than model M with respect to $\mathcal{L}$ ($M \leq_L A$) if for every $\psi \in \mathcal{L}$, $A \models \psi$ implies $M \models \psi$. It is easy to see that $\leq_L$ is a preorder (transitive and reflexive) over models.

Equivalence relations which are based on the structure of the graphs (models), can also be evaluated according to the specification languages that they strongly preserve. Since these relations are based on the structure of the graph, the problem of finding a smaller (smallest) equivalent model is reduced to a graph problem. For example, trace equivalence strongly preserves the linear-time temporal logic LTL. Another relation that is widely used is the *bisimulation equivalence* [Par81]. It is shown in [BBLS91] that bisimulation equivalence strongly preserves the Mu-Calculus logic[Koz83]. It also strongly preserves the branching-time logics LTL,CTL, and CTL*, [GL94, CE81], as these logics are expressible within the Mu-Calculus [BCM$^+$92, Dam94]. A preorder that is widely used in the context of abstractions for model checking, is the *simulation preorder* [Mil71]. This preorder is often used when abstraction is applied in order to construct a reduced model ([BBLS91],[CGL94], [DGG97],[GL94]). The preorder relates the reduced abstract model with the original one. It is shown in [BBLS91] that the simulation preorder weakly preserves the universal fragment of Mu-Calculus. It also weakly preserves LTL,ACTL, and ACTL*, the universal fragments of CTL and CTL* [GL94], as these logics are expressible in universal Mu-Calculus.

Equivalence relations like bisimulation and trace equivalence, and preorders like simulation and trace containment are defined with respect to properties of the model as a graph. This enables the development of reduction techniques, which are based on the structure of the original model rather than the properties of the system. Since reducing graphs is an easier task than reducing systems, some reduction methods have been developed for these relations.

For an automatic algorithm that receives a concrete model and constructs a correct abstract model, equivalence relations are preferable to preorders for several reasons: First, in equivalence-based reductions we are usually interested in the minimal model which is equivalent to the concrete model. Since this model is well defined, it can often be constructed in a fully automatic manner. For preorder-based reductions on the other hand, the minimal model with respect to the preorder is usually trivial and does not contain sufficient information for verification. Thus we are looking for some model that is greater by the preorder (and smaller in size) than the concrete model, but is not necessarily minimal. Since this model is not well defined, human intervention is required. Second, equivalence-based reductions, unlike preorder-based reductions, can be used in equivalence checking.

There are known algorithms for constructing the smallest model which is trace/bisimulation equivalent to a given concrete model. Since bisimulation implies trace equivalence, a minimization with respect to bisimulation preserves more properties than minimization with respect to trace equivalence. However, for the same reason, the result of minimization with respect to trace equivalent is smaller in size than the result of minimizing with respect to bisimulation. The complexity of these algorithms is different: While the complexity for language equivalence is PSPACE-complete [SVW85], the complexity for computing bisimulation is a small polynomial [PT87, Fer90, KS90].

1.2 Modular verification

It often happens that the verified system is composed of a few different parts that are loosely connected to each other, meaning that the internal mutual influence inside each part is substantially stronger than the influence of the different parts on each other. This modular structure of the system can be helpful for verifying the system more efficiently. When we are only interested in the observable behavior of a single "component" of the system, we do not need to construct a model for the whole system. Nevertheless, while verifying a single component of the system, we would like to preserve properties that are influenced by the other components of the system. Thus, we cannot verify a model of the single part by itself.

In order to verify a component C, an abstract environment A of the rest of the system M is constructed and composed with C. In this case there is a new definition for a *correct abstraction*: A is a correct abstraction of M with respect to a specification language $\mathcal{L}$ if for every formula ψ in $\mathcal{L}$, $M||C \models \psi \Leftrightarrow A||C \models \psi$.

In [CLM89] an *interface rule* for verifying a system component is defined. The interface rule restricts the abstraction method and the composition operator $||$ in a way that ensures that for every formula ϕ in the specification language, $A||C \models \phi$ iff $M||C \models \phi$. I.e., the abstraction technique strongly preserves the properties of the system. [Jos87] suggests a method, where the environment is constructed manually from formulas given by the user.

In the assume-guarantee paradigm [Fra76, Jon83, MC81, Pnu84], properties of different parts of the systems are verified separately. The environment of the verified part is represented by a formula, which describes its properties. The formula either has been verified in the environment or has been given by the user. The method proves assertions of the form $\psi M \phi$, meaning that if an environment satisfies ψ then the composition of M with the environment, satisfies ϕ. The method enables the creation of a proof schema, which is based on the structure of the system.

[GL94] suggests a framework that uses the assume-guarantee paradigm for semi-automatic verification. It presents a general method that uses models as assumptions; the models are either generated from a formula as a *tableau*, or are abstract models, which are given by the user. The proof of $\psi M \phi$ is done automatically by verifying that the composition of the tableau for ψ with M satisfies ϕ. The method requires a preorder $\leq$, a composition operator $||$ and a specification language $\mathcal{L}$

1.3 Using simulation equivalence for minimization

In the Chapter 3 we suggest the use of the *simulation equivalence* relation for minimization. The simulation equivalence relation is based on the simulation preorder [Mil71]. This equivalence relation preserves fewer properties than bisimulation but more than language equivalence. Simulation equivalence strongly preserves ACTL*, and also strongly preserves LTL and ACTL as sub logics of ACTL*. Both ACTL and LTL are widely used for model checking in practice. LTL is also the most suitable logic for sequential equivalence checking. Similarly to bisimulation, the complexity of computing simulation equivalence is polynomial. Our work includes a proof that there always exists a smallest model with respect to simulation equivalence, and presentation of an efficient algorithm that constructs this smallest model.

As an equivalence relation that is weaker than bisimulation, simulation equivalence can derive smaller minimized structures. Consider for example, the structure in part 1 of Figure 1.1. The structure in part 2 of Figure 1.1 is minimized with respect to simulation equivalence. In comparison, the minimized structure with respect to bisimulation is the structure in part 1 itself and the minimized structure with respect to language equivalence is the structure in part 3 of the figure. The results of Chapter 3 are given in [BG00].

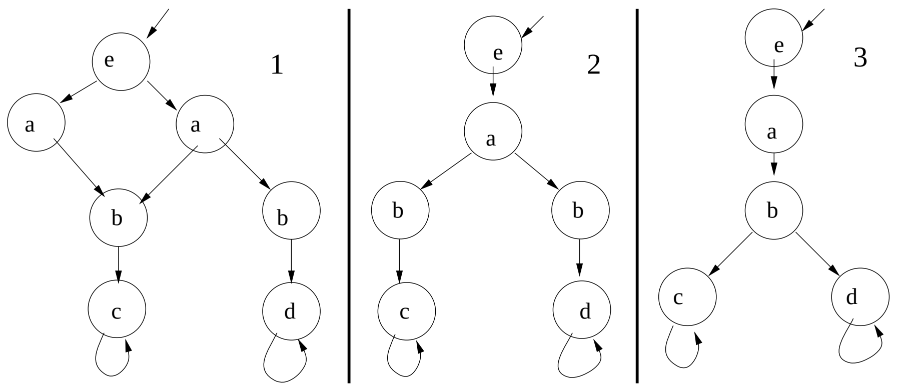

Figure 1.1: Different minimized structures with respect to different equivalence relations

1.4 Fair simulation

In methods like assume-guarantee, we often want to construct an abstract model which abstracts a property, rather than a specific system. In other words, we are looking for a model that abstracts exactly the set of all models that satisfy the property. However, some properties are not definable by a single model. A textbook example for this problem is an abstract model of a timer. In a timer an event occurs after a fixed time interval. An abstract model that abstracts all timers needs to enable the event after any finite time interval. However, any finite model that contains such behaviors enables also a behavior in which the event never occurs. Thus, an unrealistic infinite behavior is added to the model.

Consequently, we need to strengthen the model, so that it can eliminate undesirable infinite behaviors. A common way to avoid such behaviors is to add to the model fairness constraints, which distinguish between wanted (fair) and unwanted (unfair) behaviors and to exclude unfair behaviors from consideration. Indeed, there are a few definitions of such fairness constraints.

The simulation preorder does not handle fairness constraints. It is therefore desirable to extend the preorder so that it relates only the fair behaviors of the models. This extension, however, is not uniquely defined. Several distinct notions of *fair simulation* have been suggested in the literature [GL94, Lyn96, HKR97, EWS01a].

Researchers have addressed the question of which notion of fair simulation is preferable. In [HKR97], some of these notions are compared with respect to the complexity of checking for fair simulation. In [EWS01a], a different set of notions is compared with respect to two criteria: The complexity of constructing the preorder, and the ability to minimize a fair model by constructing a quotient model that is language equivalent to the original one.

In Chapter 4 we make a broader comparison of four notions of fair simulation: direct [DHWT91], delay [EWS01a], game [HKR97], and exists [GL94]. We refer to several criteria that emphasize the advantages of each of the notions.

We developed two practical applications that are based on the comparison. The first is an efficient approximated minimization algorithm for the delay, game and exists simulations. For these preorders, a unique equivalent smallest model does not exist. Therefore, an approximation is appropriate. In addition, we suggest a new implementation for the *assume-guarantee* [Fra76, Jon83, MC81, Pnu84] modular framework presented in [GL94]. The new implementation, based on the game simulation rather than the exists simulation, significantly improves the complexity of the framework.

The results of Chapter 4 are presented in [BG02].

1.5 Modular minimization

Although reductions with respect to bisimulation or simulation equivalence are polynomial and result in smaller models, computing the reduction might require a large amount of resources (time and space). This motivated the development of methods, which implement the reductions more carefully. Some of these methods are listed here.

The algorithm in [LY92] minimizes models with respect to bisimulation. In order to gain efficiency, the algorithm refers only to reachable states and computes equivalence classes for bisimulation instead of pairs of equivalent states. This appears to consume less memory for BDD-based [Bry86] implementations. In [FV98], the algorithm presented in [LY92] is applied to the intersection of the model with an automaton representing the property that should be satisfied by the model. In

[CRFJ96], a reduction with respect to symmetry equivalence is performed. Symmetry equivalence is a bisimulation equivalence, but not necessarily the maximal one. [CRFJ96] reports that computing this reduction is more efficient in the BDD framework than reduction with respect to bisimulation.

Other works exploit modularity for reduction. The modular reduction in [ASSSV94] preserves a given formula which should be checked for truth in the model. This method can result in a small model. However, since it preserves a single formula, it cannot be used for equivalence checking. In [ASSB94], the equivalence relation is a combination of language equivalence and fairness constraints. Since computing this relation is PSPACE-complete, an approximation equivalence relation is computed and the quotient model is defined with respect to the approximation equivalence relation. [GSL96] presents an algorithm that constructs an abstract model of a system through a sequence of approximations, where the final approximation is equivalent to the original system with respect to the specification language. The approximations are constructed according to *interface specifications* which are given by the user. [Shi96] suggests to decompose the model, reduce each module separately and compose the result.

In Chapter 5 we present a new modular minimization algorithm which improves the *naive modular algorithm.* The naive modular algorithm [Shi96] is based on partitioning of the system into components. It minimizes the model in iterations. In each iteration two components are selected, composed together and the result is minimized. This process is repeated until all components are composed to form the full minimized system. The advantages of this approach are:

- Time and space requirements of minimization algorithms depend on the size of the model to which they are applied. By minimizing components instead of the full system, we expect a better overall complexity. Moreover, we will be able to minimize a system in parts even when minimizing the full system is intractable due to its size.

- It is sometimes impossible to complete the construction of the minimized system due to the size of intermediate components. In such cases, it might still be possible to apply some formal verification procedures to a partially minimized model, composed of minimized components with unminimized ones.

The *improved algorithm* we present improves a single iteration in the naive algorithm. Given two components, the improved algorithm constructs the minimized model without ever constructing their full non-minimized composition. Thus the algorithm avoids the bottleneck of the naive algorithm. We present two versions of the improved algorithm. The first is for deterministic systems and the second is for nondeterministic ones. While the version for nondeterministic systems is more general, it has worse complexity. Since deterministic systems are widely used in the hardware industry, a special more efficient version for these systems is worth developing.

Our work includes an implementation of the improved algorithm. The implementation was done on an Intel verification platform at the sequential equivalence verification CAD group of Intel design technology in Haifa. We tested our algorithm on real designs. The results imply that this method has a real potential in making bisimulation minimization practical.

1.6 Using BDDs in formal verification

Symbolic verification is a technique to overcome the state explosion problem, which is orthogonal to abstraction. Symbolic verification refers to sets of states rather than each state separately. The advantage of symbolic representation is that the size of the data-structure does not depend on the size of set it represents. In practice, it often enables the representation and manipulation of large sets of states, saving a considerable amount of space.

BDDs [Bry86] are widely used in symbolic model checking and equivalence checking. Efficient representation and manipulation of sets and relations by BDDs has been the subject of extensive research. Special attention is dedicated to representing and manipulating functions. Functions are important because they represent the transition relation of most hardware systems. Since in general functions can be represented more efficiently than relations, it is worth developing special algorithms that manipulate them. Indeed, several suggestions for function manipulations have been made for BDDs [BCL91, CM90a, MKRS00, CHJ+90, CM90b].

There are two ways to represents functions: *monolithic* and *partitioned* and three approaches for manipulating them: (1) using the monolithic representation as a binary relation, (2)using early quantification [BCL91, GB94], and (3) using the *expound* and *restrict* subroutines as suggested in [CM90a, CM90b].

Although, all three approaches are used in practice for image computation [MKRS00], when it comes to preimage computation, the expound subroutine suggested in [CM90a, CM90b] cannot compete with either the monolithic algorithm or the early quantification algorithm. The inefficiency of the expound subroutine is due to the information which is attached to every BDD node. Since the expound subroutine works from the root of the BDD to its leaves, the information attached to the BDD nodes does not depend on the set of elements that the node represents. This disables some important optimizations, which are implemented for other BDD operations.

In Chapter 6 we improve the algorithm suggested in [CM90a, CM90b] for the preimage operation. We suggest a new *inverse algorithm* with the same complexity as the expound subroutine but with better constants. Furthermore, the information, which is attached to every BDD node, represents the preimage of the set represented by this node. Thus, the implementation of the inverse algorithm is much simpler and more intuitive; moreover, it is suitable for optimizations. Experimental results show that the inverse algorithm works significantly more efficiently than the expound subroutine, and in some cases even competes successfully with the monolithic algorithm and the early quantification algorithm.

Partial results of shapres 5 and 6 are presented in [BG01].

Chapter 2

Preliminaries

Let AP be a set of atomic propositions. A *Kripke structure* M over AP is a four-tuple $M = (S, s_0, R, L)$ where:

- S is a finite set of states.
- $s_0 \in S$ is the initial state.
- $R \subseteq S \times S$ is the transition relation that must be *total*, i.e., for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
- $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

The *size* $|M|$ of a Kripke structure M is the pair $(|S|, |R|)$. We say that $|M| \leq |M'|$ if $|S| \leq |S'|$ and $|R| \leq |R'|$.

Let s be a state in a Kripke structure M. A *trace* in M starting from s is an infinite sequence of states $\rho = s_0 s_1 s_2 \ldots$ such that $s_0 = s$, and for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. The i-th state of trace ρ is denoted ρ^i.

The logic ACTL* [GL94] is the universal fragment of the powerful branching-time logic CTL*. ACTL* consists of the temporal operators **X** (next-time), **U** (until) and **R** (release), as well as the universal path quantifier **A** (for all paths). We define ACTL* formulas in negation normal form, namely, negation is applied only to atomic propositions. ACTL*contains trace formulas and state formulas and is defined inductively:

- Let p be an atomic proposition, then p and $\neg p$ are both a state formulas and a trace formulas.
- Let φ and ψ be trace formulas, then
  - $(\varphi \vee \psi)$ and $(\varphi \wedge \psi)$ are trace formulas.
  - $\mathbf{X}\varphi$, $(\varphi \mathbf{U} \psi)$ and $(\varphi \mathbf{R} \psi)$ are trace formulas.
  - $A\varphi$ is a state formula.
- Let φ and ψ be state formulas, then
  - $(\varphi \vee \psi)$ and $(\varphi \wedge \psi)$ are state formulas.
  - $\mathbf{X}\varphi$, $(\varphi \mathbf{U} \psi)$ and $(\varphi \mathbf{R} \psi)$ are trace formulas.

Next we define the semantics of ACTL* with respect to Kripke structures. A state formula ϕ is satisfied by a structure M at state s, denoted $M, s \models \phi$, if the following holds (M is omitted if clear from the context):

- For $p \in AP$, $s \models p$ iff $p \in L(s)$; $s \models \neg p$ iff $p \notin L(s)$.
- $s \models \phi \wedge \psi$ iff $s \models \phi$ and $s \models \psi$; $s \models \phi \vee \psi$ iff $s \models \phi$ or $s \models \psi$.
- $s \models \mathbf{A}\varphi$ iff for every trace ρ from s, $\rho \models \varphi$.

A trace formula φ is satisfied by a trace ρ, denoted $\rho \models \varphi$, if the following holds

- $\rho \models \mathbf{X}\,\phi$ iff $\rho^1 \models \phi$.
- $\rho \models \mathbf{A}[\varphi\,\mathbf{U}\,\psi]$ iff for some $i \geq 0$, $\rho^i \models \psi$ and for all $j < i$, $\rho^j \models \phi$.
- $\rho \models \mathbf{A}(\phi\,\mathbf{R}\,\psi)$ iff for all $i \geq 0$, if for every $j < i$, $\rho^j \not\models \phi$ then $\rho^i \models \psi$.

ACTL is a subset of ACTL* where every temporal operator is immediately proceeded by the $\mathbf{A}$ quantifier.

Definition 2.0.1 *Given two structures M_1 and M_2 over AP, a relation $H \subseteq S_1 \times S_2$ is a simulation relation [Mil71] over $M_1 \times M_2$ iff the following conditions hold:*

1. *For every $s_{01} \in S_{01}$ there exists $s_{02} \in S_{02}$ such that $(s_{01}, s_{02}) \in H$.*

2. *For all* $(s_1, s_2) \in H$,

   (a) $L_1(s_1) = L_2(s_2)$ *and*

   (b) $\forall s'_1[(s_1, s'_1) \in R_1 \rightarrow \exists s'_2[(s_2, s'_2) \in R_2 \wedge (s'_1, s'_2) \in H]]$.

We say that M' *simulates* M (denoted by $M \leq M'$) if there exists a simulation relation H over $M \times M'$. The following lemma and theorem have been proven in [GL94].

Lemma 2.0.2 $\leq$ *is a preorder on the set of structures.*

Theorem 2.0.3 *Suppose* $M \leq M'$. *Then for every* $ACTL^*$ *formula* f, $M' \models f$ *implies* $M \models f$.

Given two Kripke structures M, M', we say that M is *simulation equivalent* to M' iff $M \leq M'$ and $M' \leq M$. It is easy to see that this is an equivalence relation.

A simulation relation H over $M \times M'$ is *maximal* iff for all simulation relations H' over $M \times M'$, $H' \subseteq H$.

It follows from [GL94] that if there is a simulation relation over $M \times M'$, then there is a *unique* maximal simulation over $M \times M'$.

Definition 2.0.4 *Given two structures* M_1 *and* M_2 *over* AP, *a relation* $H \subseteq S_1 \times S_2$ *is a bisimulation relation [Par81] over* $M_1 \times M_2$ *iff the following conditions hold:*

1. *For every* $s_{01} \in S_{01}$ *there exists* $s_{02} \in S_{02}$ *such that* $(s_{01}, s_{02}) \in H$ *and for every* $s_{02} \in S_{02}$ *there exists* $s_{01} \in S_{01}$ *such that* $(s_{01}, s_{02}) \in H$.

2. *For all* $(s_1, s_2) \in H$,

   (a) $L_1(s_1) = L_2(s_2)$ *and*

   (b) $\forall s'_1[(s_1, s'_1) \in R_1 \rightarrow \exists s'_2[(s_2, s'_2) \in R_2 \wedge (s'_1, s'_2) \in H]]$.

   (c) $\forall s'_2[(s_2, s'_2) \in R_2 \rightarrow \exists s'_1[(s_1, s'_1) \in R_2 \wedge (s'_1, s'_2) \in H]]$.

Definitions 2.0.1 and 2.0.4 imply that every pair of structures M and M' that are bisimulation equivalent satisfies, $M \leq M'$. Since bisimulation is a symmetric relation, they also satisfy $M' \leq M$. Thus bisimulation implies simulation equivalence.

However, bisimulation and simulation equivalence are not equivalent. As already shown in the introduction, the difference between bisimulation and simulation equivalence results in different minimal structures with respect to these relations. Next, we show that the difference in the sizes of the minimal structures can be non elementary. We present a sequence of structures $M_1, M_2, \ldots$ such that the size of the minimal structure which is bisimulation equivalent to M_n is non-elementary in n, and the size of the minimal structure which is simulation equivalent to M_n is $n+2$.

All the structures in the sequence are defined over $AP = \{a\}$ and have a single initial state. The structure M_n contains $n+1$ layers. The higher layer contains the initial state. The lower layer contains two states, one is labeled with a and the other is not labeled. The number of states in the j'th layer (counting from bottom up) is non elementary in j, the states are not labeled. For each state in the j'th layer, the set of its successors is a different subset of the states of layer $j-1$. For every subset L of the states of the j layer, there exists a state s in layer $j+1$ such that L is the set of successors of s. The initial state is the predecessor of all the states in the n'th layer. Figure 2.1 presents the structures M_1, M_2, and M_3.

It can be proven by induction over the layers that there are no bisimulation equivalent states in the same layer in M_n, thus the size of every structure that is bisimulation equivalent to M_n is non-elementary in n.

However, there exists a structure M'_n of size $n+2$ which is simulation equivalent to M_n. The structure M'_n contains $n+1$ layers. The higher layer contains the initial state. The lower layer contains two states, one is labeled with a and the other is not labeled. The j'th layer contains one state which is not labeled and has the state below it as a successor. The state at layer 2 has both states of layer 1 as successors. The structures M'_1, M'_2, and M'_3 are shown in Figure 2.1. In order to prove that $M'_n \leq M_n$, we select a path form the initial state of M_n to the state in layer 2 which is connected to both states in layer 1. We define the simulation relation simply by relating each state in M'_n to its corresponding state in the selected path. In order to prove that $M_n \leq M'_n$ we define a simulation relation H that for every layer, relates all the states in the layer to the state in the corresponding layer in M'_n.

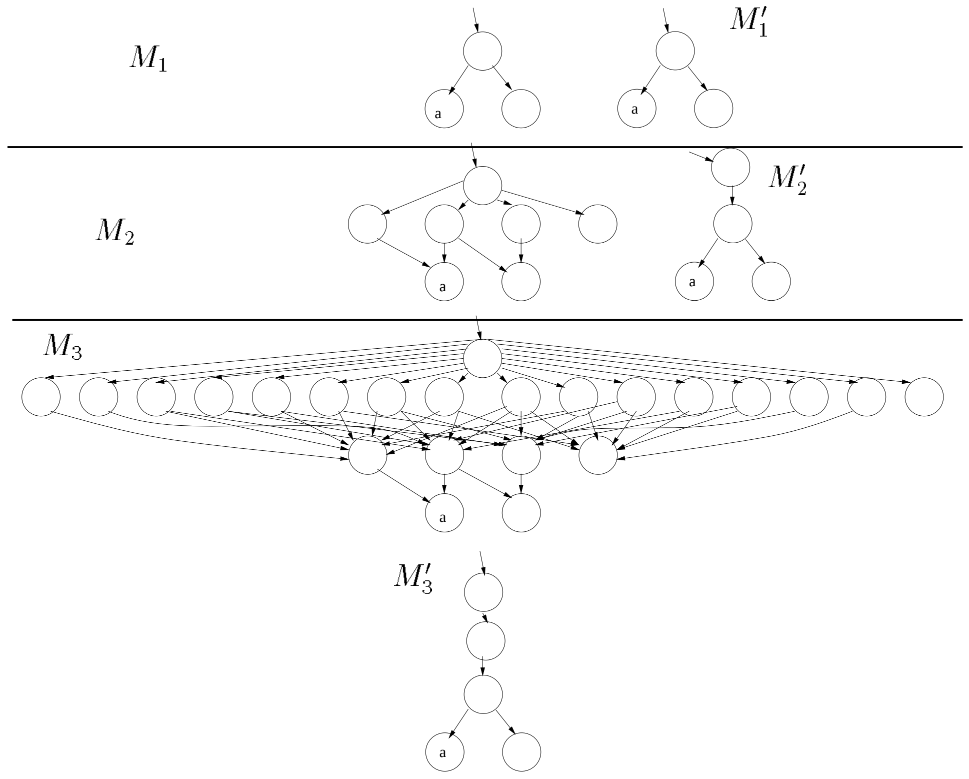

Figure 2.1: Structures M_1, M_2, and M_3 are simulation equivalent to structures M'_1, M'_2, and M'_3 respectively.

It is easy to see that H is a simulation relation.

Theorem 2.0.5 *Suppose that M, M' are bisimulation equivalent. Then for every CTL* formula f, $M' \models f$ if and only if $M \models f$.*

Chapter 3

Simulation based minimization

Given a Kripke structure M, we would like to find a structure M' that is simulation equivalent to M and is the smallest in size (number of states and transitions).

For bisimulation this can be done by constructing the *quotient structure* in which the states are the equivalence classes with respect to bisimulation. Bisimulation has the property that if one state in a class has a successor in another class then all states in the class have a successor in the other class. Thus, in the quotient structure there will be a transition between two classes if every (some) state in one class has a successor in the other. The resulting structure is the smallest in size that is bisimulation equivalent to the given structure M.

The quotient structure for simulation equivalence can be constructed in a similar manner. There are two main difficulties, however. First, when constructing the quotient structure for simulation equivalence, not all the states in an equivalence class have successors in the same class. Thus, there are a few possible ways to define a transition between classes. We define two optional transition relations, similar to those used in [DGG97]. The first defines a transition between classes whenever *all* states of one class have a successor in the other. The resulting structure is called the $\forall-$quotient structure. The second defines a transition between classes whenever there *exists* a state of one with a successor in the other. The resulting structure is called the $\exists-$quotient structure. In both cases, the structures are simulation equivalent to M, but the $\forall-$quotient structure has fewer transitions and therefore is preferable.

The other difficulty is that the quotient model for simulation equivalence is *not* the smallest in size. Actually, it is not even clear that there is a unique smallest structure that is simulation equivalent to M.

In this chapter we prove that in simulation based minimization, beside equivalent states, there exists another redundancy. Little brothers are states that are smaller by the simulation preorder than one of their brothers. Disconnecting a little brother from the mutual parent eliminates the additional redundancy, and the result of disconnecting little brothers is simulation equivalent to the original model. The definition of little brothers is not new. [KM99] shows that eliminating little brothers results in a simulation equivalent structure. However, the paper does not consider the minimization problem.

The first result in this chapter is a proof that every structure has a *unique up to isomorphism* smallest structure that is simulation equivalent to it. This structure is *reduced*, meaning that it contains no simulation equivalent states, no little brothers and no unreachable states. Our next result is the Minimizing Algorithm that given a structure M constructs the reduced structure for M. Based on the maximal simulation relation over M, the algorithm first builds the $\forall-$quotient structure with respect to simulation equivalence. Then it eliminates transitions to little brothers. Finally, it removes unreachable states. The time complexity of the algorithm is $O(|S|^3)$. Its space complexity is $O(|S|^2)$ which is due to the need to hold the simulation preorder in memory.

Since our main concern is space requirements, we suggest the Partition Algorithm, which computes the quotient structure without ever computing the simulation preorder. Similarly to [LY92], the algorithm starts with a partition Σ_0 of the state space to classes whose states are equally labeled. It also initializes a preorder H_0 over the classes in Σ_0. At iteration $i+1$, Σ_{i+1} is constructed by splitting classes in Σ_i. The relation H_{i+1} is updated based on Σ_i, Σ_{i+1} and H_i.

When the algorithm terminates (after k iterations) Σ_k is the set of equivalence classes with respect to simulation equivalence. These classes form the states of the quotient structure. The final H_k is the maximal simulation preorder over the states of the quotient structure. Thus, the Partition Algorithm replaces the first step of the Minimizing Algorithm . Since every step in the Minimizing Algorithm further reduces the size of the initial structure, the first step handles the largest structure. Therefore, improving its complexity influences most the overall complexity of the algorithm.

The space complexity of the Partition Algorithm is $O(|\Sigma_k|^2 + |S| \cdot log(|\Sigma_k|))$. We assume that in most cases $|\Sigma_k| << |S|$, thus this complexity is significantly smaller than that of the Minimizing Algorithm . Unfortunately, time complexity will probably become worse (depending on the size of Σ_k). It is bounded by $O(|S|^2 \cdot |\Sigma_k|^2 \cdot (|\Sigma_k|^2 + |R|))$. However, since our main concern is the reduction in memory requirements, the Partition Algorithm is valuable.

3.1 The reduced structure

Given a Kripke structure M, we would like to find a *reduced* structure that will be simulation equivalent to M and smallest in size. In this section we prove that a reduced structure always exists. Furthermore, we show that all reduced structures of M are *isomorphic*.

Let M be a Kripke structure and H be the maximal simulation relation over $M \times M$. We need the following two definitions in order to characterize reduced structures.

Two states $s_1, s_2 \in M$ are *simulation equivalent* iff $(s_1, s_2) \in H$ and $(s_2, s_1) \in H$. Note that simulation equivalence is an equivalence relation: the transitivity and reflexivity of H imply the transitivity and reflexivity of the relation, and the symmetry of the equivalence relation comes from its definition.
A state s_1 is a *little brother* of a state s_2 iff there exists a state s_3 such that:

- $(s_3, s_2) \in R$ and $(s_3, s_1) \in R$.
- $(s_1, s_2) \in H$ and $(s_2, s_1) \notin H$.

Definition 3.1.1 *A Kripke structure M is* reduced *if:*

1. There are no simulation equivalent states in M.

2. *There are no states s_1, s_2 such that s_1 is a little brother of s_2.*

3. *All states in M are reachable from s_0.*

Theorem 3.1.2 : *Let M, M' be two reduced Kripke structures. Then the following two statements are equivalent:*

1. *M and M' are simulation equivalent.*

2. *M and M' are isomorphic.*

The proof that 2 implies 1 is straightforward. In the rest of this section we assume that M and M' are reduced Kripke structures. We will show that if $M \leq M'$ and $M' \leq M$, then M and M' are isomorphic.

We use $H_{MM'}$ to denote the maximal simulation over $M \times M'$, and $H_{M'M}$ to denote the maximal simulation over $M' \times M$. The *composed* relation $H_{MM'M} \subseteq S \times S$ is defined by

$$H_{MM'M} = \{(s_1, s_2) | \exists s' \in S'.\ (s_1, s') \in H_{MM'} \wedge (s', s_2) \in H_{M'M}\}.$$

Lemma 3.1.3 *Given two structures M and M', and two simulation relations $H_{MM'} \subseteq S \times S'$ and $H_{M'M} \subseteq S' \times S$, the composition relation $H_{MM'M}$ of $H_{MM'}$ and $H_{M'M}$ is a simulation relation.*

Proof :

- $(s_0, s'_0) \in H_{MM'}$ and $(s'_0, s_0) \in H_{M'M}$ implies $(s_0, s_0) \in H_{MM'M}$.

- $(s_1, s_2) \in H_{MM'M}$ implies that there exists a state s' in M' such that $(s_1, s') \in H_{MM'}$ and $(s', s_2) \in H_{M'M}$. Thus, $L(s_1) = L'(s') = L(s_2)$.

- Let $(s_1, s_2) \in H_{MM'M}$ and let t_1 be a successor of s_1. We will show that there exists a successor t_2 of s_2 such that $(t_1, t_2) \in H_{MM'M}$.

  - $(s_1, s_2) \in H_{MM'M}$ implies that there exists s' such that $(s_1, s') \in H_{MM'}$ and $(s', s_2) \in H_{M'M}$.
  - $(s_1, s') \in H_{MM'}$ implies that there exists a successor $t' \in S'$ of s' such that $(t_1, t') \in H_{MM'}$.
  - $(s', s_2) \in H_{M'M}$ implies that there exists a successor $t_2 \in S$ of s_2 such that $(t', t_2) \in H_{M'M}$.

– By the above, $(t_1, t_2) \in H_{MM'M}$. □

Given two reduced Kripke structures M and M' that are simulation equivalent, we will define a *matching* relation f over $S' \times S$ based on the two simulation relations between the structures. We show that f is an isomorphism between M' and M, i.e., f is a one-to-one and onto total function that preserves the labeling of states and the transition relation.

Definition 3.1.4 *The* matching relation *$f \subseteq S' \times S$ is defined as follows: $(s', s) \in f$ iff $(s', s) \in H_{M'M}$ and $(s, s') \in H_{MM'}$.*

Lemma 3.1.5 *Let $f \subseteq S' \times S$ be the matching relation. Then f is a one-to-one, onto, and total function from S' to S.*

Proof : First we prove that f is a function from S' to S. Assume to the contrary that there are two different states, s_1, s_2, in S and a state s' in S' such that $(s', s_1) \in f$ and $(s', s_2) \in f$. Let $H_{MM'M}$ be the composed relation. Since $H_{MM'M}$ is a simulation relation, it is included in the maximal simulation over $M \times M$. We will show that $(s_1, s_2) \in H_{MM'M}$ and $(s_2, s_1) \in H_{MM'M}$, which contradicts the assumption that M is reduced.

- $(s', s_1) \in f$ implies that $(s', s_1) \in H_{M'M}$ and $(s_1, s') \in H_{MM'}$.
- $(s', s_2) \in f$ implies that $(s', s_2) \in H_{M'M}$ and $(s_2, s') \in H_{MM'}$.
- $(s_1, s') \in H_{MM'}$ and $(s', s_2) \in H_{M'M}$ implies that $(s_1, s_2) \in H_{MM'M}$.
- $(s_2, s') \in H_{MM'}$ and $(s', s_1) \in H_{M'M}$ implies that $(s_2, s_1) \in H_{MM'M}$.

The proof that f^{-1} is a function from S to S' is similar. Thus, we conclude that f is a one-to-one function.

Next, we prove that f is onto, i.e., for every state s in S there exists a state s' in S' such that $(s', s) \in f$. The proof is by induction on the distance of $s \in S$ from the initial state. (Since all states are reachable, the distance is bounded by $|S|$.)

- Base: The case where the distance is 0 follows from the fact that simulation relations relate initial states to each other. Thus, $(s'_0, s_0) \in H_{M'M}$ and $(s_0, s'_0) \in H_{MM'}$.

- Induction step: Assume that the induction hypothesis holds for every state with distance less than or equal to n. We prove it for states with distance $n+1$. Let $t_1 \in S$ be a state with distance $n+1$. Then there is a state s with distance n such that $(s, t_1) \in R$. By the induction hypothesis, there exists a state s' in S' such that$(s, s') \in H_{MM'}$ and $(s', s) \in H_{M'M}$. By the definition of simulation, for every successor of s, in particular t_1, there exists in S' a successor t'_1 of s', such that $(t_1, t'_1) \in H_{MM'}$. If, in addition, $(t'_1, t_1) \in H_{M'M}$, then $(t'_1, t_1) \in f$ and we are done.

  Assume to the contrary that $(t'_1, t_1) \notin H_{M'M}$. Then $(s', s) \in H_{M'M}$ implies that there exists t_2, such that $(s, t_2) \in R$ and $(t'_1, t_2) \in H_{M'M}$. Let $H_{MM'M}$ be the composed simulation relation. Then, $H_{MM'M}$ is included in the maximal simulation over $M \times M$. $(t_1, t'_1) \in H_{MM'}$ and $(t'_1, t_2) \in H_{M'M}$ imply $(t_1, t_2) \in H_{MM'M}$. However, t_1, t_2 are both successors of s. This implies that either t_1, t_2 are simulation equivalent or t_1 is a little brother of t_2, contradicting the assumption that M is reduced.

A similar proof can be applied to show that f^{-1} is onto, which implies that f is total. $\Box$

Lemma 3.1.6 *Let $s', t' \in S'$ be states. Then $(s', t') \in R'$ iff $(f(s'), f(t')) \in R$.*

Proof : We prove that if $(s', t'_1) \in R'$, then $(f(s'), f(t'_1)) \in R$. The proof of the other direction is similar. Let $s', t'_1 \in S'$ be two states such that $(s', t'_1) \in R'$ and let $s, t_1 \in S$ be states such that $f(s') = s$ and $f(t'_1) = t_1$. Assume to the contrary that $(s, t_1) \notin R$. Then $(s', s) \in H_{M'M}$ implies that there exists t_2 such that $(s, t_2) \in R$ and $(t'_1, t_2) \in H_{M'M}$. Moreover, $(s, s') \in H_{MM'}$ implies that there exists t'_2 such that $(s', t'_2) \in R'$ and $(t_2, t'_2) \in H_{MM'}$. We distinguish between two cases:

1. If $t'_2 = t'_1$ then $f(t'_1) = t_2$, contradicting the assumption that f is a function.

2. Otherwise, let $H_{M'MM'}$ be the composed simulation relation over $M' \times M'$. Therefore, it is included in the maximal simulation over $M' \times M'$. $(t'_1, t_2) \in H_{M'M}$ and $(t_2, t'_2) \in H_{MM'}$ imply $(t'_1, t'_2) \in H_{M'MM'}$. This implies that either t'_1, t'_2 are simulation equivalent or t'_1 is a little brother of t'_2, contradicting the assumption that M' is reduced.

□

Proposition 3.1.7 *For all $s' \in S'$, $L'(s') = L(f(s'))$.*

Proof : Immediate by definition of f.

We showed that if reduced structures M and M' are simulation equivalent, then there exists a one-to-one, onto, total function $f : S' \rightarrow S$ such that for every s', $L'(s') = L(f(s'))$ and for every s', t', $(s', t') \in R'$ iff $(f(s'), f(t')) \in R$. Thus, we conclude Theorem 3.1.2 .

Theorem 3.1.8 *Let M be a non-reduced Kripke structure. Then there exists a reduced Kripke structure M' such that M, M' are simulation equivalent and $|M'| < |M|$.*

In order to prove Theorem 3.1.8 , we present in the next sections an algorithm that receives a Kripke structure M and computes a reduced Kripke structure M'. This reduced structure is simulation equivalent to M, such that $|M'| \leq |M|$. Moreover, if M is not reduced, then $|M'| < |M|$.

The following lemma shows that the reduced structures are strictly smaller than any other structure that is simulation equivalent to them.

Lemma 3.1.9 *Let M' be a reduced Kripke structure. For every M that is simulation equivalent to M', if M and M' are not isomorphic then $|M'| < |M|$.*

Proof : By Theorem 3.1.2 , since M is not isomorphic to M', M is not reduced. By Theorem 3.1.8 , there exists a reduced Kripke structure M'' that is simulation equivalent to M, and $|M''| < |M|$. M'' and M' are both simulation equivalent to M and therefore are simulation equivalent to each other. Since they are reduced, they are also isomorphic, and therefore $|M'| = |M''|$. Thus, $|M'| < |M|$. □

3.2 The minimizing algorithm

In this section we present the Minimizing Algorithm. This algorithm gets a Kripke structure M and computes a reduced Kripke structure M', that is simulation equivalent to M and $|M'| \leq |M|$. If M is not reduced then $|M'| < |M|$.

The algorithm consists of three steps. The first step is to construct a quotient structure in order to eliminate equivalent states. The resulting quotient structure is simulation equivalent to M but is not necessarily reduced. The second step is to disconnect little brothers, and the last step is to remove all unreachable states.

If the structure obtained at each step differs from the original, then it is strictly smaller than the original.

3.2.1 The $\forall-$quotient structure

In order to compute a simulation equivalent structure that contains no equivalent states, we compute the quotient structure with respect to the simulation equivalence relation. The states of the structure are the equivalence classes and the labeling function is straightforward (all states in a given equivalence class have the same labeling, so we use this label for the class as well). However, the transition relation is not uniquely defined. We can have a transition between two equivalence classes if from *every* state of one there is a transition to some state of the other ($\forall$-transitions). We can also have a transition if there exists a state in one with a transition to some state of the other ($\exists$-transitions). Both definitions will result in a simulation equivalent structure. However, the former has a smaller transition relation, and therefore it is preferable.

In the rest of this section we present the $\forall-$quotient structure and prove that it is simulation equivalent to the original structure. If the quotient structure is not isomorphic to the original one, then it is strictly smaller in size.

For the rest of this section we fix M to be the original Kripke structure and H to be the maximal simulation relation over $M \times M$. We denote by $[s]$ the equivalence class that includes s.

Definition 3.2.1 *The $\forall-$quotient structure $M_q =< S_q, R_q, s_{0_q}, L_q >$ of M is defined as follows:*

- *S_q is the set of the equivalence classes of the simulation equivalence. (We will use Greek letters to represent equivalence classes.)*
- *$R_q = \{(\alpha_1, \alpha_2) | \forall s_1 \in \alpha_1 \; \exists s_2 \in \alpha_2 . \; (s_1, s_2) \in R\}$.*
- *$s_{0_q} = [s_0]$.*
- *$L_q([s]) = L(s)$.*

Note that $|S_q| \leq |S|$ and $|R_q| \leq |R|$. If $|S_q| = |S|$, then every equivalence class contains a single state. In this case, R_q is identical to R and M_q is isomorphic to M. Thus, when M and M_q are not isomorphic, $|S_q| < |S|$ and $|R_q| < |R|$.
Next, we show that M and M_q are simulation equivalent.

Definition 3.2.2 *Let $G \subseteq S$ be a set of states. A state $s_m \in G$ is maximal in G iff there is no state $s \in G$ such that $(s_m, s) \in H$ and $(s, s_m) \notin H$.*

Definition 3.2.3 *Let α be a state of M_q, and t_1 be a successor of some state in α. The set $G(\alpha, t_1)$ is defined as follows:*

$$G(\alpha, t_1) = \{t_2 \in S | \exists s_2 \in \alpha \wedge (s_2, t_2) \in R \wedge (t_1, t_2) \in H\}.$$

Intuitively, $G(\alpha, t_1)$ is the set of states that are greater than t_1 and are successors of states in α. Notice that since all states in α are simulation equivalent, every state in α has at least one successor in $G(\alpha, t_1)$.

Lemma 3.2.4 *Let α, t_1 be as defined in Definition 3.2.3 . Then for every maximal state t_m in $G(\alpha, t_1)$, $[t_m]$ is a successor of α.*

Proof : Let t_m be a maximal state in $G(\alpha, t_1)$, and let $s_m \in \alpha$ be a state such that t_m is a successor of s_m. We prove that for every state $s \in \alpha$, there exists a successor $t \in [t_m]$. This implies that $[t_m]$ is a successor of α.

$s, s_m \in \alpha$ implies that $(s_m, s) \in H$. This implies that there exists a successor t of s such that $(t_m, t) \in H$. By transitivity of the simulation relation, $(t_1, t) \in H$. Thus $t \in G(\alpha, t_1)$. Since t_m is maximal in $G(\alpha, t_1)$, $(t, t_m) \in H$. Thus, t and t_m are simulation equivalent and $t \in [t_m]$. $\square$

Theorem 3.2.5 *The structures M and M_q are simulation equivalent.*

Proof : First we prove that $M_q \leq M$. Let $H^{qs} \subseteq S_q \times S$ be the relation $H^{qs} = \{(\alpha, s) | s \in \alpha\}$. We prove that H^{qs} is a simulation relation.

- $s_0 \in s_{0_q}$ implies that $(s_{0_q}, s_0) \in H^{qs}$.
- By the definition of L_q, $(\alpha, s) \in H^{qs}$ implies that $L(s) = L_q(\alpha)$.
- Assume $(\alpha_1, s) \in H^{qs}$ and let α_2 be a successor of α_1. Then by the definition of R_q, there exists a successor t of s such that $t \in \alpha_2$. Thus, $(\alpha_2, t) \in H^{qs}$.

Second, we prove that $M \leq M_q$. Let $H^{sq} \subseteq S \times S_q$ be the relation $H^{sq} = \{(s_1, \alpha)|$ there exists a state $s_2 \in \alpha$ such that $(s_1, s_2) \in H\}$. We prove that H^{sq} is a simulation relation.

- $(s_0, s_0) \in H$ and $s_0 \in s_{0_q}$ imply that $(s_0, s_{0_q}) \in H^{sq}$.
- $(s_1, \alpha) \in H^{sq}$ implies that there exists a state $s_2 \in \alpha$ such that $(s_1, s_2) \in H$. Thus, $L(s_1) = L(s_2) = L_q(\alpha)$.
- Assume $(s_1, \alpha_1) \in H^{sq}$ and let t_1 be a successor of s_1. We prove that there exists a successor α_2 of α_1 such that $(t_1, \alpha_2) \in H^{sq}$. We distinguish between two cases:
  1. $s_1 \in \alpha_1$. Let t_m be a maximal state in $G(\alpha_1, t_1)$. Lemma 3.2.4 then implies that $(\alpha_1, [t_m]) \in R_q$. Since t_m is maximal in $G(\alpha_1, t_1)$, $(t_1, t_m) \in H$, which implies $(t_1, [t_m]) \in H^{sq}$.
  2. $s_1 \notin \alpha_1$. Let $s_2 \in \alpha_1$ be a state such that $(s_1, s_2) \in H$. Since $(s_1, s_2) \in H$, there is a successor t_2 of s_2 such that $(t_1, t_2) \in H$. The first case implies that there exists an equivalence class α_2 such that $(\alpha_1, \alpha_2) \in R_q$ and $(t_2, \alpha_2) \in H^{sq}$. By $(t_2, \alpha_2) \in H^{sq}$, there exists a state $t_3 \in \alpha_2$ such that $(t_2, t_3) \in H$. By transitivity of simulation, $(t_1, t_3) \in H$. Thus, $(t_1, \alpha_2) \in H^{sq}$.

$\square$

3.2.2 Disconnecting little brothers

Our next step is to disconnect the little brothers from their parents. By applying this step to a Kripke structure M with no equivalent states, we get a Kripke structure M' satisfying the following:

1. M are M' are simulation equivalent.
2. There are no equivalent states in M'.
3. There are no little brothers in M'.
4. $|M'| \leq |M|$, and if M and M' are not identical, then $|M'| < |M|$.

In Figure 3.1 we present an iterative algorithm which disconnects little brothers and results in M'.

```
change := true
while (change = true) do
    Compute the maximal simulation relation $H$
    change := false
    If there are $s_1, s_2, s_3 \in S$ such that $s_1$ is a little brother of $s_2$
             and $s_3$ is the father of both $s_1$ and $s_2$ then
       change := true
       $R = R \setminus \{(s_3, s_1)\}$
    end
 end
```

Figure 3.1: The Disconnecting Algorithm

Since each iteration of the algorithm removes one transition, the algorithm will terminate after at most $|R|$ iterations. We will show that the resulting structure is simulation equivalent to the original one.

Lemma 3.2.6 *Let $M' = < S', R', s'_0, L' >$ be the result of the Disconnecting Algorithm on M. Then M and M' are simulation equivalent.*

Proof : We prove the lemma by induction on the number of iterations.

- Base: at the beginning, M and M are simulation equivalent.
- Induction step: Let $M'' =< S'', R'', s''_0, L'' >$ be the result of the first i iterations and H'' be the maximal simulation over $M'' \times M''$. Let $M' =< S', R', s'_0, L' >$ be the result of the $(i+1)$th iteration where $R' = R'' \backslash \{(p''_1, p''_2)\}$. Assume that M and M'' are simulation equivalent. We first prove that $M' \leq M''$. We choose $H' \subseteq S' \times S''$ to be $H' = \{(s'_1, s''_2) | (s''_1, s''_2) \in H''\}$. Since M' is obtained from M'' by removing one transition, clearly H' is a simulation relation.

  We now show that $M'' \leq M'$. As in the previous case, we choose $H' \subseteq S'' \times S'$ to be $H' = \{(s''_1, s'_2) | (s''_1, s''_2) \in H''\}$. We will prove that H' is a simulation relation.

  - $(s''_0, s''_0) \in H''$ implies that $(s''_0, s'_0) \in H'$.
  - $(s''_1, s'_2) \in H'$ implies that $L''(s''_1) = L'(s'_2)$.
  - Suppose $(s''_1, s'_2) \in H'$ and t''_1 is a successor of s''_1. Since H'' is a simulation relation, there exists a successor t''_2 of s''_2 such that $(t''_1, t''_2) \in H''$. This implies that $(t''_1, t'_2) \in H'$. If $(s'_2, t'_2) \in R'$, then we are done. Otherwise, (s''_2, t''_2) is removed from R'' because t''_2 is a little brother of some successor t''_3 of s''_2. Since (s''_2, t''_2) is the only transition removed at the $(i+1)$th iteration, $(s'_2, t'_3) \in R'$. Because t''_2 is a little brother of t''_3, then $(t''_2, t''_3) \in H''$. By transitivity of the simulation relation, $(t''_1, t''_3) \in H''$, and thus $(t''_1, t'_3) \in H'$.

□

We proved that the structure M' that is computed by the Disconnecting Algorithm is simulation equivalent to the original structure M. Note that M' has the same set of states as M. We now show that the maximal simulation relation over M is identical to the maximal simulation relations for all intermediate structures M'' (including M') computed by the Disconnecting Algorithm. Therefore this relation can be computed once, at the beginning of the algorithm. Moreover, since there are no simulation equivalent states in M, there are no such states in M' either.

Lemma 3.2.7 *Let $H \subseteq S \times S$ be the maximal simulation relation over $M \times M$. Let $M' =< S, R', s_0, L >$ be the result of the Disconnect-*

ing Algorithm on M and let $H' \subseteq S' \times S'$ be the maximal simulation over $M' \times M'$. Then $H = H'$.

Proof : The Disconnecting Algorithm changes only the transition relation. Thus, for all intermediate structures M'', $S'' = S$, $s_0'' = s_0$, and $L'' = L$. We prove the lemma by induction on the number of iterations.

- Base: at the beginning, $H = H$.
- Induction step: Let $M'' =< S, R'', s_0, L >$ be the result of the first i iterations and let H'' be the maximal simulation relation over $M'' \times M''$. Assume that $H'' = H$. Let M' be the result of the $(i+1)$th iteration and H' be the maximal simulation relation over $M' \times M'$. We prove that $H' = H''$. First we prove that H'' is a simulation relation over $M' \times M'$. This implies that $H'' \subseteq H'$ (H' is maximal over $M' \times M'$).
  - $(s_0, s_0) \in H''$.
  - $(s_1, s_2) \in H''$ implies that $L(s_1) = L(s_2)$.
  - Let s_1, s_2, t_1 be states such that $(s_1, s_2) \in H''$ and $(s_1, t_1) \in R'$. $(s_1, s_2) \in H''$ implies that there exists a state t_2 such that $(s_2, t_2) \in R''$ and $(t_1, t_2) \in H''$. We distinguish between two cases:
    1. If $(s_2, t_2) \in R'$, we are done.
    2. If $(s_2, t_2) \notin R'$, then since (s_2, t_2) is removed from R'', there must exist a state t_3 such that $(t_2, t_3) \in H''$ and $(s_2, t_3) \in R''$ (t_2 is a little brother of t_3 and s_2 is the parent of both states). Since only one transition is removed, $(s_2, t_3) \in R'$. By transitivity of H'', $(t_1, t_3) \in H''$. Thus, H'' is a simulation relation over $M' \times M'$.

  Next we prove that H' is a simulation relation over $M'' \times M''$. This implies that $H' \subseteq H''$ (H'' is maximal over $M'' \times M''$).
  - $(s_0, s_0) \in H'$.
  - $(s_1, s_2) \in H'$ implies that $L(s_1) = L(s_2)$.
  - Let s_1, s_2, t_1 be states such that $(s_1, s_2) \in H'$ and $(s_1, t_1) \in R''$. We distinguish between two cases:

1. If $(s_1, t_1) \in R'$, then $(s_1, s_2) \in H'$ implies that there exists a state t_2 such that $(s_2, t_2) \in R'$ and $(t_1, t_2) \in H'$. Thus, $(s_2, t_2) \in R''$.
2. If $(s_1, t_1) \notin R'$, then since (s_1, t_1) is removed from R'', there exists a state t_3 such that $(s_1, t_3) \in R''$ and $(t_1, t_3) \in H''$ (t_1 is a little brother of t_3 and s_1 is their parent). $(t_1, t_3) \in H''$ and $H'' \subseteq H'$ implies $(t_1, t_3) \in H'$. Since (s_1, t_1) is the only transition removed from R'', $(s_1, t_3) \in R'$. This implies that there exists a state t_2 such that $(s_2, t_2) \in R'$ and $(t_3, t_2) \in H'$. By transitivity of H', $(t_1, t_2) \in H'$. Thus, $(s_2, t_2) \in R''$ and H' is a simulation relation over $M'' \times M''$.

□

The Disconnecting Algorithm can be greatly simplified as a result of the last theorem. The maximal simulation relation is computed once on the original structure M and is used in all iterations. The *Simplified Algorithm* is presented in Figure 3.2

```
Compute the maximal simulation relation $H$
for every node $s_i \in S$ do
   $G = \emptyset$
   for every transition $(s_i, s_j) \in R$ do
      for every states $s_g \in G$ do
         if $(s_g, s_j) \in H$ and $(s_j, s_g) \notin H$ then
            remove the transition $(s_i, s_g)$ from $R$
         end
         if $(s_g, s_j) \notin H$ and $(s_j, s_g) \in H$ then
            remove the transition $(s_i, s_j)$ from $R$
         end
      end
      add $s_j$ to $G$
   end
end
```

Figure 3.2: The Simplified Algorithm

If the algorithm is executed symbolically (with BDDs), then this operation can be performed efficiently in one step:

$$R' = R - \{(s_1, s_2) | \exists s_3 : (s_1, s_3) \in R \wedge (s_2, s_3) \in H \wedge (s_3, s_2) \notin H\}.$$

3.2.3 The algorithm

In Figure 3.3 we present our algorithm for constructing the reduced structure for a given one.

```
1.  Compute the $\forall-$quotient structure $M_q$ of $M$ and
          the maximal simulation relation $H$ over
$M_q \times M_q$.
2.  $R' = R_q - \{(s_1, s_2) | \exists s_3 : (s_1, s_3) \in R_q \wedge (s_2, s_3) \in H\}$.
3.  Remove all unreachable states.
```

Figure 3.3: The Minimizing Algorithm

Note that the check $(s_3, s_2) \notin H$ is eliminated in the second step. This is because M_q does not contain simulation equivalent states. Removing unreachable states in the third step does not change the properties of simulation with respect to the initial states. The size of the resulting structure is equal to or smaller than the original one. Again, if the resulting structure is not identical, then it is strictly smaller in size.

We have proved that the structure M' that results from applying the Minimizing Algorithm is simulation equivalent to the original structure M. Thus we can conclude that Theorem 3.1.8 is correct.

Figure 3.4 presents an example of the three steps of the Minimizing Algorithm applied to a Kripke structure.

1. Part 1 contains the original structure, where the maximal simulation relation is (not including the trivial pairs):
   $\{(2,3),(3,2),(11,2),(11,3),(4,5),(6,5),(7,8),(8,7),(9,10),(10,9)\}$.
   The equivalence classes are : $\{\{1\},\{2,3\},\{11\},\{4\},\{5\},\{6\},\{7,8\},\{9,10\}\}$.

2. Part 2 presents the $\forall-$structure M_q. The maximal simulation relation H is (not including the trivial pairs):
   $H = \{(\{11\},\{2,3\}),(\{4\},\{5\}),(\{6\},\{5\})\}$.

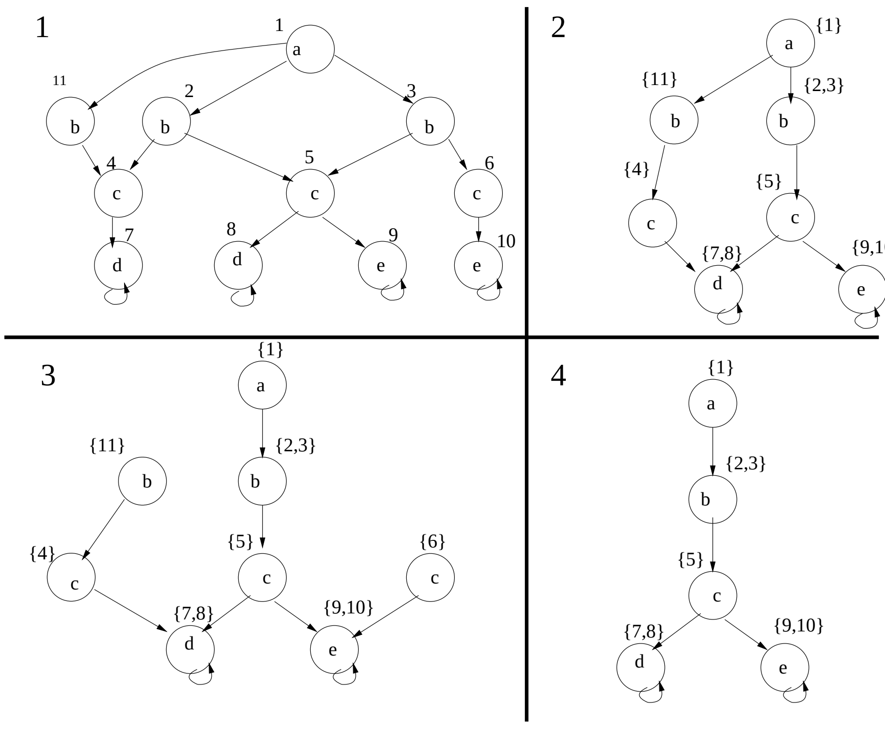

Figure 3.4: An example of the Minimizing Algorithm

3. $\{11\}$ is a little brother of $\{2, 3\}$ and $\{1\}$ is their parent. Part 3 presents the structure after the removal of the transition $(\{1\}, \{11\})$.

4. Finally, part 4 contains the reduced structure, obtained by removing the unreachable states.

3.2.4 Complexity

The complexity of each step of the algorithm depends on the size of the Kripke structure obtained from the previous step. In the worst case, the Kripke structure does not change; then all three steps depend on the original Kripke structure. Let M be the given structure. We analyze each step separately:

1. First, the algorithm constructs equivalence classes. It computes the maximal simulation relation. [BP96, HHK95] showed that

this can be done in time $O(|S| \cdot |R|)$. Once the algorithm has the simulation relation, the equivalence classes can be constructed in time $O(|S|^2)$. Next, the algorithm constructs the transition relation. This can be done in time $O(|S| + |R|)$. Building the entire quotient structure can be done in time $O(|S| \cdot |R|)$.

2. Next, little brothers are disconnected from their parents. Looking at the Simplified Algorithm in Figure 3.2, we can see that the number of iterations of the two external *for* loops is exactly $|R|$ and that the number of iterations in the inner loop is bounded by $|S|$. This implies that the overall complexity of this step is $O(|S| \cdot |R|)$.

3. Unreachable states can be removed in time $O(|R|)$.

The entire algorithm works in time $O(|S| \cdot |R|)$.

The space bottleneck of the algorithm is the computation of the maximal simulation relation, which is bounded by $|S|^2$.

3.3 Partition classes

In the previous section, we presented the Minimizing Algorithm. The algorithm consists of three steps, each of which results in a structure that is smaller in size. Since the first step handles the largest structure, improving its complexity will have the greatest influence on the overall complexity of the algorithm.

In this section we suggest an alternative algorithm for computing the set of equivalence classes. The algorithm avoids the construction of the simulation relation over the original structure. As a result, it has a better space complexity, but its time complexity is worse. Since the purpose of the Minimizing Algorithm is to reduce space requirements, reducing its own space requirement takes precedence.

3.3.1 The partition algorithm

Let $M =< S, R, s_0, L >$ be a Kripke structure and H be the maximal simulation over $M \times M$. We would like to build the equivalence classes of the simulation equivalence relation without first calculating H. Our algorithm, called the *Partitioning Algorithm*, starts with a *partition* Σ_0

of S to classes. The classes in Σ_0 differ from one another only by their state labeling. In each iteration, the algorithm refines the partition and forms a new set of classes. We use Σ_i to denote the set of the classes obtained after i iterations. In order to refine the partitions, we build an *ordering* relation H_i over $\Sigma_i \times \Sigma_i$. This relation is updated at every iteration according to the previous and current partitions (Σ_{i-1} and Σ_i) as well as the previous ordering relation (H_{i-1}). Initially, H_0 includes only the identity pairs (of classes).

In the algorithm, we use *succ(s)* for the set of successors of s. We use $[s]^i$ to denote the equivalence class of s in Σ_i. $[s]$ is used whenever Σ_i is clear from the context. We also use a function Π that associates with each class $\alpha \in \Sigma_i$ the set of classes $\alpha' \in \Sigma_{i-1}$ that contain a successor of some state in α.

$$\Pi(\alpha) = \{[t]^{i-1} | \exists s \in \alpha.\ \ (s,t) \in R\}.$$

We use the following notations:

- English letters to denote states.
- Capital English letters to denote sets of states.
- Greek letters to denote equivalence classes.
- Capital Greek letters to denote sets of equivalence classes.

The partition algorithm is presented in Figure 3.5.

Definition 3.3.1 *The partial order $\leq_i$ on S is defined as follows: $s_1 \leq_i s_2$ iff*

- *$L(s_1) = L(s_2)$.*
- *If $i > 0$, then for every successor t_1 of s_1 there exists a successor t_2 of s_2 such that $([t_1],[t_2]) \in H_{i-1}$.*

If $i = 0$, $s_1 \leq_0 s_2$ iff $L(s_1) = L(s_2)$.

Definition 3.3.2 *Two states, s_1, s_2, are $i-$*equivalent *iff $s_1 \leq_i s_2$ and $s_2 \leq_i s_1$. (Since $\leq_i$ is a partial order, $i-equivalence$ is an equivalence relation.)*

```
Initialize the algorithm:
    change := $true$
    for each label $a \in 2^{AP}$ construct $\alpha_a \in \Sigma_0$ such that $s \in \alpha_a \Leftrightarrow L(s) = a$.
    $H_0 = \{(\alpha, \alpha) | \alpha \in \Sigma_0\}$
    $i = 0$
    while $change = true$ do begin
        $change := false$
refine $\Sigma$:
        $\Sigma_{i+1} := \emptyset$
        for each $\alpha \in \Sigma_i$ do begin
            while $\alpha \neq \emptyset$ do begin
                choose $s_p$ such that $s_p \in \alpha$
                $GT := \{s_g | s_g \in \alpha \wedge \forall t_p \in succ(s_p)\ \exists t_g \in succ(s_g).\ ([t_p], [t_g]) \in H_i\}$
                $LT := \{s_l | s_l \in \alpha \wedge \forall t_l \in succ(s_l)\ \exists t_p \in succ(s_p).\ ([t_l], [t_p]) \in H_i\}$
                $\alpha' := GT \cap LT$
                if $\alpha \neq \alpha'$ then $change := true$
                $\alpha := \alpha \setminus \alpha'$
                Add $\alpha'$ as a new class to $\Sigma_{i+1}$.
            end
        end
update $H$:
        $H_{i+1} = \emptyset$
        for every $(\alpha_1, \alpha_2) \in H_i$ do begin
            for each $\alpha'_2, \alpha'_1 \in \Sigma_{i+1}$ such that $\alpha_2 \supseteq \alpha'_2$, $\alpha_1 \supseteq \alpha'_1$ do begin
                $\Phi = \{\phi | \exists \xi \in \Pi(\alpha'_2)\ (\phi, \xi) \in H_i\}$
                if $\Phi \supseteq \Pi(\alpha'_1)$ then
                    insert $(\alpha'_1, \alpha'_2)$ to $H_{i+1}$
                else
                    change := true
            end
        end
        i = i + 1
    end
```

Figure 3.5: The partition algorithm

In the rest of this section we explain how the algorithm works. There are two invariants (formally proved later) which are preserved during the execution of the algorithm.

Invariant 1: For all states $s_1, s_2 \in S$, s_1 and s_2 are in the same class $\alpha \in \Sigma_i$ iff s_1 and s_2 are $i-$equivalent.

Invariant 2: For all states $s_1, s_2 \in S$, $s_1 \leq_i s_2$ iff $([s_1], [s_2]) \in H_i$.

Σ_i is a set of equivalence classes with respect to the $i-$equivalence relation. In the ith iteration we split the equivalence classes of Σ_{i-1} so that only states that are i-equivalent remain in the same class.

A class $\alpha \in \Sigma_{i-1}$ is repeatedly split by choosing an arbitrary state $s_p \in \alpha$ (called the *splitter*) and identifying the states in α that are $i-$equivalent to s_p. These states form an $i-$equivalence class α' that is inserted into Σ_i.

α' is constructed in two steps. First we calculate the set of states $GT \subseteq \alpha$ that contains all states s_g such that $s_p \leq_i s_g$. Next we calculate the set of states $LT \subseteq \alpha$ that contains all states s_l such that $s_l \leq_i s_p$. The states in the intersection of GT and LT are the states in α that are $i-$equivalent to s_p.

H_i captures the partial order $\leq_i$, i.e., $s_1 \leq_i s_2$ iff $([s_1], [s_2]) \in H_i$. We later prove (Lemma 3.3.6) that the sequence $\leq_0, \leq_1, \ldots$ satisfies $\leq_0 \supseteq \leq_1 \supseteq \leq_2 \supseteq \ldots$. Therefore, if $s_1 \leq_i s_2$, then $s_1 \leq_{i-1} s_2$. Hence, $([s_1], [s_2]) \in H_i$ implies $([s_1], [s_2]) \in H_{i-1}$. Thus, when constructing H_i, it is sufficient to check $(\alpha'_1, \alpha'_2) \in H_i$ only when $\alpha_2 \supseteq \alpha'_2$, $\alpha_1 \supseteq \alpha'_1$, and $(\alpha_1, \alpha_2) \in H_{i-1}$.

For suitable α'_1 and α'_2, we first construct the set Φ of classes that are "smaller" than the classes in $\Pi(\alpha'_2)$. By checking if $\Phi \supseteq \Pi(\alpha'_1)$, we determine whether every class in $\Pi(\alpha'_1)$ is "smaller" than some class in $\Pi(\alpha'_2)$. If so, then (α'_1, α'_2) is inserted into H_i.

When the algorithm terminates, $\leq_i$ is the maximal simulation relation and the $i-$equivalence is the simulation equivalence relation over $M \times M$. Moreover, H_i is the maximal simulation relation over the corresponding quotient structure M_q.

The algorithm runs until there is no change in the partition Σ_i and no change in the relation H_i. A change in Σ_i is the result of a partitioning of some class $\alpha \in \Sigma_i$. The number of changes in Σ_i is bound by the number of classes in the last iteration k, i.e., $|\Sigma_k|$.

We show that a change in H_i can happen at most $|\Sigma_k|^2$ times. Thus, the algorithm terminates after at most $|\Sigma_k|^2 + |\Sigma_k|$ iterations. It is possible that in some iteration i, Σ_i will not change but H_i will, and in a later iteration $j > i$, Σ_j will change again.

Example: In this example we show how the partition algorithm is applied to the Kripke structure presented in Figure 3.6 .

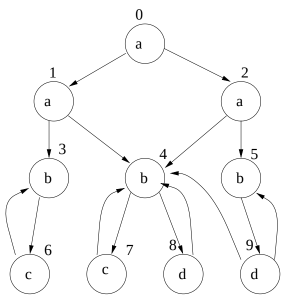

Figure 3.6: An example of a structure to be reduced

- We initialize the algorithm as follows:
  $\Sigma_0 = \{\alpha_0, \beta_0, \gamma_0, \delta_0\}$, $H_0 = \{(\alpha_0, \alpha_0), (\beta_0, \beta_0), (\gamma_0, \gamma_0), (\delta_0, \delta_0)\}$,
  where $\alpha_0 = \{0, 1, 2\}, \beta_0 = \{3, 4, 5\}, \gamma_0 = \{6, 7\}, \delta_0 = \{8, 9\}$.
- The first iteration results in the relations:
  $\Sigma_1 = \{\alpha_1, \alpha_2, \beta_1, \beta_2, \beta_3, \gamma_0, \delta_0\}$,
  $H_1 = \{(\alpha_1, \alpha_1), (\alpha_2, \alpha_2), (\beta_1, \beta_1), (\beta_2, \beta_2), (\beta_3, \beta_3), (\beta_1, \beta_2), (\beta_3, \beta_2), (\gamma_0, \gamma_0), (\delta_0, \delta_0)\}$,
  where $\alpha_1 = \{0\}, \alpha_2 = \{1, 2\}, \beta_1 = \{3\}, \beta_2 = \{4\}$, $\beta_3 = \{5\}, \gamma_0 = \{6, 7\}, \delta_0 = \{8, 9\}$.
- The second iteration results in the relations:
  $\Sigma_2 = \{\alpha_1, \alpha_2, \beta_1, \beta_2, \beta_3, \gamma_1, \gamma_2, \delta_0\}$,
  $H_2 = \{(\alpha_1, \alpha_1), (\alpha_2, \alpha_2), (\beta_1, \beta_1), (\beta_2, \beta_2), (\beta_3, \beta_3), (\beta_1, \beta_2), (\beta_3, \beta_2), (\gamma_1, \gamma_1), (\gamma_2, \gamma_2), (\gamma_1, \gamma_2), (\delta_0, \delta_0)\}$,
  where $\alpha_1 = \{0\}, \alpha_2 = \{1, 2\}, \beta_1 = \{3\}, \beta_2 = \{4\}, \beta_3 = \{5\}, \gamma_1 = \{6\}, \gamma_2 = \{7\}, \delta_0 = \{8, 9\}$.
- The third iteration results in the relations:
  $\Sigma_3 = \Sigma_2, H_3 = H_2$ - $change = false$.
  The equivalence classes are:
  $\alpha_1 = \{0\}, \alpha_2 = \{1, 2\}, \beta_1 = \{3\}, \beta_2 = \{4\}, \beta_3 = \{5\}, \gamma_1 = \{6\}, \gamma_2 = \{7\}, \delta_0 = \{8, 9\}$.

Since the third iteration results in no change to the computed partition or ordering relations, the algorithm terminates. Σ_2 is the final set of equivalence classes, which constitutes the set S_q of states of M_q. H_2 is the maximal simulation relation over $M_q \times M_q$.

3.3.2 The correctness of the partition algorithm

In order to prove the correctness of the Partitioning Algorithm, we prove three invariants. We have already mentioned the first two. The third invariant is necessary to prove them.

Invariant 1: For all states $s_1, s_2 \in S$, s_1 and s_2 are in the same class $\alpha \in \Sigma_i$ iff s_1 and s_2 are $i-$equivalent.

Invariant 2: For all states $s_1, s_2 \in S$, $s_1 \leq_i s_2$ iff $([s_1], [s_2]) \in H_i$.

Invariant 3: H_i is transitive.

We will prove these invariants by induction on i.
Base:

1. s_1, s_2 are in the same class in Σ_0 iff $L(s_1) = L(s_2)$ iff $s_1 \leq_0 s_2$ and $s_2 \leq_0 s_1$ iff s_1 is $0-$equivalent to s_2.
2. $([s_1], [s_2]) \in H_0$ iff $[s_1] = [s_2]$ iff s_1, s_2 are in the same class iff $L(s_1) = L(s_2)$ iff $s_1 \leq_0 s_2$.
3. $(\alpha_1, \alpha_2) \in H_0$ iff $\alpha_1 = \alpha_2$. Thus, for every $\alpha_1, \alpha_2, \alpha_3$, if $(\alpha_1, \alpha_2) \in H_0$ and $(\alpha_2, \alpha_3) \in H_0$, then $\alpha_1 = \alpha_2 = \alpha_3$. This implies that $(\alpha_1, \alpha_3) \in H_0$.

In the next three sections we prove the induction step. We assume that for every $j \leq i$, the invariants hold for j. We prove that the invariants hold for $i+1$.

3.3.3 Proving invariant 1

In this section we fix s_p (the splitter) to be the state that was chosen in the partition of class α, and in the construction of class $\alpha' = GT \cap LT$.

Proposition 3.3.3 *For every $\alpha' \in \Sigma_{i+1}$ there exists $\alpha \in \Sigma_i$ such that $\alpha' \subseteq \alpha$.*

We use α'_{pre} to denote the class $\alpha \in \Sigma_i$ that contains $\alpha' \in \Sigma_{i+1}$.

Proposition 3.3.4 *Let $\alpha_1, \alpha_2 \in \Sigma_{i+1}$. Then $(\alpha_1, \alpha_2) \in H_{i+1}$ implies that $(\alpha_{1pre}, \alpha_{2pre}) \in H_i$.*

Corollary 3.3.5 *If states s_1 and s_2 are in the same class, then $L(s_1) = L(s_2)$.*

Lemma 3.3.6 *If $s_1 \leq_{i+1} s_2$, then $s_1 \leq_i s_2$.*

Proof : First, $s_1 \leq_{i+1} s_2$ implies $L(s_1) = L(s_2)$. Next, we distinguish between two cases:

1. If $i = 0$, then $L(s_1) = L(s_2)$ implies $s_1 \leq_0 s_2$.
2. Suppose $i > 0$. We will show that for every successor t_1 of s_1, there exists a successor t_2 of s_2 such that $([t_1]^{i-1}, [t_2]^{i-1}) \in H_{i-1}$.

   Let t_1 be a successor of s_1. Then $s_1 \leq_{i+1} s_2$ implies that there exists a successor t_2 of s_2 such that $([t_1]^i, [t_2]^i) \in H_i$. Let $[t_1]^{i-1} = ([t_1]^i)_{pre}$ and $[t_2]^{i-1} = ([t_2]^i)_{pre}$. Then, by Proposition 3.3.4, $([t_1]^{i-1}, [t_2]^{i-1}) \in H_{i-1}$, as required.

□

Lemma 3.3.7 *Let α' be a class in Σ_{i+1} and s_1 and s_2 be states in α'. Then s_1 and s_2 are $(i+1)-$equivalent.*

Proof : Let $\alpha' \in \Sigma_{i+1}$, $s_1, s_2 \in \alpha'$. We prove that for every successor t_1 of s_1 there exists a successor t_2 of s_2 such that $([t_1], [t_2]) \in H_i$. This implies that $s_1 \leq_{i+1} s_2$.

$s_1 \in \alpha'$ implies $s_1 \in LT$. By the definition of LT, there exists a successor t_p of s_p such that $([t_1], [t_p]) \in H_i$. $s_2 \in \alpha'$ implies $s_2 \in GT$. Then by the definition of GT, there exists a successor t_2 of s_2 such that $([t_p], [t_2]) \in H_i$. By Invariant 3, H_i is transitive, and therefore $([t_1], [t_2]) \in H_i$. □

The proof that $s_2 \leq_{i+1} s_1$ is similar. Thus s_1, s_2 are $(i+1)-$equivalent.

Lemma 3.3.8 *Let s_1 and s_2 be $(i+1)-$equivalent states. Then s_1 and s_2 are in the same class in Σ_{i+1}.*

Proof : We will prove that $s_2 \in [s_1]$. Let s_p be the splitter, used to construct $[s_1]$.

- By Lemma 3.3.6, s_1 and s_2 are $(i+1)$equivalent, and this implies that s_1 and s_2 are $i-$equivalent. By the induction hypothesis, s_1 and s_2 are in the same equivalence class in H_i and thus are candidates for being in the same equivalence class in H_{i+1}.
- Since s_1, s_2 are $(i+1)-$equivalent, then for every successor t_2 of s_2, there exists a successor t_1 of s_1 such that $([t_2],[t_1]) \in H_i$.
- Since $s_1 \in [s_1]$, then $s_1 \in LT$. By the definition of LT, there exists a successor t_p of s_p such that $([t_1],[t_p]) \in H_i$.
- By Invariant 3, H_i is transitive. Therefore $([t_2],[t_p]) \in H_i$. We proved that for every successor t_2 of s_2 there exists a successor t_p of s_p such that $[t_2],[t_p] \in H_i$. Thus, by definition of LT, $s_2 \in LT$.
- Since $s_1 \in [s_1]$, then $s_1 \in GT$. Then by the definition of GT, for every successor t_p of s_p, there exists a successor t_3 of s_1, such that $([t_p],[t_3]) \in H_i$.
- Since s_1, s_2 are $(i+1)-$equivalent, there exists a successor t_4 of s_2 such that $([t_3],[t_4]) \in H_i$.
- H_i is transitive, and therefore $([t_p],[t_4]) \in H_i$. We proved that for every successor t_p of s_p there exists a successor t_4 of s_2 such that $([t_p],[t_4]) \in H_i$. Thus, by the definition of GT, $s_2 \in GT$.
- $s_2 \in GT$, and $s_2 \in LT$ implies that $s_2 \in [s_1]$.

□

By Lemma 3.3.8 and Lemma 3.3.7 we can conclude the proof of Invariant 1.

3.3.4 Proving invariant 2

In this section we prove for H_{i+1} the property defined by Invariant 2. Since the construction of H_{i+1} is based on both Σ_i and Σ_{i+1}, we need to distinguish between classes in these sets. We use $[s]^i$ and $[s]^{i+1}$ to denote equivalence classes in Σ_i and Σ_{i+1} respectively.

Lemma 3.3.9 *Let $([s_1]^{i+1}, [s_2]^{i+1}) \in H_{i+1}$. Then for every successor t_1 of s_1, there exists a successor t_2 of s_2 such that $([t_1]^i, [t_2]^i) \in H_i$.*

Proof : Let $([s_1]^{i+1}, [s_2]^{i+1}) \in H_{i+1}$, and let t_1 be a successor of s_1. Then $[t_1]^i \in \Pi([s_1]^{i+1})$. Since $\Pi([s_1]^{i+1}) \subseteq \Phi$, then $[t_1]^i \in \Phi$. By definition of Φ, there is a state t_3 such that $[t_3]^i$ is in $\Pi([s_2]^{i+1})$ and $([t_1]^i, [t_3]^i) \in H_i$. $[t_3]^i \in \Pi([s_2]^{i+1})$ implies that t_3 is a successor of some state s_3 in $[s_2]^{i+1}$.

Since s_2, s_3 are in the same class in Σ_{i+1}, by Lemma 3.3.7, they are $(i+1)-$equivalent. Thus, there exists a successor t_2 of s_2 such that $([t_3]^i, [t_2]^i) \in H_i$. By Invariant 3, H_i is transitive and therefore $([t_1]^i, [t_2]^i) \in H_i$. $\Box$

Corollary 3.3.10 *If $([s_1]^{i+1}, [s_2]^{i+1}) \in H_{i+1}$, then $s_1 \leq_{i+1} s_2$.*

Lemma 3.3.11 *If $s_1 \leq_{i+1} s_2$, then $([s_1]^{i+1}, [s_2]^{i+1}) \in H_{i+1}$.*

Proof : Assume $s_1 \leq_{i+1} s_2$.

- By Lemma 3.3.6, $s_1 \leq_i s_2$.
- By the induction hypothesis, Invariant 2 holds for i. Thus, $([s_1]^i, [s_2]^i) \in H_i$.
- Clearly, $[s_1]^{i+1} \subseteq [s_1]^i$ and $[s_2]^{i+1} \subseteq [s_2]^i$. Since $([s_1]^i, [s_2]^i) \in H_i$, the pair $([s_1]^{i+1}, [s_2]^{i+1})$ is considered for inclusion in H_{i+1} in the $(i+1)$th **update** step of the algorithm.
- In order to prove that $([s_1]^{i+1}, [s_2]^{i+1}) \in H_{i+1}$, we show that $\Pi([s_1]^{i+1}) \subseteq \Phi$, i.e., every class α in $\Pi([s_1]^{i+1})$ is also in Φ.
- Let $\alpha \in \Sigma_i$ be a class in $\Pi([s_1]^{i+1})$. Then there exists a state $s_3 \in [s_1]^{i+1}$ and a successor t_3 of s_3 such that $t_3 \in \alpha$.
- By Lemma 3.3.7 , s_1, s_3 being in the same class of Σ_{i+1} implies that there exists a successor t_1 of s_1 such that $(\alpha, [t_1]^i) \in H_i$.
- Since $s_1 \leq_{i+1} s_2$, then there exists a successor t_2 of s_2 such that $([t_1]^i, [t_2]^i) \in H_i$.
- Since H_i is transitive, $(\alpha, [t_2]^i) \in H_i$.

- The definition of $\Pi([s_2]^{i+1})$ implies that $[t_2]^i \in \Pi([s_2]^{i+1})$. Hence, $(\alpha, [t_2]^i) \in H_i$ implies that $\alpha \in \Phi$.

□

Corollary 3.3.10 and Lemma 3.3.11 prove Invariant 2.

3.3.5 Proving invariant 3

Lemma 3.3.12 *Let $[s_1], [s_2], [s_3]$ be classes in Σ_{i+1} such that $([s_1], [s_2]) \in H_{i+1}$ and $([s_2], [s_3]) \in H_{i+1}$. Then $L(s_1) = L(s_3)$.*

Proof : By Corollary 3.3.10, $s_1 \leq_{i+1} s_2$ and $s_2 \leq_{i+1} s_3$. By Definition 3.3.1, $L(s_1) = L(s_2) = L(s_3)$. □

Lemma 3.3.13 *H_{i+1} is transitive.*

Proof : Let $\alpha_1, \alpha_2, \alpha_3$ be classes in Σ_{i+1} such that $(\alpha_1, \alpha_2) \in H_{i+1}$ and $(\alpha_2, \alpha_3) \in H_{i+1}$. We prove that $(\alpha_1, \alpha_3) \in H_{i+1}$. To do so, we show that for all states s_1, s_3 in α_1, α_3 respectively, the following holds: For every successor t_1 of s_1, there exists a successor t_3 of s_3 such that $([t_1]^i, [t_3]^i) \in H_i$. By Lemma 3.3.11 and Lemma 3.3.12, this implies that $(\alpha_1, \alpha_3) \in H_{i+1}$.

Let s_1, s_2, s_3 be states in $\alpha_1, \alpha_2, \alpha_3$ respectively, and let t_1 be a successor of s_1. By Lemma 3.3.9 , $(\alpha_1, \alpha_2) \in H_{i+1}$ implies that there exists a successor t_2 of s_2 such that $([t_1]^i, [t_2]^i) \in H_i$. By Lemma 3.3.9 , $(\alpha_2, \alpha_3) \in H_{i+1}$ implies that there exists a successor t_3 of s_3 such that $([t_2]^i, [t_3]^i) \in H_i$. By the induction hypothesis, $([t_1]^i, [t_3]^i) \in H_i$. Thus, we conclude that $(\alpha_1, \alpha_3) \in H_{i+1}$. □

The above lemma proves Invariant 3. This completes the proof of the three invariants.

3.3.6 Equivalence classes

In this section we will show that when the algorithm terminates after k iterations, $\leq_k$ is the maximal simulation relation over $M \times M$ and Σ_k is the set of equivalence classes with respect to simulation equivalence over $M \times M$. Moreover, H_k is the maximal simulation relation over the corresponding quotient structure M_q.

Lemma 3.3.14 *For every $i \geq 0$ and every state s, $s \leq_i s$.*

Proof : We will prove it by induction on i:

- Base: For $i = 0$, $L(s) = L(s)$ implies $s \leq_0 s$.
- Induction step: Assume that the lemma holds for i. Let t be a successor of s. The induction hypothesis implies that $t \leq_i t$. Based on Invariant 2, $([t],[t]) \in H_i$. Thus, for every successor t of s, we choose t as the successor of s such that $([t],[t]) \in H_i$. By the definition of $\leq_{i+1}$, this implies $s \leq_{i+1} s$.

□

Proposition 3.3.15 *When the algorithm terminates, $\leq_k = \leq_{k-1}$.*

Lemma 3.3.16 *$\leq_k$ is a simulation over $M \times M$.*

Proof :

- By Lemma 3.3.14 , $s_0 \leq_k s_0$.
- $(s_1, s_2) \in \leq_k$ implies that $L(s_1) = L(s_2)$.
- $(s_1, s_2) \in \leq_k$ implies that for every successor t_1 of s_1 there exists a successor t_2 of s_2 such that $([t_1],[t_2]) \in H_{k-1}$. By Corollary 3.3.10, $t_1 \leq_{k-1} t_2$. Thus, since $\leq_k = \leq_{k-1}$, $t_1 \leq_k t_2$.

□

Lemma 3.3.17 *$\leq_k$ is the maximal simulation over $M \times M$.*

Proof : Let H' be the maximal simulation over $M \times M$. We prove that $H' \subseteq \leq_k$. By Invariant 2, it is sufficient to prove that $(s_1, s_2) \in H'$ implies $([s_1],[s_2]) \in H_k$.

We prove by induction on i that $(s_1, s_2) \in H'$ implies that $([s_1],[s_2]) \in H_i$.

- Base: $(s_1, s_2) \in H'$ implies that $L(s_1) = L(s_2)$. Therefore, $[s_1]^0 = [s_2]^0$ and $([s_1]^0, [s_2]^0) \in H_0$.
- Induction step: Assume that the lemma holds for $i-1$. Let (s_1, s_2) be in H'. Then for every successor t_1 of s_1 there exists a successor t_2 of s_2 such that $(t_1, t_2) \in H'$. By the induction hypothesis, $([t_1],[t_2]) \in H_{i-1}$ which, by Lemma 3.3.11, implies that $([s_1],[s_2]) \in H_i$.

□

Theorem 3.3.18 *When the algorithm terminates, Σ_k is the set of equivalence classes of the simulation equivalence relation.*

Proof : States s_1, s_2 are simulation equivalent iff $(s_1, s_2) \in \leq_k$, and $(s_2, s_1) \in \leq_k$ iff s_1, s_2 are $k-$*equivalent* iff (by Invariant 1) s_1, s_2 are in the same class in Σ_k. □

We proved that Σ_k is the set of equivalence classes which are used as the set of states S_q in the quotient structure M_q. Next, we show that H_k is the maximal simulation relation over $M_q \times M_q$.

Lemma 3.3.19 *H_k is a simulation over $M_q \times M_q$.*

Proof :

- By Invariant 2, $(s_0, s_0) \in \leq_k$ implies that $([s_0], [s_0]) \in H_k$.
- Assume that $([s_1], [s_2]) \in H_k$. By Invariant 2, $(s_1, s_2) \in \leq_k$. Thus $L(s_1) = L(s_2)$, which implies that $L_q([s_1]) = L_q([s_2])$.
- Let $([s_1], [s_2])$ be a pair in H_k and α a successor of $[s_1]$. By the definition of R_q, there exists a successor t_1 of s_1 in α. Since $\leq_k$ is a simulation relation, there is a successor t_2 of s_2 such that $(t_1, t_2) \in \leq_k$. Let t_m be a maximal state in $G([s_2], t_2)$ (Definition 3.2.2). By Lemma 3.2.4, $[t_m]$ is a successor of $[s_2]$. t_m is maximal in $G([s_2], t_2)$. Hence $(t_2, t_m) \in \leq_k$. Since $\leq_k$ is transitive, $(t_1, t_m) \in \leq_k$. Thus, by Invariant 2, $(\alpha, [t_m]) \in H_k$.

□

Theorem 3.3.20 *H_k is the maximal simulation relation over $M_q \times M_q$.*

Proof : Let H' be the maximal simulation relation over $M_q \times M_q$. We prove that the relation defined by $H = \{(s_1, s_2) | ([s_1], [s_2]) \in H'\}$ is the maximal simulation relation over $M \times M$. Thus, $H = \leq_k$. By Invariant 2, $\leq_k = \{(s_1, s_2) | ([s_1], [s_2]) \in H_k\}$; hence, $H_k = H'$.

- By $H_k \subseteq H'$ we have $\leq_k \subseteq H$. Since H includes the maximal simulation relation $\leq_k$, it is sufficient to show that H is a simulation relation.

- By transitivity of H', H is transitive.
- Since $([s_0],[s_0]) \in H'$, $(s_0,s_0) \in H$.
- Assume that $(s_1,s_2) \in H$. Then $L(s_1) = L_q([s_1]) = L_q([s_2]) = L(s_2)$.
- Suppose that $(s_1,s_2) \in H$ and t_1 is a successor of s_1. Let t_m be a maximal state in $G([s_1],t_1)$ (Definition 3.2.2). By Lemma 3.2.4, $[t_m]$ is a successor of $[s_1]$. t_m is maximal in $G([s_1],t_1)$. Hence $(t_1,t_m) \in \leq_k$. Because $\leq_k \subseteq H$, $(t_1,t_m) \in H$. Since H' is a simulation relation, there is a successor α of $[s_2]$ such that $([t_m],\alpha) \in H'$. By the definition of R_q, there exists a successor t_2 of s_2 in α. It follows from $([t_m],\alpha) \in H'$ that $(t_m,t_2) \in H$ and by transitivity of H, $(t_1,t_2) \in H$.

$\square$

3.3.7 Space complexity

The space complexity of the partition algorithm depends on the size of Σ_i. We assume that the algorithm is applied to Kripke structures with some redundancy. Thus $|\Sigma_i| << |S|$.

We measure the space complexity with respect to the size of the following three relations:

1. The relation R.
2. The relations H_i whose size depends on Σ_i. We can bound the size of H_i by $|\Sigma_i|^2$.
3. A relation that relates each state to its equivalence class. Since every state belongs to a single class, the size of this relation is $O(|S| \cdot log(|\Sigma_i|))$.

In the ith iteration we do not need to keep all $H_0, H_1, \ldots$ and $\Sigma_0, \Sigma_1, \ldots$, since we only refer to H_i, H_{i+1} and Σ_i, Σ_{i+1}. This fact, along with the three relations cited above, leads us to conclude that the total space complexity is $O(|R| + |\Sigma_k|^2 + |S| \cdot log(|\Sigma_k|))$.

In practice, we often do not hold the transition relation R in the memory. Rather, we use it to provide, whenever needed, the set of successors of a given state. Thus, the space complexity is $O(|\Sigma_k|^2 + |S| \cdot log(|\Sigma_k|))$. Recall that the space complexity of the naive algorithm for computing the equivalence classes of the simulation equivalence relation is bounded by $|S|^2$, which is the size of the simulation relation over $M \times M$. When $|\Sigma_k| << |S|$, the partition algorithmachieves a much better space complexity.

3.3.8 Time complexity

First, we would like to bound the number of main iterations in the algorithm. The algorithm continues as long as the splitting of equivalence classes continues in the **refine** step, meaning $\Sigma_i \neq \Sigma_{i+1}$, or changes occur in the **update** step. The equivalence classes split at most $|\Sigma_k|$ times during the algorithm. We claim that there is a change in the **update** step at most $|\Sigma_k|^2$ times. Next we prove this claim. We use Proposition 3.3.3 to deduce Corollary 3.3.21.

Corollary 3.3.21 *For every $0 \leq i \leq k$ and every class α in Σ_k there exists a class α' in Σ_i, such that $\alpha \subseteq \alpha'$.*

Next we define a relation $H_{ik} \subseteq \Sigma_k \times \Sigma_k$, which relates pairs of classes over Σ_k to H_i. Corollary 3.3.21 implies that H_{ik} is well defined.

Definition 3.3.22 *Let $0 \leq i \leq k$. Then $H_{ik} = \{(\alpha_1, \alpha_2) | let\ \alpha'_1, \alpha'_2 \in \Sigma_i$ be such that $\alpha_1 \subseteq \alpha'_1,\ \alpha_2 \subseteq \alpha'_2$ then $(\alpha'_1, \alpha'_2) \in H_i\}$.*

Proposition 3.3.4 implies that for every $0 \leq i < k$, $H_{(i+1)k} \subseteq H_{ik}$. Furthermore, if there is a change in the **update** step, then $H_{(i+1)k} \subset H_{ik}$. Thus we can bound the number of iterations in which there is a change in the **update** step by $|H_{0k} \setminus H_{kk}| \leq |\Sigma_k|^2$.

We showed that the algorithm runs $O(|\Sigma_k|^2)$ iterations. In every iteration it performs one **refine** step and one **update** step. First we analyze the time complexity of the **refine** step: Figure 3.7 shows how the set GT is computed.

The set LT is computed in a similar manner and the computation $\alpha' = LT \cap GT$ is simple; thus it is sufficient to analyze the total time it takes to compute GT.

```
$GT = \emptyset$
$\Gamma_p = \{[s'] | (s_p, s') \in R\}$
for every $s \in \alpha$ do
    $\Gamma_s = \{[s'] | (s, s') \in R\}$
    $\Gamma_{<s} = \{\gamma' | \exists \gamma \in \Gamma_s \quad (\gamma', \gamma) \in H_i\}$
    if $\Gamma_p \subseteq \Gamma_{<s}$ then inserts $s$ to $GT$
end
```

Figure 3.7: The construction of GT

The sets Γ_p and Γ_s are computed in time $O(|S|)$. The set $\Gamma_{<s}$ is computed in time $O(|\Sigma_k|^2)$. Thus the total time complexity of the **refine** step is the number of times a state is tested for being in GT times $O(|S| + |\Sigma_k|^2)$. We distinguish between two cases:

1. α is split in the internal loop. The number of such sub-iterations is bounded by Σ_k. For each iteration in which α is split, the number of times every state in α is tested for being in GT is at most equal to the number of partitions. The number of partition is bounded by Σ_k. Since $|\alpha| \leq |S|$, the number of times a state is tested for being in GT is $O(|S| \cdot |\Sigma_k|)$. Thus the total time complexity of these iterations is $O(|\Sigma_k| \cdot |S| \cdot (|S| + |\Sigma_k|^2))$.

2. α is not split in the internal loop. In each iteration where α is not split, every state in α is tested for being in GT exactly once. Since $\sum_{\alpha \in \Sigma_k} |\alpha| = |S|$, the time complexity of one iteration is $O(|S| \cdot (|S| + |\Sigma_k|^2))$. Since the number of such iterations is bounded by $|\Sigma_k|^2$, the time complexity of these iterations is $O(|\Sigma_k|^2 \cdot |S| \cdot (|S| + |\Sigma_k|^2))$.

Thus the total time complexity of the **refine** step is $O(|\Sigma_k|^2 \cdot |S| \cdot (|S| + |\Sigma_k|^2))$.

Next we analyze the complexity of the **update** step. The construction of $\Pi(\alpha)$ can be done in time $O(|R|)$. Given $\Pi(\alpha'_2)$, the construction of Φ can be done in time $O(|\Sigma_k|^2)$. The time required to check whether $\Phi \supseteq \Pi(\alpha'_1)$ is $O(|\Sigma_k|)$. Thus the time complexity of internal loop in the **update** step is $O(|R| + |\Sigma_k|^2)$.

Since the number of sub-iterations in each **update** step is bounded by $O(|\Sigma_k|^2)$, the total time complexity of an **update** step is $O(|\Sigma_k|^2 \cdot$

$(|R| + |\Sigma_k|^2))$. Since the number of main iterations in the algorithm is bounded by $|\Sigma_k|^2$, the total complexity of all **update** steps is $O(|\Sigma_k|^4 \cdot (|R| + |\Sigma_k|^2))$.

Thus the total time complexity of the algorithm is $O(|\Sigma_k|^4 \cdot (|R| + |\Sigma_k|^2) + |\Sigma_k|^2 \cdot |S| \cdot (|S| + |\Sigma_k|^2))$.

Chapter 4

Applicability of fair simulation

In this chapter we make a broader comparison of four notions of fair simulation: direct [DHWT91], delay [EWS01a], game [HKR97], and exists [GL94]. We refer to several criteria that emphasize the advantages of each of the notions. The results of the comparison are summarized in Table 4.1.

We developed two practical applications that are based on the comparison. The first is an efficient approximated minimization algorithm for the delay, game and exists simulations. For these preorders, a unique equivalent smallest model does not exist. Therefore, an approximation is appropriate. In addition, we suggest a new implementation for the *assume-guarantee* [Fra76, Jon83, MC81, Pnu84] modular framework presented in [GL94]. The new implementation, based on the game simulation rather than the exists simulation, significantly improves the complexity of the framework.

Our comparison refers to three main aspects of fair simulation. The first is the time complexity of constructing the preorder. There, we mainly summarize results of other works (see Figure 4.1). We see that constructing the direct, delay, and game simulations is polynomial in the number of states n and the number of transitions m [EWS01a]. In contrast, constructing the exists simulation is PSPACE-complete [KV96] , which is a great disadvantage.

The second aspect that we consider is the ability to use the preorder for minimization. We say that two models are *equivalent* with respect to a preorder if each is smaller by the preorder than the other. The goal of minimization is to find the smallest in size model that is equivalent with respect to the preorder to the original one[1].

In [BG00] it has been shown that for every model with no fairness constraints there exists a unique smallest in size model which is simulation equivalent to it. The minimization algorithm that constructs this smallest in size model [BG00] identifies and eliminates two types of redundancies in the given model. One is the existence of equivalent states. This redundancy is eliminated by constructing a *quotient model*. The other is the existence of a successor of a state whose behavior is contained in the behavior of another successor of the same state. Such a state is called *a little brother*. This redundancy is eliminated by *disconnecting little brothers*.

We thus examine, for each of the fair simulation preorders, the following three questions. Given a model M:

- 1. Is there a unique smallest in size model that is simulation equivalent to M?
- 2. Is the quotient model of M simulation equivalent to M?
- 3. Is the result of disconnecting little brothers in M simulation equivalent to M?

Our examination (see Figure 4.1) leads to a new minimization algorithm that uses the direct and delay simulations as approximations for the game and exists simulations. The new algorithm obtains a better reduction than the algorithm suggested in [EWS01a].

The third aspect that we investigate is the relationship between the simulation preorders and universal branching-time logics. A basic requirement of using a preorder in verification is that it preserves the specification logic, i.e., if $M_1 \leq M_2$ then, for every formula ϕ in the logic, $M_2 \models \phi$ implies $M_1 \models \phi$. Indeed, all four notions of fair simulation satisfy this requirement. A stronger requirement is that the preorder have a *logical characterization* by some logic. This means

[1]Note that this is a stronger criterion than the one used in [EWS01a], where only language equivalence is required.

that $M_1 \leq M_2$ **if and only if** for every formula ϕ in the logic, $M_2 \models \phi$ implies $M_1 \models \phi$.

Logical characterization is useful in determining if model M_2 can be used as an abstraction for model M_1, when the logic $\mathcal{L}$ should be preserved. If the preorder $\leq$ is logically characterized by $\mathcal{L}$ then checking $M_1 \leq M_2$ is a necessary and sufficient condition and will never give a false negative result.

Another important relationship between a logic and a preorder is the existence of a *maximal model* $\mathcal{T}_\phi$ for a formula ϕ with respect to the preorder. The maximal model $\mathcal{T}_\phi$ for a formula ϕ is such that for every model M', $M' \leq \mathcal{T}_\phi$ if and only if $M' \models \phi$. Maximal models are used as tableaux in the framework described in [GL94] for the *assume-guarantee paradigm.* The assume-guarantee is an inductive modular verification paradigm in which the environment of the verified part can be represented by a formula. The result method is a proof schema which is based on the modular structure of the system.

In [GL94], a semi-automatic framework for the assume-guarantee paradigm is presented. The framework uses the exists preorder and is defined with respect to the logic ACTL. It uses a tableau to represent an ACTL formula. This tableau is the maximal model for the formula with respect to the exists preorder.

In this work we show that there is also a maximal model for ACTL formulas with respect to the game simulation. In addition, we show that other conditions required for a sound implementation of the assume-guarantee paradigm hold for the game simulation. Once the game simulation replaces the exists simulation, the complexity of the implementation is dramatically reduced.

The results of our comparison are presented in Table 4.1. The proofs of the claims for which no citation is given appear in the next sections.

The rest of the chapter is organized as follows: In Section 4.1 we define the Büchi fairness constraints and the different notions of fair simulation. Section 4.2 investigates simulation minimization. For each of the fair simulations we check whether there exists a unique minimal structure, and whether constructing a quotient structure or disconnecting little brothers results in an equivalent structure. We then present a

[3] In [EWS01a] it is shown that the quotient model is $\underline{\text{language equivalent}}$ to the original model. Here, we show that they are delay equivalent.

		minimization			relation to logic	
notion	time complexity of constructing the preorder	unique smallest model	quotient model	little brothers	has logical characterization	max model
Direct	$O(m \cdot n)$ [EWS01a]	*true*	true	*true*	*false*	*false*
Delay	$O(m \cdot n^3)$ [EWS01a]	*false*	true [3]	*false*	*false*	*false*
Game	$O(m \cdot n^3)$ [EWS01a]	*false*	false [EWS01a]	*false*	$\forall AFMC$ [HKR97]	*true*
Exists	PSPACE complete [KV96]	*false*	*false*	*false*	$ACTL^*$	true [GL94]

Table 4.1: The properties of the different notions of fair simulation

new minimization algorithm for the game and exists simulations. Section 4.3 investigates the relationships between fair simulation and logic. Each notion is checked for logical characterization and for the existence of a maximal structure. In Section 4.4 we prove that the game simulation can replace the exists simulation in the implementation of the assume-guarantee paradigm.

4.1 Fairness constraints and fair simulation

In this work we refer to Büchi fairness constraints. Given a Kripke structure $M =< S, L, S_0, R >$, we add a Büchi fairness constraints which distinguish between fair traces and unfair traces in M. A Büchi fairness constraints is a set $F \subseteq S$. In order to capture the infinite behavior of ρ, we define

$$\inf(\rho) = \{ s \mid s = \rho^i \text{ for infinitely many } i \}.$$

We say that a trace ρ is *fair* according to the fair set F iff $\inf(\rho) \cap F \neq \emptyset$.

Next, we define the different notions of fair simulation. The first notion is the direct simulation, which is the most straightforward extension of the ordinary simulation.

Definition 4.1.1 *$H \subseteq S_1 \times S_2$ is a* direct simulation relation *[DHWT91] ($\leq_{di}$) over $M_1 \times M_2$ iff it satisfies the conditions of Definition 2.0.1,*

except that here 2a is replaced by:
2(a') $L_1(s_1) = L_2(s_2)$ *and* $s_1 \in F_1$ *implies* $s_2 \in F_2$.

We now define the exists simulation:

Definition 4.1.2 *[GL94]* $H \subseteq S_1 \times S_2$ *is an* exists simulation *(*$\leq_\exists$*) over* $M_1 \times M_2$ *iff it satisfies the conditions of Definition 2.0.1, except that here 2b is replaced by:*
2(b') *for every fair trace* ρ_1 *from* s_1 *in* M_1 *there exists a fair trace* ρ_2 *from* s_2 *in* M_2 *such that for all* $i \in IN$, $(\rho_1^i, \rho_2^i) \in H$.[4]

The next definitions are based on games over Kripke structures. We start with a game that characterizes the simulation over structures with trivial fairness constraints. Given two Kripke structures M_1, M_2, we define a game of two players over M_1, M_2. The players are called the adversary and the protagonist, where the adversary plays on M_1 and the protagonist plays on M_2.

Definition 4.1.3 *Given two Kripke structures,* M_1 *and* M_2, *a* simulation game *consists of a finite or infinite number of rounds. At the beginning, the adversary selects an initial state* s_{01} *in* M_1, *and the protagonist responds by selecting an initial state* s_{02} *in* M_2 *such that* $L_1(s_{01}) = L_2(s_{02})$. *In each round, assume that the adversary is at* s_1 *and the protagonist is at* s_2. *The adversary then moves to a successor* s_1' *of* s_1 *on* M_1, *after which the protagonist moves to a successor* s_2' *of* s_2 *on* M_2 *such that* $L_1(s_1') = L_2(s_2')$.

If the protagonist does not have a matching state, the game terminates and the protagonist fails. Otherwise, if the protagonist always has a matching successor to move to, the game proceeds ad infinitum for ω rounds and the protagonist wins. The adversary wins iff the protagonist fails.

Definition 4.1.4 *Given two Kripke structures* M_1 *and* M_2, *a* strategy π *of the protagonist is a partial function* $\pi : (S_1 \times S_2 \rightarrow S_2) \cup (S_{01} \times \{\perp\} \rightarrow S_{02})$. *The function* π *should satisfy the following: If* $s_2' = \pi(s_1', s_2)$ *then* $(s_2, s_2') \in R_2$.

[4]In such a case we use the notation $(\rho_1, \rho_2) \in H$.

The protagonist plays according to a strategy π if when the adversary initially selects $s_{0_1} \in S_{0_1}$, the protagonist selects $s_{0_2} = \pi(s_{0_1}, \perp)$ and, for every round i, when the adversary moves to s_1' and the protagonist is in s_2, the protagonist moves to $s_2' = \pi(s_1', s_2)$. π is a winning strategy for the protagonist if the protagonist wins whenever it plays according to π.

We can now present an alternative definition to the simulation preorder. This definition is equivalent to Definition 2.0.1 [HKR97].

Definition 4.1.5 *Given two Kripke structures, M_1 and M_2, M_2 simulates M_1 ($M_1 \leq M_2$) iff the protagonist has a winning strategy in a simulation game over M_1, M_2.*

In order to extend the simulation game to fair simulation, we add a winning condition which refers to the infinite properties of the game. We then give two additional definitions of fair simulation, the delay ($\leq_{de}$) and the game ($\leq_g$) simulations.

Definition 4.1.6 *[EWS01a] The protagonist* delay wins *a game over two fair Kripke structures M_1 and M_2 iff the game is played for infinitely many rounds. Moreover, whenever the adversary reaches a fair state then the protagonist reaches a fair state within a finite number of rounds.*

Definition 4.1.7 *[HKR97] The protagonist* game wins *a game over two fair Kripke structures M_1 and M_2 iff the game is played for infinitely many rounds. Moreover, if the adversary moves along a fair trace, then the protagonist moves along a fair trace as well.*

We say that π is a delay/game winning strategy for the protagonist if the protagonist delay/game wins whenever it plays according to π.

Definition 4.1.8 *[HKR97, EWS01a] Given two fair Kripke structures, M_1 and M_2, M_2 delay/game simulates M_1 iff the protagonist has a delay/game winning strategy over M_1, M_2.*

Definitions 4.1.1,4.1.2 and 4.1.8 are extensions of Definition 2.0.1 and its equivalent Definition 4.1.5. Consequently, on structures with trivial fairness constraints ($F = S$), all four definitions are equivalent. On structures with non-trivial fairness constraints ($F \neq S$), the direct, delay and game simulations imply ordinary simulation, the exists simulation, however, does not imply ordinary simulation. In [HKR97, EWS01a] the following relationships over the fair simulation preorders are shown:

$$M_1 \leq_{di} M_2 \;\Rightarrow\; M_1 \leq_{de} M_2 \;\Rightarrow\; M_1 \leq_{g} M_2 \;\Rightarrow\; M_1 \leq_{\exists} M_2.$$

Note that the definitions of game/exists simulation are not limited to specific types of fairness constraints. They hold even if M_1 and M_2 have different types of fairness constraints. Finally, we extend the delay/game simulations for states.

Definition 4.1.9 *For all states s_1 and s_2 in a structure M, $s_1 \leq_{de/g} s_2$ if the protagonist has a winning delay/game strategy in a game over $M \times M$ where the adversary starts at s_1 and the protagonist starts at s_2.*

4.2 Minimization with respect to fair simulation

For structures with trivial fairness constraints ($F = S$), two forms of redundancy are considered [BG00]. These redundancies are handled in [BG00], by first constructing a quotient structure that results in a structure without equivalent states and then disconnecting *little brothers* to eliminate the other redundancy. For structures with trivial fairness constraints, eliminating these redundancies results in a unique, smallest in size structure that is simulation equivalent to the original structure [BG00].

The following lemma is a direct consequence of the result in [BG00] if we refer to states in F as having additional labeling.

Lemma 4.2.1 *For every structure, there exists a unique, smallest in size structure that is direct simulation equivalent to it.*

The proof of Lemma 4.2.1 and the construction of the smallest structure can be obtained as in Chapter 3. Unfortunately, performing the same operations for the other notions of fair simulations might result in an inequivalent structure. In this section we investigate minimization with respect to each notion of fair simulation. We start by checking whether the quotient structure is equivalent to the original one. Next we check whether it is safe to disconnect little brothers. We then determine whether there exists a unique smallest in size equivalent structure. Finally, we use the results of this section to suggest a new and better minimizing algorithm.

In this section we use language equivalence and language containment.The definitions are given below.

Definition 4.2.2

- *The language of s_1 is contained in the language of s_2 ($s_1 \subseteq s_2$) if for every fair trace ρ_1 from s_1 there is a fair trace ρ_2 from s_2 such that $\forall i \geq 0$, $L(\rho_1^i) = L(\rho_2^i)$.*
- *$M_1 \subseteq M_2$ if for every fair trace starting at an initial state $s_{01} \in S_{01}$ there is a fair trace starting at an initial state $s_{02} \in S_{02}$ such that $\forall i \geq 0$, $L_1(\rho_1^i) = L_2(\rho_2^i)$.*
- *M_1 is language equivalent to M_2 if $M_1 \subseteq M_2$ and $M_2 \subseteq M_1$.*

Clearly, all notions of fair simulation imply language containment.

4.2.1 Quotient structure

The quotient structure is the result of unifying all equivalent states into equivalence classes. Recall that states s_1 and s_2 are equivalent if $s_1 \leq s_2$ and $s_2 \leq s_1$. The equivalence classes are the states of the quotient structure. There is a transition from one equivalence class to another iff there exists a transition from a state in the former to a state in the latter. An equivalence class is initial if it contains an initial state and is fair if it contains a fair state. For the delay simulation, we present the following lemma.

Lemma 4.2.3 *Let M^Q be the quotient structure of a structure M. Then $M \equiv_{de} M^Q$.*

The proof of Lemma 4.2.3 appears in Section 4.5.1 and is similar to the proof in [EWS01b].

In [EWS01a] it is shown that the quotient structure with respect to game simulation is not equivalent to the original one. We show that for every preorder $\leq_{\clubsuit}$ that lies between game simulation and language containment, the quotient structure with respect to this preorder might not be equivalent to the original structure.

Lemma 4.2.4 *Let $\leq_{\clubsuit}$ be any preorder such that for every M_1, M_2,*

$$M_1 \leq_g M_2 \Rightarrow M_1 \leq_{\clubsuit} M_2 \Rightarrow M_1 \subseteq M_2.$$

Then there exists a structure M whose quotient structure with respect to $\leq_{\clubsuit}$ is not equivalent to M with respect to $\leq_{\clubsuit}$.

Proof Consider the structure M_1 in Figure 4.1. States s_0 and s_2 are equivalent with respect to game simulation. This can be seen by considering a strategy that instructs the protagonist to move to the same state the adversary moves to. This strategy proves both directions of the game equivalence. Since $M_1 \leq_g M_2 \Rightarrow M_1 \leq_{\clubsuit} M_2$, s_0 and s_2 are also equivalent with respect to $\leq_{\clubsuit}$.

However, the quotient structure that is the result of unifying states s_0 and s_2 is not equivalent to M_1 with respect to $\leq_{\clubsuit}$. Since $M_1 \leq_{\clubsuit} M_2 \Rightarrow M_1 \subseteq M_2$, it is sufficient to prove that the quotient structure is not language equivalent to M_1: the language of M_1 contains all words in which both a and b occur infinitely often, but the language of the quotient structure contains the word a^ω.

Furthermore, there is no other definition of a quotient structure of M_1 that is language equivalent to M_1. Such a quotient structure contains two states, one in which a is true and another in which b is true and at least one of the states is fair. Assume that the state where a is true is fair. We distinguish between two cases: If there exists a transition from this state to itself, then the language of the quotient structure includes a word where b occurs only finitely many times, a contradiction. Otherwise, the word $(aab)^\omega$ is not in the language of the quotient structure, a contradiction. Assuming that the state where b is true is fair, will lead to a contradiction in a similar way. $\Box$

Corollary 4.2.5 *For exists/game simulation, the quotient structure is not necessarily equivalent to the original structure.*

4.2.2 Disconnecting little brothers

A state s_2 is a *little brother* of another state s_3 if both states are suc-

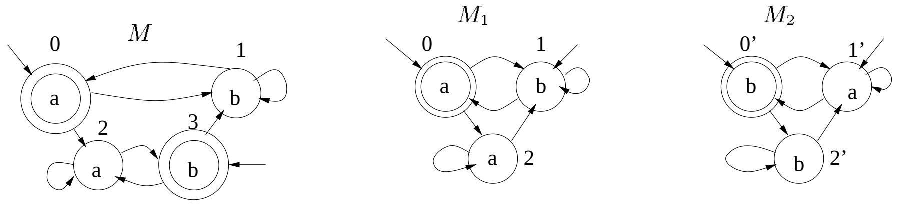

Figure 4.1: The structures M_1 and M_2 are equivalent to M with respect to game/exists simulation, and they are both minimal. Note that states 0 and 2 ($0'$ and $2'$) are equivalent but cannot be unified. (Double circles denote fair states.)

cessors of the same state s_1, $s_2 \leq s_3$, and $s_3 \not\leq s_2$. Little brothers s_2 is disconnected by removing the transition (s_1, s_2) from R.

Lemma 4.2.6 *Let $\leq_\spadesuit$ be a preorder such that*

$$M_1 \leq_{de} M_2 \Rightarrow M_1 \leq_\spadesuit M_2 \Rightarrow M_1 \subseteq M_2.$$

Assume that structure M' is the result of disconnecting little brothers in structure M with respect to $\leq_\spadesuit$. M' might not be equivalent to M with respect to $\leq_\spadesuit$.

Proof Consider the structure M_1 in Figure 4.2. State s_2 is a little brother of state s_1 with respect to $\leq_{de}$. This can be seen by considering the strategy that instructs the protagonist to move from state s_1 to state s_0 in the first round and to move to the same state the adversary moves to in the other rounds. This strategy shows that $s_2 \leq_{de} s_1$, because
$M_1 \leq_{de} M_2 \;\Rightarrow\; M_1 \leq_\spadesuit M_2$, $s_2 \leq_\spadesuit s_1$. Next, note that $s_1 \not\subseteq s_2$, since s_1 has a successor labeled c and s_2 does not. Thus $s_1 \not\leq_\spadesuit s_2$, and s_2 is a little brother of s_1 with respect to $\leq_\spadesuit$.

Next we show that the result of disconnecting s_2 from s_0 is not equivalent to M_1 with respect to $\leq_\spadesuit$. Since $M_1 \leq_\spadesuit M_2 \Rightarrow M_1 \subseteq M_2$, it is sufficient to show that the result of disconnecting s_2 from s_0 is not language equivalent to M_1. But this is true since disconnecting s_2 results in a structure with no fair traces from s_1. $\square$

Corollary 4.2.7 *The structure that results when little brothers are disconnected with respect to delay/game/exists simulation might not be* *equivalent to the original structure with respect to delay/game/exists simulation.*

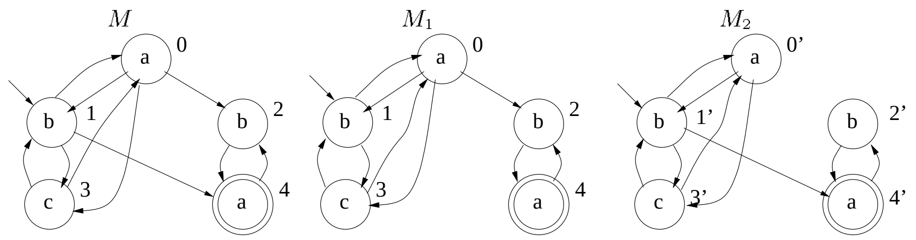

Figure 4.2: The structures M_1 and M_2 are equivalent with respect to delay/game/exists simulation to M, and they are both minimal. Note that state 2 ($4'$) is a little brother of 1 ($0'$) but cannot be disconnected.

4.2.3 Unique smallest in size structure

Lemma 4.2.8 *Let $\leq_\spadesuit$ be a preorder such that*

$$M_1 \leq_{de} M_2 \Rightarrow M_1 \leq_\spadesuit M_2 \Rightarrow M_1 \subseteq M_2.$$

Then there exists a structure M that has no unique smallest in size structure with respect to $\leq_\spadesuit$.

Proof Consider the structures in Figure 4.2. Structures M_1 and M_2 are delay equivalent but are not isomorphic. In order to see that $M_2 \leq_{de} M_1$, consider the strategy in which in every round the protagonist moves to the same state as the adversary, except for the transition from $1'$ to $4'$, when the protagonist moves to state 0. Similarly, we can show that $M_1 \leq_{de} M_2$. Since $M_1 \leq_{de} M_2 \Rightarrow M_1 \leq_\spadesuit M_2$, M_1 and M_2 are equivalent with respect to $\leq_\spadesuit$.

Next, we show that there is no smaller structure that is equivalent to M_1 and M_2 with respect to $\leq_\spadesuit$. Since $M_1 \leq_\spadesuit M_2 \Rightarrow M_1 \subseteq M_2$, it is sufficient to show that there is no smaller structure that is language equivalent to M_1 and M_2. Note that every equivalent structure must contain a strongly connected component with three states labeled $\{a\}$, $\{b\}$ and $\{c\}$. However, these states cannot be fair because there are no fair traces in M_1 and M_2 which have infinitely many states labeled

$\{c\}$. Thus, there should be two other states labeled $\{a\}$ and $\{b\}$ on a fair, strongly connected component. Consequently, there have to be at least five states in any structure that is language equivalent to M_1 and M_2. $\square$

Corollary 4.2.9 *There is no unique smallest in size structure with respect to delay/game/exists simulation.*

An interesting observation is that the minimization operations are not *independent*[5] [KP92]. For example, in structure M in Figure 4.1, states s_0 and s_1 are game/exists equivalent to states s_2 and s_3 respectively. Unifying states s_0 and s_2 results in structure M_2. Unifying states s_1 and s_3 results in structure M_1. Both structures are equivalent to M and neither can be further minimized. A similar phenomenon occurs in structure M of Figure 4.2: for delay/exists/game simulation, states s_4 and s_2 are little brothers of states s_0 and s_1 respectively. Disconnecting state s_4 from state s_1 results in M_1, and disconnecting state s_2 from state s_0 results in M_2. Again, both structures are equivalent to M, and neither structure can be further minimized.

4.2.4 An approximated minimization algorithm for delay/game/exists simulation

In Chapter 3, two efficient procedures for minimizing with respect to ordinary simulation are presented. In the previous sections we have shown that these procedures cannot be used for delay/game/exists simulation. Furthermore, we have shown that there is no equivalent unique smallest in size structure with respect to these simulations. As a result, we suggest an algorithm that performs some minimization but does not necessarily construct a minimal structure. Our algorithm uses the direct/delay simulations as an approximation of the game/exists simulation. The algorithm is presented in Figure 4.3. The first step results in $M' \equiv_{de} M$. The second step results in $M'' \equiv_{di} M'$. Since direct simulation implies delay simulation, $M'' \equiv_{de} M$. M'' is also equivalent to M with respect to game/exists simulation. Thus, the algorithm combines the advantages of the direct and the delay simulations in order

[5]Operations are not independent if one operation disables another.

Given a structure M,

1. Construct a quotient structure M' with respect to delay simulation.
2. Construct M'' by disconnecting little brothers in M' with respect to direct simulation.

Figure 4.3: Minimization algorithm for the delay/game/exists simulations.

to produce a reduced structure that is equivalent with respect to delay/game/exists simulations to the original one. The complexity of the first step is $O(m \cdot n^3)$ [EWS01a], and of the second step $O(m \cdot n)$ [BG00]. Thus the total complexity of the algorithm is $O(m \cdot n^3)$.

4.3 Relating the simulation notions to logics

In this section we investigate the relationship between the different notions of fair simulation and the logics ACTL and ACTL*. First we check for each notion whether it has a logical characterization. Next we check whether there exists a maximal structure for ACTL with respect to each notion.

4.3.1 Logical characterization

Definition 4.3.1 *Logic $\mathcal{L}$ characterizes a preorder $\leq$ if for all structures M_1 and M_2, $M_1 \leq M_2$ if and only if for every formula ϕ in $\mathcal{L}$, $M_2 \models \phi$ implies $M_1 \models \phi$.*

In [GL94], it is shown that if $M_1 \leq_\exists M_2$, then the following property holds: $\forall \phi \in$ ACTL*, $M_2 \models \phi$ implies $M_1 \models \phi$. Since all other simulation notions imply the exists simulation, this property holds for all of these notions.

We now investigate which of the fair simulations satisfy the other direction of logical characterization. We show that ACTL* characterizes the exists simulation but not the game/delay/direct simulation. On the other hand, ACTL does not characterize any of these notions.

First we prove that the exists simulation is characterized by the ACTL* logic. We prove that if $M \not\leq_{\exists} M'$, then there exists an ECTL* formula ϕ and an initial state s_0 of M such that $M, s_0 \models \phi$. Furthermore, for all initial states s_0' of M', $M', s_0' \not\models \phi$. This implies that there exists an ACTL* formula ψ which is equivalent to $\neg\phi$ such that $M' \models \psi$ but $M \not\models \psi$.

Our proof is similar to the proof in [ASS$^+$94] for fair bisimulation. It is based on a different definition of fair simulation. This definition called rational simulation, is presented below.

Definition 4.3.2 *Let ρ be a trace through a Kripke structure M. ρ is a rational trace if $\exists N, K$ such that $\forall i(i > N \rightarrow \rho^i = \rho^{((i-N) mod\ K)+N})$.*

Thus, a rational trace is a trace with a prefix of length N followed by a cycle of length K.

Definition 4.3.3 *A state s is smaller by rational simulation than a state t ($s \leq_{rat} t$) if they lie in the coarsest preorder H that satisfies*

- *$L(s) = L(t)$.*
- *for every fair rational trace ρ_s starting at s there exists a fair rational trace ρ_t starting at t such that $(\rho_s, \rho_t) \in H$.*

$M \leq_{rat} M'$ if for every $s_0 \in S_0$ there exists $s_0' \in S_0'$ such that $s_0 \leq_{rat} s_0'$.

Lemma 4.3.4 *Let s and t be states in structure M. If there exists a fair trace ρ_s from s such that for all fair traces ρ_t from t, $(\rho_s, \rho_t) \notin H$, then there exists a fair <u>rational</u> trace ρ_{sr} from s such that for all fair traces ρ_t from t, $(\rho_{sr}, \rho_t) \notin H$.*

The proof of Lemma 4.3.4 appears in Section 4.5.2. Corollary 4.3.5 is straightforward from Lemma 4.3.4.

Corollary 4.3.5 *If $M \not\leq_{\exists} M'$ then $M \not\leq_{rat} M'$.*

In the proof we refer to one structure instead of two. This can be done when we refer to M'' which is the union of M and M' where, $S'' = S \cup S'$, (assume $S \cap S' = \emptyset$), $R'' = R \cup R'$, and $F'' = F \cup F'$. having deduced Corollary 4.3.5, it is now sufficient to prove the following:

Lemma 4.3.6 *For every structure M and states s and t, If for all ECTL* formulas ϕ, $M, s \models \phi$ implies $M, t \models \phi$, then $s \leq_{rat} t$.*

Proof We prove that $s \not\leq_{rat} t$ implies that there exists an ECTL* formula ϕ such that $M, s \models \phi$ but $M, t \not\models \phi$.

We first inductively define a sequence of preorders over $S \times S$.

Definition 4.3.7

- *$(s, t) \in H_0$ iff $L(s) = L(t)$.*
- *$(s, t) \in H_{i+1}$ iff for every fair rational trace ρ_s starting at s there exists a fair rational trace ρ_t starting at t such that $(\rho_s, \rho_t) \in H_i$.*

Note that for every $i \geq 0$, $H_{i+1} \subseteq H_i$. Thus, after at most $|S|^2$ preorders, we reach a fixpoint. We use H_∞ to denote the preorder at the fixpoint. It is easy to see that H_∞ is exactly the fair rational simulation.

For every state s, we define the following ECTL* formulas. For every t such that $(s, t) \notin H_i$, we define $D_i(s, t)$ such that for every $(s, v) \in H_i$, $v \models D_i(s, t)$ and $t \not\models D_i(s, t)$. We also define formulas $C_i(s)$ such that for all states $v \in S$, $v \models C_i(s)$ iff $(s, v) \in H_i$.

We define $D_i(s, t)$ and $C_i(s)$ inductively.

- Let P be the set of atomic propositions true in s. Then for all $t \in S$, such that $(s, t) \notin H_0$ $D_0(s, t) = C_0(s) = \wedge_{(p \in P)} p \wedge_{(p \in AP \setminus P)} \neg p$.
- Let s and t be states such that $(s, t) \notin H_{i+1}$. Then there exists a fair rational path ρ from s for which there is no H_i-corresponding trace from t.

  Assume that $\rho = s_1, s_2, \ldots, s_N, (s_{N+1}, \ldots s_{N+K})^\omega$. We first define for $1 \leq j \leq K$ a formula that describes the cycle from place $j+N$, namely the trace
  $s_{N+j}, s_{N+j+1}, \ldots s_{N+K}, s_{N+1}, \ldots, s_{N+j-1}$.
  $cycle^j_{i+1}(s, t) = C_i(s_{(N+1+((j-1) mod\ K))}) \wedge \mathbf{X}(C_i(s_{(N+1+(j\ mod\ K))}) \wedge \mathbf{X}(C_i(s_{(N+1+((j+1)\ mod\ K))}) \wedge \ldots \mathbf{X}(C_i(s_{(N+1+(j+K-2)\ mod\ K)})) \ldots)$.
  Let $cycle_{i+1}(s, t) = \vee_{j=1}^{K} cycle^j_{i+1}(s, t)$.
  Let $trace_{i+1}(s, t) = C_i(s_1) \wedge \mathbf{X}(C_i(s_2) \ldots \wedge \mathbf{X}(C_i(s_{N+K}) \wedge \mathbf{X}\ \mathbf{G}(cycle_{i+1}(s, t)) \ldots)$.
  Let $D_{i+1}(s, t) = \mathbf{E}\, trace_{i+1}(s, t)$.

  Let $C_{i+1}(s) = \wedge_{(s,t) \notin H_i} D_{i+1}(s, t)$.

Note that $\rho^{N+1} \models cycle_{i+1}(s,t)$. Furthermore,
$\rho \models C_i(s_1) \wedge \mathbf{X}(C_i(s_2), \ldots, \wedge \mathbf{X}(C_i(s_N))\ldots)$, thus $s \models D_{i+1}(s,t)$.

Given a state v, if $v \models D_{i+1}(s,t)$, then there is a fair trace ρ' starting at v such that $\rho' \models trace_{i+1}(s,t)$. We prove that $(\rho, \rho') \in H_i$. First, for each $1 \leq j \leq N+K$, $\rho'^j \models C_i(s_j)$. Further, it is true that for $j \geq N+1$, $\rho'^j \models cycle_{i+1}(s,t)$. Using these facts, one can show by induction that for $j \geq 1$, $\rho'^{N+j} \models cycle_{i+1}^{((j-1) mod\ K)+1}(s,t)$. This implies that for each $j \geq 1$, $\rho^{N+j} \models C_i(s_{N+1+(j-1) mod\ K})$.

Once we know that for every state v such that $(s,v) \in H_{i+1}$, $v \models D_{i+1}(s,t)$, it is easy to see that for every state v, $v \models C_{i+1}(s)$ iff $(s,v) \in H_{i+1}$.

Let $C_\infty(s)$ be the formula such that for all $v \in S$, $v \models C_\infty(s) \Leftrightarrow (s,v) \in H_\infty$. Then for all $t \in S$, $t \not\models C_\infty(s) \Leftrightarrow (s,t) \notin H_\infty$. Since $s \models C_\infty(s)$, for all $t \in S$ such that $(s,t) \notin H$ there exists $\psi \in$ECTL* that differentiates between s and t.$\Box$

Assume that $M \not\leq_{rat} M'$. Then there exists an initial state $s_0 \in S_0$ such that for all initial states $s_0' \in S_0'$, $s_0 \not\leq_{rat} s_0'$. Thus, $M, s_0 \models C_\infty(s_0)$, and for all $s_0' \in S_0'$, $M', s_0' \not\models C_\infty(s_0)$. Let $\psi \in$ACTL* be the formula equivalent to $\neg C_\infty$. Then, since $M, s_0 \not\models \psi$, $M \not\models \psi$ and since for all $s_0' \in S_0'$, $M', s_0' \models \psi$, $M' \models \psi$.

From Corollary 4.3.5 and Lemma 4.3.6 we deduce Corollary 4.3.8.

Corollary 4.3.8 *If for all ACTL* formulas ψ, $M' \models \psi$ implies $M \models \psi$, then $M \leq_\exists M'$.*

Unlike ACTL*, ACTL does not characterize the exists simulation. In [ASS+94] two structures, M_1 and M_2, are given. It is shown in [ASS+94] that for every ϕ in ACTL, $M_2 \models \phi$ implies $M_1 \models \phi$. However, there exists an ACTL* formula φ such that $M_2 \models \varphi$ but $M_1 \not\models \varphi$. Since ACTL* characterizes the exists simulation, $M_1 \not\leq_\exists M_2$.

Unfortunately, the game, direct, and delay simulations cannot be characterized by either ACTL* or ACTL. In [HKR97] two structures, M_1 and M_2, are given such that $M_1 \leq_\exists M_2$ but $M_1 \not\leq_g M_2$. Since ACTL* characterizes the exists simulation, for every ϕ in ACTL* (and therefore ACTL), $M_2 \models \phi$ implies $M_1 \models \phi$. Therefore, ACTL* (ACTL) does not characterizes the game simulation. Since the direct/delay simulation implies the game simulation, ACTL* (ACTL) does not characterize them either.

We have shown that ACTL* characterizes the exists simulation but not the game/delay/direct simulation. Furthermore, ACTL does not characterize any of these notions. The question arises whether the direct/delay/game simulation can be characterized by any other logic. [HKR97] shows that the game simulation can be characterized by the Universal Alternating Free μ-Calculus ($\forall$**AFMC**) logic when interpreted over fair structures.

We show that no reasonable logic that describes the fair branching behavior of a structure can characterize the direct/delay simulation. Consider structures M_1 and M_2 in Figure 4.4. M_1 and M_2 cannot be distinguished by a temporal logic formula. This is because they have computation trees, with exactly the same fair traces. However, $M_1 \not\leq_{de} M_2$ and therefore, $M_1 \not\leq_{di} M_2$. To see that $M_1 \not\leq_{de} M_2$ note that if the adversary chooses the path 123^{∞} the protagonist must choose the path $1'2'3'^{\infty}$. However 2 is a fair state while $2'$ and $3'$ are not. Thus neither simulation can be characterized by any such logic.

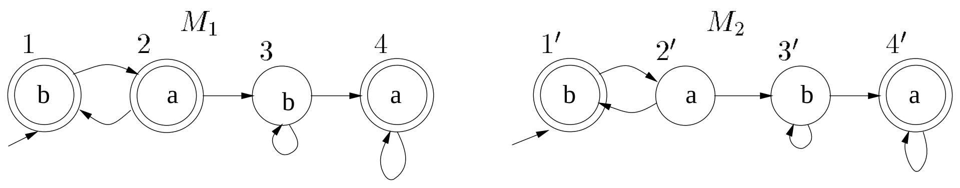

Figure 4.4: The direct/delay simulations cannot be characterized by temporal logics.

4.3.2 Maximal structure

Next we check for the existence of a maximal structure for a formula with respect to a preorder.

Definition 4.3.9 *A structure M_ϕ is maximal for formula ϕ with respect to preorder $\leq$ if for every structure M, $M \models \phi \Leftrightarrow M \leq M_\phi$.*

In [GL94] a construction of a maximal structure for ACTL formulas with respect to the exists simulation is presented. The maximal structure is used as a tableau for the formula. In this section we check whether the direct/delay/game simulations have a maximal structure. We prove that the maximal structure constructed in [GL94] is maximal with respect to the game simulation as well. On the other hand, we show that the formula $\mathbf{A}[a \,\mathbf{U}\, b]$ has no maximal structure with respect to the direct and delay simulations. This formula is contained in both ACTL and ACTL*.

4.3.3 A maximal structure for ACTL with respect to game simulation

We prove that for every ACTL formula, the tableau of the formula as defined in [GL94] is the maximal structure for the formula with respect to the game simulation. First, we describe the construction of the tableau as shown in [GL94]. In [GL94], a different type of fairness constraint, the generalized Büchi acceptance condition, is used. A *generalized Büchi acceptance* condition is a set $F = \{f_1, f_2, \dots f_n\}$ of subsets of S. A trace ρ is *fair* according to F iff for every $1 \leq i \leq n$, $\inf(\rho) \cap f_i \neq \emptyset$. Since the game simulation is not limited to a certain type of fairness constraint, we do not have to change anything in its definition.

For the remainder of this section, fix an ACTL formula ψ. Let AP_ψ be the set of atomic propositions in ψ. The tableau associated with ψ is a structure $\mathcal{T}_\psi = (S_T, R_T, S_{0T}, L_T, F_T)$. The set of *elementary formulas* of ψ, $el(\psi)$, is defined as follows:

1. $el(p) = el(\neg p) = \{p\}$ if $p \in AP_\psi$.
2. $el(\phi_1 \vee \phi_2) = el(\phi_1 \wedge \phi_2) = el(\phi_1) \cup el(\phi_2)$.
3. $el(\mathbf{AX}\, \phi) = \{\mathbf{AX}\, \phi\} \cup el(\phi)$.
4. $el(\mathbf{A}[\phi_1 \,\mathbf{U}\, \phi_2]) = \{\mathbf{AX}\, False, \mathbf{AX}(\mathbf{A}[\phi_1 \,\mathbf{U}\, \phi_2])\} \cup el(\phi_1) \cup el(\phi_2)$.
5. $el(\mathbf{A}[\phi_1 \,\mathbf{R}\, \phi_2]) = \{\mathbf{AX}\, False, \mathbf{AX}(\mathbf{A}[\phi_1 \,\mathbf{R}\, \phi_2])\} \cup el(\phi_1) \cup el(\phi_2)$.

The set of tableau states is $S_T = \mathcal{P}(el(\psi))$[6]. The labeling function is $L_T(s_t) = s_t \cap AP_\psi$. In order to specify the set S_{0T} of initial states and the transition relation R_T, we need an additional function *sat* that associates with each sub-formula ϕ of ψ a set of states in S_T. Intuitively, $sat(\phi)$ will be the set of states that satisfy ϕ.

1. $sat(\phi) = \{s \mid \phi \in s\}$ where $\phi \in el(\psi)$.

[6]Some of the states are deleted in order to keep R_T total.

2. $sat(\neg\phi) = \{s \mid \phi \notin s\}$ where ϕ is an atomic proposition. Recall that only atomic propositions can be negated in ACTL.

3. $sat(\phi \vee \varphi) = sat(\phi) \cup sat(\varphi)$.

4. $sat(\phi \wedge \varphi) = sat(\phi) \cap sat(\varphi)$.

5. $sat(\mathbf{A}[\phi\ \mathbf{U}\ \varphi]) = (sat(\varphi) \cup (sat(\phi) \cap sat(\mathbf{AX}(\mathbf{A}[\phi\ \mathbf{U}\ \varphi])))) \cup sat(\mathbf{AX}\,False)$.

6. $sat(\mathbf{A}[\phi\ \mathbf{R}\ \varphi]) = (sat(\varphi) \cap (sat(\phi) \cup sat(\mathbf{AX}(\mathbf{A}[\phi\ \mathbf{R}\ \varphi])))) \cup sat(\mathbf{AX}\,False)$.

The set of initial states of the tableau is $S_{0T} = sat(\psi)$. The transition relation is defined so that if $\mathbf{AX}\,\phi$ is included in some state then all its successors should satisfy ϕ.

$$R_T(s_1, s_2) = \bigwedge_{AX\,\phi \in el(\psi)} (\mathbf{AX}\,\phi) \in s_1 \Rightarrow s_2 \in sat(\phi).$$

The fairness constraint guarantees that *eventuality* properties are fulfilled. This is done by requiring that for every fair trace ρ, for every elementary formula $\mathbf{AX}\,\mathbf{A}[\phi\,\mathbf{U}\,\varphi]$ of ψ, and for every state s on ρ, if $s \in sat(\mathbf{AX}\,\mathbf{A}[\phi\,\mathbf{U}\,\varphi])$, then there is a later state t on ρ such that $t \in sat(\varphi)$. Thus, we obtain the following fairness constraints:

$$F_T = \{\, ((S_T - sat(\mathbf{AX}\,\mathbf{A}[\phi\,\mathbf{U}\,\varphi])) \cup sat(\varphi)) \mid \mathbf{AX}\,\mathbf{A}[\phi\,\mathbf{U}\,\varphi] \in el(\psi)\,\}.$$

4.3.4 The tableau is the maximal structure for game simulation

In this section we prove that for every Kripke structure M, $M \models \psi$ iff $M \leq_g \mathcal{T}_\psi$. Most lemmas were proved in [GL94] for the exists simulation. We give proofs only for the lemmas that are different due to the change of the simulation preorder.

Lemma 4.3.10 *[GL94] For all subformulas ϕ of ψ, if $t \in sat(\phi)$, then $t \models \phi$.*

The main result of Lemma 4.3.10 is that the tableau for ψ satisfies ψ. This is because any initial state of $\mathcal{T}_\psi$ is in $sat(\psi)$, and therefore every initial state of $\mathcal{T}_\psi$ satisfies ψ. Consequently, since ACTL is preserved by the $\leq_g$ preorder, for every Kripke structure M, if $M \leq_g \mathcal{T}_\psi$, then $M \models \psi$.

Our next step is to prove that $M \models \psi$ implies $M \leq_g \mathcal{T}_\psi$. We show that if $M \models \psi$ then the protagonist has a winning strategy function in a game over $M \times \mathcal{T}_\psi$. We define the strategy function π as follows: $\pi(s_0, \perp) = \{ \phi \mid \phi \in el(\psi), s_0 \models \phi \}$ and $\pi(s', t) = \{ \phi \mid \phi \in el(\psi), s' \models \phi \}$. Thus, whenever the adversary moves to a state s', the protagonist moves to $t' = \pi(s', t)$, such that both s', t' satisfy exactly the same set of elementary formulas of ψ. The following lemma extends this result for all subformulas of ψ.

Lemma 4.3.11 *[GL94] If $t' = \pi(s', t)$, then for every subformula or elementary formula ϕ of ψ, $s' \models \phi$ implies $t' \in sat(\phi)$.*

Lemma 4.3.12 *π is a winning strategy.*

Proof

1. Any given state s' satisfies a unique subset of $el(\psi)$. Thus, for every s', t' is unique and π is a function.

2. For every $s_0 \in S_0$, by Lemma 4.3.11 $M, s_0 \models \psi$ implies $t_0 = \pi(s_0, \perp) \in sat(\psi)$. By the definition of S_{0T}, this implies $t_0 \in S_{0T}$.

3. Assume that $t' = \pi(s', t)$. Then for every $p \in AP_\psi$, $p \in L(s') \Leftrightarrow s' \models p \Leftrightarrow p \in L_T(t')$.

4. Assume that $t' = \pi(s', t)$. Let (s, t) be the position of the game in the previous round. Let $\mathbf{AX}\,\phi_1, \mathbf{AX}\,\phi_2, \ldots, \mathbf{AX}\,\phi_n$ be all the formulas of the form $\mathbf{AX}\,\phi$ in $el(\psi)$ which s satisfies. Then we have $s' \models \phi_1$, $s' \models \phi_2$, $\ldots, s' \models \phi_n$. By Lemma 4.3.11, $t' \in sat(\phi_1)$, $t' \in sat(\phi_2)$, $\ldots, t' \in sat(\phi_n)$. Now by the definition of π, the formulas of the form $\mathbf{AX}\,\phi$ in t must be exactly $\mathbf{AX}\,\phi_1$, $\mathbf{AX}\,\phi_2$, $\ldots$, $\mathbf{AX}\,\phi_n$. Then by the definition of R_T, we see that $(t, t') \in R_T$.

5. We prove that if ρ is a fair run, then $\pi(\rho)$ is also a fair run. Assume that $\pi(\rho)$ is not fair. By the definition of F_T, there must be some elementary subformula $\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b]$ such that

$$inf(\pi(\rho)) \cap ((S_T - sat(\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b])) \cup sat(\phi_b)) = \emptyset.$$

This means that there is an $i \geq 0$ such that for all $j \geq i$, $\pi(s_j, t_{j-1}) \in sat(\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b])$ but $\pi(s_j, t_{j-1}) \notin sat(\phi_b)$.

Consider the state $t_i = \pi(s_i, t_{i-1})$. $t_i \in sat(\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b])$ iff $\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b] \in t_i$. The definition of π then implies that $s_i \models \mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b]$. In addition, Lemma 4.3.11 implies that if $t_i \notin sat(\phi_b)$, then $s_i \not\models \phi_b$. Since $\pi(s_i, t_{i-1}) \in sat(\mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b])$ and for all $j \geq i$, $\pi(s_j, t_{j-1}) \notin sat(\phi_b)$, then $s_i, s_{i+1}, \ldots$ is a fair trace in M starting at s_i, and every state on this trace satisfies $\neg\phi_b$. But $s_i \models \mathbf{AX}\,\mathbf{A}[\phi_a\,\mathbf{U}\,\phi_b]$, a contradiction. Hence $\pi(\rho)$ is in fact a fair trace in $\mathcal{T}_\psi$. $\Box$

Corollary 4.3.13 *For any structure M, $M \models \psi$ iff $M \leq_g \mathcal{T}_\psi$. Thus, $\mathcal{T}_\psi$ is the maximal structure for ψ with respect to game simulation.*

4.3.5 A maximal structure for direct/delay simulation

We now show that it is impossible to construct a maximal structure for the formula $\phi = \mathbf{A}[a\,\mathbf{U}\,b]$ with respect to the direct/delay simulations. Thus, any logic that contains this formula or an equivalent formula, in particular ACTL and ACTL*, does not have a maximal structure with respect to these simulations. More specifically, we show that there is no finite structure $\mathcal{T}_\phi$ such that $\mathcal{T}_\phi \models \phi$ and $\mathcal{T}_\phi$ is greater by the direct/delay simulation than any structure that satisfies ϕ. Since the direct simulation implies the delay simulation, it is sufficient to prove this result for the delay simulation. In Figure 4.5 we present a sequence of structures $M_0, M_1, \ldots$ such that for every n in $I\!N$, $M_n \models A(aUb)$. We prove that for every n and every structure M', if $M_n \leq_{de} M'$ and $M' \models \mathbf{A}[a\,\mathbf{U}\,b]$ then $|M'| \geq n$. Thus, any structure that satisfies $\mathbf{A}[a\,\mathbf{U}\,b]$ and is greater by the delay simulation than all the structures in the sequence has to be infinite.

Lemma 4.3.14 *For every $n > 0$ and every structure M', if $M_n \leq_{de} M'$ and $M' \models \mathbf{A}[a\,\mathbf{U}\,b]$, then $|M'| \geq n$.*

Proof Let $n \in I\!N$ be a natural number and M' be a structure such that $M' \models \mathbf{A}[a\,\mathbf{U}\,b]$ and $M_n \leq_{de} M'$. In a game over $M_n \times M'$ the protagonist has a winning strategy and thus it wins in every game no matter how the adversary plays. Consider the following strategy of the adversary. It starts from the initial state. As long as the protagonist

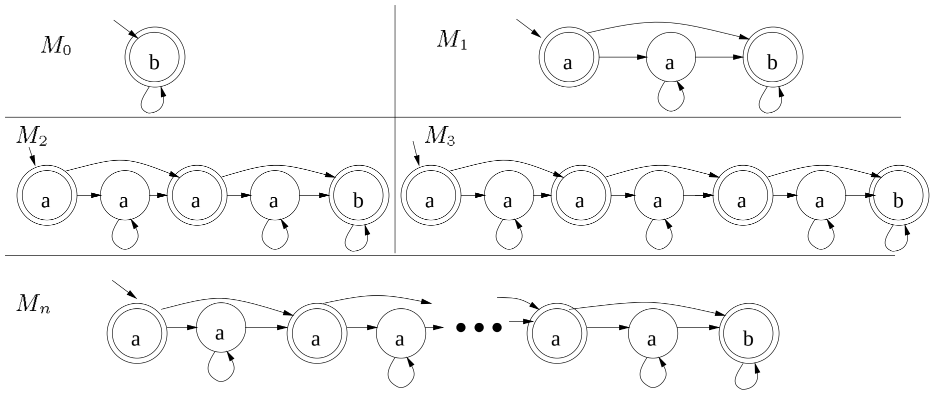

Figure 4.5: There is no finite structure M' such that for every n in $I\!N$, M' is greater by direct/delay simulation than M_n, and $M' \models \mathbf{A}[a\,\mathbf{U}\,b]$.

moves to a fair state the adversary moves to the next fair state (until it reaches the last one). If the protagonist moves to a state that is not fair, then the adversary moves to the successor which is not fair in M_n and stays there until the protagonist moves to a fair state in M'. We distinguish between two cases:

1. The suffix of the game is an infinite sequence of unfair states in both structures. In this case the adversary is the last player who was in a fair state. Thus it wins the game. This means that M' is not greater than M_n by the delay simulation, a contradiction.

2. Otherwise, the adversary moves through n fair states in M_n that are labeled a to the state labeled b. Since the adversary moves to a fair state only when the protagonist is in a fair state, the protagonist has been in n fair states that are labeled a. Since $M' \models \mathbf{A}[a\,\mathbf{U}\,b]$, these states must be different (otherwise there would be an infinite fair trace which is labeled a). Thus the size of M' is at least n. $\Box$

We proved that there is no maximal structure for $\mathbf{A}[a\,\mathbf{U}\,b]$ with respect to the direct/delay simulations.

4.4 A new implementation for the assume-guarantee framework

This section shows that the game simulation can replace the exists simulation in the implementation of the assume-guarantee paradigm [Fra76, Jon83, MC81, Pnu84], as suggested in [GL94].

In the assume-guarantee paradigm, properties of different parts of the systems are verified separately. The environment of the verified part is represented by a formula that describes its properties. The formula either has been verified or is given by the user. The method proves assertions of the form $\psi M \phi$, meaning that if the environment satisfies ψ then the composition of M with the environment satisfies ϕ. The method enables the creation of a proof schema which is based on the structure of the system. [GL94] suggests a framework that uses the assume-guarantee paradigm for semi-automatic verification. It presents a general method that uses models as assumptions; the models are either generated from a formula as a *tableau* or are abstract models given by the user. The proof of $\psi M \phi$ is done automatically by verifying that the composition of the tableau for ψ with M satisfies ϕ. The method requires a preorder $\leq$, a composition operator $||$, and a specification language $\mathcal{L}$ which satisfy the following properties:

1. For every two structures M_1, M_2, if $M_1 \leq M_2$, then for every formula ψ in $\mathcal{L}$, $M_2 \models \psi$ implies $M_1 \models \psi$.
2. For every two structures M_1, M_2, $M_1 || M_2 \leq M_1$.
3. For every three structures M_1, M_2, M_3, $M_1 \leq M_2$ implies $M_1 || M_3 \leq M_2 || M_3$.
4. Let ψ be a formula in $\mathcal{L}$ and $\mathcal{T}_\psi$ be a tableau for ψ. Then $\mathcal{T}_\psi$ is the maximal structure with respect to the preorder $\leq$.
5. For every structure M, $M \leq M || M$.

An implementation for this framework was presented in [GL94]. The implementation uses the $ACTL$ logic as the specification language, the exists simulation preorder, and a composition operator which satisfy the properties above. In this section we suggest a new implementation which is similar to that of [GL94], except that the game simulation is used as the preorder. We show that the game simulation can replace the exists simulation. As we have stated, the game simulation preserves the ACTL logic, and thus property one is satisfied. In Section 4.3 we proved that the game simulation satisfies property four. Thus, it is left to show that the game simulation preorder and the composition operator as defined in [GL94] satisfy properties two, three and five. Again we use generalized Büchi constraints. In order to prove these properties we need to define the composition operator $\|$.

Definition 4.4.1 *Let M_1, M_2 be Kripke structures. The parallel composition of M_1 and M_2, denoted $M_1\|M_2$, is the structure M defined as follows.*

- $AP = AP_1 \cup AP_2$.
- $S = \{(s_1,s_2) | L_1(s_1) \cap AP_2 = L_2(s_2) \cap AP_1\}$[7].
- $R = \{((s_1,s_2),(t_1,t_2)) | (s_1,t_1) \in R_1 \wedge (s_2,t_2) \in R_2\}$.
- $S_0 = (S_{0_1} \times S_{0_2}) \cap S$.
- $L((s_1,s_2)) = L_1(s_1) \cup L_2(s_2)$.
- $F = \{(f_i \times S_2) \cap S | f_i \in F_1\} \cup \{(S_1 \times f_i) \cap S | f_i \in F_2\}$.

Remark: In all notions of simulation, there is a requirement that if $s_1 \leq s_2$, then $L_1(s_1) = L_2(s_2)$. When M_1 and M_2 are defined over different **AP** we replace this requirement with $L_1(s_1) \cap AP_2 = L_2(s_2) \cap AP_1$.

Lemma 4.4.2 *(* **property 2.** *) For every pair of Kripke structures M_1, M_2,*
$M_1\|M_2 \leq_g M_1$.

Proof We define a strategy π as follows: $\pi((s_{01},s_{02}),\perp) = s_{01}$ and $\pi((s'_1,s'_2),s_1) = s'_1$, i .e., the protagonist moves on the projection of the adversary's trace on M_1. It is easy to see that π is a function. Let $((s_1,s_2),s_1)$ be the previous position in the game and assume that the adversary moves to (s'_1,s'_2). Then $s'_1 = \pi((s'_1,s'_2),s_1)$. Clearly, $L_{12}(s'_1,s'_2) \cap AP_1 = L_1(s'_1)$. The definition of composition implies that if

[7]Some of the states might have to be deleted in order to keep R total.

$((s_1, s_2), (s'_1, s'_2))$ is a transition in $M_1 \| M_2$ then (s_1, s'_1) is a transition in M_1. Furthermore, if the adversary's trace is fair then the protagonist's trace is fair as well. □

Lemma 4.4.3 *(* **property 3.** *) Let M_1, M_2, M_3 be Kripke structures. Then $M_1 \leq_g M_2$ implies $M_1 \| M_3 \leq_g M_2 \| M_3$.*

Proof Let π be a strategy in a game over $M_1 \times M_2$. We define a strategy π' as follow: $\pi'((s_{01}, s_{03}), \perp) = (\pi(s_{01}, \perp), s_{03})$ and $\pi'((s'_1, s'_3), (s_2, s_3)) = (\pi(s'_1, s_2), s'_3)$, i.e., whenever the adversary moves to s'_1 in M_1 and s'_3 in M_3, the protagonist moves to the same state in M_3 and to $s'_2 = \pi(s'_1, s_2)$ in M_2.

It is easy to see that π' is a function. Let $((s_1, s_3), (s_2, s_3))$ be the previous position in the game and assume that the adversary moves to (s'_1, s'_3). Then
$\pi'((s'_1, s'_3), (s_2, s_3)) = (\pi(s'_1, s_2), s'_3)$. Let $s'_2 = \pi(s'_1, s_2)$. Since π is a winning strategy, $L_1(s'_1) = L_2(s'_2)$ and (s_2, s'_2) is a transition in M_2. Thus, $L_{13}(s'_1, s'_3) = L_{23}(s'_2, s'_3)$. Furthermore, the definition of composition implies that if $((s_1, s_3), (s'_1, s'_3))$ is a transition in $M_1 \| M_3$ then $((s_2, s_3), (s'_2, s'_3))$ is a transition in $M_2 \| M_3$.

Whenever the adversary moves on a fair trace in $M_1 \| M_3$, the traces projected on M_1 and M_3 are both fair. The protagonist moves on the same trace on M_3. Thus this trace is fair. Let ρ_1 be the trace on M_1 along which the adversary moves. Since ρ_1 is fair and π is a strategy, the trace $\pi(\rho_1)$ along which the protagonist moves on M_2 is fair as well. The definition of $\|$ implies that the protagonist moves on a fair trace in $M_2 \| M_3$. □

Lemma 4.4.4 property 5. *For every structure M, $M \leq_g M \| M$.*

Proof Consider the strategy $\pi(s_0, \perp) = (s_0, s_0)$ and $\pi(s', (s, s)) = (s', s')$. Clearly π is a winning strategy. □

We proved that the game simulation preorder and the composition operator satisfy the properties required in [GL94]. Therefore, game simulation can replace the exists simulation in the assume-guarantee framework presented in [GL94].

4.4.1 Complexity

Verifying a formula of the form $\psi M \varphi$ is PSPACE-complete in the size of ψ [KV98]. However, the real bottleneck of this framework is checking for fair simulation between models, which for the exists simulation is PSPACE complete in the size of the models. (Typically, models are much larger than formulas). Thus, replacing the exists simulation with the game simulation reduces this complexity to polynomial and eliminates the bottleneck of the framework. However, the algorithm for game simulation presented in [EWS01a] refers to Kripke structures with regular Büchi constraints, and the implementation presented in [GL94] refers to Kripke structures with generalized Büchi constraints. In order to apply the algorithm suggested in [EWS01a] within the assume-guarantee framework, we need a translation between these types of fairness constraints.

[CVWY91] defines a transformation of a Büchi automaton with generalized fairness constraints into a Büchi automaton with regular fairness constraints. Here we show that applying this transformation to a Kripke structure with generalized Büchi constraints results in a Kripke structure with regular Büchi constraints that is game simulation equivalent to the original one. The translation affects the size of the structure and thus the complexity of the construction of the preorder. The sizes of S and R are multiplied by $|F|$, where $|F|$ is the number of sets in F. Thus the complexity of constructing the preorder is $|F| \cdot |R| \cdot (|S| \cdot |F|)^3 = |R| \cdot |S|^3 \cdot |F|^4$. Note that in the tableau for a formula, $|F|$ is bounded by the size of the formula and the size of the tableau is exponential in the size of the formula; thus, the transformation of the tableau to regular fairness constraints result in a strucuture that is logarithmic bigger than the original one.

Definition 4.4.5 *[CVWY91] Let $M =< S, R, S_0, L, \{f_1, f_2, \dots f_n\} >$ be a Kripke structure with generalized Büchi constraints. We define the Kripke structure $M_r =< AP, S_r, R_r, L_r, F_r >$ with a regular Büchi constraint, as follows:*

- $S_r = S \times \{1, 2, \dots n\}$.
- $R_r = \cup_{i=1}^{n}\{((s_1, i), (s_2, i)) | (s_1, s_2) \in R \wedge s_1 \notin f_i\} \cup$
  $\cup_{i=1}^{n-1}\{((s_1, i), (s_2, i+1)) | (s_1, s_2) \in R \wedge s_1 \in f_i\} \cup$
  $\{((s_1, n), (s_2, 1)) | (s_1, s_2) \in R \wedge s_1 \in f_n\}$.
- $S_{r_0} = S_0 \times \{1\}$.
- $L_r(s, i) = L(s)$.

- $F_r = \{(s,n) | s \in f_n\}$.

In the proof below M denotes a Kripke structure with generalized Büchi constraints. M_r denotes the transformation of M to a Kripke structure with regular Büchi constraints. We show that $M_r \leq_g M$ and $M \leq_g M_r$.

Lemma 4.4.6 *$M \leq_g M_r$.*

Proof : First we define a strategy π for the protagonist: $\pi(s_0, \perp) = (s_0, 1)$ and

$$\pi(s', (s,i)) = \begin{cases} (s', i) & s \notin f_i \\ (s', i+1) & i < n \wedge s \in f_i \\ (s', 1) & i = n \wedge s \in f_n. \end{cases}$$

Next, we prove that π is a winning strategy. It is easy to see that π is a function. The definition of the transformation implies that if $(s,i) = \pi(s,(t,j))$ then $L_r((s,i)) = L(s)$ and that $((t,j),(s,i))$ is a transition in M_r.

It is left to prove that if the adversary moves on a fair trace ρ in M then the protagonist moves on $\pi(\rho)$, which is a fair trace in M_r.

First, we prove that for every $i \in 1, 2, \ldots n$

(*) there are infinitely many states of the form (s,i) in $\pi(\rho)$.

Assume to the contrary that there is an index $i \in \{1, 2, \ldots n\}$ which does not satisfy (*). Let j be the minimal index which does not satisfy (*) and let k be the index before j ($k = ((j-2) mod\ n) + 1$). Then there exists a suffix of $\pi(\rho)$ in which all the states are of the form (s,k). This implies that there exists a suffix of ρ without states in f_k. Thus, ρ is not fair, a contradiction.

Next we prove that $\pi(\rho)$ is fair. Since $\pi(\rho)$ contains infinitely many states of the form (s,n) and infinitely many states of the form $(s,1)$, then there exist infinitely many states in F_r. $\Box$

Lemma 4.4.7 *$M_r \leq_g M$.*

Proof We define the strategy π for the protagonist: $\pi((s_0,1),\perp) = s_0$ and $\pi((s_2,i),s_1) = s_2$. It is easy to see that π is a function and that $s_2 = \pi((s_2,j),s_1)$ implies that $L_r((s_2,j)) = L(s_2)$. $((s_1,i),(s_2,j)) \in R_r$

also implies $(s_1, s_2) \in R$. It is left to prove that if the adversary moves on a fair trace in M_r then the protagonist moves on a fair trace in M. Let $\rho = (s_0, i_0), (s_1, i_1), (s_2, i_2), \ldots$ be a fair trace in M_r. We prove that

$$\pi(\rho) = s_0, \pi((s_1, i_1), s_0), \pi((s_2, i_2), s_1), \pi((s_3, i_3), s_2), \ldots = s_0, s_1, s_2, \ldots$$

is a fair run in M. Assume to the contrary that $\pi(\rho)$ is not fair. Then there exists an index $i \in \{1 \ldots n\}$ such that $\pi(\rho)$ contains only finitely many states in f_i. Thus, there is a suffix of $\pi(\rho)$ without any state in f_i. This implies that

(**) there exists a suffix of ρ, without any states of the form (s, i), where s is an element in f_i.

Let j be the minimal index that satisfies (**). Then there exists a suffix of ρ in which all the states are of the form (s, j). This implies that this suffix does not contain any states in $\{(s, n) | s \in f_n\}$. Thus ρ is not fair, a contradiction. $\Box$

4.5 Complementary proofs

In this section we complete the proofs of Lemma 4.2.3 and Lemma 4.3.4.

4.5.1 A quotient structure for the delay simulation

In this section we prove Lemma 4.2.3. For every structure M, let M^Q be its quotient structure with respect to the delay simulation. Then M and M^Q are equivalent with respect to the delay simulation.

The proof that $M \leq_{de} M^Q$ is straightforward. Consider the strategy $\pi(s_0, \perp) = [s_0]$ and $\pi(s', [s]) = [s']$. It easy to see that π is a winning strategy.

Before we prove the other direction, we need some new definitions. First, we extend the definition of delay simulation to a relation over the states of a structure.

In [EWS01a] it is shown that there exists a strategy π^* such that π^* is a winning strategy for every simulation game over $M \times M$, where the adversary and the protagonist start at states s_1 and s_2 such that $s_1 \leq_{de} s_2$. Another property of π^* is presented in Proposition 4.5.1.

Proposition 4.5.1 *[EWS01a] Let s_1 and s_2 be states in M such that $s_1 \leq_{de} s_2$. Let s'_1 be a successor of s_1 and $s'_2 = \pi^*(s'_1, s_2)$. Then $s'_1 \leq_{de} s'_2$.*

Since the delay simulation is transitive, the following proposition is straightforward.

Proposition 4.5.2 *Let M be a structure and let M^Q be its quotient structure. Let s_1 and s_2 be states in M such that $s_1 \leq_{de} s_2$. Then every state s_3 that is in the same equivalence class as s_1 satisfies $s_3 \leq_{de} s_2$.*

We denote by $[s]$ the equivalence class of s. Lemma 4.5.3 and Lemma 4.5.4 imply that $M^Q \leq_{de} M$.

Lemma 4.5.3 *Let s_1 and s_2 be states in M such that $s_1 \leq_{de} s_2$. Then the protagonist has a strategy in a game over $M^Q \times M$ in which the adversary starts at $[s_1]$ and the protagonist starts at s_2. In each round assume that the adversary is at $[s_3]$ and the protagonist is at s_4. Then $s_3 \leq_{de} s_4$.*

Proof Let π^* be the winning strategy over $M \times M$. We define the strategy π' as follows: At the beginning, $\pi'([s_1], \perp) = s_2$. Assume that the previous position of the game was $([s_3], s_4)$ such that $s_3 \leq_{de} s_4$ and that the adversary moves to $[s'_3]$. The definition of M^Q implies that there exists a transition (t_3, t'_3) in M such that s_3 and t_3 are in the same class, as are s'_3 and t'_3. Proposition 4.5.2 implies that $t_3 \leq_{de} s_4$. We define $\pi'([s_3]', s_4) = \pi^*(t'_3, s_4)$. By the definition of π^*, π' is well-defined. Moreover, since s'_3 and t'_3 are in the same equivalence class, $s'_3 \leq_{de} t'_3$. Furthermore, by Proposition 4.5.1, since $s'_4 = \pi^*(t'_3, s_4)$, $t'_3 \leq_{de} s'_4$, and therefore $s'_3 \leq_{de} s'_4$.$\Box$

Note that this strategy ensures that in every round $L^Q([s_3]) = L(s_4)$. However, it does not ensure that whenever the adversary moves to a fair state, the protagonist moves to a fair state after finitely many rounds.

Lemma 4.5.4 *Let M be a structure and let M^Q be its quotient structure. Then $M^Q \leq_{de} M$.*

Proof We describe a strategy π'' which uses memory. In [EJ91, EWS01a] it is shown that if there exists a strategy with memory then there exists a memoryless strategy. The strategy π'' "remembers" two arguments:

the first argument is called the *status*, which can be either *fulfilled* or *unfulfilled*. The status is unfulfilled if the protagonist has not visited a fair trace since the last time the adversary did. Otherwise, the status is fulfilled. The second argument called the *middle*, and it "remembers" a state in M.

Let π^* be a winning strategy over $M \times M$ and π' a strategy over $M^Q \times M$ as defined in Lemma 4.5.3. We define π'' as follows: $\pi''([s_0], \bot) = s_0$, If the status is fulfilled, then $\pi''([s_3'], s_4) = \pi'([s_3'], s_4)$. Thus the middle argument is ignored. In a round where the status becomes unfulfilled, meaning that $[s_3]$ is fair and s_4 is not, we assign middle to be a fair state in the class of s_3 (there is at least one).

If the status is not fulfilled, assume that the adversary moves to $[s_3']$. Then we assign $middle' = \pi'([s_3'], middle)$ and $\pi''([s_3'], s_4) = \pi^*(middle', s_4)$.

In order to see that π'' is a winning strategy, first consider the round where the status becomes unfulfilled. In this round, s_3 and $middle$ are in the same class. Thus, if the position is $([s_3], s_4)$, then $s_3 \leq_{de} middle$. Furthermore, as long as the status does not become fulfilled, $middle$ moves along a trace in M such that whenever the adversary moves to $[s_3]$, $s_3 \leq_{de} middle$. Since $middle$ starts at a fair state and moves on a trace in M, by the definition of π^*, after a finite number of rounds, the protagonist moves to a fair state as well. □

4.5.2 Proving Lemma 4.3.4

Lemma 4.3.4 cliams the following:
Let s and t be states in structure M. If there exists a fair trace ρ_s from s such that for all fair traces ρ_t from t, $\rho_s \not\leq_{rat} \rho_t$, then there exists a fair <u>rational</u> trace ρ_{sr} from s such that for all fair traces ρ_t from t, $\rho_{sr} \not\leq_{rat} \rho_t$.

We define an equivalence relation with respect to $\leq_{rat}$, such that states s and t are equivalent with respect to $\leq_{rat}$ if $s \leq_{rat} t$ and $t \leq_{rat} s$. We denote by $[s]$ the equivalence class of s. We say that $[s_1] \leq_{rat} s_2$ iff $s_1 \leq_{rat} s_2$.

Definition 4.5.5 *Let M be a structure. We define the preorder structure M^P as follows:*

- $AP = \{C_1, C_2, \dots C_n\}$ *where* $\{C_1, C_2, \dots C_n\}$ *are the equivalence classes with respect to* $\leq_{rat}$.
- $S^P = \{(s, C_i) | s \in S$ *and there exists* $s' \in C_i$ *such that* $(s', s) \in H\}$.
- $((s, C_i), (t, C_j)) \in R^P \Leftrightarrow (s, t) \in R$.
- $S_0^P = \{(s_0, C_i) | s_0 \in S_0\}$.
- $L^P((s, C_i)) = C_i$.
- $(s, C_i) \in F^P \Leftrightarrow s \in F$.

Given a state s^P in M^P, we denote by $head(s^P)$ the first element of s^P and by $tail(s^P)$ the second element of s^P.

Lemma 4.5.6 *Given a fair trace* ρ_s *from a state* s *and a state* t *in a structure* M*, the following conditions are equivalent:*

1. *There exists a fair trace* ρ_t *from* t *such that* $\rho_s \leq_{rat} \rho_t$.
2. *There exists a fair trace* ρ_{tp} *from* $(t, [s])$ *such that for all* $i \geq 0$, $L^P(\rho_{tp}^i) = [\rho_s^i]$.

Proof For the first direction, assume that there exists a fair trace ρ_t from t such that $\rho_s \leq_{rat} \rho_t$. Consider the trace ρ_{tp} such that for all $i \geq 0$, $head(\rho_{tp}^i) = \rho_t^i$ and $tail(\rho_{tp}^i) = [\rho_s^i]$. By the definition of M^P, ρ_{tp} is a trace in M^P. Since ρ_t is fair, ρ_{tp} is fair as well.

For the second direction, assume that there exists a fair trace ρ_{tp} from $(t, [s])$ such that for all $i \geq 0$, $L^P(\rho_{tp}^i) = [\rho_s^i]$. Consider the trace ρ_t that satisfies $\rho_t^i = head(\rho_{tp}^i)$. By the definition of M^P, ρ_t^i is a trace in M. Furthermore, $\rho_s \leq_{rat} \rho_t$. Since ρ_{tp} is a fair trace, ρ_t is fair as well. $\Box$

Lemma 4.5.7 *Let* ρ_{sp} *be a fair trace from* $(s_1, [s_2])$ *in* M^P *such that* $L^P(\rho_{sp})$ *is an* ω*-regular word. Then there exists a rational trace* ρ_{s_1} *from* s_1 *such that for all* $i \geq 0$, $tail(\rho_{sp}^i) \leq_{rat} \rho_{s_1}^i$.

Proof Since $L^P(\rho_{sp})$ is an ω-regular word, we can write it as $w_1 w_2^\omega$. Let $N = |w_1|$ and $K = |w_2|$. Consider the trace ρ_{s_1} that satisfies, $\rho_s^i = head(\rho_{sp}^i)$. Then, for all $i \geq 0$, $tail(\rho_{sp}^i) \leq_{rat} \rho_{s_1}^i$. Let $[s_2] = tail(\rho_{sp}^N)$.

Then for all $i \geq 0$, $[s_2] \leq_{rat} \rho_{s_1}^{N+K\cdot i}$. Since M is a finite structure there exists a state $s_\spadesuit$ such that for infinitely many numbers i, $\rho_{s_1}^{N+K\cdot i} = s_\spadesuit$. Since ρ_{s_1} is fair, there are $i < j$ such that $\rho_{s_1}^{N+K\cdot i} = \rho_{s_1}^{N+K\cdot j} = s_\spadesuit$ and an index $N + K \cdot i \leq k \leq N + K \cdot j$ such that $\rho_{s_1}^k$ is a fair state.

Let $\rho_{s'}$ be the following trace: For all $0 \leq l \leq N + K \cdot i$, $\rho_{s'}^l = \rho_{s_1}^l$ and for all $l > N + K \cdot i$, $\rho_{s'}^l = \rho_{s_1}^{((l-N-K\cdot i) mod\ ((j-i)\cdot K))+N+K\cdot i}$. It is easy to see that $\rho_{s'}$ is a fair rational trace. Furthermore, the construction of $\rho_{s'}$ implies that for all $l \geq 0$, $tail(\rho_{sp}^l) \leq_{rat} \rho_{s'}^l$. □

Finally we prove Lemma 4.3.4: Assume that there exists a fair trace from s such that for every fair trace ρ_t from t, $\rho_s \not\leq_{rat} \rho_t$. By Lemma 4.5.6, there is no fair trace ρ_{tp} such that for all $i \geq 0$, $L^P(\rho_{tp}^i) = [\rho_s^i]$. We refer to $(s,[s])$ and $(t,[t])$ as two copies M_s^P and M_t^P of M^P where the former has $(s,[s])$ as a single initial state and the latter has $(t,[t])$ as a single initial state. Then the language of $M_s^P \setminus M_t^P$ is not empty. This implies that the language of $M_s^P \setminus M_t^P$ contains an ω-regular word. Thus, there exists an ω-regular word w_s in the language of $(s,[s])$ that is not in the language of $(t,[t])$. This implies that their exists a fair trace ρ'_{sP} that starts at $(s,[s])$ and $w_s = L^P(\rho'_{sP})$.

By Lemma 4.5.7 there exists a rational fair trace ρ_s that starts at s, such that for all $i \geq 0$, $tail(\rho'^i_{sP}) \leq_{rat} \rho_s^i$. Assume to the contrary that there exists a fair trace ρ_t from t such that $\rho_s \leq_{rat} \rho_t$. Consider the trace ρ_{tP} such that for all $i \geq 0$, $head(\rho_{tP}^i) = \rho_t^i$ and $tail(\rho_{tP}^i) = tail(\rho'^i_{sP})$. Clearly, ρ_{tP} is a fair trace from $(t,[s])$. Furthermore, $L^P(\rho'_{sP}) = L^P(\rho_{tp})$, thus $L^P(\rho_{tP}) = w_s$. This implies that w_s is in the language of M_t^P, a contradiction. □

4.6 Conclusion

The comparison shows that there is no notion of fair simulation, which has all desired advantages. However, it is clear that their relationship with temporal logics gives the exists and game simulations several advantages over the delay and direct simulations. On the other hand, the delay and direct simulations are better for minimization. Since this research is motivated by usefulness to model checking, relationships with logic are important. Thus, it is advantageous to refer to the delay and direct simulations as approximations of the game/exists simulations. These approximations enable some minimization with respect to the exists and game simulations. Out of the four notions, we consider the game simulation to be the best. This is due to its complexity and its applicability in modular verification.

Chapter 5

Modular reduction

In this chapter we develop a novel technique for modular reduction called the improved algorithm. The innovation of this algorithm is that it avoids constructing intermediate models, which consume unnecessary space. Given two reduced models M_1 and M_2, our technique directly constructs their reduced composition. The algorithm creates a copy of M_1 (M_2), and use it as an abstract environment of M_2 (M_1). First, it reduces M_1 with respect to the outputs of M_1 that are also inputs of M_2 ($O_1 \cap I_2$). We call the result M_1^r. Similarly it reduces M_2 to M_2^r. Next, M_2 and M_1^r are composed. The result is called M_2^e. M_2^e represents M_2 as if it is composed to M_1. Similarly, the improved algorithm constructs $M_1^e = M_1||M_2^r$. In the next step M_2^e is reduced with respect to the outputs of M_2 (O_2). The result is called M_2^d. Similarly M_1^d is constructed. Finally, the algorithm composes M_1^d and M_2^d, using a restricted composition. The resulting structure $M_d = M_1^d||M_2^d$ is smallest with respect to states and transitions, which is equivalent to $M_1||M_2$. In this chapter we use the bisimulation equivalence relation, we prefer bisimulation over simulation equivalence because its definition is simpler.

The rest of the chapter is organized as follows: In Section 5.1 we define an FSM, FSMs composition and bisimulation equivalence. Section 5.2 presents some properties of bisimulation and modularity. Section 5.3 presents the modular minimization algorithm for deterministic and nondeterministic FSMs. Section 5.4 describes the implementation and the experimental results.

5.1 Basic definitions

We model systems as finite-state machines (FSMs) in the form of *Moore machines* in which the states are labeled with outputs and the edges are labeled with inputs. Such machines are commonly used for modeling hardware designs.

Definition 5.1.1 *[Moo56] An FSM is a tuple $M =< S, S_0, I, O, L, R >$ where*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is a set of initial states.*
- *I is a finite set of input propositions.*
- *O is a finite set of output propositions.*
- *$I \cap O = \emptyset$.*
- *L is a labeling function that maps each state to the set of output propositions true in that state.*
- *$R \subseteq S \times 2^I \times S$ is the transition relation. We assume that for every $s \in S$ and $i \subseteq I$ there exists at least one state s' such that $(s, i, s') \in R$.*

An FSM is *deterministic* iff for every state s and $i \subseteq I$ there exists exactly one state s' such that $(s, i, s') \in R$, and $|S_0| = 1$.

Two FSMs are composed only if their outputs are disjoint. There is a transition from a pair of states in the composed FSM if and only if the output of each state match the input on the transition leaving the other state. This models the input-output connections between the two machines.

Definition 5.1.2 *Let $M_1 =< S_1, S_{01}, I_1, O_1, L_1, R_1 >$ and $M_2 =< S_2, S_{02}, I_2, O_2, L_2, R_2 >$ be two FSMs such that $O_1 \cap O_2 = \emptyset$. The composition $M = M_1 || M_2 =< S, S_0, I, O, L, R >$ is an FSM such that:*

- *$S = S_1 \times S_2$.*

- $S_0 = S_{01} \times S_{02}$.
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$.
- $O = O_1 \cup O_2$.
- $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.
- $((s_1, s_2), i, (s'_1, s'_2)) \in R$ *iff* $(s_1, (i \cup L_2(s_2)) \cap I_1, s'_1) \in R_1$ *and* $(s_2, (i \cup L_1(s_1)) \cap I_2, s'_2) \in R_2$.

Lemma 5.1.3 *Let M_1 and M_2 be deterministic FSMs, then the composition M of M_1 and M_2 is deterministic as well.*

Proof : Obviously, $|S_0| = 1$. Let (s_1, s_2) be a state in S and $i \subseteq I$ be an input. Let $i_1 = (i \cup L_2(s_2)) \cap I_1$ and $i_2 = (i \cup L_1(s_1)) \cap I_2$. Since M_1 is deterministic, there exists exactly one state s'_1 such that $(s_1, i_1, s'_1) \in R_1$. Similarly, there exists exactly one state s'_2 such that $(s_2, i_2, s'_2) \in R_2$. By the definition of composition, (s'_1, s'_2) is the only state such that $((s_1, s_2), i, (s'_1, s'_2)) \in R$. $\square$

We now define the basic notion of equivalence that we use in this work, namely, *bisimulation.*

Definition 5.1.4 *Let $M_1 =< S_1, S_{01}, I_1, O_1, L_1, R_1 >$ and $M_2 =< S_2, S_{02}, I_2, O_2, L_2, R_2 >$ be two FSMs such that $O_1 \cap O_2 \neq \emptyset$ and $I_1 = I_2$. We say that M_1 and M_2 are bisimulation equivalent with respect to $O' \subset O_1 \cap O_2$ iff there exists a relation $H \subseteq S_1 \times S_2$ (called bisimulation relation) such that:*

- *For every state $s_{01} \in S_{01}$ there exists a state $s_{02} \in S_{02}$ such that $(s_{01}, s_{02}) \in H$ and for every state $s_{02} \in S_{02}$ there exists a state $s_{01} \in S_{01}$ such that $(s_{01}, s_{02}) \in H$.*
- *For every pair (s_1, s_2) in H the following three conditions hold:*
  - $L_1(s_1) \cap O' = L_2(s_2) \cap O'$.
  - *For every $i \subseteq I_1$ (recall that $I_1 = I_2$), and for every state s'_1 such that $(s_1, i, s'_1) \in R_1$ there exists a state s'_2 such that $(s_2, i, s'_2) \in R_2$ and $(s'_1, s'_2) \in H$.*

- *For every $i \subseteq I_2$, and for every state s'_2 such that $(s_2, i, s'_2) \in R_2$ there exists a state s'_1 such that $(s_1, i, s'_1) \in R_1$ and $(s'_1, s'_2) \in H$.*

Proposition 5.1.5 *For every FSM M, let s be a state in M that is not reachable from any initial state. The result of removing s from M is bisimulation equivalent to M.*

As a consequence of Proposition 5.1.5, we refer only to FSMs where all the states are reachable from the initial states.

Bisimulation is an equivalence relation over FSMs. [Mil89] shows that for every two FSMs M_1 and M_2, there exists a maximal bisimulation relation, which contains every relation that satisfies the conditions of Definition 5.1.4. The maximal bisimulation relation $H \subseteq S \times S$ over the states of an FSM M is an equivalence relation over S. As such, it induces a partition of S to equivalence classes. These classes can be used to form the *quotient FSM* of M, which is the minimal FSM that is bisimulation equivalent to M.
We will denote by $[s]$ the equivalence class of a state s.

Definition 5.1.6 *Let $M =< S, S_0, I, O, L, R >$ be an FSM and let $H \subseteq S \times S$ be the maximal bisimulation relation with respect to $O' \subseteq O$ over M. The quotient FSM $M_Q =< S_Q, S_{0_Q}, I_Q, O_Q, L_Q, R_Q >$ of M with respect to H is defined as follows:*

- $S_Q = \{\alpha | \alpha$ *is an equivalence class in* $H\}$.
- $S_{0_Q} = \{[s_0] | s_0 \in S_0\}$.
- $I_Q = I$.
- $O_Q = O'$.
- *For $\alpha \in S_Q$, $L_Q(\alpha) = L(s) \cap O'$, for some (all) states $s \in \alpha$.*
- $R_Q = \{(\alpha, i, \alpha') |$ *there are states* $s \in \alpha, s' \in \alpha'$ *such that* $(s, i, s') \in R\}$.

Definition 5.1.7 *An FSM M is minimized iff it is isomorphic to its quotient FSM.*

5.2 Properties of modularity and reduction

The improved algorithm uses both modularity and bisimulation-based reduction. In the following we present some properties of the bisimulation relation, bisimulation reduction, and the relationships between bisimulation and modularity. The proofs for these claims are given in Section 5.5.

Lemma 5.2.1 *Let M be an FSM, and let M_Q be the quotient FSM of M. Let (α, i, α') be an element in R_Q. Then for every state s in α there exists a state s' in α' such that $(s, i, s') \in R$.*

Proposition 5.2.2 *If M is deterministic then M_Q is deterministic.*

Lemma 5.2.3 *M is minimized iff the maximal bisimulation relation over $M \times M$ contains exactly the identity pairs.*

Lemma 5.2.4 *Let M be an FSM and M_Q be the quotient FSM of M with respect to O', then M and M_Q are bisimulation equivalent with respect to O'.*

Lemma 5.2.5 *Let M be an FSM and M_Q be the quotient FSM of M with respect to O'. Then M_Q is the smallest (in number of states and transitions) FSM which is bisimulation equivalent to M with respect to O'.*

Proposition 5.2.6 *Let M_1 and M_2 be FSMs and let $H \subseteq S_1 \times S_2$ be bisimulation relation over M_1 and M_2 with respect to $O \subseteq O_1 \cap O_2$. Then H is a bisimulation relation with respect to every $O' \subseteq O$.*

Lemma 5.2.7 *Let M_1 and M_2 be minimized FSMs. If $O_1 \cap I_2 = \emptyset$ and $O_2 \cap I_1 = \emptyset$, then $M = M_1 || M_2$ is minimized.*

5.3 The improved algorithm

In this section we present the improved algorithm. Like the naive algorithm, the improved algorithm receives a design, given as a set of n components. The improved algorithm works in iterations. In each iteration two minimized components M_1 and M_2 are selected and a new minimized component, which is equivalent to $M_1 \| M_2$, is constructed. The algorithm terminates when an iteration results in a single component. In this case, the final component is the smallest in terms of states and transitions which is equivalent to the composition of the n original components.

In this section we focus on a single iteration of the improved algorithm. Unlike the naive algorithm, where the two components are first composed and then the result is minimized, the improved algorithm constructs a minimized FSM which is equivalent to $M_1 \| M_2$ without constructing the composition. Thus, the improved algorithm requires less time and space.

By Lemma 5.2.7, if M is the result of a composition of two different FSMs, which do not interact with each other, then, M can be minimized by minimizing M_1 and M_2 separately. This however, does not hold in the general case, namely, given two minimized components M_1 and M_2, their composition $M_1 \| M_2$ is not necessarily minimized. This is demonstrated in Figure 5.1.

Figure 5.1 shows two FSMs M_1 and M_2 for which $O_1 \cap I_2 \neq \emptyset$. M_1 and M_2 are minimized but their composition M is not. Figure 5.1 also contains M_Q which is the result of minimizing M. The FSMs in Figure 5.1 are Moore machines and we use the following convention in their description. The labels in the states represent the outputs of the Moore machines. The inputs are represented by a boolean formula on the edges. For states $s, s' \in S$ and $i \subseteq I$, (s, i, s') is an element in R iff i satisfies the formula on the edge from s to s'.

The observation demonstrated in Figure 5.1 implies that a more sophisticated algorithm is needed for components that interact between themselves. We will present two versions of the improved algorithm, one for deterministic FSMs and another for nondeterministic FSMs. While the former is less general, it has a better complexity. Since often hardware designs are modeled by a deterministic FSM, it worth developing a special algorithm for deterministic designs.

5.3.1 Deterministic FSMs

We now describe a single iteration of the improved algorithm. The version for deterministic FSMs and the version for nondeterministic FSMs are differ only in the last stage of single the iterations. We

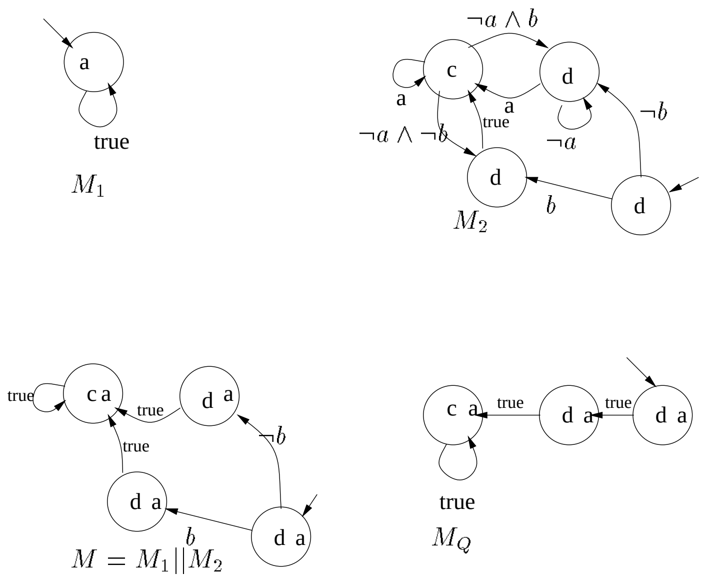

Figure 5.1: The composition of two minimized FSMs is not always minimized.

first present the version for deterministic systems, which is simpler, and then we present the change in the last stage for nondeterministic FSMs. In each iteration, the algorithm is given two minimized FSMs M_1 and M_2 such that $O_1 \cap O_2 = \emptyset$. We use the notation $M = M_1||M_2$, $O'_1 = O_1 \cap I_2$, and $O'_2 = O_2 \cap I_1$. The algorithm performs the following stages:

1. Reduce M_1 with respect to O'_1, resulting in M_1^r.
2. Reduce M_2 with respect to O'_2, resulting in M_2^r.
3. Compose $M_1^e = M_1||M_2^r$.
4. Compose $M_2^e = M_1^r||M_2$.
5. Reduce M_1^e with respect to O_1, resulting in M_1^d.
6. Reduce M_2^e with respect to O_2. resulting in M_2^d.
7. Compose $M_d = M_1^d||M_2^d$.

FSM	Input	Output
M_1	I_1	O_1
M_2	I_2	O_2
M_1^r	I_1	O_1'
M_2^r	I_2	O_2'
M	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2$
M_1^e	$(I_1 \setminus O_2') \cup (I_2 \setminus O_1) = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2'$
M_2^e	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1') = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_2 \cup O_1'$
M_1^d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	O_1
M_2^d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	O_2
M_d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2$

Table 5.1: The inputs and outputs of the intermediate FSMs in the improved algorithm.

Table 5.1 presents the inputs and outputs of the FSMs constructed by the improved algorithm.

An example for the *improved algorithm* is presented in Figure 5.2. The intuition behind the improved algorithm is as follows. When two FSMs are composed, each restricts the behavior of the other by providing a real environment, rather than an open one. In the restricted environment, states that behave differently in the open environment are now indistinguishable and can be collapsed into the same equivalence class.

Our goal is to minimize M_1 and M_2 separately, while taking into account the environment each runs in. While minimizing M_2 it is sufficient to consider only the part of M_1 which influences M_2. M_1^r is exactly that part. Therefore, states in M_2 that become indistinguishable in $M = M_1 \| M_2$ are also indistinguishable in $M_2^e = M_1^r \| M_2$. These states are collapsed, resulting in M_2^d. Similarly, in M_1^e states of M_1 that are indistinguishable in M are collapsed (resulting in M_1^d). When M_1^d and M_2^d are finally composed, M_d is the result of a composition of two minimized FSMs which do not interact with each other and therefore M_d is minimized.

The skeleton of the correctness proof for the algorithm is given in the lemma below. In the rest of the section we prove each of the claims, thus prove the correctness of our algorithm.

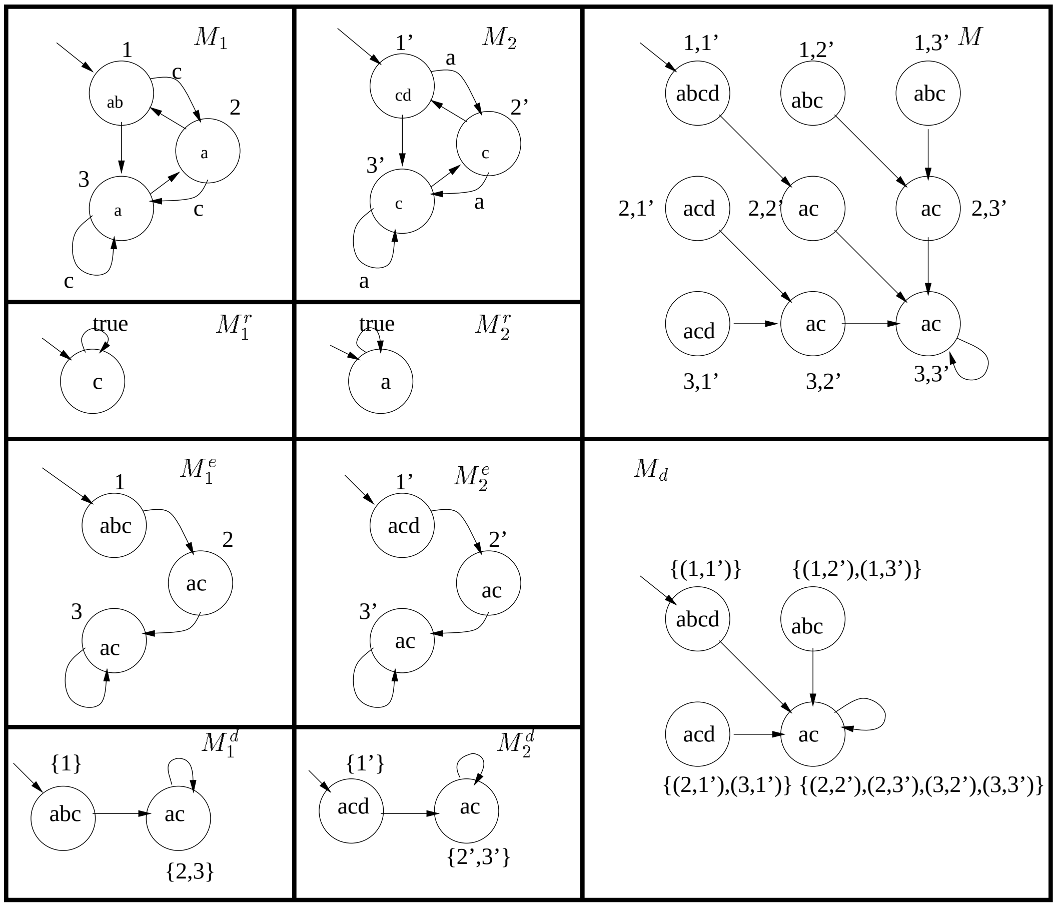

Figure 5.2: An example of the deterministic version of the improved algorithm: M_1 has input set $I_1 = \{c\}$ and output set $O_1 = \{a, b\}$. M_2 has input set $I_2 = \{a\}$ and output set $O_2 = \{c, d\}$. Note that, even though M_1 and M_2 are minimized, M is not. M_d is the quotient model of M. It can also be obtained by composing M_1^d and M_2^d. The states of M_1^d, M_2^d and M_d are given as the sets of states in the equivalence classes the states represent

Lemma 5.3.1

- *M_1^e and M are bisimulation equivalent with respect to $O_1 \cup O_2'$.*
- *M_2^e and M are bisimulation equivalent with respect to $O_2 \cup O_1'$.*

- *M_1^d and M are bisimulation equivalent with respect to O_1.*
- *M_2^d and M are bisimulation equivalent with respect to O_2.*
- *M_d and M are bisimulation equivalent with respect to $O_1 \cup O_2$*
- *M_d is minimized with respect to $O_1 \cup O_2$.*

Lemma 5.3.2 *M_1^e and M are bisimulation equivalent with respect to $O_1 \cup O_2'$.*

Proof: Let $H_1^e \subseteq S \times S_1^e$ be $H_1^e = \{((s_1, s_2), (s_1, s_2^r)) | s_2^r$ is the equivalence class of $s_2\}$. We prove that H_1^e is a bisimulation relation.

- For every $(s_{10}, s_{20}) \in S_0$, we have $((s_{10}, s_{20}), (s_{10}, [s_{20}])) \in H_1^e$. Similarly, For every $(s_{10}, \alpha_0) \in S_{10}^r$, let s_{20} be the initial state in α, then $((s_{10}, s_{20}), (s_{10}, [s_{20}])) \in H_1^e$.

Let $((s_1, s_2), (s_1, s_2^r)) \in H_1^e$:

- Since the labeling of an equivalence class is equal to the labeling of the states it contains, $L_2(s_2) \cap O_2' = L_2^r(s_2^r)$. The definition of composition therefore implies, $L((s_1, s_2)) \cap (O_1 \cup O_2') = L_1^e((s_1, s_2^r))$.
- Let $((s_1, s_2), i, (s_1', s_2'))$ be an element in R. This implies that for $i_1 = (i \cup L_2(s_2)) \cap I_1$, $(s_1, i_1, s_1') \in R_1$ and for $i_2 = (i \cup L_1(s_1)) \cap I_2$, $(s_2, i_2, s_2') \in R_2$. Let ${s_2}'^r$ be the equivalence class of s_2', then $(s_2^r, i_2, {s_2}'^r) \in R_2^r$. Since $L_2(s_2) \cap I_1 = L_2(s_2) \cap O_2' = L_2^r(s_2^r)$, $i_1 = (i \cup L_2^r(s_2^r)) \cap I_1$. The definition of composition implies $((s_1, s_2^r), i, ({s_1}', {s_2}'^r)) \in R_1^e$. By the definition of H_1^e, $((s_1', s_2'), ({s_1}', {s_2}'^r)) \in H_1^e$.
- Let $((s_1, s_2^r), i, ({s_1}', {s_2}'^r))$ be an element in R_1^e. This implies that for $i_1 = (i \cup L_2^r(s_2^r)) \cap I_1$, $(s_1, i_1, s_1') \in R_1$ and for $i_2 = (i \cup L_1(s_1)) \cap I_2$, $(s_2^r, i_2, {s_2}'^r) \in R_2^r$. By Lemma 5.2.1, there exists a state s_2' such that $(s_2, i_2, s_2') \in R_2$ and ${s_2}'^r$ is the equivalence class of s_2'. Since $L_2(s_2) \cap I_1 = L_2(s_2) \cap O_2' = L_2^r(s_2^r)$, $i_1 = (i \cup L_2(s_2)) \cap I_1$. By the definition of composition, $((s_1, s_2), i, (s_1', s_2')) \in R$ and by the definition of H_1^e, $((s_1', s_2'), ({s_1}'^r, {s_2}'^r)) \in H_1^e$. $\Box$

Lemma 5.3.3 *M_2^e and M are bisimulation equivalent with respect to $O_1' \cup O_2$.*

The proof is similar to the proof of Lemma 5.3.2.

Lemma 5.3.4 *M_1^d and M are bisimulation equivalent with respect to O_1.*

Proof : Proposition 5.2.6 together with Lemma 5.3.2 implies that M_1^e and M are bisimulation equivalent with respect to O_1. Lemma 5.2.4 implies that M_1^e and M_1^d are bisimulation equivalent with respect to O_1. Since bisimulation equivalence is transitive, then M and M_1^d are bisimulation equivalent with respect to O_1. $\Box$

Lemma 5.3.5 *M_2^d and M are bisimulation equivalent with respect to O_2.*

The proof is similar to the proof of Lemma 5.3.4.

Lemma 5.3.6 *If M_1 and M_2 are deterministic, then M_d and M are bisimulation equivalent with respect to $O_1 \cup O_2$.*

Note that both M_1^d and M_2^d have the same input (I), and that $I \cap O_1 = I \cap O_2 = \emptyset$.

Proof : Let $H_1^d \subseteq S \times S_1^d$ and $H_d^2 \subseteq S \times S_2^d$ be bisimulation relations over $M \times M_1^d$ and $M \times M_2^d$ respectively. Let $H_d \subseteq S \times S_d$, be the following relation: $H_d = \{((s_1, s_2), (s_1^d, s_2^d)) | ((s_1, s_2), s_1^d) \in H_d^1$ and $((s_1, s_2), s_2^d) \in H_d^2\}$. We prove that H_d is a bisimulation relation.

- $((s_{01}, s_{02}), s_{01}^d) \in H_d^1$ and $((s_{01}, s_{02}), s_{02}^d) \in H_d^2$ implies that $((s_{01}, s_{02}), (s_{01}^d, s_{02}^d)) \in H_d$.

Let $((s_1, s_2), (s_1^d, s_2^d))$ be a pair in H_d.

- $((s_1, s_2), s_1^d) \in H_d^1$ implies $L((s_1, s_2)) \cap O_1 = L_1^d(s_1^d)$. $((s_1, s_2), s_2^d) \in H_d^2$ implies $L((s_1, s_2)) \cap O_2 = L_2^d(s_2^d)$. Thus, $L((s_1, s_2)) = L_d((s_1^d, s_2^d))$.
- Let $((s_1, s_2), i, (s_1', s_2'))$ be an element in R. Since $((s_1, s_2), s_1^d) \in H_d^1$, there exists a state ${s_1}'^d$ such that $(s_1^d, i, {s_1}'^d) \in R_1^d$ and $((s_1', s_2'), {s_1}'^d) \in H_d^1$. Since $((s_1, s_2), s_2^d) \in H_d^2$, there exists a state ${s_2}'^d$ such that $(s_2^d, i, {s_2}'^d) \in R_2^d$ and $((s_1', s_2'), {s_2}'^d) \in H_d^2$. The definition of composition implies that $((s_1^d, s_2^d), i, ({s_1}'^d, {s_2}'^d)) \in R_d$ and by the definition of H_d, $((s_1', s_2'), ({s_1}'^d, {s_2}'^d)) \in H_d$.

- Let $((s_1^d, s_2^d), i, ({s_1}'^d, {s_2}'^d))$ be an element in R_d. Then $(s_1^d, i, {s_1}'^d) \in R_1^d$ and $(s_2^d, i, {s_2}'^d) \in R_2^d$. Since $((s_1, s_2), s_1^d) \in H_d^1$ then there exists a state (s_1', s_2') such that $((s_1, s_2), i, (s_1', s_2')) \in R$ and $((s_1', s_2'), {s_1}'^d) \in H_d^1$. Since $((s_1, s_2), s_2^d) \in H_d^2$ then there exists a state (s_1'', s_2'') such that $((s_1, s_2), i, (s_1'', s_2'')) \in R$ and $((s_1'', s_2''), {s_2}'^d) \in H_d^2$. Since M is deterministic, $(s_1', s_2') = (s_1'', s_2'')$. By the definition of H_d, $((s_1', s_2'), ({s_1}'^d, {s_2}'^d)) \in H_d$. $\square$

M_1^d and M_2^d are minimized with respect to O_1 and O_2 respectively. Furthermore,$I \cap O_1 = I \cap O_2 = \emptyset$, thus Lemma 5.2.7 induces the following corollary.

Corollary 5.3.7 *M_d is minimized with respect to $O_1 \cup O_2$.*

5.3.2 Time and space complexity

In this section we compare between the complexity of the naive algorithm and the complexity of the improved algorithm.

The algorithms include two basic operations:

1. Composing two FSMs $M'' = M \| M'$. The most costly part in time and space of this operation is the computation of the transition relation R''. This can be done in time and space complexity of $O(|R''|)$.

2. Minimizing an FSM M into its quotient FSM M_Q. The algorithms have the same complexity as the one in [Hop71, PT87]. Their space complexity is $O(|R|)$ and the time complexity is $O(|R| \cdot log(|S|))$.

Thus, the minimization is the dominant part of the algorithm. In the naive algorithm there is only one minimization of $M = M_1 \| M_2$. In the improved algorithm however, there are 4 minimizations: The minimization of M_1 that results in M_1^r, the minimization of M_2 that results in M_2^r, the minimization of M_1^e that results in M_1^d and the minimization M_2^e that results in M_2^d.

Since the complexity of a minimization depends on the size of the minimized FSM, we need to compare the sizes of M_1, M_2, M_1^e, M_2^e, versus the size of M. We assume that the size of M_1 is equal to the size of M_2.

The differences between the sizes of M_1 and M_2 and the size of M depend on the interactions between M_1 and M_2. The interaction between M_1 and M_2 is measured by the number of inputs of one that are outputs of the other. The size of the state spaces of M_1 and M_2, is square root of the size of the state space of M. However, the size of the transition relation depends on the interactions. When the interaction between M_1 and M_2 is high, many inputs of M_1 and M_2 are connected to outputs of M_2 and M_1 respectively. These inputs are not part of the inputs of M[1]. In this case, every component in M_2 is an input of M_1 and vice versa thus, $|S_1| \cdot |2^{I_1}| \approx |S_2| \cdot |2^{I_2}| \approx |S| \cdot |2^I|$. Since $|R_1| \approx |S_1| \cdot |2^{I_1}|$ and $|R_2| \approx |S_2| \cdot |2^{I_2}|$, $|R_1| \approx |R_2| \approx |R|$, and $|M_1| \approx |M_2| \approx |M|$.

Next we compare the sizes of M_1^e and M_2^e with the size of M. Note that $M = M_1 \| M_2$, $M_1^e = M_1 \| M_2^r$ and $M_2^e = M_1^r \| M_2$. This implies that the difference between the sizes of M and M_1^e depends on the difference between M_2 and M_2^r. Similarly, the difference between the sizes of M and M_2^e depends on the difference between M_1 and M_1^r. When there is no redundancy, $|M_1| = |M_1^r|$ and $|M_2| = |M_2^r|$. In this case, $|M_1^e| = |M_2^e| = |M|$.

The worst-case scenario is when the interaction between M_1 and M_2 is high and there is no redundancy in M_1 and M_2, $|M_1| = |M_1^r|$ and $|M_2| = |M_2^r|$. In this case the improved algorithm performs four minimizations, each requires the same time as the single minimization of the naive algorithm. Since, we need to keep at most three different models at the same time, the space requirement of the improved algorithm is three time larger than that of the naive algorithm.

In the best scenario however, $|M_1| = |M_2| = |M_1^e| = |M_2^e| = \sqrt{|M|}$. In this scenario, instead of time complexity of $|R| \cdot log(|S|)$ in the naive algorithm, the improved algorithm has time complexity, $4 \cdot \sqrt{|R|} \cdot log(\sqrt{|S|})$, which is significantly better.

5.3.3 Nondeterministic FSMs

In this section we extend the modular method to nondeterministic FSMs. When we consider nondeterministic FSMs, Lemma 5.3.6 does not hold. The result M_d of composing M_1^d and M_2^d might be inequiva-

[1]Recall that $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$

lent to M due to "illegal states" in M_d which are not equivalent to any state in M.

In order to understand this inequality, we inspect the role of the states of M_1 in M (the role of the states of M_2 is similar). When we look at M as a composition of M_1 and M_2, every state s_1 has two functionalities, the first is to determine the outputs and next state of M_1 and the second to determine the inputs of M_2.

In M_d these two functionalities are fulfilled by two states of S_1. Let $([(s_1,[s_2])],[([t_1],t_2)])$ be a state in M_d. Then s_1 fulfills the first functionality of determining the outputs and the next states of M_1, and t_1 fulfills the second functionality of determining the inputs of M_2. A state $([(s_1,[s_2])],[([t_1],t_2)])$ of M_d might be illegal when $s_1 \notin [t_1]$. In this case, the combination of next state in M_1 and input of M_2 does not occur in any state of S_1.

The problem of illegal states is demonstrated in Figure 5.3. In this figure, all the states of M_1 and M_2 are initial states, which makes M_1 and M_2 nondeterministic. M_1^r and M_2^r cannot be further reduced, the same holds for M_1^e and M_2^e. Since the result M_d of composing M_1^d and M_2^d is minimized and contains 16 initial states, it cannot be bisimulation equivalent to $M = M_1||M_2$. The error in the algorithm is due to illegal states like $((0,2),(1,2))$ in M_d which is related to both s_0 and s_1 in M_1 and is not equivalent to any state in M.

We now present the version for nondeterministic systems of the improve algorithm. This algorithm, restricts the states of M_d to legal states only. As before, the minimized FSM is constructed without constructing the composition $M_1||M_2$ itself. First we define two functions.

Definition 5.3.8 *The function $f_1 : S_1 \times S_2 \to S_1^d$ is defined as follows: $f_1(s_1,s_2) = [(s_1,[s_2])]$.*

Definition 5.3.9 *The function $f_2 : S_1 \times S_2 \to S_2^d$ is defined as follows: $f_2(s_1,s_2) = [([s_1],s_2)]$.*

Next, we define a new FSM M_d' which is similar to M_d except that the set of states is restricted.
$S_d' = \{(s_1^d,s_2^d) | \exists s_1,s_2,\ s_1^d = f_1(s_1,s_2) \wedge s_2^d = f_2(s_1,s_2)\}$. The definitions for the other components of M_d' are straightforward. $S_0^{d'} = {S_0}^d \cap S_d'$, the inputs, outputs, and labeling function remain the same, and $R_d' =$

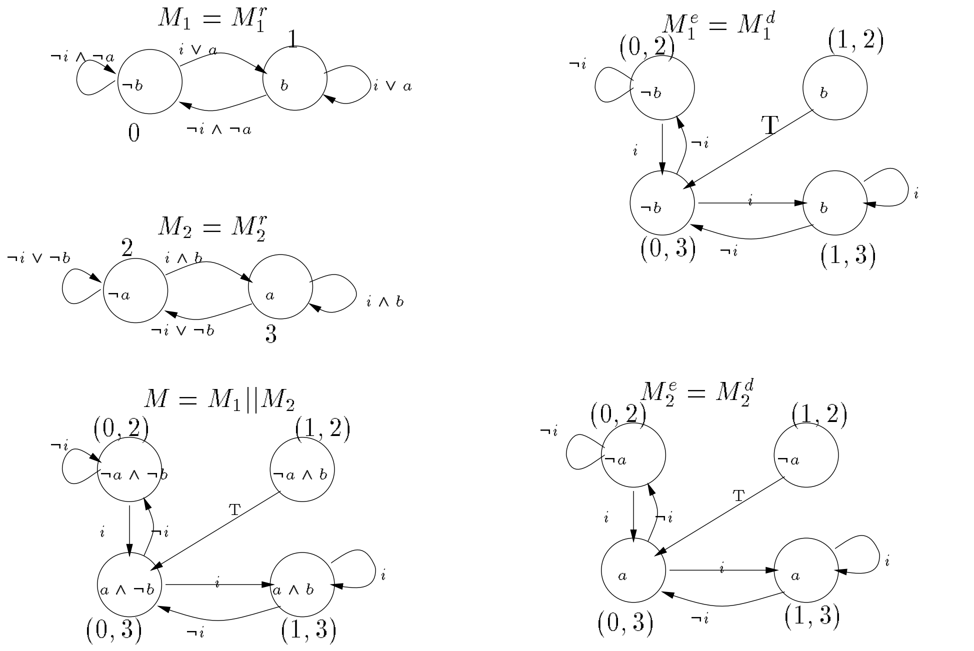

Figure 5.3: An example of inequivalent result of the version for deterministic systems of the improved algorithm, where M_1 and M_2 are not deterministic

$R_d \cap (S'_d \times S'_d)$. We now prove that M'_d is bisimulation equivalent to M.

Lemma 5.3.10 *M'_d and M are bisimulation equivalent with respect to*
$O_1 \cup O_2$.

Proof : Let $H \subseteq S \times S'_d$ be defined as follows: $H = \{((s_1, s_2), (s_1^d, s_2^d)) | \, s_1^d = f_1(s_1, s_2) \wedge s_2^d = f_2(s_1, s_2)\}$. We prove that H is a bisimulation relation.

- The definition of S'_{d0} implies that for every state $(s_{10}, s_{20}) \in S_0$ there exists a state $(s_{10}^d, s_{20}^d) = ([(s_{10}, [s_{20}])], [([s_{10}], s_{20})]) \in S'_{d0}$ such that $((s_{10}, s_{20}), (s_{10}^d, s_{20}^d)) \in H$. For the other direction, assume that $(s_{10}^d, s_{20}^d) = ([(s_{10}, [s_{20}])], [([s_{10}], s_{20})])$ is a state in S'_{d0}. Then, the state (s_{10}, s_{20}) is in S_0 and $((s_{10}, s_{20}), (s_{10}^d, s_{20}^d)) \in H$.

- Let $((s_1, s_2), (s_1^d, s_2^d))$ be an element in H. Since $L_1^d([(s_1, [s_2])]) = L_1^e(s_1, [s_2]) \cap O_1 = L_1(s_1)$ and $L_2^d([([s_1], s_2)]) = L_2^e([s_1], s_2) \cap O_2 = L_2(s_2)$, $L((s_1, s_2)) = L_d(([(s_1, [s_2])], [([s_1], s_2)]))$.

- Let $((s_1, s_2), (s_1^d, s_2^d))$ be in H and let i be an element in I. Let $i_1 = (i \cup L_2(s_2)) \cap I_1 = (i \cup L_2^r([s_2])) \cap I_1$ and
  - $i_2 = (i \cup L_1(s_1)) \cap I_2 = (i \cup L_1^r([s_1])) \cap I_2$. Then, $((s_1, s_2), i, (s_1', s_2')) \in R$ iff
  - $(s_1, i_1, s_1') \in R_1$ and $(s_2, i_2, s_2') \in R_2$ iff (Definition 5.1.6 and Lemma 5.2.1)
  - $([s_1], i_1, [s_1']) \in R_1^r$ and $([s_2], i_2, [s_2']) \in R_2^r$. $(s_1, i_1, s_1') \in R_1$ and $([s_2], i_2, [s_2']) \in R_2^r$ iff $((s_1, [s_2]), i, (s_1', [s_2'])) \in R_1^e$.
  - Similarly, $([s_1], i_1, [s_1']) \in R_1^r$ and $(s_2, i_2, s_2') \in R_2$ iff $(([s_1], s_2), i, ([s_1'], s_2')) \in R_2^e$.
  - Therefore, $((s_1, [s_2]), i, (s_1', [s_2'])) \in R_1^e$ and $(([s_1], s_2), i, ([s_1'], s_2')) \in R_2^e$ iff
  - $(s_1^d, i, s_1^{d'}) = ([(s_1, [s_2])], i, [(s_1', [s_2'])])$ is in R_1^d and $(s_2^d, i, s_2^{d'}) = ([([s_1], s_2)], i, [([s_1'], s_2')])$ is in R_2^d iff
  - $((s_1^d, s_2^d), i, (s_1^{d'}, s_2^{d'})) \in R_d$.

□

Next, we prove that M_d' is minimized. First, we show that the maximal bisimulation over M_d' includes a bisimulation over M_2^d.

Lemma 5.3.11 *Let H_d' be the maximal bisimulation relation over M_d'. We define a relation H_1^d over $S_1^d \times S_1^d$ as follows: $([(s_1, [s_2])], [(t_1, [t_2])]) \in H_1^d$ iff $(([(s_1, [s_2])], [([s_1], s_2)]), ([(t_1, [t_2])], [([t_1], t_2)])) \in H_d'$. Then H_1^d is a bisimulation relation.*

Proof :

- Since H_d' contains all identity pairs, H_1^d contains all identity pairs as well. This implies that for every initial state, the pair of the initial state with itself is an element in H_1^d.

Let $([(s_1, [s_2])], [(t_1, [t_2])])$ be an element in H_1^d:

- $L_d(([(s_1, [s_2])], [([s_1], s_2)])) = L_d(([(t_1, [t_2])], [([t_1], t_2)]))$ implies $L_d^1([(s_1, [s_2])]) = L_d^1([(t_1, [t_2])])$.
- Let $([(s_1, [s_2])], i, [(s_1', [s_2'])])$ be an element in R_1^d. Let $i_1 = (i \cup L_2(s_2)) \cap I_1 = (i \cup L_2^r([s_2])) \cap I_1$ and $i_2 = (i \cup L_1(s_1)) \cap I_2 = (i \cup L_1^r([s_1])) \cap I_2$.

1. By Lemma 5.2.1, $((s_1, [s_2]), i, (s'_1, [s'_2])) \in R^e_1$.
2. Thus, $(s_1, i_1, s'_1) \in R_1$ and $([s_2], i_2, [s'_2]) \in R^r_2$.
3. By Definition 5.1.6 and Lemma 5.2.1 $([s_1], i_1, [s'_1]) \in R^r_1)$ and $(s_2, i_2, s'_2) \in R_2$.
4. This implies that, $([([s_1], s_2)], i, [([s'_1], s'_2)]) \in R^d_2$.
5. Thus, $(([(s_1, [s_2])], [([s_1], s_2)]), i, ([(s'_1, [s'_2])], [([s'_1], s'_2)])) \in R'_d$.
6. Since H'_d is a bisimulation, there exists a state $([(t'_1, [t'_2])], [([t'_1], t'_2)])$ such that $(([(t_1, [t_2])], [([t_1], t_2)]), i, ([(t'_1, [t'_2])], [([t'_1], t'_2)])) \in R'_d$ and $(([(s'_1, [s'_2])], [([s'_1], s'_2)]), ([(t'_1, [t'_2])], [([t'_1], t'_2)])) \in H'_d$.
7. This implies that $([(t_1, [t_2])], i, [(t'_1, [t'_2])]) \in R^d_1$ and $([(s'_1, [s'_2])], [(t'_1, [t'_2])]) \in H^d_1$.

- Similarly, we can prove that for every state $[(t'_1, [t'_2])]$ such that $([(t_1, [t_2])], i, [(t'_1, [t'_2])]) \in R^d_1$ there exists a state $[(s'_1, [s'_2])]$ such that $([(s_1, [s_2])], i, [(s'_1, [s'_2])]) \in R^d_1$ and $([(s'_1, [s'_2])], [(t'_1, [t'_2])]) \in H^d_1$.

□

Lemma 5.3.12 *Let H'_d be the maximal bisimulation relation over M'_d. We define a relation H^d_2 over $S^d_2 \times S^d_2$ as follows: $([([s_1], s_2)], [([t_1], t_2)]) \in H^d_2$ iff $(([(s_1, [s_2])], [([s_1], s_2)]), ([(t_1, [t_2])], [([t_1], t_2)])) \in H'_d$. Then H^d_2 is a bisimulation relation.*

The proof of Lemma 5.3.12 is similar to the proof of Lemma 5.3.11.

Lemma 5.3.13 *M'_d is minimized.*

Proof Let H_d be the maximal bisimulation over $M'_d \times M'_d$. Assume to the contrary that the lemma does not hold. Then by Lemma 5.2.3, there are two different states $(s^d_1, s^d_2), (t^d_1, t^d_2)$ such that $((s^d_1, s^d_2), (t^d_1, t^d_2)) \in H_d$. Since $(s^d_1, s^d_2) \neq (t^d_1, t^d_2)$, either $s^d_1 \neq t^d_1$ or $s^d_2 \neq t^d_2$. Assume w.l.o.g. that $s^d_1 \neq t^d_1$. Let H^d_1 be the relation defined in Lemma 5.3.11. By Lemma 5.3.11 H^d_1 is a bisimulation. By the definition of H^d_1, $(s^d_1, t^d_1) \in H^d_1$. By Lemma 5.2.3, M^d_1 is not minimized, a contradiction. □

5.3.4 Additional complexity

The additional complexity is due to the computation of S'_d which forces us to refer to the whole state space of M. Nevertheless, since we only compute the state space and do not use it in the reduction method, the version for nondeterministic systems of the improved algorithm is still better than the naive algorithm. Computing f_1 and f_2, can be done during the construction of M_1^r and M_2^r and the construction of M_1^d and M_2^d without any additional time complexity. However, since the function operates on the states of $M_1||M_2$, the space complexity is $|S| = |S_1| \cdot |S_2|$. In a worst-case scenario, the complexity of the nondeterministic improved algorithm is identical to the complexity of the of the deterministic improved algorithm. However, when $M_1^r \ll M_1$ and $M_2^r \ll M_2$, this complexity is worse than the complexity for the deterministic version.

5.4 An implementation of the improved algorithm

In this section we describe an implementation of the improved algorithm. Our goal is to compare between the improved algorithm, the naive algorithm and the ordinary algorithm. The ordinary algorithm minimizes a given FSM directly and does not use modularity. The implementation has been developed within the sequential equivalence verification CAD group of Intel design technologies in Haifa. The designs, which are tested in the equivalence department, have the following properties:

1. $S_0 = S$, i.e., every state in the model is an initial state.
2. The transition relation is a function, meaning that for every state s and input i there exists exactly one state t, such that (s, i, t) is a transition in R.

Note that the first property makes these designs nondeterministic. These properties guide us to use the version for nondeterministic systems of the improved algorithm. However, we represent the transition relation as a function, which can be represented more concisely than regular relation.

```
typedef struct fsm {
    VarList     inputs;
    BddFunction outputs;
    BddFunction latches;
    BDD         domain;
    BddFunction equivFunc;
}FSM;
```

Figure 5.4: The data structure that models FSMs

A general description of the implementation is given in Section 5.4.1. The improved algorithm uses the ordinary algorithm as a subroutine. The same ordinary algorithm is used for comparison with the improved algorithm. Since we deal with FSMs that have a transition relation that is a function, we use an algorithm that is similar to the algorithm presented in [Hop71]. The experimental results are presented in Section 5.4.2.

5.4.1 The implementation framework

The minimization algorithms (either the improved algorithm, the naive algorithm or the ordinary algorithm) receive an FSM from an Intel program, which compiles the RTL description of the design into an FSM. The given FSM contains three lists: A list of inputs, a list of latches, and a list of outputs. The list of *inputs* contains BDD variables only. The list of *latches* which encodes the state space, is made of pairs, where, every pair contains a BDD variable and a BDD which represents the next state function. The list of *outputs*, which encodes the labeling function is made of pairs where every pair contains a BDD variable and a BDD, which represents the output function.

We modeled an FSM by the *FSM* data structure shown in Figure 5.4. In addition to the inputs, latches and outputs fields, the FSM data structure has two more fields: The first is the *domain* field, which is a BDD over the latches which represents the set of states. The second, is the *equivFunc* field. When a minimization of an FSM is performed, a set of equivalence classes is constructed. These classes are the states of the resulting FSM. The field equivFunc of the resulting FSM contains a function that relates the states of the original FSM to their equivalence classes.

The information about the modular structure of the tested designs was lost during the development stage. Thus, instead of a set of components, the improved algorithm receives one FSM. In order to perform the minimization, it first partitions the FSM and then performs the improved algorithm. A basic description of the implementation of the improved algorithm is presented in Figure 5.5.

The algorithm receives an FSM om and partitions it into two FSMs $m1$ and $m2$. Then it uses the improved algorithm to construct a minimized model md which is equivalent to om. The algorithm partitions the model by partitioning the set of latches and the set of outputs, (it is possible for $m1$ and $m2$ to share inputs). The goal of the partition is that the interaction between the models will be minimal. Since finding such a partition is hard, the algorithm uses a heuristic to find a partition with low interaction.

The improved algorithm uses the subroutine *reduction*, which performs the ordinary algorithm. The algorithm is an adaptation of the algorithm given in [Hop71] for constructing the quotient automaton for a given regular deterministic automaton. The algorithm is adapted for FSMs for which the transition relation is a function. Given an FSM, it constructs its quotient FSM. The main difference between the algorithm in [Hop71] and the ordinary algorithm is in the initial partitioning. While for automata the initial partition forms two sets (accepting and rejecting), the states of the FSM are initially partitioned into $2^{|AP|}$ sets, one for each state labeling.

Both minimization algorithms (improve algorithm and ordinary algorithm) minimize the FSM with respect to its outputs. Thus before they minimize M_1 into M_1^r (M_2 into M_2^r), they need to remove the outputs in $O_1 \setminus I_2$ ($O_2 \setminus I_1$). In order to remove these external outputs the algorithms use the *rmExternalOutputs* subroutine.

In order to construct the set rd of "legal states" of the form $([(s_1, [s_2])], [([s_1], s_2)])$, the algorithm constructs two functions $f1d : S \to S_1^d$ and $f2d : S \to S_1^d$. In order to construct $f1d$, the algorithm composes the functions $M1d.equivFunc : S_1^e \to S_1^d$ and the function $m2r.equivFunc : S_2 \to S_2^r$. Since $S_1^e = S_1 \times S_2^r$, the resulting function relates the states of $S_1 \times S_2$ to the states of S_1^d. The function $f2d$ is constructed in a similar way. Then the algorithm calculates $rd = fd(om.domain)$, where $fd : S \rightarrow S_d$ is defined as follows: $fd(s) = (f1d(s), f2d(s))$.

The sets, functions and relations are represented by BDDs. We use Intels BDD package for the implementation.

5.4.2 Experimental results

We compared between the ordinary algorithm, the naive algorithm, and the improved algorithm. While we tested the improved algorithm, we found out that the minimization of M_1^r and M_2^r does not improve the performance of the algorithm. Thus we tested the algorithm also without these minimizations. In this case M_1^e (M_2^e) are simply M with only part of the outputs. We tested the improved algorithm without the construction of M_1^r and M_2^r when the design is partitioned only once (appears in the tables as improved2), and when the design is recursively partitioned until it has one output only (appears in the tables as improved3).

The results are presented in the following tables. In Table 5.2 we present general properties of the tested designs. Table 5.3 compares the minimization times of the minimization algorithms. Table 5.4 compares the space requirements of the minimization algorithms. The algorithms were tested on a machine with two CPUs 550 MHZ each and 2GB memory.

The experimental results imply that in most designs, all versions of the improved algorithm have better performances than the ordinary and naive algorithms both in time and space. The improved algorithm which does not reduce M_1^r and M_2^r and partitions the outputs recursively, has the best time performance and the improved algorithm which does not reduce M_1^r and M_2^r and partitions the outputs only once, has the best space performance.

The differences between the two versions of the improved algorithm that do not reduce M_1^r and M_2^r, demonstrate the tradeoff between the efficiency of the improved algorithm and its overhead. On the one hand, the efficiency of improved the algorithm results in a better running time, and on the other hand the overhead results in larger space requirements. This tradeoff is taken into account in the subroutine

```
FSM improvedAlgorithm(FSM om){
  FSM         m1, m2, m1r, m2r, m1e, m2e, m1d, m2d, md;
  BddFunction fd, f1d, f2d;
  BDD         re, rd;

  /* the recursion tail condition - based on the size of the model */
  if (!shouldSplit(om))
    return reduction(om);

  /* partition om to m1 and m2 */
  partModel(om, m1, m2);

  m1r = rmExternalOutputs(m1);
  m1r = improvedAlgorithm(m1r);
  m2e = modelComposition(m1r, m2);

  m2r = rmExternalOutputs(m2);
  m2r = improvedAlgorithm(m2r);
  m1e = modelComposition(m1, m2r);

  m1d = reduction(m1e);
  m2d = reduction(m2e);

  f1d = composeFunc(m1d.equivFunc, m2r.equivFunc);
  f2d = composeFunc(m2d.equivFunc, m1r.equivFunc);
  fd = joinBddFunc(f1d,f2d);

  rd = bdd_image(om.domain, fd);
  md = disjointComposition(m1d, m2d, rd);

  return md;
}
```

Figure 5.5: The improved algorithm

Name	No. of inputs	No. of latches	No of outputs
$s298$	5	14	7
$s298d2$	5	11	3
$s298d3$	5	13	5
$s400d1$	5	17	2
$s400d2$	5	17	2
$s400d3$	5	19	4
$s400$	5	21	6
$s349$	11	15	11
$s444.2$	5	20	5
$s444$	5	21	6

Table 5.2: General properties of the tested designs

`shouldSplit` that decides whether to reduce the sub-model by further partitioning it with the improved algorithm or should it use the ordinary reduction algorithm. In general, if the sub-model is too small, then the overhead the improved algorithm become too large.

Note that, while in some cases the improved algorithm is up to 12 times faster than the ordinary minimization algorithm, in cases where the ordinary minimization algorithm has better performance, the differences between the algorithms are small.

5.5 Properties of bisimulation

In this section, we prove the claims presented in Section 5.2. Note that whenever two FSMs M_1 and M_2 are composed, they must satisfy $O_1 \cap \emptyset_2 = \emptyset$.

Lemma 5.2.1 *Let M be an FSM, and let M_Q be the quotient FSM of M. Let (α, i, α') be an element in R_Q. Then for every state s in α there exists a state s' in α' such that $(s, i, s') \in R$.*

Proof :Assume that $(\alpha, i, \alpha') \in R_Q$. Let $H \subseteq S \times S$ be the maximal bisimulation relation over $M \times M$. The definition of quotient FSM implies that there are states t, t' in S such that $t \in \alpha$, $t' \in \alpha'$ and $(t, i, t') \in R$. Let s be a state in α. Since s and t are in the same equivalence class, $(t, s) \in H$. Thus, there exists a state s' such that

Name	ordinary algorithm	naive algorithm	improved algorithm	improved2 algorithm	improved3 algorithm
s298	46	72	34	27	27
s298d2	21	26	22	22	23
s298d3	32	41	29	26	26
s400d1	190	469	385	206	206
s400d2	173	575	500	239	239
s400d3	1,336	2,048	1,209	590	601
s400	12,396	space overflow	2,302	1,129	999
s349	2,640	4,496	1,759	1,474	255
s444.2	3,512	3,379	1,380	828	799
s444	9,362	space overflow	2,891	1,055	1,038

Table 5.3: The time in seconds for minimization of the different minimization algorithms.

Name	ordinary algorithm	naive algorithm	improved algorithm	improved2 algorithm	improved3 algorithm
s298	1,482,712	1,919,032	451,793	326,359	467,705
s298d2	151,885	209,452	85,108	125,577	144,624
s298d3	759,032	716,557	278,283	337,834	362,691
s400d1	8,691,398	12,521,457	10,105,732	5,118,214	5,118,214
s400d2	9,368,512	12,775,984	7,009,955	4,820,907	4,820,907
s400d3	28,436,930	41,089,896	12,540,649	10,714,419	25,165,669
s400	105,175,584	space overflow	32,491,632	17,740,753	44,641,501
s349	27,567,754	36,747,754	12,442,018	2,552,658	3,876,761
s444.2	49,964,703	66,788,414	19,376,000	17,451,240	33,190,412
s444	97,687,526	space overflow	21,972,212	17,168,679	43,223,054

Table 5.4: The maximal number of BDD nodes required by the different minimization algorithms.

$(s, i, s') \in R$ and $(t', s') \in H$. Since $(t', s') \in H$, t' and s' are in the same equivalence class, thus $s' \in \alpha'$. $\square$

Proposition 5.5.1 *If M is deterministic then M_Q is deterministic.*

Lemma 5.2.3 *M is minimized iff the maximal bisimulation relation over $M \times M$ contains exactly the identity pairs.*

Proof : For the first direction, assume that H is the maximal bisimulation over $M \times M$ and that H contains exactly the identity pairs. Then every equivalence class contains exactly one state. Let M_Q be the quotient FSM of M. We define a function $f : S \rightarrow S_Q$ as follows: $f(s) = \alpha$ iff s is in α. Obviously f is a total and onto function. Since every equivalence class contains exactly one state, f is also one to one. Furthermore, by Lemma 5.2.1 and the definition of quotient FSM, $(s, i, s') \in R$ iff $(f(s), i, f(s')) \in R_Q$. Thus, M and M_Q are isomorphic and M is minimized.

For the second direction, assume that there is a pair $(s_1, s_2) \in H$ such that $s_1 \neq s_2$. Then s_1, s_2 are in the same equivalence class. Since the equivalence classes partition the states set and at least one class contains more than one state, $|S_Q| < |S|$. Thus M and M_Q are not isomorphic. $\square$

Lemma 5.5.2 *Let M be an FSM. The identity relation $H_{ID} = \{(s, s) | s \in S\}$ is a bisimulation relation over $M \times M$.*

Proof :

- For every $s_0 \in S_0$, $(s_0, s_0) \in H_{ID}$.

Let (s, s) be a pair in H_{ID}:

- $L(s) = L(s)$.
- Let (s, i, s') be an element in R. Then (s, i, s') is an element in R, and $(s', s') \in H_{ID}$. $\square$

Lemma 5.5.3 *Let M_Q be the quotient FSM of M, and let H_{QQ} be the maximal bisimulation relation over $M_Q \times M_Q$. Let $H_q = \{(s_1, s_2) | ([s_1], [s_2]) \in H_{QQ}\}$, then H_q is a bisimulation relation over $M \times M$.*

Proof :

- By the definition of the quotient FSM, for every $s_0 \in S_0$, $[s_0] \in S_{0Q}$. Since $([s_0],[s_0]) \in H_{QQ}$, $(s_0, s_0) \in H_q$.

Let (s_1, s_2) be a pair in H_q.

- $([s_1],[s_2]) \in H_{QQ}$ implies that $L_Q([s_1]) = L_Q([s_2])$ which implies that $L(s_1) = L(s_2)$.
- Let (s_1, i, s_1') be an element in R. Then $([s_1], i, [s_1']) \in R_Q$. Since $([s_1],[s_2]) \in H_{QQ}$ there exists a class α_2' such that $([s_2], i, \alpha_2') \in R_Q$ and $([s_1'], \alpha_2') \in H_{QQ}$. $([s_2], i, \alpha_2') \in R_Q$ together with Lemma 5.2.1, implies that there exists a state s_2' such that $(s_2, i, s_2') \in R$. The definition of H_q implies $(s_1', s_2') \in H_q$.
- Similarly, we can prove that for every successor s_2' of s_2 there exists a successor s_1' of s_1 such that $(s_1', s_2') \in H_q$. $\Box$

Lemma 5.5.4 *Let M_Q be the quotient FSM of M, and let H_{QQ} be the maximal bisimulation relation over $M_Q \times M_Q$. Then H_{QQ} is the identity relation.*

Proof : Lemma 5.5.2 implies that the identity relation is bisimulation relation over $M_Q \times M_Q$, thus it is contained in H_{QQ}. Assume to the contrary that H_{QQ} contains a pair (α_1, α_2) such that $\alpha_1 \neq \alpha_2$. Let s_1 and s_2 be states in α_1 and α_2 respectively and let H_q be the relation that defined in Lemma 5.5.3. By the definition of H_q, $(s_1, s_2) \in H_q$. By Lemma 5.5.3, H_q is a bisimulation over $M \times M$, thus (s_1, s_2) is an element in the maximal bisimulation over $M \times M$. This implies that s_1 and s_2 are in the same equivalence class, a contradiction. $\Box$

Corollary 5.5.5 *Every quotient FSM is minimized.*

For the rest of this paper, we will use the term "minimized FSM" for quotient FSM.

Lemma 5.2.4 *Let M be an FSM and M_Q be the quotient FSM of M with respect to O', then M and M_Q are bisimulation equivalent with respect to O'.*

Proof : Let $H_Q \subseteq S \times S_Q$ be the following relation: $H_Q = \{(s, \alpha) | s \text{ is in } \alpha\}$. We prove that H_Q is a bisimulation relation.

- By the definition of the quotient FSM, for every $s_0 \in S_0$, s_0 is in $\alpha_0 \in S_{0Q}$. Similarly for every $\alpha_0 \in S_{0Q}$ there exists $s_0 \in S_0$ such that $s_0 \in \alpha_0$.

Let (s, α) be a pair in H_Q:

- By the definition of the quotient FSM, $L(s) \cap O' = L_Q(\alpha)$.
- Let (s, i, s') be an element in R. Let α' be the equivalence class of s', then by the definition of the quotient FSM, $(\alpha, i, \alpha') \in R_Q$ and by the definition of H_Q, $(s', \alpha') \in H_Q$.
- Let (α, i, α') be an element in R_Q. By Lemma 5.2.1, there exists a state s' such that $(s, i, s') \in R$ and s' is in α'. Thus $(s', \alpha') \in H_Q$. $\square$

Lemma 5.5.6 *Let M_1 and M_2 be two FSMs that are bisimulation equivalent. Let $H \subseteq S_1 \times S_2$ be a bisimulation relation over $M_1 \times M_2$. Then, the relation $H' = \{(s_1, s'_1) | \text{there exists } s_2 \in S_2 \text{ such that } (s_1, s_2) \in H \text{ and } (s'_1, s_2) \in H\}$ is a bisimulation relation over M_1 with respect to $O_1 \cap O_2$.*

Proof : We prove that H' is a bisimulation relation.

- Since H is a bisimulation relation, for every initial state $s_{01} \in S_{01}$ there exists an initial state $s_{02} \in S_{02}$ such that $(s_{01}, s_{02}) \in H$. Thus, for every initial state $s_{01} \in S_{01}$, $(s_{01}, s_{01}) \in H'$.

For every pair $(s_1, s'_1) \in H'$ the following holds:

- Since $(s_1, s'_1) \in H'$, there exists a state $s_2 \in S_2$ such that $(s_1, s_2) \in H$ and $(s'_1, s_2) \in H$. This implies that $L_1(s_1) \cap (O_1 \cap O_2) = L_2(s_2) \cap (O_1 \cap O_2) = L_1(s'_1) \cap (O_1 \cap O_2)$.
- Let (s_1, i, t_1) be a transition in R_1. Since $(s_1, s'_1) \in H'$, there exists a state $s_2 \in S_2$ such that $(s_1, s_2) \in H$ and $(s'_1, s_2) \in H$. Since H is a bisimulation, there exists a state $t_2 \in S_2$ such that $(s_2, i, t_2) \in R_2$ and $(t_1, t_2) \in H$. This implies that there exists a state $t'_1 \in S_1$ such that $(s'_1, i, t'_1) \in R_1$ and $(t'_1, t_2) \in H$. Thus $(t_1, t'_1) \in H'$.

- Similarly, for every transition $(s'_1, i, t'_1) \in R_1$ there exists a transition $(s_1, i, t_1) \in R_1$ such that $(t_1, t'_1) \in H'$.

$\Box$

Lemma 5.2.5 *Let M be an FSM and M_Q be the quotient FSM of M with respect to O'. Then M_Q is the smallest (in number of states and transitions) FSM which is bisimulation equivalent to M with respect to O'.*

Proof : First, we prove that M_Q is smallest with respect to the number of states. Assume to the contrary that there exists an FSM M' that is bisimulation equivalent to M and smaller than M_Q. Since bisimulation is transitive, M_Q and M' are bisimulation equivalent. Let H be a bisimulation relation over $M_Q \times M'$. Then, there exists two different states s_q and t_q in S_Q that are equivalent to the same state in M'. Let H_q be the relation $H_q = \{(s_q, t_q) | \text{there exists } s' \in S' \text{ such that } (s_q, s') \in H \text{ and } (t_q, s') \in H\}$. By Lemma 5.5.6, H_q is a bisimulation relation. Thus s_q and t_q are bisimulation equivalent, contradicting Lemma 5.5.4.

Next, we prove that M_Q is smallest with respect to number of transitions. Assume to the contrary that there exists an FSM M' that is bisimulation equivalent to M and smaller than M_Q. Since bisimulation is transitive, M_Q and M' are bisimulation equivalent. Let H be a bisimulation relation over $M_Q \times M'$. Since the number of states in M_Q is not larger than the number of states in M', there exists a pair $(s_q, s') \in H$ such that the number of transitions from s_q is greater than the number of transitions from s'. Since for every transition $(s', i, t') \in R'$ there exists a matching transition from s_q, there exists a transition $(s', i, t') \in R'$ which has two transitions (s_q, i, t_{q1}) and (s_q, i, t_{q2}) in R_q which match it. This implies that $(t_{q1}, t') \in H$ and $(t_{q2}, t') \in H$. Let H_q be the relation $H_q = \{(s_q, t_q) | \text{there exists } s' \in S' \text{ such that } (s_q, s') \in H \text{ and } (t_q, s') \in H\}$. By Lemma 5.5.6, H_q is a bisimulation relation. Thus t_{q1} and t_{q2} are bisimulation equivalent, contradicting Lemma 5.5.4. $\Box$

5.5.1 Composition and bisimulation

Next, we present some properties of composition and bisimulation.

Lemma 5.5.7 *Let $M = M_1 || M_2$ and let H_1 and H_2 be the maximal bisimulation relations over $M_1 \times M_1$ and $M_2 \times M_2$ with respect to O_1 and*

O_2 respectively. Let H be the relation $H = \{((s_1, s_2), (t_1, t_2)) | (s_1, t_1) \in H_1,\ (s_2, t_2) \in H_2\}$ then H is a bisimulation over $M \times M$.

Proof :

- Let $(s_{10}, s_{20}) \in S_0$, since $(s_{10}, s_{10}) \in H_1$ and $(s_{20}, s_{20}) \in H_2$, $((s_{10}, s_{20}), (s_{10}, s_{20})) \in H$.

Let $((s_1, s_2), (t_1, t_2))$ be a pair in H.

- By the definition of H, $(s_1, t_1) \in H_1$ and $(s_2, t_2) \in H_2$, thus $L_1(s_1) = L_1(t_1)$ and $L_2(s_2) = L_2(t_2)$. Since $O_1 \cap O_2 = \emptyset$, $L((s_1, s_2)) = L((t_1, t_2))$.
- Let $((s_1, s_2), i, (s'_1, s'_2))$ be an element in R. By the definition of composition, $(s_1, (i \cup L_2(s_2)) \cap I_1, s'_1) \in R_1$ and $(s_2, (i \cup L_1(s_1)) \cap I_2, s'_2) \in R_2$. Since $(s_1, t_1) \in H_1$ and $L_2(s_2) = L_2(t_2)$, there exists a state t'_1 such that $(t_1, (i \cup L_2(t_2)) \cap I_1, t'_1) \in R_1$ and $(s'_1, t'_1) \in H_1$. Similarly, there exists a state t'_2 such that $(t_2, (i \cup L_1(t_1)) \cap I_2, t'_2) \in R_2$ and $(s'_2, t'_2) \in H_2$. The definition of composition implies that $((t_1, t_2), i, (t'_1, t'_2)) \in R$ and by the definition of H, $((s'_1, s'_2), (t'_1, t'_2)) \in H$.
- In a similar way we can show that for every successor (t'_1, t'_2) of (t_1, t_2) there exists a successor (s'_1, s'_2) of (s_1, s_2) such that $((s'_1, s'_2), (t'_1, t'_2)) \in H$. $\square$

Lemma 5.5.8 *If $M = M_1 || M_2$ is minimized then M_1 and M_2 are also minimized.*

Proof : Assume to the contrary that the lemma does not hold. W.l.o.g. assume that M_1 is not minimized. By Lemma 5.2.3 there are two different states s_1, t_1 such that $(s_1, t_1) \in H_1$. Since every bisimulation relation contains the identity pairs, there exists a state s_2 such that $(s_2, s_2) \in H_2$. Let H be the relation defined in Lemma 5.5.7, then $((s_1, s_2), (t_1, s_2)) \in H$. By Lemma 5.5.7 H is a bisimulation relation, thus it is contained in the maximal bisimulation relation over $M \times M$. This implies that $((s_1, s_2), (t_1, s_2))$ is an element in the maximal bisimulation relation. By Lemma 5.2.3, M is not minimized, a contradiction. $\square$

Lemma 5.5.9 *Let $M = M_1||M_2$ and H be a bisimulation over $M \times M$. If $O_2 \cap I_1 = \emptyset$ then the relation*
$H_1 = \{(s_1, t_1)|s_1, t_1 \in S_1$ and $\exists\ s_2, t_2\ ((s_1, s_2), (t_1, t_2)) \in H\}$ is a bisimulation relation over $M_1 \times M_1$.

Proof :

- Let $s_{10} \in S_{10}$ and $s_{20} \in S_{20}$. Since $((s_{10}, s_{20}), (s_{10}, s_{20})) \in H$, $(s_{10}, s_{10}) \in H_1$.

Let (s_1, t_1) be a pair of states such that $(s_1, t_1) \in H_1$ and let s_2, t_2 be states such that $((s_1, s_2), (t_1, t_2)) \in H$.

- $((s_1, s_2), (t_1, t_2)) \in H$ implies $L((s_1, s_2)) = L((t_1, t_2))$. Since $O_1 \cap O_2 = \emptyset$, we conclude that $L_1(s_1) = L_1(t_1)$.
- Let (s_1, i_1, s_1') be an element in R_1. Since $O_2 \cap I_1 = \emptyset$, $I_1 \subseteq I$. Let $i \subseteq I$ be such that $i_1 = i \cap I_1$. Since $O_2 \cap I_1 = \emptyset$, $(i \cup L_2(s_2)) \cap I_1 = i \cap I_1 = i_1$. Let s_2' be a state such that $(s_2, (i \cup L_1(s_1)) \cap I_2, s_2') \in R_2$. Such s_2' exists by the receptiveness of Moore machines. Then $((s_1, s_2), i, (s_1', s_2')) \in R$. Since $((s_1, s_2), (t_1, t_2)) \in H$, there exists a state (t_1', t_2') such that $((t_1, t_2), i, (t_1', t_2')) \in R$ and $((s_1', s_2'), (t_1', t_2')) \in H$. This implies that $(t_1, i_1, t_1') \in R_1$. By the definition of H_1, $(s_1', t_1') \in H_1$.
- In a similar way we can show that for every successor t_1' of t_1 there exists a successor s_1' of s_1 such that $(s_1', t_1') \in H_1$. □

Lemma 5.2.7 *Let M_1 and M_2 be minimized FSMs. If $O_1 \cap I_2 = \emptyset$ and $O_2 \cap I_1 = \emptyset$, then $M = M_1||M_2$ is minimized.*

Proof Let H be the maximal bisimulation over $M \times M$. Assume to the contrary that the lemma does not hold. Then by Lemma 5.2.3, there are two different states $(s_1, s_2), (t_1, t_2)$ such that $((s_1, s_2), (t_1, t_2)) \in H$. Since $(s_1, s_2) \neq (t_1, t_2)$, either $s_1 \neq t_1$ or $s_2 \neq t_2$. We assume w.l.o.g. that $s_1 \neq t_1$. Let H_1 be the relation defined in Lemma 5.5.9. By Lemma 5.5.9 H_1 is a bisimulation. By the definition of H_1, $(s_1, t_1) \in H_1$. By Lemma 5.2.3, M_1 is not minimized, a contradiction. □

Chapter 6

Using BDDs for preimage calculations

In this chapter we improve the algorithm suggested in [CM90a, CM90b] for the preimage operation. We suggest a new *inverse algorithm* with the same complexity as the *expound subroutine* but with better constants. Furthermore, the information which is attached to every BDD node, represents the preimage of the set it represents. Thus, the implementation of the inverse algorithm is much simpler and more intuitive; moreover, it is suitable for optimizations. Experimental results show that the inverse algorithm works much more efficiently than the expound subroutine, and in some cases even competes successfully with the monolithic algorithm and the early quantification algorithm.

6.1 Preliminaries

We describe BDDs as presented in [Bry86]. We use $x_1, x_2, \ldots, x_n$ to denote boolean variables and $g(x_1, x_2, \ldots x_n)$ to denote a boolean function. Let $\gamma \in \{0,1\}^n$, we use the functions $v_i(\gamma)$ to denote the value of the i'th bit in γ. Sometimes we use x_i as the boolean function $g(\gamma) = v_i(\gamma)$ and $\overline{x_i}$ as $g(\gamma) = \neg v_i(\gamma)$.

A BDD is always defined with respect to an order over the variables. Given an order π over the BDD variables, a BDD is defined as follows:

Definition 6.1.1 *A BDD is a DAG (Directed Acyclic Graph) with one root and at most two leaves. The leaves are labeled with* 0, 1 *and* *every non-leaf node nd is labeled by a variable $x_i = L(nd)$. Every non-leaf node nd has exactly two successors nd.low and nd.high. If a non-leaf node nd' is a successor of another node nd then $\pi(L(nd')) > \pi(L(nd))$.*

Every BDD node nd represents a boolean function g_{nd}. The BDD represents the boolean function of its root. The boolean function g_{nd} is defined inductively on the structure of the BDD:

- If nd is a leaf then it represents the label of nd (0 or 1).
- If nd in non-leaf node which is labeled with variable x_i, then $g_{nd} = (x_i \wedge g_{nd.high}) \vee (\overline{x_i} \wedge g_{nd.low})$.

Every boolean function $g : \{0,1\}^n \rightarrow \{0,1\}$ characterizes a set $A \subseteq \{0,1\}^n$, such that $\gamma \in \{0,1\}^n$ is an element of A if and only if $g(\gamma) = 1$. In the rest of this work we will not differ between a BDD Γ, the boolean function g that Γ represents, and the set A that g characterizes.

Next, we define a reduced BDD:

Definition 6.1.2 *A BDD Γ is reduced if it satisfies the followings:*

1. *There are no two different nodes in Γ which represent the same function.*
2. *Each non-leaf node nd in Γ satisfies: $nd.high \neq nd.low$.*

[Bry86] shows that given an order over the BDD variables, for every boolean function there exists a unique reduced BDD which represents it; for the rest of this paper we refer only to reduced BDDs. In addition, [Bry86] suggests efficient procedures that implement operations over boolean functions represented by BDDs. Table 6.1 shows the operations and their time complexity. We use Γ_1, Γ_2 as BDDs with the same variable order.

In model checking and equivalence checking of deterministic models, BDDs are used to characterize sets of states of the verified FSM, its transition relation and its labeling function. The set of states and set of initial states are represented by their characterizing boolean functions. The transition relation is represented as a function, $R : S \times 2^I \rightarrow S$. The labeling function is represented as a function $L : S \rightarrow 2^{AP}$.

Operation	time complexity
$\Gamma_1 \wedge \Gamma_2$	$O(\lvert\Gamma_1\rvert \cdot \lvert\Gamma_2\rvert)$
$\Gamma_1 \vee \Gamma_2$	$O(\lvert\Gamma_1\rvert \cdot \lvert\Gamma_2\rvert)$
$\neg\Gamma_1$	$O(1)$
$g_1\rvert_{x_i=b}$	$O(\lvert\Gamma_1\rvert \log \lvert\Gamma_1\rvert)$
$\exists x_1 \ldots \exists x_n \Gamma_1$	$O(\lvert\Gamma_1\rvert^n)$

Table 6.1: The complexity of BDDs operations

6.1.1 Using BDDs for function manipulations

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ and a subset $\Gamma \subseteq \{0,1\}^n$, we define $f(\Gamma)$ called the image of Γ, to be $f(\Gamma) = \{\gamma' | \exists \gamma \in \Gamma$ such that $f(\gamma) = \gamma'\}$. Similarly, given a set $\Gamma' \subseteq \{0,1\}^m$ the operation $f^{-1}(\Gamma')$ called the preimage of Γ', is defined as $f^{-1}(\Gamma') = \{\gamma | \exists \gamma' \in \Gamma'$ such that $f(\gamma) = \gamma'\}$.

Let $f : \{0,1\}^n \rightarrow \{0,1\}^m$ be a total function. There are two ways to represent f: The partitioned [BCL91] representation and the monolithic representation.

We first describe the partitioned representation of f. Let the elements of $\{0,1\}^n$ be encoded by $x_1, x_2, \ldots x_n$ and the elements of $\{0,1\}^m$ by $x'_1, x'_2, \ldots, x'_m$. Given an element $\gamma \in \{0,1\}^n$, $\gamma' = f(\gamma)$ is a unique element of $\{0,1\}^m$. Thus the values of the variables that encode γ' depend only on γ. A function f can therefore be defined by m boolean functions $f_1, f_2, \ldots, f_m$, where f_j determines the value of x'_j in the result of f. Every boolean formula is represented by a BDD; thus we represent f as m BDDs over $x_1, x_2, \ldots, x_n$.

The monolithic representation of functions refers to f as a relation where the pair (γ, γ') is an element in f iff $f(\gamma) = \gamma'$. Such a relation is constructed as follows $f = \wedge_{i=1}^{m} x'_i \leftrightarrow f_i$.

There are three different approaches for manipulating functions which are represented by BDDs: The monolithic algorithm, the early quantification algorithm [BCL91], and the expound subroutine.

When a function f is represented by the monolithic representation, the operation $f^{-1}(\Gamma')$ is calculated by the monolithic algorithm as follows: $f^{-1}(\Gamma') = \exists x'_1, x'_2, \ldots x'_n (f \wedge \Gamma')$. The major drawback in this method is the size of the monolithic relation, which often becomes too large to handle.

The other two algorithms for manipulating functions operate on the partitioned representation of functions. First, we describe the early quantification algorithm for the f^{-1} operation. This algorithm is an improvement of the monolithic algorithm. In the early quantification algorithm the f^{-1} operation is performed as $f^{-1}(\Gamma') = \exists x'_n((f_n \leftrightarrow x'_n) \wedge \exists x'_{n-1}((f_{n-1} \leftrightarrow x'_{n-1}) \wedge \ldots \wedge \exists x'_1((f_1 \leftrightarrow x'_1) \wedge \Gamma')\ldots))$. In this way the intermediate BDDs remain small and the whole operation requires less space.

Next, we describe the *expound* subroutine which calculates $f^{-1}(\Gamma')$ in the following way: The operation is calculated inductively on the graph of the BDD Γ', in a top down direction. For each node nd in Γ' a function nd_{pre} is calculated. When the algorithm terminates, the resulting set is characterized by the function of the 1 leaf of Γ'. At the beginning of the algorithm, the function of the root of Γ' is 1. For each node nd' in Γ' the function nd'_{pre} is calculated as follows: Let $nd_1, nd_2, \ldots nd_m$ be the predecessors of nd'. Since the algorithm is inductive, the functions $nd_{1pre}, nd_{2pre}, \ldots nd_{mpre}$ have already been calculated. Let $L(nd_1), L(nd_2), \ldots L(nd_m)$, be the variables that label $nd_1, nd_2, \ldots nd_m$, respectively. The function nd'_{pre} is calculated as $\bigvee_{i=1}^{m}(nd_{ipre} \wedge c_i)$ where

$$c_i = \begin{cases} f_j & nd_i \text{ is connected to } nd' \text{ by its high edge and } x'_j = L(nd_i) \\ \neg f_j & nd_i \text{ is connected to } nd' \text{ by the low edge and } x'_j = L(nd_i) \end{cases}$$

The advantages of the expound subroutine are:

- It does not use next state variables.
- It uses only "cheap" BDD operations.
- The number of BDD operations is linear in the number of BDD nodes in Γ'.

However, the top down direction of the subroutine creates some practical problems.

1. In most BDD packages, in each BDD node there are pointers only to its successors and there are no pointers to predecessors. For every BDD node nd', the value of the function nd'_{pre}, depends on the predecessors of nd'. Since nd' does not contain pointers to

his predecessors, the information for calculating nd'_{pre} should be pushed from the predecessors when the algorithm reaches them. Thus, before the algorithm reaches nd', it should reach all of its predecessors. However, because nd' does not have pointers to its predecessors, its does not "know" which BDD nodes are they. Thus, the algorithm have to reach all the BDD nodes above nd' before it reaches it. This implies, that the algorithm needs an additional data structure which enables access to the nodes of Γ' according to their level.

2. In the implementation of expound subroutine, the algorithm attaches an extra function *pre* to every BDD node. In most BDD packages, the BDD nodes contain only two pointers, and there is no space for the extra pointer nd_{pre}. For other BDD operations that need to attach extra information to the BDD nods, using cache solves this problem. A cache is a data structure in which for every node the extra information is stored. In order to make the cache effective, the cache enables collisions, where different nodes are mapped to the same cell in the cache. In case of collision, some of the information is lost and have to be recalculated again. This is not efficient for the expound subroutine, because it will require to access all the nodes above the node for which the function should be recalculated.

3. Most BDD packages use complementary edges. A complementary edge, which points at a BDD node nd, represents the complement set of the set that nd represents. The use of complementary edges forces the expound subroutine to store two functions nd_{pre1} and nd_{pre2} in each BDD node, increasing the space requirements of the algorithm.

6.2 The inverse algorithm

In this section we present our inverse algorithm as an alternative way to compute the f^{-1} operation. Similarly to the expound subroutine, the inverse algorithm stores data in the BDD nodes. However, the computation is bottom up. For every node nd, the algorithm calculates the preimage of the set that nd represents, this makes the algorithm more intuitive and easier to implement. The algorithm is presented in Figure 6.1.

```
BDD inverse(BDD $\Gamma'$){
    return inverseNode($\Gamma'.root$)
}

inverseNode(node $nd$){
    if (($res$ = getCache($nd$)) $\neq$ NULL) then return $res$
    if ($nd$ is a terminal node ) then return $nd.value$
    $j = nd.index$
    $res = (\neg f_j \wedge$ inverseNode($nd.low$)$) \vee (f_j \wedge$ inverseNode($nd.high$)$)$
    insertCache($nd$,$res$);
    return($res$)
}
```

Figure 6.1: The inverse algorithm

Our algorithm have all the advantages of the expound subroutine. Furthermore, it is easy to implement. Since the function, which is attached to the BDD nodes, depends on the sets they represent, the algorithm is suitable for using cache. Finally, since $f^{-1}(\neg A) = \neg f^{-1}(A)$, using complementary edges in the BDD does not increase the amount of space it requires.

Next, we prove the correctness of the algorithm. The first proposition is immediate from the definition of the operation f^{-1} and from the definition of the partitioned representation of f.

Proposition 6.2.1 *Given an element $\gamma' \in \{0,1\}^m$. An element $\gamma \in \{0,1\}^n$ satisfies $f(\gamma) = \gamma'$ iff for every $1 \leq j \leq m$, $\gamma \in f_j \Leftrightarrow v_j(\gamma') = 1$.*

Definition 6.2.2 *We define $\hat{f}_j$ as follows:*

$$\hat{f}_j(b) = \begin{cases} f_j & b = 1 \\ \neg f_j & b = 0 \end{cases} .$$

The next proposition rephrases Proposition 6.2.1, using the notation $\hat{f}_j$. Let $b'_1, b'_2, \ldots b'_m$ be the boolean representation of γ', then every $\gamma \in f^{-1}(\gamma')$ should satisfy that for every $1 \leq j \leq m$, $f_j(\gamma) = b'_j$.

Proposition 6.2.3 *Given an element γ', the following holds: $f^{-1}(\gamma') = \wedge_{j=1}^{m} \hat{f}_j(v_j(\gamma'))$.*

We now extend the previous proposition to a set of elements in $\{0,1\}^m$.

Corollary 6.2.4 *Given a subset $Q' \subseteq \{0,1\}^m$, let $f^{-1}(Q') = \{\gamma | \exists \gamma'. \gamma' \in Q' \text{ and } f(\gamma) = \gamma'\}$ then $f^{-1}(Q') = \vee_{\gamma' \in Q'}(\wedge_{j=1}^{m} \hat{f}_j(v_j(\gamma')))$.*

Definition 6.2.5 *Let Q' be a subset of $\{0,1\}^m$. Let $j \in \{1,\ldots,m\}$. Then, $Q'_j = Q' \wedge \{\gamma' | v_j(\gamma') = 1\}$ and $\overline{Q'_j} = Q' \wedge \{\gamma' | v_j(\gamma') = 0\}$.*

We now calculate the operation $f^{-1}(Q')$ in two separate stages, based on the fact that $f^{-1}(Q') = f^{-1}(Q'_j) \vee f^{-1}(\overline{Q'_j})$. This enables to pull f_j out as shown in Lemma 6.2.6.

Lemma 6.2.6 *Let $Q = f^{-1}(Q')$, then $Q = (f_j \wedge \vee_{\gamma' \in Q'_j}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma')))) \vee (\neg f_j \wedge \vee_{\gamma' \in \overline{Q'_j}}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma'))))$.*

Proof :

- $Q = \vee_{\gamma' \in Q'}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma')))$.
- Since $Q' = Q'_j \vee \overline{Q'_j}$, $Q = \vee_{\gamma' \in (Q'_j \vee \overline{Q'_j})}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma')))$; thus $Q = \vee_{\gamma' \in Q'_j}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma'))) \vee \vee_{\gamma' \in \overline{Q'_j}}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma')))$.
- For every $\gamma' \in Q'_j$, $v_j(\gamma') = 1$; thus for every $\gamma' \in Q'_j$, $\hat{f}_j(v_j(\gamma')) = f_j$.
- This implies that $\vee_{\gamma' \in Q'_j}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma'))) = \vee_{\gamma' \in Q'_j}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma')) \wedge f_j) = f_j \wedge \vee_{\gamma' \in Q'_j}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma')))$.
- Similarly, $\vee_{\gamma' \in \overline{Q'_j}}(\wedge_{k=1}^{m} \hat{f}_k(v_k(\gamma'))) = \neg f_j \wedge \vee_{\gamma' \in \overline{Q'_j}}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma')))$.
- Thus $Q = (f_j \wedge \vee_{\gamma' \in Q'_j}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma')))) \vee (\neg f_j \wedge \vee_{\gamma' \in \overline{Q'_j}}(\wedge_{k \neq j} \hat{f}_k(v_k(\gamma'))))$. $\square$

Lemma 6.2.7 *Let Q' be a subset of $\{0,1\}^m$ and let $Q = f^{-1}(Q')$. Let Γ be the BDD that represents Q'. Let x_j be the variable labeling the root of Γ. Then $Q = (f_j \wedge \vee_{\gamma' \in \Gamma_{root.high}} \wedge_{k \neq j} \hat{f}_k(v_k(\gamma'))) \vee (\neg f_j \wedge \vee_{\gamma' \in \Gamma_{root.low}} \wedge_{k \neq j}^{m} \hat{f}_k(v_k(\gamma')))$.*

Proof : The definition of BDD implies that $root.low = Q|_{v_j \leftarrow 0}$ which is exactly $\overline{Q'_j}$. Similarly, $root.high = Q|_{v_j \leftarrow 1} = Q'_j$. $\Box$

Corollary 6.2.8 can be concluded by Corollary 6.2.4 and Lemma 6.2.7.

Corollary 6.2.8 *Let Q' be a subset of $\{0,1\}^m$. Let Γ'_Q be the BDD that represents Q'. Let j be the index of the root of Γ'_Q. Then $Q = (f_j \wedge f^{-1}(root.high)) \vee (\neg f_j \wedge f^{-1}(root.low))$.*

The computation follows the following intuition. Suppose the set $Q' \subseteq \{0,1\}^m$ is represented by a BDD $\Gamma_{Q'}$ so that x'_j is the variable in the root of $\Gamma_{Q'}$. Elements of Q', represented by $root.low$ are those in which $x'_j = 0$, thus, their preimage is contained in $\neg f_j$. Similarly, the elements of Q', represented by $root.high$ are those in which $x'_j = 1$, and therefore their preimage is contained in f_j.

This form of calculation can be implemented by recursion over the graph of the BDD which represents Q'. Next, we prove the correctness of our algorithm.

Lemma 6.2.9 *Let nd be a node in the BDD Γ'_Q and let $Q'' \subseteq \{0,1\}^m$ be the set represented by nd. Then* `inverseNode(`nd`)` *returns $f^{-1}(Q'')$.*

Proof : We prove the lemma by induction on the levels of Γ'_Q from bottom up.

- Base: Let nd be a terminal node, we distinguish between two cases:
  - $nd.value = 1$, then nd represents $\{0,1\}^m$; in this case the function returns 1, which represents $f^{-1}(\{0,1\}^m) = \{0,1\}^n$.
  - $nd.value = 0$, then nd represents $\emptyset$; in this case the function returns 0, which represents $f^{-1}(\emptyset) = \emptyset$.
- Induction step: Assume that the lemma holds for all the levels below nd; we prove that it holds for nd. Both $nd.low$ and $nd.high$ belong to levels which are lower, in the induction order than the level of nd. Thus the induction hypothesis implies that $inverseNode(nd.low) = f^{-1}(nd.low)$ and $inverseNode(nd.high) = f^{-1}(nd.high)$. By Corollary 6.2.8, $inverseNode(nd) = f^{-1}(nd)$. $\Box$

Corollary 6.2.10 *The algorithm returns a BDD that represents $f^{-1}(Q')$.*

Complexity: As the expound subroutine, our inverse algorithm performs for each node in $\Gamma_{Q'}$ two conjunction operations and one disjunction. Thus, the number of BDD operation is $O(|\Gamma_{Q'}|)$ and it uses only "cheap" operations $(\wedge, \vee, \neg)$.

Unlike the expound algorithm, the inverse algorithm, is easy to implement, does not require any additional data structure, and is suitable for using cache and complement edges. This explains the much better performance shown in Section 6.2.6.

6.2.1 An Example: Modeling a deterministic FSM by BDDs for functions

The example in this section demonstrates how the BDD representation for functions can be used for representing an FSM. In addition, we show how to compute the set of predecessors Q for a given set of states Q'.

Consider the FSM in Figure 6.2. Its set of states is $S = \{00, 01, 10, 11\}$. Its input set is $I = \{a\}$. The transition function $R : S \times I \to S$ is shown in Table 6.2. In this table we use the variables (x_0, x_1) to encode S, i_0 to encode I and x'_0, x'_1 to encode the next states in the transition relation.

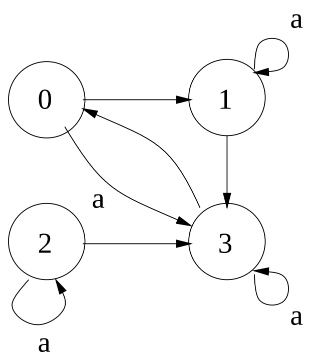

Figure 6.2: An example FSM

In order to define R in our BDD framework we partition it into two boolean functions R_0 and R_1, where, R_0 consists of the set of elements

x_0	x_1	i_0	x'_0	x'_1
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Table 6.2: The transition relation of the FSM

of the form (s, i) for which $x'_0 = 1$. Similarly, R_1 is the set of elements for which $x'_1 = 1$. The sets are: $R_0 = \{001, 010, 100, 101, 111\}$ and $R_1 = \{000, 001, 010, 011, 100, 111\}$.

Next we show how to use this representation in order to compute $R^{-1}(Q')$ using the `inverse` algorithm. The result of this operation is the set of pairs (s, i) such that $(s, i) \in R^{-1}(Q')$. Thus, the set Q of predecessors of Q' is computed by $Q = \{s | \exists i \in I.(s, i) \in R^{-1}(Q')\}$.

Let $Q' = \{01, 10\}$. The BDD $\Gamma_{Q'}$ is shown in Figure 6.3. The inverse

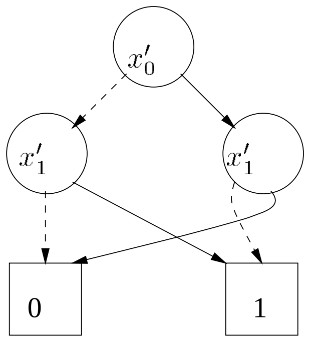

Figure 6.3: The BDD $\Gamma_{Q'}$. Dashed lines lead to *low* successors; full lines lead to *high* successors.

algorithm results in $R^{-1}(Q) = (\neg R_0 \wedge ((\neg R_1 \wedge 0) \vee (R_1 \wedge 1))) \vee (R_0 \wedge ((\neg R_1 \wedge 1) \vee (R_1 \wedge 0))) = (\neg R_0 \wedge R_1) \vee (R_0 \wedge \neg R_1) = \{000, 011, 101\}$. The set of predecessor is now computed by $Q = \{q | \exists i.(q, i) \in R^{-1}(Q')\} = \{00, 01, 10\}$.

6.3 Experimental results

We ran experiments to compare the four algorithms that calculate the preimage operation. We implemented these algorithm on the platform used in the Equivalence Department of the Israel Design Center at Intel Haifa.

Each algorithm was inserted as a subroutine into a tool which uses the preimage subroutine to calculate the transitive preimage of a given set Q'. That is, the tool calculates a set Q of all states from which there exists a trace to states in Q'. The calculation is done by repeating the preimage operation until a fix-point is reached.

We made the comparison over test cases of the equivalence department. The results are presented in Table 6.3, Table 6.4 and Table 6.5. Table 6.3 shows the properties of the tested designs. Table 6.4 presents the computation times in seconds of the different methods over the test cases, and Table 6.5 presents the the space requirements in BDD nods of the different methods over the test cases. The algorithms were tested on a machine with two CPUs of 550 MHZ each and 2GB memory.

The experimental results show that the inverse algorithm is strictly better than the expound algorithm both in time and space. It also shows that although the monolithic algorithm has the best average performances, for some designs it explodes, thus it needs a backup algorithm. A comparison between the inverse algorithm and the early quantification algorithm shows that the early quantification algorithms have an advantage although it is not a strict one.

Design	inputs number	latches number	outputs number
s1488	10	6	19
s1196	16	18	14
s713	36	19	23
s400	4	21	6
s444	4	21	6
s386	8	6	7
s349	5	15	11
s1238	15	18	14
s832	19	5	19
s820	19	5	19
x7	36	172	33

Table 6.3: The properties of the designs

Design	our algorithm	monolithic	early quantification	expound subroutine
s1488	47	30	37	78
s1196	22	34	22	23
s713	10,306	1,240	4,952	15,606
s400	11,597	1,569	6,581	55,936
s444	9,373	803	7,611	58,375
s386	21	21	21	22
s349	5,173	1,342	12,193	space overflow
s1238	22	37	22	23
s832	46	46	43	104
s820	49	41	44	100
x7	241	space overflow	286	194

Table 6.4: Time of calculations in seconds

Design	our algorithm	monolithic	early quantification	expound subroutine
s1488	3,523,525	472,289	1,880,253	5,020,289
s1196	127,317	1,898,538	123,801	171,971
s713	13,394,762	5,710,104	8,883,805	17,254,376
s400	6,695,438	2,487,305	11,224,721	10,459,001
s444	6,462,535	5,034,860	5,574,310	11,016,241
s386	35,256	16,349	41,636	54,249
s349	18,535,262	5,741,429	19,329,050	space overflow
s1238	130,900	1,984,082	121,053	168,110
s832	3,544,107	511,101	2,917,210	5,057,697
s820	3,544,061	511,058	2,917,169	5,057,648
x7	6,111,667	space overflow	2,325,368	5,580,730

Table 6.5: Space required in the calculations in BDD nodes

Chapter 7

Conclusion and Future Research

In this work we concentrated on methods for overcoming the state explosion problem. Chapters 3, 4, and 5 refer to the use of equivalence relations and preorders for abstraction. In Chapter 6 we improve an existing symbolic algorithm for the preimage operation.

In Chapter 3 we discussed minimization with respect to the simulation preorder. We proved that for every Kripke structure M there exists a unique smallest in size structure A such that M and A are simulation equivalent. We proved that given a structure M the minimal abstract structure A can be obtained by eliminating two redundancies: Equivalent states and little brothers. We presented two algorithms that construct the minimal equivalent structures: The minimizing algorithm and the partition algorithm. The former algorithm has a better time complexity and the latter has a better space complexity.

The results in Chapter 3 can be extended in several directions. In Chapter 3 we showed that minimization with respect to simulation equivalence, can result in smaller models than bisimulation minimization. We also showed that minimization with respect to language equivalence can result in an even smaller model, however the complexity of such minimization is exponential. An interesting research direction is to find a sequence of equivalence relations $E_0, E_1, \ldots, E_n$ where E_0 is the simulation equivalence relation, and E_n is the language equivalence relation. For each $i < n$ the following should holds: (1) $E_i \subseteq E_{i+1}$, thus E_{i+1} is less restrictive than E_i. (2) The result of the reduction with respect to E_{i+1} is smaller than the result of reduction with respect to E_i. (3) The complexity of reducing with respect to E_{i+1} is greater than or equal to the complexity of reducing with respect to E_i. Having such a sequence, a parameterized reduction algorithm can be developed. The algorithm will receive an equivalence relation as a parameter and reduce with respect to this relation.

In Chapter 4 we made a broad comparison between four notions of fair simulation: direct [DHWT91], delay [EWS01a], game [HKR97], and exists [GL94]. The comparison shows that there is no notion of fair simulation which has all desired advantages. However, it is clear that their relationship with temporal logics gives the exists and game simulations several advantages over the delay and direct simulations. On the other hand, the delay and direct simulations are better for minimization. Since this research is motivated by usefulness to model checking, relationships with logic are important. Thus, it is advantageous to refer to the delay and direct simulations as approximations of the game/exists simulations. These approximations enable some minimization with respect to the exists and game simulations. Out of the four notions, we consider the game simulation to be the best. This is due to its complexity and its applicability in modular verification.

Modularity is extensively used in the development of systems. As a result, most systems have a modular structure. In Chapter 5 we showed how this structure can be used for a more efficient minimization algorithm. Given an FSM M the algorithm constructs two disjoint FSMs M_1^e and M_2^e such that M is equivalent to the restricted composition of M_1^e and M_2^e. Once the algorithm constructs these FSMs, the problem of minimizing M is reduced to minimizing M_1^e and M_2^e separately and composing the result. Since the complexity of minimizing M might be quadratically greater than minimizing M_1^e and M_2^e separately, the potential of the algorithm is huge. The experimental results showed that the improved algorithm outperformed both the naive algorithm and the ordinary algorithm.

In Chapter 6 we improved the algorithm suggested in [CM90a, CM90b] for preimage calculation. We suggested a new *inverse algorithm* with the same complexity as the expound subroutine but with better constants.

The experimental results show that the inverse algorithm is strictly better than the expound algorithm both in time and space. It also shows that although the monolithic algorithm has the best average performance, for some designs it explodes, thus it needs a backup algorithm. A comparison between the inverse algorithm and the early quantification algorithm shows that the early quantification algorithm have an advantage although it is not a significant one. Given a function f and a set of elements Q' there is a high probability that the inverse algorithm will perform the operation $f^{-1}(Q')$ faster than the other algorithms. Thus, it is worth while to have it as an alternative to the other algorithms.

It would be interesting to find criteria for functions to determine which algorithm is preferable. Based on that a procedure should be implemented which selects an algorithm according to the computed function. A similar work has been done in [MKRS00] with respect to the image operation.

Bibliography

[ASS$^+$94] A. Aziz, V. Singhal, T.R. Shiple, A.L. Sangiovanni-Vincentelli, F. balarin, and R.K. Brayton. Equivalences for fair kripke structures. In *ICALP*, LNCS 840, pages 364–375, 1994.

[ASSB94] A. Aziz, V. Singhal, G.M. Swamy, and R.K. Brayton. Minimizing interacting finite state machines: A compositional approach to language containment. In *Proceedings of the International Conference on Computer Design*, pages 255–261, 1994.

[ASSSV94] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincetelly. Formula-dependent equivalence for compositional CTL model checking. In D. Dill, editor, *Proceedings of the Sixth Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 324–337, 1994.

[BBLS91] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulation. In *Computer-aided Verification*, volume 663 LNCS, pages 260–273, 1991.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Int. Conference on Very Large Scale Integration*, 1991.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BG00] D. Bustan and O. Grumberg. Simulation based minimization. In *Conference on Automated Deduction*, volume 17, pages 255–270, 2000.

[BG01] D. Bustan and O. Grumberg. Modular minimization of deterministic finite-state machines. In *6th International Workshop on Formal Methods for Industrial Critical Systems*, pages 163–178, 2001.

[BG02] D. Bustan and O. Grumberg. Applicability of fair simulation. In *TACAS*, LNCS 2280, pages 401–414. Springer, 2002.

[BP96] B. Bloom and R. Paige. Transformational design and implementation of new efficient solution to the ready simulation problem. In *Science of Computer Programming*, volume 24, pages 189–220, 1996.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, volume 131 of *LNCS*. Springer Verlag, 1981.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16, 5:1512–1542, September 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CHJ$^+$90] H. Cho, G. Hachtel, S. Jeong, B. Plessier, E. Shwarz, and F. Somenzi. ATPG aspects of FSM verification. In *ICCAD*, pages 134–137, 1990.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings, Fourth Annual*

Symposium on Logic in Computer Science, pages 353–362, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[CM90a] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Computer Aided Design*, pages 126–129, 1990.

[CM90b] O. Coudert and J.C. Madre. Verifying temporal properties of sequential machines without building their state diagrams. In *Computer Aided Verification*, LNCS 531, pages 23–29, 1990.

[CRFJ96] E.M. Clarke, R.Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Formal Methods in System Design*, pages 77–104, 1996.

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proceedings of Computer-Aided Verification*, volume 531 of *LNCS*, pages 233– 242, 1991.

[Dam94] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 11 April 1994.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.

[DHWT91] D.L. Dill, A.J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relation. In *Computer-Aided Verification*, LNCS 575, pages 255–265, 1991.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science*, pages 368–377, San Juan, Puerto Rico, 1–4 October 1991. IEEE.

[EWS01a] K. Etessami, Th. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for Bchi automata. In *Automata, Languages and Programming, 28th international collquium*, LNCS 2076, pages 694–707, 2001.

[EWS01b] K. Etessami, Th. Wilke, and R. Schuller. Faster algorithms for computing fair simulation relation, and how to use them for state space reduction. Technical Report ITD-01-40643, Bell-Labs, 2001.

[Fer90] J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. In *Science of Computer Programing*, volume 13, 1990.

[Fra76] N. Francez. *The Analysis of Cyclic Programs.* PhD thesis, Weizmann Institute of Science, 1976.

[FV98] K. Fisler and M. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 115–132, 1998.

[GB94] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer Aided Verification*, LNCS 818, pages 299–310, 1994.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[GSL96] Susanne Graf, Bernhard Steffen, and Gerlad Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulation on finite and infinite graphs. In *Proc. Symp. Foundations of Computer Science*, pages 453–462, 1995.

[HKR97] T.A. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Proc. 8th Conference on Concurrency Theory*, LNCS 1234, 1997.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*. Academic Press, New York, 1971.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *In International Federation for Information Processing (IFIP)*, pages 321–332, 1983.

[Jos87] B. Josko. MCTL - an extension of CTL for modular verification of concurrent systems. In *In Workshop on Temporal Logic in Specification, Manchester*, volume LNCS 398, Springer Verlag, 1987.

[KM99] A. Kucera and R. Mayr. Simulation preorder on simple process algebras. In *International Colloquium on Automata, Languages and Programing*, volume 1644 LNCS, 1999.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

[KS90] P.C. Kanellakis and S.A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *Information and computation*, volume 1, pages 43–68, 1990.

[KV96] O. Kupferman and M.Y. Vardi. Verification of fair transition systems. In *Computer Aided Verification (CAV'96)*, LNCS 1102, pages 372–382, 1996.

[KV98] O. Kupferman and M.Y. Vardi. Modular model checking. In *Proc. Compositionality Workshop*, LNCS 1536. Springer-Verlag, 1998.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the 24th ACM Symp. on Theory of Computing*, 1992.

[Lyn96] N.A. Lynch. *Distributed Algorithm.* Morgan Kaufmann Publishers, 1996.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 481–489, London, UK, 1971.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[MKRS00] I. Moon, J. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Design Automation Conf*, pages 23–28, 2000.

[Moo56] E. F. Moore. Gedanken–experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Annals of Mathematics Studies (34), Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*. sv, 1984.

[PT87] R. Paige and R.E. Tarjan. Three partition refinement algorithms. In *SIAM Journal on COMPUTING*, volume 16, 1987.

[Shi96] T.R. Shiple. *Formal Analysis of synchronous circuits.* PhD thesis, University of California at Berkeley, 1996.

[SVW85] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. In *In Proc. 10th Int. Colloquium on Automata, Languages and Programming*, volume LNCS 194, pages 465–474, 1985.