Graph Algorithms for Massive Data-Sets

Gentilini Raffaella

May 10, 2005

To my mother.

Abstract

In this dissertation we face a number of fundamental graph problems which, on the one hand, share applications to the state explosion problem in model checking, and, on the other hand, pose interesting algorithmic questions when very large graphs are dealt with. In more detail, this thesis is divided into three main parts.

Part I (*Symbolic Graph Algorithms*) deals with the algorithmic solution of fundamental graph problems (especially graph connectivity problems) assuming a symbolic OBDD-based graph representation. Working on symbolically represented data has potential: the standards achieved in graph sizes (playing a crucial role, for example, in verification, VLSI design, and CAD) are definitely higher. On the other hand, symbolic algorithm's design generates constraints as well. For example, Depth First Search is not feasible in the symbolic setting and our approach suggests a symbolic framework to tackle those problems which, in the standard case, are naturally solved by a DFS-based algorithm.

Part II (*Equivalence Based Graph Reduction*) deals with well known bisimulation and simulation graph reductions. In particular, our purpose is to show a *fil rouge* connecting such notions, which consists in casting them into partition refinement problems (coarsest partition problems). Such a formulation is well known to be the engine of efficient computational strategies for bisimulation. In this thesis, we extend the picture to the, computationally more challenging, notion of simulation. Besides (we hope) a deeper understanding of the algebraic nature of simulation equivalence, the final byproduct is the definition of a new space efficient simulation algorithm refining a partition rather than a relation.

Part III discusses some applications to model checking and builds connecting bridges among the material introduced in the first two parts of the dissertation. In particular, the strongly connected components symbolic algorithm defined in Part I is used to obtain a symbolic version of an efficient *rank*-based bisimulation reduction, recently defined in [39]. Finally, our symbolic procedure for strongly connected components turns out to be the key to cut down from $\mathcal{O}(V \log V)$ to $\mathcal{O}(V)$ the number *symbolic steps* necessary to perform a number of tasks in model checking as *bad cycle detection* and *Büchi* as well as *Streett* automata language emptiness problems.

Acknowledgments

There are a number of travels whose final destination turns out to be less important than the whole journey. The path that has taken me to write this Ph.D. thesis is such a travel.

I give to Alberto Policriti, my advisor, most of the credit for making these three years as Ph.D. student so significant, and I would like to deeply thank him. He is that kind of person who finds pleasure in sharing his wide knowledge and most of the time I felt more as an equal than as a wild young student. The many scientific and "life" lessons that he patiently tried to teach me have still a lot of new starting points for learning.

If having a constant and trusted guide is fundamental in any challenging trip, having good fellow travellers is also important. I found many of them in these three years and I am very grateful to any one of them. In particular, I would like to thank Nadia Ugel and Toto Paxia for being so helpful in my New York brief stop, and all my co-authors: Agostino Dovier, Carla Piazza and, again, Alberto Policriti.

Finally, I would like to thank my Ph.D. thesis referees, Klaus Schneider and Roberto Sebastiani, for giving me many useful suggestions about the writing of this manuscript, the final effort of my Ph.D. travel.

Contents

Introduction

"640K ought to be enough for anybody"
Bill Gates

The need for tools and algorithmic techniques to deal with very large graphs is likely to be a primary one in the very next future. In fact, many emerging and/or fundamental areas, such as WWW and mobile network modeling, VLSI, hardware verification and geographic information systems, must be able to cope with larger and larger graphs structures. On this ground, for example, a large part of the algorithmic community is nowaday devoted to the study of classical graph algorithms assuming the use of external memory and the I/O-complexity framework [112]. As a matter of fact technological advances are increasing CPU-time speeds at an annual rate of $40 - 60\%$ while disk transfer rates are only increasing of $7 - 10\%$ annually [112]. This is a strong motivation for the above community, however, the same considerations should motivate new research efforts for the algorithmic solution of classical graph problems assuming an *implicit* succinct representation. Implicit representations allow *sets* of nodes (and edges) to partly share their representation (encoding) improving on space requirements: in this framework, graphs can be (hopefully) completely handled in main memory with considerable savings not only in space but also in time resources.

Ordered Binary Decision Diagrams and Implicit Graph Manipulation in Symbolic Model Checking

In this thesis, we consider a type of implicit representation whose benefits are well known for practitioners and researchers in such areas as formal verification, VLSI, and boolean optimization: the *Ordered Binary Decision Diagram* (OBDD) based representation. Ordered Binary Decision Diagrams were originally developed by Bryant [14, 15] as a canonical representation of boolean functions. Basically, OBDDs are a kind of acyclic diagrams resulting from merging isomorphic subgraphs and redundant information in the complete binary decision tree of a boolean function. The original paper of Bryant [14] introducing OBDD, is now one of the few *super* cited papers in the most used scientific literature digital library (1558 citations in CiteSeer database). In fact, many important boolean functions (such as adders, counters, and symmetric functions, see [114, 40, 79] for a survey) have an OBDD representation whose size is polynomial in the number of variables; moreover there are efficient (polynomial in the size of the input OBDDs) procedures for boolean combinations of OBDD represented function and, due to the canonicity of the representation, the comparison operation is done in constant time. However, on the back-stage of the success of the OBDD representation there is another, impressive, application of this kind of representation: OBDDs allowed the automatic verification technique called *model checking*, to be applied to real industrial size protocols and circuits. In *symbolic model checking* [78, 26, 99] the nodes and edges of the structure modeling the tested system are assigned a binary code: on this ground the characteristic boolean functions relative to sets of nodes and edges (graphs) can be OBDD represented. Graphs having an explicit dimension of 10^{20} [16], 10^{100} and more [26] could be handled by symbolic model checking. Notably, the technique of symbolic model checking was endowed of the ACM Kanellakis Award for Theory and Practice in 1998 and many leader companies (IBM, Motorola, Intel, ...) make use of symbolic model checking in their verification process.

Challenges in Symbolic Graph *Algorithmica*

Despite the above mentioned great amount of practical evidence for the effectivness of implicit OBBD based graph structure representations, their use and study is still confined to the computer aided verification and engineering community. As Bryant observed in a recent talk surveying OBDDs and symbolic model checking:

"No people with greatest talent in algorithms work on OBDDs: there are no papers in STOC/FOCS/SODA..."

Bryant–2001

The fundamental dilemma that prevents algorithmics to appreciate the potential of OBDD symbolic computation lies, in our opinion, in the weak assessments about efficiency that can be made to date. In fact the effectiveness of the framework can be nowaday demonstrated only empirically. Worst, the few formal results known seem to be deeply negative: in [45] the PSPACE completeness of some simple problems assuming an OBDD representation is proved (see section 1.3 for a discussion).

Despite this general scepticism within the pure algorithmic community, we think that the many successes in dealing with enormous OBDD represented graphs, coming from the many above cited areas (VLSI, CAD, verification, ... see [40, 26, 79, 99] for more details), can not be reduced to a repository of good examples. Thus, due to the greater and greater importance of the space parameter in algorithmic design, the analysis of algorithms assuming an implicit (OBDD based) representation (*symbolic algorithms*) deserves a deeper investigation. In this context, there are many questions that could be a good starting point for inspiration[1]:

- *Symbolic Algorithmic Design Issue*
  Symbolic OBDD based representations merge as much information as possible among encoding of nodes and edges. The way this is obtained will be clearer in the sequel, however, it is natural that algorithms on symbolically represented graphs must take advantage of this sharing, working on *sets* of nodes and edges rather than single nodes and edges. Many algorithms in the explicit setting do not proceed processing sets of nodes, rather they gain efficiency manipulating

[1] "One moment of inspiration. A life of perspiration." A.Einstein.

and building structures upon each node and edge. In these cases, new algorithmic strategies must be developed to take advantage of having data symbolically represented.

Researchers in the area of model checking faced this problem with respect to specific tasks occurring within their verification algorithms. In this context only a few general graph problems were analyzed in the symbolic setting: the reachability problem (finding all nodes that can be reached from a source node v) and the problem of determining the strongly connected components in a graph [5, 117, 78, 26, 99]. In [115] Ingo Wegener, who developed different variants of OBDD data structures, addressed the problem of analyzing the algorithmic solution of general graph problems assuming an OBDD representation:

"(...) However, we have to look for a new type of algorithms. Loops like – For all $v \in V$ do sequentially – *are intractable. OBDD synthesis work for all $v \in V$ in parallel"*

- *Complexity Issue*
  Which parameters make a symbolic OBDD based algorithm a good algorithm?

  The evaluation of a symbolic algorithm only in terms of its asymptotic complexity, as for classical explicit algorithms, is not satisfactory. In fact, on the one hand the distance between the best case and the worst case is often enormous. For example, breadth first search symbolic analysis of a graph $G = \langle V, E \rangle$ is logarithmic in $|V|$ in the best case and $\Theta(|V|^3)$ in the worst case. On the other hand, there are parameters, as the maximal size of the OBDDs involved in the computation, that must be taken into consideration.

  Today, most authors simply count the number of OBDD operations (number of *simbolic steps*)[5, 78]. Other authors perform a classical complexity analysis of their symbolic algorithm, *but* assuming to work on simple very structured graphs (as grid networks, see [116, 97, 98]).

- *Explicit vs Symbolic Algorithms*
  To date, symbolic algorithms are mostly used when explicit procedures are simply useless because of the prohibitive input structure size. However, in symbolic computation, the more space is saved, the more *time* is saved (combining little OBDDs has a low cost). Are there some problems whose symbolic algorithmic solution is better than the explicit one, assuming an amortized analysis?

The previous items motivate the work in the first part of this thesis and are some of the parameters that should be considered in a wider project, concerning a systematic development of a symbolic counterpart to classical graph algorithmica.

Model Checking, State Explosion Problem, and Graph Reductions

A travel around the challenges of Symbolic Graph Algorithms can look at model checking [26, 99] as both a hopeful departure point and a sure final arrival land.

On the one hand, the success of symbolic model checking inspires the possibility of achieving similar outstanding new standard problems dimensions in other areas dealing with graph computation. On the other hand, model checking can be an application field of symbolic algorithmic solution to classical graph problems.

To this purpose, it must be said, however, that symbolic graph manipulation does not resolve the so called notorious *state explosion problem*, which is somehow inherent in the model checking approach. In the literature, many other tools to deal with the modeling structures' dimensions have been developed, either in alternative or in addition to symbolic computation. Roughly, these techniques can be divided in two categories. The first category includes methods exploiting compositional reasoning [60]. The second one relies on abstraction: models are reduced before any kind of reasoning takes place on them (see, for example, [33] for a survey). Abstraction techniques can be further split into *strongly preserving reductions* and *weakly preserving reductions*, where strong and weak refer to the way properties (expressed in a given temporal logic language $\mathcal{L}$) are preserved upon reductions. If only the truth of $\mathcal{L}$ properties is preserved within the abstraction, we have weak preservation. Otherwise, we have strong preservation. Weak preservation techniques are mainly based on abstract interpretation refinements. Strong preservation techniques, instead, rely on the definition of equivalence relations on the graph models vertex-sets. In [77] the bisimulation equivalence is recognized as preserving the whole μ-calculus language [72]. In [77, 25], the simulation equivalence is proved to preserve the universal fragment of the μ-calculus as well as of $\forall$CTL* and $\forall$CTL logics [26, 99]. As a matter of fact, bisimulation and simulation are the most used equivalences in the above reduction context as the complexity of computing them is a small polynomial (while, for example, the complexity for language equivalence is PSPACE-complete [102]).

Scope and Main Contributions of the Thesis

In this thesis, we deal with most of the above outlined topics. Broadly, we face a number of fundamental graph problems which, on the one hand, share applications to the state explosion problem in model checking, and, on the other hand, pose interesting algorithmic questions when very large graphs are dealt with.

More in detail, the graph problems discussed in the first part of this thesis are difficult to be solved when a symbolic data representation is assumed. Thus, part one of the dissertation can be view as the attempt of moving some preliminary steps toward the development of a symbolic counterpart to classical graph algorithms. Graph decomposition in its strongly connected components is exactly the first graph problem approached in this context. Apart from model checking, where it can be used to enhance bad cycle detection [5], this problem finds applications in many fields where large graphs are manipulated as, for example, WWW analysis [13]. For this reasons it is largely studied and recognized to pose challenges, both in the external memory framework [112] and assuming a symbolic OBDD-based representation. As far as the symbolic setting is concerned, difficulties rely on the fact that depth first search graph traversal is not applicable in a symbolic setting [104, 117, 5], since it can not be adapted to work on sets of nodes. Hence, the well known (depth first search based) efficient strongly connected components algorithm by Tarjan [106] can not be rephrased symbolically. In Tarjan's procedure the final DFS node labelling defines an order that efficiently drives the computation of strongly connected components. In this thesis we propose *spine-sets* as a symbolic counterpart to DFS for connectivity graph analysis. Spine-sets allow to obtain strongly connected components of a symbolic graph within $\mathcal{O}(|V|)$ *symbolic steps* improving on previous results [117, 5]. The role of spine-sets as a counterpart to DFS in the symbolic setting, is enhanced by the possibility of reusing the above notion to devise an $\mathcal{O}(V)$ symbolic steps algorithm for biconnected components analysis.

In the second part of the dissertation, we consider systematically *bisimulation* and *simulation* equivalence based graph reduction, from a (classical) algorithmic point of view. In particular, our purpose is to show how there is a *fil rouge* connecting such notions which consists in casting them into partition refinement problems (coarsest partition problems). Such a formulation is well known to be the engine of efficient computational strategies to compute bisimulation reductions [67, 84, 70]. As an original contribution, we extend the picture to the, computationally more challenging, notion of simulation. Casting the simulation problem into a *generalized coarsest partition* problem seems to shed some new light on the concrete nature of this kind of graph reduction. Moreover, it is the first step along a path that allows us to define a novel efficient simulation algorithm that gains space resources in determining a partition, rather than a relation, on graph nodes.

The third final part of this thesis discusses some applications, in model checking, of the material previously introduced. Within the model checking framework, it is also possible to outline a number of links relating part I and part II of the dissertation. In particular, as an original contribution, we develop a symbolic version of the rank-based efficient bisimulation reduction algorithm, recently introduced in [39]. The key tool of *rank*, inspired by set theory and fundamental for the complexity of [39], is proved to be symbolically computable either upon our symbolic SCC algorithm or with a specialized symbolic procedure. Our symbolic SCC procedure is finally shown to be the engine of new symbolic algorithms, checking (various types of) ω-automata language emptiness: these procedures cut down, from $\mathcal{O}(V \log V)$ to $\mathcal{O}(V)$, the number of symbolic steps necessary to check language emptiness of (generalized) Büchi and Streett automata [5].

Structure of the Dissertation

We outline here the structure of the dissertation and the origin of each chapter. As previously anticipated, this thesis is divided into three main parts:

Part I: *Symbolic Graph Algorithms*

- In Chapter 1, we present the Binary Decision Diagram (OBDD) data structure and we review the main results of the literature about OBDD based graph representation and manipulation. We state here also an opening discussion on challenges, limits, and difficulties underling both the design and the analysis of symbolic graph algorithms.

- In Chapter 2, we develop our symbolic algorithm to compute, within $\mathcal{O}(V)$ symbolic steps, a graph decomposition in its strongly connected components. The work presented in Chapter 2, that improves previous procedures requiring $\mathcal{O}(V^2)$ [117] and $\mathcal{O}(V \log(V))$ symbolic steps [5], is partly based on [55].

- In Chapter 3, we develop a symbolic approach to biconnected components computation. We finally show how to take advantage of spine-sets to enhance the overall computational strategy, obtaining an algorithm that computes biconnected components within $\mathcal{O}(V)$ symbolic steps. The results in this chapter are partly based on [57].

Part II: ***Equivalence Based Graph Reductions***

- In Chapter 4, we present the state of the art concerning both algebraic characterizations and computational strategies for obtaining bisimulation and simulation reductions.

- In Chapter 5, we encode the simulation problem into a (generalized) coarsest partition problem. On this ground, we develop a new simulation algorithm that improves either in the time or in the space resources used, on previously defined procedures [56]. The results of Chapter 5 have been partly presented in [53, 56].

Part III: ***Applications to Model Checking***

- In Chapter 6, we give the reader an overview on model checking verification technique, that we consider to discuss some applications of the material introduced in the first two parts of the thesis.

- In Chapter 7, we present a symbolic version of the rank-based bisimulation reduction algorithm developed in [39]. The results in this chapter are partly based on [37] and build a bridge linking some material introduced in the first and in the second part of the dissertation.

- In Chapter 8, we collect a number of tasks, in symbolic model checking that can take advantage of the performances of our SCC symbolic algorithm.

I

Symbolic Graph Algorithms

1

Preliminaries: OBDDs and Symbolic Graph Algorithms

In this chapter, we present the Binary Decision Diagram (OBDD) data structure and we review the main results of the literature about OBDD based graph representation and manipulation. We state here also an opening discussion on challenges, limits, and difficulties underling both the design and the analysis of symbolic graph algorithms.

1.1 Ordered Binary Decision Diagrams (OBDDs)

Ordered Binary Decision Diagrams (OBDDs) [14, 15] were introduced in 1986 by Bryant, as a canonical representation for boolean functions. An OBDD represents a boolean function as a rooted, acyclic, directed, and labeled graph. As an example, we represent in Figure 1.1, on the right, the OBDD of the boolean function

$$f = x_1 \wedge (x_2 \iff x_3)$$

with respect to the variable order $\pi = \langle x_1 < x_2 < x_3 \rangle$. Each branch defines a set of variables (truth value) assignments, and the leaf labeling gives the corresponding function value. The rightmost branch, for example, corresponds to the computation paths for $f(0\_\_)$, all evaluating to 0. The OBDD for f maintains exactly the same information of the ordered binary decision tree in Figure 1.1, on the left; however, it avoids redundant information like isomorphic subgraphs and useless test nodes. More precisely, the OBDD data structure can be defined upon the transformation rules below, that apply to ordered binary decision trees (see Definition 1.1.1). Let $\pi = \langle x_1, \ldots, x_n \rangle$ be a linear order over the set of variables of a boolean function f.

Definition 1.1.1 (Ordered Binary Decision Tree) *The* Ordered Binary Decision Tree *of the boolean function $f : \{0,1\}^n \mapsto \{0,1\}$, with respect to the variable order $\pi = \langle x_1, \ldots, x_n \rangle$, is the complete labeled binary tree in which:*

- *each internal node v, at depth i, is labeled $var(v) = x_i$ and has two outgoing edges labeled* 0 *(dotted edge) and* 1 *(not dotted edge), respectively.*

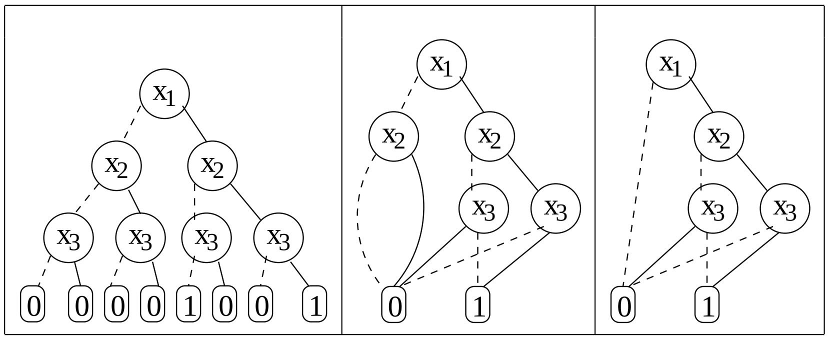


Figure 1.1: The ordered binary decision tree, an unreduced OBDD and the OBDD for $f = x_1 \wedge (x_2 \iff x_3)$ and $\pi = \langle x_1, x_2, x_3 \rangle$

- *each leaf is labeled with a value in* $\{0, 1\}$
- *for all* $\bar{x} \in \{0, 1\}^n$ *and* $z \in \{0, 1\}$*,* $f(\bar{x}) = z$ *if and only if the only path of edges labeled* $\bar{x}$ *in the tree ends onto a leaf whose label is* z*.*

In the transformation rules below, we call $low(v)$ the node pointed by the edge having label 0 and leaving vertex v. We call $high(v)$ the node pointed by the 1 labeled edge leaving v.

1. *Remove Duplicate vertices*

   If the vertices u and v are such that $var(u) = var(v)$, $low(u) = low(v)$, and $high(u) = high(v)$, then eliminate one of the two vertices and redirect all incoming edges to the other vertex.

2. *Remove Redundant Tests*

   If the nonterminal vertex v has $low(v) = high(v)$, then eliminate v and redirect all incoming edges to $low(v)$.

Applying the above rules to an ordered binary decision tree, T, an arbitrary number of times we obtain a diagram structure called *unreduced OBDD* (see Figure 1.1). Once no reduction is possible, any more, we have an OBDD.

Definition 1.1.2 (Ordered Binary Decision Diagram (OBDD)) *The* Ordered Binary Decision Diagram *of the boolean function* $f : \{0, 1\}^n \mapsto \{0, 1\}$*, with respect to the variable order* $\pi = \langle x_1, \ldots, x_n \rangle$*, is the acyclic directed labeled graph obtained by removing all duplicate vertices and redundant tests from the* π *ordered binary decision tree of* f*.*

The idea of avoiding redundancy, in a decision tree, by the transformation rules introduced above cames back to Lee and Akers [74, 3, 83]. However, at the beginning these ideas were not constrained to any variable ordering and result in structures computationally hard to deal with. In [14] Bryant proved that, by fixing a variable ordering, it was possible to obtain a *canonical* boolean function representation. Besides canonicity, the ordering over variables was also the engine of computationally efficient composition procedures.

1.1.1 The Variable Order Problem

The size and the form of an OBDD representing a boolean function strongly depends on the chosen variable ordering. As an example, consider the function:

$$g_6 = (x_1 \iff x_2) \vee (x_3 \iff x_4)$$

with respect to the two variable orderings π_1 and π_2, where $\pi_1 = \langle x_1, x_2, x_3, x_4 \rangle$ and $\pi_2 = \langle x_1, x_3, x_2, x_4 \rangle$. In the first case the OBDD has 6 internal nodes (see Figure 1.2, on the left). In the second case, instead, the OBDD number of internal nodes explodes to 9 (see Figure 1.2, on the right).

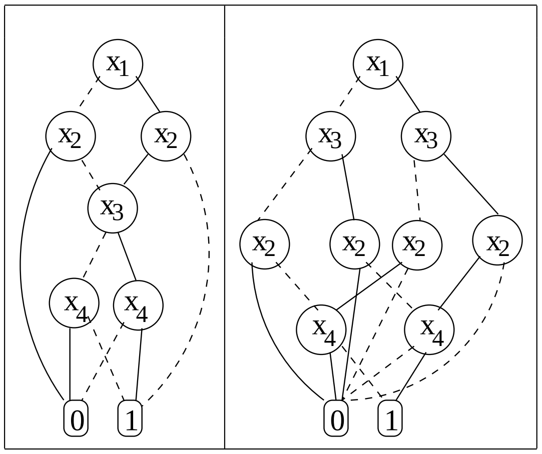


Figure 1.2: The OBDDs representing $(x_1 \iff x_2) \vee (x_3 \iff x_4)$ with respect to the variable orderings $\pi_1 = \langle x_1, x_2, x_3, x_4 \rangle$ and $\pi_2 = \langle x_1, x_3, x_2, x_4 \rangle$, respectively

In general, the OBDD representing the function $g_{2n} = (x_1 \iff x_2) \vee \ldots \vee (x_{2n-1} \iff x_{2n})$ has a size which is linear, with respect to n, if the underling variable ordering is $\pi_1 = \langle x_1, x_2, x_3, \ldots, x_{2n} \rangle$; it exposes instead an exponential size, with respect to the number of variables, if the chosen variable ordering is $\pi_2 = \langle x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n} \rangle$. The intuition, underling the above behavior, is that the first n levels of the OBDD having exponential size must form a complete binary tree: in fact, the result of g_{2n} is completely unpredictable if the value of none even variable is known. Such an intuition is formalized by Lemma 1.1.3, below, due to Wegener and Sieling [114, 100]. We use the notation $f\mid_{x_i=0}$ to indicate the boolean function $f(x_1, \ldots, x_n) \wedge (x_i = 0)$ and we say that a function f essentially depends on x_i if $f\mid_{x_i=0} \neq f\mid_{x_i=1}$.

Lemma 1.1.3 *Let G be the OBDD for $f : \{0,1\}^n \mapsto \{0,1\}$ with respect to the variable ordering $\langle x_1, \ldots, x_n \rangle$. Let S_i be the set of different subfunctions $f\ |_{x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ which essentially depend on x_i, where $c_1, \ldots, c_{i-1} \in \{0,1\}$. For each function $g \in S_i$ the OBDD G contains exactly one node labeled x_i which represent g. The OBDD G does not contain further internal nodes.*

In the terminology of [114] functions like the one presented in Figure 1.2, for which there exist both polynomial and exponential size OBDDs, are called *well behaved*. There exist functions that have only exponential size OBDDs: these functions are referred to as *bad functions* in [114]. As an example, the boolean functions representing either of the middle two outputs of an n bit multiplier have exponential size OBDDs, as proved by Bryant in [14, 15]. A further class of boolean function is that of *good functions* which have polynomial size OBDDs for any variable ordering. An example of such kinds of functions is given by symmetric functions, whose value depends only on the number of arguments equal to 1 [15].

In [9] Bollig and Wegener proved that the problem of finding the best ordering, with respect to the OBDD size of a given function, is NP-complete. Later, in [100] Sieling proved that the best variable ordering problem is hard to approximate, also. Namely, for a boolean function f and for $\alpha \geq 1$, let us call a variable ordering π an α-approximation, if the OBDD-size for f and π is larger than the minimum OBDD size for f, by a factor of at most α. For each constant $\alpha \geq 1$ the computation of α-approximations is NP-hard [100]. Hence, heuristics are the only possibility to obtain good variable orderings. To date many of such heuristics have been introduced in literature and successfully implemented in many OBDD packages: we refer to [114, 40] for more details on the topic.

1.1.2 OBDD Construction and Composition

Building the OBDD representation of a boolean function, as well as performing boolean composition, involves the same core OBDD operation: the Apply operation [14, 15]. The Apply operation takes as input two argument OBDDs, G_f and G_g, representing two boolean functions, f and g, and a boolean operation to perform their composition (e.g. And, Or, Nand, Xor). The result of computing Apply$(G_f, G_g, \langle op \rangle)$ is the OBDD representing $f \langle op \rangle g$.

The Apply algorithm relies on the *shannon expansion* of a boolean function around a variable x:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1})$$

Any operation of boolean composition, $\langle op \rangle$, commute with shannon expansion i.e. it holds:

$$f \langle op \rangle g = (\neg x \wedge f|_{x=0} \langle op \rangle g|_{x=0}) \vee (x \wedge f|_{x=1} \langle op \rangle g|_{x=1})$$

Thus, it is possible to devise a recursive procedure which compute $f \langle op \rangle g$ while depth first traversing the input OBDD arguments. More precisely, the computation starts on the couple of vertices of the input OBDDs, say m and m', and recursively performs one of steps below. The completion of each step produces a node, $\langle m, m' \rangle$ in the OBDD for $f \langle op \rangle g$.

- If m and m' are sinks labels $a, b \in \{0, 1\}$, then $\langle m, m' \rangle$ is a sink node with label $a\langle op \rangle b$;
- if m and m' are internal nodes labeled with the same variable x_i, then $\langle m, m' \rangle$ is a node having:
  - $var(\langle m, m' \rangle) = x_i$;
  - $high(\langle m, m' \rangle)$ equal to the result of Apply$(high(m), high(m'), \langle op \rangle)$;
  - $low(\langle m, m' \rangle)$ equal to the result of Apply$(low(m), low(m'), \langle op \rangle)$.
- otherwise, if m, m' are labeled with different variables, assume, without loss of generality that $var(m) > var(m')$. Then, $\langle m, m' \rangle$ is a node such that:
  - $var(\langle m, m' \rangle) = var(m')$;
  - $high(\langle m, m' \rangle)$ equal to the result of Apply$(high(m), m', \langle op \rangle)$;
  - $low(\langle m, m' \rangle)$ equal to the result of Apply$(low(m), m', \langle op \rangle)$.

To prevent multiple recursive evaluations on the same couple of OBDDs subgraphs, an hash table, called the *computed table*, is used. The computed table keeps trace of each couple of nodes that causes a recursive call to be done, together with the output OBDD node produced. This way, whenever a couple of nodes in the computed table is touched, recursive calls on subgraphs are prevented and the OBDD node in the corresponding table position is returned. Hence, the time complexity of Apply gets bounded by $\mathcal{O}(\sum_{\langle m,n \rangle \in V_f \times V_g} |E_f^{-1}(m)||E_g^{-1}(n)|) = \mathcal{O}(|G_f||G_g|)$, where $G_f = \langle V_f, E_f \rangle$ and $G_g = \langle V_g, E_g \rangle$. A second hash table, called *unique table* prevents to produce an unreduced OBDD. The unique table maintains triples of the kind $\langle x_i, R_0, R_1 \rangle$, that refer to existing OBDD nodes having variable labeling equal to x_i, and pointing with the 0 labeled edge (1 labeled edge) to R_0 (R_1).

The Apply algorithm allows to implement a number of derived boolean functions, also. In particular:

- The complement $\bar{f}$ of a boolean function can be computed as the xor of f and the constant **1**. A more efficient implementation consists in redirecting to 1 sink each edge linked to a 0 sink, and viceversa.
- The restriction of a function around one of its variables can be defined as $f|_{x=a} = f \wedge (x = a)$. A more efficient implementation involves a unique depth first traversal of the input OBDD. Such a visit redirects each edge reaching a node labeled x to $low(v)$, if $a = 0$, or $high(v)$ if $a = 1$; moreover it uses a unique table to avoid having an unreduced OBDD in output.
- Universal quantification around a variable, $\forall x f$ can be computed as $f|_{x=0} \wedge f|_{x=1}$.
- Existential quantification around a variable, $\exists x f$ can be computed as $f|_{x=0} \vee f|_{x=1}$.

By the canonicity of the OBDD representation, the operation of comparing two functions consists in checking if the pointer to their OBDD representation is the same.

As far the reduction of an unreduced OBDD, G_f, is concerned, an $\mathcal{O}(V_f \log V_f)$ algorithm is presented in [14, 15]. Later, Wegener improved the result by presenting an $\mathcal{O}(V_f)$ Reduce algorithm [101]. Finally, we briefly recall the possibility of efficiently performing both satisfiability tests of OBDD represented functions, and the count of the number of inputs evaluating to 1 (SAT-count). To perform a satisfiability test, it is sufficient to check if there is some edge linked to the 1-sink. To count the number of paths leading to a 1-sink, an $\mathcal{O}(|G_f|)$ algorithm can be designed, relying on the following considerations. If $N(w)$ paths lead to the OBDD node w, then half of these paths goes on along $low(w)$ and the other half follows $high(w)$. Thus, it is sufficient to initialize $N(w) = 0$ for all nodes in G_f but for the root, r, that is assigned $N(r) = 2^n$. The values $N(w)$ are then updated upon a levelwise top-down visit of G_f. The final label N, that comes to be associated to the 1-sink, represents the number of inputs evaluating to 1. We refer to surveys in [15, 114, 40, 104] for more details on the OBDD operations described in this section.

We conclude by recalling that various packages have been developed to manipulate OBDDs: Somenzi's CUDD at Colorado University [105], Lind-Nielsen's BuDDy, Biere's ABCD package, Janssen's OBDD package from Eindhoven University of Technology, Carnegie Mellon's OBDD package, the Berkeley's CAL [96], K. Milvang-Jensen's parallel package BDDNOW, and Yang's PBF package. All these packages are endowed with a number of built-in operations such as equality test and the boolean operations described throughout this section.

1.2 Symbolic Graph Representation and Manipulation via OBDDs

Any notion or structure that can be encoded in terms of boolean functions can be represented by means of OBDDs. OBDDs were used to deal with hard combinatorial problems, relying on the boolean representations of variables and constraints [73]. In verification, OBDDs were used to encode the graph structure resulting from the modeling process. The way sets and graphs can be expressed in terms of boolean functions was first suggested by Bryant [14, 15]. Later, Ken McMillan proposed symbolic model checking i.e. a model checking framework in which the computation is completely performed on OBDD represented graph structures. In this context basic operations to deal with graph analysis, as computing the successors of a given node, were studied.

1.2.1 OBDD based Graph Representation

Consider a graph $G = \langle V, E \rangle$. G can be represented by means of OBDDs relying on the following considerations:

- Each node is encoded as a binary number i.e. $V = \{0, 1\}^v$. Hence, a set $U \subseteq$

V is a set of binary strings of length v whose characteristic (boolean) function, $\chi_U(u_1, \ldots, u_v)$, can be represented by an OBDD;

- $E \subseteq V \times V$ is a set of binary strings of length $2v$ whose characteristic (boolean) function, $\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v)$, can be represented by an OBDD.

Hence, in the best case the OBDD representation of a graph has a size that is linear with respect to the $\log(E)$ variables necessary to encode E. In the worst case the size of an OBDD represented graph explodes to $\Theta(V + E)$.

Example 1.2.1 Consider the graph represented in Figure 1.3. To give an OBDD representation of such a graph, we need to encode its nodes with a 3-variables boolean code, $\langle x_1, x_2, x_3 \rangle$. Accordingly, 6 variables are necessary, namely $\langle x_1, x_2, x_3, x'_1, x'_2, x'_3 \rangle$, to give a symbolic representation of the graph relation. We depict in Figure 1.3, on

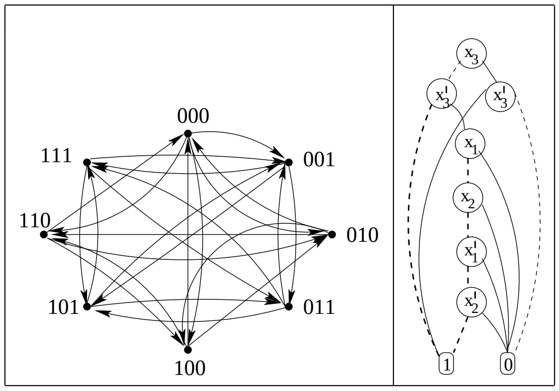


Figure 1.3: A graph and the OBDD representation of its relation, with respect to the variable ordering $\langle x_3, x'_3, x_1, x_2, x'_1, x'_2 \rangle$

the right, the OBDD encoding of the graph relation, with respect to the variable ordering $\langle x_3, x'_3, x_1, x_2, x'_1, x'_2 \rangle$. Note that, using the alternative variable ordering $\langle x_1, x'_1, x_2, x'_2, x_3, x'_3 \rangle$, we would obtain an OBDD having exponential size with respect to the number of variables. In fact, the first three levels of the resulting OBDD would be a complete binary tree.

In [98], Sawitzki, discusses a number of graph topologies for which there exists a variable ordering that guarantees an OBDD representation having logarithmic size with respect to the graph size. ■

1.2.2 Computing on Symbolic Graphs

Symbolic computation on OBDD represented graphs is based on symbolic manipulation of sets of nodes and edges. The following operations on sets of nodes and edges are usually considered as basic graph operations in symbolic computation [26, 104]:

- boolean set-composition operations. We will use the standard notation $\cup, \cap \ldots$ to denote set-composition operations;

- image (post) and pre-image (pre) computation: given a direct graph $G = \langle V, E\rangle$ and a set of nodes $A \subseteq V$, $\mathsf{post}(A)$ and $\mathsf{pre}(A)$ are defined as follows:

  $$\mathsf{post}(A) = \{v \in V \mid v \in E(A)\} \quad \mathsf{pre}(A) = \{v \in V \mid v \in E^{-1}(A)\}$$

  In an undirected graph the operations pre and post get to be equivalent and, in this context, we will use the notation img.

- In [5, 104] the operation of extraction of a singleton from a set of graph nodes, $\mathsf{pick}(A)$, $A \subseteq V$, is used;

- graph relation composition.

Each of the above mentioned basic graph operations is implemented as a single OBDD operation. More precisely, using the APPLY procedure it is possible to perform each boolean operation of set-composition. For example, the union of two OBDD represented sets of nodes, A, B, is implemented as $\textsc{Apply}(A, B, \vee)$ in time $\mathcal{O}(|A||B|)$. Given a symbolic set, A, extracting a singleton is simply implemented by retrieving an OBDD path evaluating to 1.

The basic task of obtaining the set of nodes on the adjacency list of a given (set of) vertices is performed using the so called RELATIONAL PRODUCT OBDD operation [26, 78, 104]. Composing relations and, hence, computing on sets of edges is also realized through relational products. Unlike the OBDD operations implementing set-composition, the relational product can cause an exponential blow up of the manipulated OBDDs. More precisely, while APPLY complexity is polynomial in the sizes of the input OBDDs, the relational product complexity is exponential in the *number of variables* of the manipulated OBDDs [15, 78, 104].

The Relational Product

Consider a set of nodes in $G = (V, E)$ and assume to have it represented by means of an OBDD $A(x_1, \dots, x_n)$. Let $E(x_1, \dots x_n, y_1, \dots y_n)$ to be the OBDD representing the graph relation. Then, the set of nodes on the adjacency list of at least one element in $A(x_1, \dots, x_n)$, is given by the following boolean function

$$\exists x_1, \dots x_n (A(x_1, \dots, x_n) \wedge E(x_1, \dots, x_n, y_1, \dots, y_n))$$

More precisely, the above boolean function evaluates to 1, on the free variables $y_1, \dots, y_n$, if and only if the code of a node in the adjacency list of at least one element in $A(x_1, \dots, x_n)$ is given. The general operation consisting in a combination of conjunction and existential quantification is called relational product.

It is possible to compute a relational product, $\exists \bar{x}(P(\bar{y})\langle op\rangle Q(\bar{z}))$, by first building the OBDD for $P(\bar{y})\langle op\rangle Q(\bar{z})$, and then quantifying on each required variable. However, this way it is necessary to define an OBDD that depends on a number of variables greater than the final result. This overload is of a double (number of variables) in the special case of determining the set of nodes linked to a (set of) vertices: thus the worst case size OBDD occurring overall the computation is quadratic with respect to the size of the output OBDD. For these reasons a special algorithm, RelProd, was developed in [78], that never build intermediate OBDDs depending on the quantified variables. The pseudocode for the algorithm RelProd is given in Figure 1.4.

```
RelProd($f, g$ : OBDD represented boolean functions, $X$ : set of variables)

(1)  if ($f = \mathbf{0} \vee g = \mathbf{0}$) then return $\mathbf{0}$;
(2)  if ($f = \mathbf{1} \wedge g = \mathbf{1}$) then return $\mathbf{1}$;
(3)  if ($f, g, r$) is in the computed table then return $r$;
(4)  let $x$ be the topmost variable between $f$ and $g$ top variables
(5)  $r_0 = $ RelProd($f|_{x=0}, g|_{x=0}, X$);
(6)  $r_1 = $ RelProd($f|_{x=1}, g|_{x=1}, X$);
(7)  if $x \in E$ then let $r = r_0 \vee r_1$
(8)      else let $r = (x \wedge r_1) \vee (\neg x \wedge r_0)$;
(9)  insert ($f, g, r$) into the computed table;
(10) return $r$
```

Figure 1.4: The Relational Product algorithm

The procedure works upon a combined depth first visit of the two input OBDDs, similarly to the Apply operation. Thus, each recursive call is associated to a couple of nodes of the input OBDDs. If the leading labelling variable, say x_i, is quantified, then an OBDD node is created: such node creation depends on an execution of Apply subprocedure on two (recursively built) OBDDs, that do not depend on x_i (line 7).

As some recursive calls need an Apply composition (lines 7–8), it seems not possible to amortize a constant time to each couple of input OBDD nodes. In fact, in [78], it is proved that, unless $P = NP$, quantifying a vector of $\mathcal{O}(n)$ variables, in an OBDD that depends on n variables, requires a number of steps exponential in n. A simple bound to the complexity of RelProd is $\mathcal{O}(|C|2^{2(n-k)})$, where $|C|$ is the size of the produced final OBDD and k is the number of quantified variables. The bound simply assess that there are at most $|C|$ recursive calls, each one of cost at most $\mathcal{O}(2^{2(n-k)})$: in fact the maximum sizes of the OBDDs built, overall the computation, is $\mathcal{O}(2^{2(n-k)})$. Note that no better bound is known, to date, and that performing composition and quantification in two separate steps results in a procedure having the same (time) complexity.

1.3 Symbolic Graph Algorithms: State-of-the-Art and Opening Discussion

To take advantage of OBDD-synthesis, symbolic graph algorithms should operate on *sets* of nodes and/or edges rather than on single graph elements. In fact, on the one hand, this way each OBDD operation hopefully processes many nodes and/or edges in parallel. On the other hand, for example, computing the vertices on the adjacency list of one node has the same worst case complexity as obtaining the vertices linked to a set of nodes. The above characteristic should be somehow inherent to a proper definition of symbolic algorithm and poses strict constraint on the possibility of rearranging classical algorithm in the symbolic setting.

Apart from the agreement on this general suggestion [115, 104, 6], there is no definite assessment, in the literature, of what a *good* symbolic algorithm is, on how evaluating complexity of symbolic procedures, as well as on how comparing symbolic algorithms.

1.3.1 Complexity of Symbolic Graph Algorithms

A natural question that arises when algorithmically approaching graph problems, assuming a symbolic data representation, is the following: which is the complexity of the graph problem with respect to the dimension of the input symbolic representation? This question has been considered in [45], for very simple graph problems. In the next subsection we show how it deserves an answer that, one one hand sounds bad but, on the other hand, is rather natural and leaves open a number of ways to proceed.

The PSPACE Completeness Barrier [45]

In [45] the GAP problem is considered; namely, given a directed graph G and two nodes, s, t, the GAP problem simply asks if there is a path in G from s to t. The authors of [45] show that it is possible to encode as a GAP problem each decision problem on a polynomial space bounded Turing machine. The graph underling the encoding represents all possible configurations and transitions of the Turing machine and it is shown to have a succinct, polynomial in the number of variables, OBDD representation: hence the GAP problem is PSPACE complete with respect to the dimensions of a symbolic input representation.

However, if the number of nodes of the manipulated (explicit) graph are considered, the above result simply assess that it is not possible to use OBDDs to define polylogarithmic algorithms for GAP as well as more complex graph problems. It remains possible to devise OBDD-based algorithms that solve always polinomially the above problems and "often" with sublinear space and/or time. In fact, these characteristics are shared by all the symbolic graph algorithms defined, since now, in the verification field. It follows that, beyond the PSPACE barrier, a number of questions as well as of promising research perspective remain open:

- Having large distance between worst and best case for most symbolic graph algorithms defined in the verification field, classical worst case analysis is not satisfiable (consider, for example, exactly the GAP problem which is polylogarithmic in the best case and polynomial in the worst case, with respect to the dimensions of the explicit represented graph). Which parameters have to be considered to compare symbolic algorithms?

- Are there classes of graphs for which GAP, or more complex graph problems, result polylogarthmic assuming an OBDD representation?;

- Are there classes of graph problems whose worst case (polynomial) complexity is the same assuming both an explicit and a symbolic representation (thus, the middle case space and time complexities are better for symbolic representations)?

Comparing Symbolic Algorithms: Discussion on Current Approaches

Even if there is no assessment, in the literature, with respect to how evaluating and comparing symbolic algorithms, some directions and parameters have been isolated. In this subsection, we start discussing them and we finally give the framework we use in this thesis, to evaluate and comparing symbolic algorithms.

Most authors in the verification and model checking community, evaluate a symbolic algorithm counting the number of OBDD operations performed [78, 48, 16, 26]. In [5, 104] the notion of *symbolic steps* was introduced, capturing the fact that symbolic operations based on relational product can cause an explosion of the input OBDD size. Thus, the asymptotic number of symbolic steps, i.e. of relational product operations, is taken as a measure of symbolic algorithms performance. Indeed, each computation that processes an OBDD on n variables with $\log(n)$ set-composition operations, can get to a worst case size OBDD. Thus, the symbolic steps approach is worthwhile as long as the number of packages of $\log(n)$ consecutive set-composition operations, manipulating the same OBDD, does not grow too much.

Indeed, comparing symbolic procedures exclusively on the ground of bounds on the number of general, as well as specific, OBDD operations is not fair: this way, in fact, the sizes of the manipulated OBDD sizes is not taken in account anyway. To consider OBDD sizes also, it is possible to use, as a performance parameter, the number of variables of intermediate OBDDs manipulated, also. The importance of having the variables number as low as possible was recognized by various authors [104, 48, 6]. As a matter of fact multiplying by a constant k the number of variables on which an OBDD depends, means exploding its worst case size, say S, to S^k. The algorithms defined overall this thesis will use at most $2n$ variables, where n is the number of variables needed to encode the graph vertex set. $2n$ variables allow to compute on sets of nodes using pre and post operations and *leaving untouched* the relation of the graph. In a certain sense, hence, this is the minimum number of variables needed to get some information from the graph relation.

In [97, 98, 116] Sawitzki and Woelfel do not pose any strict constraint to the number of variables in the computed OBDD. As an example the topological algorithm in [116] requires $4n$ variables and relies on computing the transitive closure of the input graph relation. The purpose in [97, 98, 116] is, on the one hand, that of exploiting the power of graph relation composition to which it is possible to apply speedup strategies as *iterative squaring* [16]. On the other hand, the author in [97, 98, 116] avoid the problems related to the likely huge intermediate OBDD sizes, largely experimentally recognized in the verification community [5, 104, 109], by imposing strict constraint on their graph topology (very regular graphs as grids are considered). Computing on so regular graphs guarantees to have all intermediate OBDDs of size polynomial in the number of variables: hence, the complexity analysis is carried on in the classical way and the costs obtained are polylogarithmic.

Comparing Symbolic Algorithms: Parameters used in this Thesis

In this dissertation, we take deeply into account suggestions coming from the great deal of experimental work done in symbolic model checking. In fact, on the one hand we do not want to pose too much strict constraints on graph topologies dealt with. On the other hand, we want our algorithms to be applicable in verification.

Following [78, 26, 104, 5, 48], our graph connectivity symbolic algorithms will work on sets of nodes, rather than edges, thus minimizing the number of variables in intermediate OBDDs (as well as their worst case sizes).

Fixed the number of variables at disposal, we will use the asymptotic number of *symbolic steps*, that we define as OBDD single operations (rather than special operations as relational products [5]) to compare algorithms. This way, we would be able to compare our procedure with symbolic algorithms for the same purpose previously defined [117, 5, 42].

1.3.2 Symbolic Graph Algorithms

We now turn out to present the symbolic graph algorithms developed, to date, in the literature. As we already observed, since symbolic graph algorithms should work on sets of graph elements, it is not always possible to translate classical algorithms into efficient symbolic procedures. The simple problem of exploring a graph is significant to this purpose. Symbolically, the problem can be efficiently tackled in a breadth first manner: sets of nodes having increasing distance from the source-node are considered and the diameter of the graph is a bound on the number of symbolic steps necessary to complete the discovering. To perform depth first search (DFS) instead, you need computing an order over nodes. This order is associated to depth of nodes in the paths subsequently discovered. Thus, it is necessary to maintain an overall (discovering time) vertex-labelling and to subsequently take into consideration a node at a time (according to such a labelling). As a result, DFS is intractable in a symbolic setting [104, 5]. Recently, a graph-problem that, in the explicit setting, is linearly solved starting from a DFS visit, has been tackled in the area of verification. Motivated by the possibility of speeding up some model checking algorithms, the authors of [117, 5] proposed two symbolic algorithms to find the strongly connected components of a directed graph. Both the algorithms compute the strongly connected components of a node, v, by intersecting two, breadth first discovered, vertex-sets: the set of nodes having a path to v (backward-set) and the set of nodes that can be reached by v (forward-set). Moreover, both algorithms need the same number of variables, in that they both manipulate sets of nodes: the procedure in [5] outperforms the one in [117] in that it needs $\mathcal{O}(V \log(V))$ rather than $\mathcal{O}(V^2)$ symbolic steps.

The designing of the symbolic graph algorithms for *bisimulation* reduction has also been proposed in the area of model checking [12, 47, 26, 99]. The first symbolic bisimulation algorithms [12, 26] obtained the bisimulation equivalence over the graph vertex set as a minimal fix-point relation, starting from the universal relation. Later, [48] proposed bisimulation symbolic algorithms that do not manipulate relations but, rather, vertex-sets partitions: this way the number of variables involved in the computation is cut in half.

All the above symbolic procedures were deeply experimentally analyzed on a data set derived from graphs modeling systems in verification. A data set of random graphs, besides that of graphs deriving from model checking, was used to test the algorithm in [62]. Such procedure solves, symbolically, the maxflow problem on 0-1 networks on graphs having more than 10^{30} nodes.

Finally, the authors in [97, 98, 116], developed a number of symbolic graph algorithms (topological sorting [116], maxflow on 0-1 networks, all pair shortest paths problem [98]) all relying on symbolic computation of transitive closure of graph relation. Transitive closure computation is well known, in model checking, to likely lead to huge intermediate OBDDs. Hence, even if it can be computed with only $\log(n)$ iterations by *iterative squaring* it is avoided in model checking. However, the purpose of [97, 98, 116] is that of posing strict constraint to graph topologies (very regular graphs as grids are considered) so as to guarantee succinct intermediate OBDDs and polylogarithmic procedures.

2 DFS and Strongly Connected Components in a Symbolic Setting

Strongly Connected Components (SCC) decomposition is a fundamental graph problem. Applications of strongly connected components analysis are well known in contexts where massive graphs are manipulated as WWW [13, 112] and model checking [5, 117]. The classical SCC algorithm by Tarjan [106] can not be translated in frameworks used, in the above contexts, to deal with massive data-sets as external memory and symbolic (OBDD based) algorithms. In this chapter, we face exactly the problem of symbolically computing the strongly connected components of a given graph, enhancing on previous works [117, 5]. More broadly, our results will prove the key for efficient translating, in the symbolic framework, those graph (connectivity) algorithms that are classically solved upon a graph depth first search traversal.

2.1 Strongly Connected Components in the Explicit and Symbolic Frameworks

In this section we briefly recall necessary basic concepts concerning the strongly connected components problem. Further, we review existing computational strategies (and challenges) to solve the SCC problem assuming classical and symbolic graph representations.

2.1.1 The SCC problem

A digraph $G = \langle V, E\rangle$ is *strongly connected* if, for any pair of nodes (v, u), v and u are mutually reachable in G ($v \leftrightsquigarrow u$). Given a digraph $\langle V, E\rangle$ the relation of mutual reachability $\leftrightsquigarrow$ is an equivalence relation over V.

Definition 2.1.1 (Strongly Connected Components) *The* strongly connected components *of $G = \langle V, E\rangle$ are the elements of the partition on V induced by the relation of mutual reachability.*

The SCC problem consists in determining the strongly connected components of a given digraph $G = \langle V, E\rangle$. We will use the notation $scc_G(v)$ (or simply $scc(v)$) to refer to the strongly connected component of v in $G = \langle V, E\rangle$. $scc_G(v)$ is said to be *trivial* if it is equal to $\{v\}$.

Definition 2.1.2 (Scc-Closed Vertex-Set) *Given $G = \langle V, E\rangle$, let $U \subseteq V$. U is said to be* scc-closed *if, for all vertices $v \in V$, either $scc(v) \cap U = \emptyset$ or $scc(v) \subseteq U$.*

It is immediate to see that boolean combinations of scc-closed sets are scc-closed. Lemma 2.1.4, whose proof is immediate (see [117]), relates some of the above defined notions and ensures the correctness of the symbolic scc-algorithms in [117, 5].

Definition 2.1.3 (Backward and Forward Sets) *Let $G = \langle V, E\rangle$ be a digraph and $U \subseteq V$ be a set of nodes. We define the* backward set *of U, denoted by $BW_G(U)$, as the set of nodes that reach U. Conversely, we define the* forward set *of U, denoted by $FW_G(U)$, as the set of nodes reachable from U.*

Lemma 2.1.4 *Let $G = \langle V, E\rangle$ be a digraph. Consider the subgraph $G' = \langle U, E \restriction U\rangle$ where $U \subseteq V$ is scc-closed. For all $v \in U$, both $FW_{G'}(v)$ and $BW_{G'}(v)$ are scc-closed and*

$$scc_G(v) = FW_{G'}(v) \cap BW_{G'}(v)$$

2.1.2 Explicit vs Symbolic SCC Computation

Using explicit data structures (as adjacency lists) for graph representation, the SCC problem can be efficiently solved by the well known Tarjan algorithm [106]. Tarjan's algorithm decomposes a digraph, $G = \langle V, E\rangle$, in its strongly connected components within time (and space) $\Theta(V + E)$. The key for obtaining an optimal complexity, in [106], is that of having an *ordering* that drives SCC computation. Such an ordering is devised upon the finishing labeling of a depth first graph traversal.

If DFS visit is the engine of efficiency in a classical explicit framework, it turns out to be a source of problems when the symbolic framework is considered. In fact, on the contrary of BFS traversal, DFS visit can not be adapted to work symbolically on sets of nodes and needs proceeding upon a sequential discovering, and labeling, of each graph vertex.

In symbolic model checking, SCC decomposition is a useful preprocessing step for many verification algorithms [5, 117]. Hence, a number of authors try to devise efficient symbolic computational strategies for SCC analysis. The first symbolic algorithm devised dates back to 1995 [16], and relies on symbolic computation of the transitive closure of the graph relation. Computing the transitive closure of a graph is largely experimentally proved, in model checking [117, 109, 104, 5], to suffer of huge intermediate OBDD sizes: indeed it relies on manipulating and composing a relation and, hence, needs the double of variables than an OBDD algorithm that simply manipulates sets of nodes.

In [117] an SCC algorithm manipulating sets of nodes, leaving untouched the graph relation, is devised. The simple algorithm in [117] is based on Lemma 2.1.4. In each iteration, a node v is chosen at random. Both the backward set, $BW(v)$, and the forward-set, $FW(v)$, are then computed by a breadth first traversal, and finally intersected to obtain $scc(v)$. The number of symbolic steps used in [117] is $\mathcal{O}(V^2)$. In [5] an SCC symbolic algorithm improving on [117] is proposed. Namely, the procedure in [5] uses a sort of *process the smallest half* strategy to obtain strongly connected components decomposition within $\mathcal{O}(|V|\log(|V|)$ symbolic steps.

No symbolic SCC algorithm, to date, uses any ingredient providing an ordering that drives computation (whereas classical Tarjan SCC procedure takes advantage exactly of DFS-based nodes ordering to obtain linear performances). In this chapter we try cover this gap introducing a special technique, the *spine-sets*, that serves as a symbolic counterpart of DFS ordering and allow to define a symbolic $\mathcal{O}(V)$ SCC procedure.

2.2 Spine-Sets: a Symbolic Counterpart to Depth-First Search Visit

We introduce here the notion of *spine-sets*, that will result central in our framework. A spine-set allows to implicitly encode an ordering, suitable for the efficient symbolic computation of strongly connected components. In this sense, spine-sets are, within the design of connectivity algorithms, a sort of symbolic counterpart to the use of DFS.

A path $(v_0, \dots, v_p)$ in a graph G is said a *chordless path* if and only if for all $0 \leq i < j \leq p$ such that $j - i > 1$, there is no edge from v_i to v_j in G.

Definition 2.2.1 (Spine-set) *Consider a graph G having vertex-set V and edge-set E. Let $\mathcal{S} \subseteq V$. The pair $\langle \mathcal{S}, v \rangle$ is a* spine-set *of G if and only if G contains a chordless path with vertex-set $\mathcal{S}$ that ends at v. The node v is called the* spine-anchor *of the spine-set $\langle \mathcal{S}, v \rangle$.*

It is immediate to verify that a spine-set is associated to a unique chordless path. On this ground, we use the notation $\overline{v_0 \dots v_p}$ to express the fact that $\langle \{v_0, \dots, v_p\}, v_p \rangle$ is a spine-set of G associated to the chordless path $(v_0, \dots, v_p)$. Though simple, Lemma 2.2.2 is significant in that it allows to view a spine-set as an *implicitly ordered set*.

Lemma 2.2.2 *If $\overline{v_0 \dots v_p}$ and $p > 0$, then $E^{-1}(v_p) \cap \{v_0, \dots v_p\} = \{v_{p-1}\}$ and $\overline{v_0 \dots v_{p-1}}$*

Proof. Each subpath of a chordless path is clearly a chordless path. Hence, $\overline{v_0 \dots v_{p-1}}$ and $v_{p-1} \subseteq E^{-1}(v_p) \cap \{v_0, \dots v_p\}$. The expression $v_{p-1} \supseteq E^{-1}(v_p) \cap \{v_0, \dots v_p\}$ also holds in that, otherwise, there would be some edge from a spine-set node v_j to v_p, with $p - j > 1$. ■

Subsection 2.2.1 relates spine-sets to the strongly connected components of a digraph.

2.2.1 Spine-Sets and Strongly Connected Components

Consider digraph $G = \langle V, E \rangle$.

Lemma 2.2.3 *Given the spine-set* $\overline{v_0 \ldots v_p}$ *in* $G = \langle V, E \rangle$*, there is a minimum* $0 \leq t \leq p$ *and a maximum* $0 \leq l \leq p$ *such that:*

1. $scc(v_p) \cap \{v_0, \ldots, v_p\} = \{v_t, \ldots, v_p\} \ \wedge \ scc(v_0) \cap \{v_0, \ldots, v_p\} = \{v_0, \ldots, v_l\}$;

2. *if* $t \neq 0$*, then* $\overline{v_0 \ldots v_{t-1}}$ *and* $\{v_{t-1}\} = E^{-1}(scc(v_p) \cap \{v_0, \ldots, v_p\}) \cap \mathcal{S}$*;*

3. *if* $l \neq p$*, then* $\overline{v_{l+1} \ldots} v_p$*.*

Proof.

1. Let $scc(v_p)$ to contain the spine-set node v_j. We prove that, for all $j < k < p$, $v_k \in scc(v_p)$. Let $j < k < p$. By definition of spine-set, there is a path from v_k to v_p. By $v_j \in scc(v_p)$, there is a path from v_p to v_j and hence, using again the definition of spine-set, there is a path from v_p to v_k.

   An analogous reasoning schema allows to conclude that there exists a maximum $0 \leq l \leq p$ such that $\mathrm{scc}(v_0) \cap \{v_0, \ldots, v_p\} = \{v_0, \ldots, v_l\}$.

2. Follows from Item 1, from definition of spine-set, and from the fact that each subpath of a chordless path is a chordless path.

3. Follows from Item 1 and from the fact that each subpath of a chordless path is a chordless path.

■

By the above lemmas, in a digraph the nodes of a spine-set can be assigned to their strongly connected components in the order induced on them by the spine-set. In the next section we state an analogous result in the framework of the biconnected components of an undirected graph. This comes not by chance and depends on the fact that the order induced by a spine-set gives information on the *depth* of the spine-set nodes in a path of the graph. Both the symbolic connectivity algorithms presented in the rest of this thesis, use spine-sets to *drive* the computation on opportune breadth-first discovered scc-closed (bcc-closed) subgraphs. Thus, despite the vertex-set is always explored in a breadth first search manner, globally, the strongly connected components (biconnected components) get produced in a *piecewise depth first order*. In the symbolic scc-algorithm, the scc-closed sets computed are forward-sets of a spine-anchor. Some important properties about such sets are stated in Lemma 2.2.4.

Lemma 2.2.4 *Let* $\langle \mathcal{S}, u \rangle$ *be a spine-set in* $G = \langle V, E \rangle$ *and* $scc(\langle \mathcal{S}, u \rangle) = \bigcup_{w \in \mathcal{S}} scc(w)$*. Then,* $FW(u) \cap scc\,(\langle \mathcal{S}, u \rangle) = scc\,(u)$*.*

Proof. Clearly $FW(u) \cap \mathrm{scc}(\langle \mathcal{S}, u \rangle) \supseteq \mathrm{scc}(u)$. To prove the opposite inclusion, notice that since both $FW(u)$ and $\mathrm{scc}(\langle \mathcal{S}, u \rangle)$ are scc-closed, $FW(u) \cap \mathrm{scc}(\langle \mathcal{S}, u \rangle)$ is scc-closed. Let w be such that $\mathrm{scc}(w) \subseteq FW(u) \cap \mathrm{scc}(\langle \mathcal{S}, u \rangle)$. We prove that $\mathrm{scc}(w) = \mathrm{scc}(u)$. From the fact that $\mathrm{scc}(w) \subseteq FW(u)$, we have that w is reachable from u. By $\mathrm{scc}(w) \subseteq \mathrm{scc}(\langle \mathcal{S}, u \rangle)$, there exists w' such that $w' \in \mathcal{S}$ and $\mathrm{scc}(w) = \mathrm{scc}(w')$. u is reachable from w' since $w' \in \mathcal{S}$ and $\langle \mathcal{S}, u \rangle$ is a spine-set. By $\mathrm{scc}(w) = \mathrm{scc}(w')$, we conclude that u is reachable from w. ■

The *skeleton of a forward set*, introduced below, relates spine-sets and forward-sets. It is practically a particular spine-set that will be used in our symbolic scc-algorithm to drive the computation.

Definition 2.2.5 (Skeleton of $FW(v)$) *Let $FW(v)$ be the forward-set of the vertex $v \in V$. $\langle \mathcal{S}, u \rangle$ is a skeleton of $FW(v)$ iff u is a node in $FW(v)$ whose distance from v is maximum and $\mathcal{S}$ is the set of nodes on a shortest path from v to u.*

Lemma 2.2.6 *Let $FW(v)$ be the forward-set of $v \in V$. If $\langle \mathcal{S}, u \rangle$ is a skeleton of $FW(v)$, then $\langle \mathcal{S}, u \rangle$ is a spine-set in $G = \langle V, E \rangle$.*

Proof. It follows immediately from the definition of spine-set. ■

2.3 Strong Connectivity on Symbolic Graphs

We focus here on strong connectivity. We show how the notions introduced in Section 2.2 allow us to design a symbolic scc-algorithm performing a linear number of symbolic steps. We start by giving some intuitions about the procedure we are going to arrange. In each iteration the strongly connected component of a node, v, is simply determined by first computing $FW(v)$ and then identifying those vertices in $FW(v)$ having a path to v. The choice of the node to be processed in any given iteration is driven by the implicit order associated to an opportune spine-set. More specifically, whenever a forward-set $FW(v)$ is built, a skeleton of such a forward-set, $\overline{v_0 \ldots v_p}$, is also computed. The order induced by the skeleton is then used for the subsequent computations. Stated in other words, $scc(v_p)$ will be the first strongly connected component isolated in the scc-closed subset $FW(v) \setminus scc(v)$. In this way, the symbolic steps performed to produce $FW(v)$ are distributed over the computation of the strongly connected components of the nodes in a skeleton of $FW(v)$. This amortized analysis is the key point for the linear complexity of the algorithm.

2.3.1 From Spine-Sets to a Linear Symbolic Strongly Connected Components Algorithm

With this intuition, we outline in Figure 2.1 the pseudocode for our linear symbolic scc-algorithm. The parameters of the procedure depicted in Figure 2.1 are a graph $\langle V, E \rangle$ and a pair $\langle \mathcal{S}, N \rangle$. $\langle \mathcal{S}, N \rangle$ is either $\langle \emptyset, \emptyset \rangle$ or $\mathcal{S} = \{v_0, \ldots, v_p\} \subseteq V$, $N = \{v_p\}$, with $\overline{v_0 \ldots v_p}$ (i.e. $\langle \{v_0, \ldots, v_p\}, v_p \rangle$ is a spine-set in $\langle V, E \rangle$).

In case V is empty the routine terminates, otherwise the vertex for which the next strongly connected component is computed is chosen.

```
SYMBOLIC-SCC($V, E, \langle \mathcal{S}, N \rangle$)

(1)  if $V = \varnothing$ then return;

– Determine the node for which the scc is computed –
(2)  if $\mathcal{S} = \varnothing$ then $\mathrm{N} \leftarrow Pick(\mathrm{V})$;

– Compute the forward-set of the the singleton N and a skeleton –
(3)  $\langle FW, New\mathcal{S}, NewN \rangle \leftarrow$ SKEL-FORWARD($V, E, N$);

– Determine the scc containing N –
(4)  $SCC \leftarrow N$;
(5)  while $((\mathsf{pre}(SCC) \cap FW) \setminus SCC) \neq \varnothing$ do
(6)          $SCC \leftarrow SCC \cup (\mathsf{pre}(SCC) \cap FW)$;

– Insert the scc in the scc-Partition –
(7)  $SCC$-Partition $\leftarrow$ $SCC$-Partition $\cup\, SCC$

– First recursive call: computation of the scc's in $V \setminus FW$ –
(8)  $V' \leftarrow V \setminus FW$;  $E' \leftarrow E \upharpoonright V'$;
(9)  $\mathcal{S}' \leftarrow \mathcal{S} \setminus SCC$;  $N' \leftarrow \mathsf{pre}(SCC \cap \mathcal{S}) \cap (\mathcal{S} \setminus SCC)$;
(10) SYMBOLIC-SCC($V', E', \langle \mathcal{S}', N' \rangle$)

– Second recursive call: computation of the sccs in $FW \setminus SCC$ –
(11) $V' \leftarrow FW \setminus SCC$;  $E' \leftarrow E \upharpoonright V'$;
(12) $\mathcal{S}' \leftarrow New\mathcal{S} \setminus SCC$;  $N' \leftarrow NewN \setminus SCC$;
(13) SYMBOLIC-SCC($V', E', \langle \mathcal{S}', N' \rangle$)
```

Figure 2.1: The scc-algorithm performing a linear number of symbolic steps.

In case ($\mathcal{S} \neq \emptyset$ and) $N = \{v_p\}$, v_p is chosen. Otherwise ($\mathcal{S} = \emptyset$) an arbitrary element $v \in V$ is picked (assigning the singleton $\{v\}$ to N). Then the subprocedure SKEL-FORWARD is invocated to compute the forward-set of the singleton N and a skeleton, $\langle \mathcal{S}', u' \rangle$, of such a forward-set. The local variable FW maintains the just mentioned forward-set whereas $New\mathcal{S}$ and $NewN$ maintain $\mathcal{S}'$ and $\{u'\}$, respectively. In line (4) the local variable SCC is initialized to be the singleton N and then it is augmented with the strongly connected component containing N (loop of lines (5)-(6)). In line (7) the partition of scc's is updated and finally the procedure is recursively called over:

1. the subgraph of $\langle V, E\rangle$ induced by $V \setminus FW$ and the spine-set of such a subgraph obtained from $\langle \mathcal{S}, N\rangle$ by subtracting SCC (cf. Item 2 in Lemma 2.2.3);

2. the subgraph of $\langle V, E\rangle$ induced by $FW \setminus SCC$ and the spine-set of such a subgraph obtained from $\langle New\mathcal{S}, NewN\rangle$ by subtracting SCC (cf. Item 3 in Lemma 2.2.3).

In Figure 2.2 the pseudocode of the subprocedure SKEL-FORWARD is presented. It is used within the execution of SYMBOLIC-SCC to obtain the forward-set of a node together with a skeleton of such a forward-set. The parameters of such a routine are

```
SKEL-FORWARD $(V, E, N)$

(1)  Let stack be an empty stack of sets of nodes
(2)  $L \leftarrow N$

– Compute the Forward-set of $N$ and push onto stack the onion rings –
(3)  while $(L \neq \emptyset)$ do
(4)      push(stack, $L$);
(5)      $FW \leftarrow FW \cup L$;
(6)      $L \leftarrow$ post($L$) $\setminus FW$;

– Determine a Skeleton of the Forward-set of $N$–
(7)  $L \leftarrow$ pop(stack);
(8)  $\mathcal{S}' \leftarrow N' \leftarrow$ pick($L$);
(9)  while stack $\neq \emptyset$ do
(10)     $L \leftarrow$ pop(stack);
(11)     $\mathcal{S}' \leftarrow \mathcal{S}' \cup$ pick(pre($\mathcal{S}'$) $\cap L$);
(12) return$\langle FW, \mathcal{S}', N'\rangle$;
```

Figure 2.2: The procedure for computing the forward-set of a node v together with a skeleton.

a graph $G = \langle V, E\rangle$ and a singleton $N = \{v\} \subseteq V$. The forward-set of the node in input, v, is simply computed with a symbolic breadth first search [104, 93] i.e. by a loop that discovers, in each iteration i, all the nodes of V having distance i from v. In [93] the just mentioned sets are referred as *onion-rings* and are enqueued onto a set-priority-queue to produce a counterexample of minimum length. In this context, they are pushed onto a stack to produce a skeleton of $FW(v)$.

Notice that the restriction of the transition relation in the two recursive calls is only for the sake of clarity. As done in [5] the results of the image and pre-image computations are intersected with V', so that we store in memory only the original transition relation.

2.3.2 Soundness and Complexity Results

The soundness and completeness of the algorithm in Figure 2.1 are stated in Theorems 2.3.3 and 2.3.4, respectively. Throughout this section, we will use the notation V_c^l to indicate the variable V on line l within the c-th execution of SYMBOLIC-SCC. V_c will denote the value of the parameter V to the c-th call to SYMBOLIC-SCC. An analoguous notation will be adopted for all the variables in SYMBOLIC-SCC. Lemma 2.3.1, below, states that the subprocedure SKEL-FORWARD$(V, E, \{v\})$ computes the forward-set of v and a skeleton of $FW(v)$.

Lemma 2.3.1 *Let $G = \langle V, E\rangle$. Given $v \in V$, the procedure* SKEL-FORWARD$(V, E, \{v\})$ *returns the triple of sets $\langle FW(v), \mathcal{S}', \{u\}\rangle$ where $\langle \mathcal{S}', u\rangle$ is a skeleton of $FW(v)$.*

Proof. Upon the termination of each iteration, i, of the first loop in SKEL-FORWARD, it holds that

- L maintains the set of nodes at distance i from $\{v\}$;
- L is pushed onto the stack of vertex-sets, `stack`, and it is united to FW.

It follows that the first loop in SKEL-FORWARD is executed d_v times, where d_v is the maximum distance from v, of a vertex in V. Upon the termination of such a loop, FW maintains the forward-set of $\{v\}$. Moreover, `stack` has length d_v and w belongs to the set in position i of `stack` if and only if w has distance i from v. Thus line (8) assigns, to both N' and $\mathcal{S}'$, a singleton $\{u\}$, where u is a node having maximum distance from v. Then, the while-loop of lines (9)-(11) augment $\mathcal{S}'$ with the set of nodes on a path of minimum distance from v to u. By Definition 2.2.5, we conclude that $\langle \mathcal{S}', u\rangle$ is a skeleton of $FW(v)$. ■

Lemma 2.3.2 *Consider the execution of* SYMBOLIC-SCC *on $(V, E, \langle \emptyset, \emptyset\rangle)$ on $\langle V, E\rangle$. In each recursive call c, to* SYMBOLIC-SCC*:*

(a) $\langle V_c, E_c\rangle$ is a subgraph of $\langle V, E\rangle$ and V_c is scc-closed.

(b) $N_c^3 = \{v\} \subseteq V$ and if $\mathcal{S}_c \neq \emptyset$, then $\langle \mathcal{S}_c, N_c\rangle$ is a spine-set of $\langle V_c, E_c\rangle$.

(c) $scc_G(v)$ is assigned to SCC_c^7 and $\langle SCC_c^7, V_c^7 \setminus FW_c^7, FW_c^7 \setminus SCC_c^7\rangle$ is a partition over V_c.

Proof. By induction on the number of recursive calls to SYMBOLIC-SCC within the execution of SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset\rangle)$. The base case is immediate. We assume our lemma true for the first c recursive calls and we sketch here the inductive step. Let SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'}\rangle)$ be the recursive call from which the $c + 1$ call to SYMBOLIC-SCC is executed (hence $c' \leq c$).

Inductive Step (Item (a)) V_{c+1}^7 is either $V_{c'}^7 \setminus FW_{c'}^7$ or $FW_{c'}^7 \setminus SCC_{c'}^7$. Hence, by inductive hypothesis, $\langle V_{c+1}, E_{c+1}\rangle$ is a subgraph of $\langle V, E\rangle$ and V_{c+1} is a scc-closed subset of V.

Inductive Step (Item (b)) The values of $\mathcal{S}_{c+1}$ and N_{c+1} depend on whether the $c + 1$-th call to SYMBOLIC-SCC corresponds to the first or to the second recursive call within SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'}\rangle)$. In the first case, if $\mathcal{S}_{c'} = \emptyset$, then line (9) initializes $\langle \mathcal{S}^{c+1}, N^{c+1}\rangle$ to $\langle \emptyset, \emptyset\rangle$. Otherwise, by inductive hypothesis, $\langle \mathcal{S}_{c'}, N_{c'} = \{v\}\rangle$ is a spine-set in $\langle V_{c'}, E_{c'}\rangle$ and $SCC^7_{c'} = scc_G(v)$. Hence, by Lemma 2.2.3 (item 2) and Lemma 2.2.4, lines (8)-(9) in SYMBOLIC-SCC initialize $\langle \mathcal{S}_{c+1}, N_{c+1}\rangle$ to a spine-set in $\langle V_{c+1}, E_{c+1}\rangle$. In the second case, the inductive hypothesis, Lemma 2.2.3 (item 3), Lemma 2.2.6, and Lemma 2.3.1, ensure that lines (11)-(12) in SYMBOLIC-SCC initialize $\langle \mathcal{S}_{c+1}, N_{c+1}\rangle$ to a spine-set in $\langle V_{c+1}, E_{c+1}\rangle$.

Inductive Step (Item (c)) By Lemma 2.3.1 and by inductive hypothesis, we have that the strongly connected component of v in G is assigned to SCC^7_{c+1} and the forward-set of v in $\langle V_{c+1}, E_{c+1}\rangle$ is assigned to FW^7_{c+1}.

Thus, $\langle SCC^7_{c+1}, V^7_{c+1} \setminus FW^7_{c+1}, FW^7_{c+1} \setminus SCC^7_{c+1}\rangle$ is a partition over V_{c+1}. ∎

Theorem 2.3.3 (Soundness) *Consider $G = \langle V, E\rangle$. If $SCC \subseteq V$ is added to $SCC\_Partition$ within* SYMBOLIC-SCC*$(V, E, \langle \emptyset, \emptyset\rangle)$, then SCC is a scc of G.*

Proof. Follows directly from Lemma 2.3.2 and Lemma 2.1.4. ∎

Theorem 2.3.4 (Completeness) *Consider $G = \langle V, E\rangle$. If $v \in V$, then upon the termination of* SYMBOLIC-SCC*$(V, E, \langle \emptyset, \emptyset\rangle)$, all the sccs of G belong to scc_Partition.*

Proof. By Lemma 2.3.2, in each recursive call, SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$, within SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset\rangle)$:

- $V_c \subset V$;
- a strongly connected component of G, $scc(v)$, is computed;
- SYMBOLIC-SCC is recursively called over $(V', E', \langle \mathcal{S}', N'\rangle)$ and $(V'', E'', \langle \mathcal{S}'', N''\rangle)$, where $\langle V', V'', scc(v)\rangle$ is a partition over V.

The thesis follows immediately. ∎

By Lemma 2.3.5, below, each node can be inserted at most two times into a spine-set, within the execution of SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset\rangle)$. Lemma 2.3.5 is preliminary to Theorem 2.3.6 about the complexity of SYMBOLIC-SCC.

Lemma 2.3.5 *Consider $G = (V, E)$. Within* SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset\rangle)$*, each $v \in V$ is inserted at most two times into a spine-set.*

Proof. Consider the c-th recursive call, SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$, within the execution of SYMBOLIC-SCC$(V, E, \emptyset, \emptyset)$. We proceed by induction on the number of recursive calls and we prove that, $\forall v \in V_c$,

(a) If $v \in V_c \setminus \mathcal{S}_c$, then v has never been inserted into a spine-set within the first $c-1$ recursive calls;

(b) if $v \in \mathcal{S}_c$, then v has been inserted exactly one time into a spine-set;

(c) within lines (1)–(7) of the procedure SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$, $v \in V_c$ is inserted into a spine-set at most one time;

(d) if the node $v \in \mathcal{S}_c$ is inserted into a spine-set within lines (1)–(7) of the procedure SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$, then $SCC^7_c = scc(v)$.

The thesis follows immediately from items (a)–(d). We sketch here the proof of the inductive step for items (a)–(d) (the base case is immediate).

Consider $v \in V_c$, with $c > 1$. Item (c) follows from the fact that a node can be inserted into a spine-set only within line (4) of SKEL-FORWARD; by Lemma 2.3.1, the sets considered on such lines are mutually disjoint.

To prove the inductive step for Items (a)–(b), assume that the c-th recursive call to SYMBOLIC-SCC was called within the c'-th execution of the procedure. If $c = c'+1$, then, by Lemma 2.3.2, $V_c = V_{c'} \setminus FW(u)$ and $\mathcal{S}_c = \mathcal{S}_{c'} \setminus scc(u)$, where $\{u\} = N^3_{c'}$. Hence, Items (a)–(b), for the inductive step, follow directly from the inductive hypothesis and from the fact that the nodes inserted into a spine-set, within the c'-th execution of SKEL-FORWARD, belong to $FW(u)$. Otherwise ($c > c'+1$), the c-th execution of SYMBOLIC-SCC corresponds to the second recursive call within SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle))$. In this case, note that the two vertex-sets considered by the two recursive calls within SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle)$ are disjoint. Hence, for all c'' with $c' < c'' < c$, v can not be inserted into a spine-set within SYMBOLIC-SCC$(V_{c''}, E_{c''}, \langle \mathcal{S}_{c''}, N_{c''} \rangle)$ (as $v \notin V_{c''}$). By Lemma 2.2.3, $FW(N^3_{c'}) \cap \mathcal{S}_{c'} = scc(N^3_{c'})$ i.e. $V_c = FW(N^3_{c'}) \setminus scc(N^3_{c'})$ is disjoint from $\mathcal{S}_{c'}$. Moreover, the nodes in $\langle \mathcal{S}_c, N_c \rangle$ are nodes inserted into a spine within the c' recursive call, and not yet assigned to their scc. Thus, by inductive hypothesis, we have the validity of items (a)–(b) for the inductive step.

We finally obtain the inductive step for item (d). By Lemma 2.3.1, if $\langle \mathcal{S}_{c'}, N_{c'} \rangle$ is not empty spine-set, then $\langle New\mathcal{S}^3_c, NewN^3_c \rangle$ is a spine set in $FW(N_{c'})$. Hence, by Lemma 2.2.4, $\mathcal{S}_{c'} \cap New\mathcal{S}^3_c \subseteq scc(N^3_c)$ and we obtain the validity of item (d) for the c-th recursive call. ■

By Lemma 2.3.5, the linear number of symbolic steps performed by SYMBOLIC-SCC is immediate. In fact, each symbolic step in the loop of lines (5)–(6) of SYMBOLIC-SCC assigns a set of nodes to its scc. Hence the global number of these symbolic steps is $\mathcal{O}(|V|)$. The remaining symbolic steps, within SKEL-FORWARD, are proportional to the number of insertions into a spine-set. By Lemma 2.3.5 the number of these insertions is bounded by $\mathcal{O}(|V|)$. Notice that counting boolean operations as well as symbolic steps would not change the asymptotic complexity of our algorithm.

Theorem 2.3.6 (Complexity) *Consider $G = \langle V, E \rangle$.* SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$ *performs $\mathcal{O}(|V|)$ symbolic steps.*

Proof. Lines (1)–(2) and (7)–(13) of SYMBOLIC-SCC contain a constant number of symbolic steps. Since each recursive call produces a strongly connected component, lines (1)–(2) and (7)–(13) of SYMBOLIC-SCC perform, globally, $\mathcal{O}(|V|)$ symbolic steps. Each symbolic step in the loop of lines (5)–(6) in the main procedure assigns a set of nodes to its scc. Hence the global number of these symbolic steps is $\mathcal{O}(|V|)$. The remaining symbolic steps are performed within the procedure SKEL-FORWARD. Within an execution of SKEL-FORWARD, the number of symbolic steps is proportional to the number of sets popped from the stack upon the second loop termination. Whenever a stack is popped from the stack, an insertion into a spine is performed. Hence, by

Lemma 2.3.5, the global cost (in term of symbolic steps) of executing SKEL-FORWARD within SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$, is linear in the size of V. ■

Figure 2.3 shows the relative sizes of the computed forward-sets and scc's . $FW(v_1)$ and $scc(v_1)$ are already computed when SYMBOLIC-SCC$(V, E, \{v_l, \ldots, v_p\}, v_p)$ is called and subsequently, $FW(v_p)$ is computed and $scc(v_p)$ is given in output. It is also

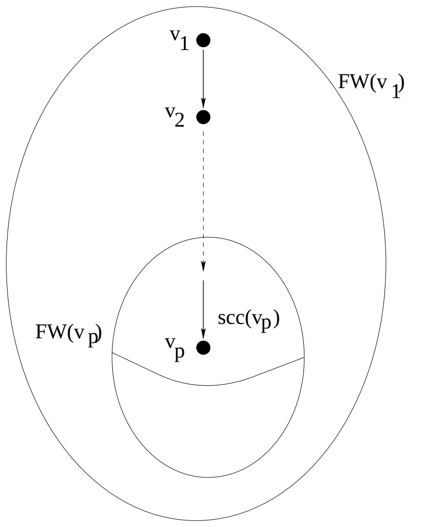


Figure 2.3: Relative sizes of the computed forward-sets.

possible to express the complexity of the algorithm in Figure 2.1 in terms of the diameter of $G = \langle V, E \rangle$, d_G, and of the size of the *scc*-partition, N_G. In detail, since in each iteration of SYMBOLIC-SCC a scc is detected through a forward-set computation, the complexity expression $\mathcal{O}(\min(|V|, N_G * d_G))$ is immediately obtained.

We finally observe that several heuristics to optimize the implicit scc algorithm in Figure 2.1 are possible. In particular, the set $T \subseteq V$ of nodes belonging to trivial scc's which do not reach any not-trivial scc could be quickly determined by the following fix-point pre-computation: while $(V \neq pre(V))$ do $\{V \leftarrow pre(V);\ T \leftarrow (V \setminus pre(V)) \cup T\}$. Each node in T is a trivial scc of the graph in input, thus a non-expensive preprocessing determining T *a-priori* (alternative to an explicit enumeration of each node in T within the main procedure), would make the algorithm in Figure 2.1 somehow “more symbolic”.

3

A Symbolic Approach to Biconnected Components Computation

In this chapter we reuse the notion of spine-sets, introduced in Chapter 2, to devise a symbolic algorithm for biconnected components graph decomposition in $\mathcal{O}(V)$ symbolic steps. The result enhances the role of spine-sets as a symbolic counterpart to DFS graph traversal.

3.1 Biconnected Components in the Explicit and in the Symbolic Framework

In this section we briefly recall necessary basic concepts concerning the biconnected components problem and its algorithmic solution within classical and symbolic frameworks.

3.1.1 The BCC Problem

The notion of *biconnectivity* is the counterpart, in an undirected graph, of the notion of strong connectivity.

Consider $G = (V, E)$ and, from now on, assume that $|E| \geq 1$, G is connected, and contains no self loops. A vertex $a \in V$ is said an *articulation point* of G if there exist two distinct vertices $v \neq a$ and $w \neq a$ such that every path between v and w contains a. Stated another way, a is an articulation point if the removal of a splits G into two or more parts. $G = (V, E)$ is *biconnected* if it contains no articulation point. The biconnected components of $G = (V, E)$ can be defined starting from the following equivalence relation $\circlearrowleft$ on E.

Definition 3.1.1 *Given $G = (V, E)$, consider two edges $(v, u) \in E$ and $(w, z) \in E$. (v, u) and (w, z) are in relation $\circlearrowleft$ if and only if either they are the same edge or there is a simple cycle in G containing both (v, u) and (w, z).*

Definition 3.1.2 (Biconnected Components) *Given $G = (V, E)$, consider the partition of $E, \langle E_1, \ldots, E_k \rangle$, induced by $\circlearrowleft$. For each $1 \leq i \leq k$, let V_i be the set of endpoints of the set of edges E_i. The biconnected components of G are the subgraphs $(V_1, E_1), \ldots (V_k, E_k)$.*

In the rest of this thesis, we use the notation $bcc_G(v, u)$ (or simply $bcc(v, u)$) to indicate the biconnected component containing edge (v, u) in $G = (V, E)$. Lemma 3.1.3, proved in [2], gives useful information on biconnectivity.

Lemma 3.1.3 *For $1 \leq i \leq k$, let $G_i = (V_i, E_i)$ be the biconnected components of $G = (V, E)$. Then:*

1. *G_i is biconnected;*
2. *For all $i \neq j$, $V_i \cap V_j$ contains at most one vertex;*
3. *a is an articulation point of G if and only if $a \in V_i \cap V_j$ for some $i \neq j$.*

Definition 3.1.4 (Bcc-Closed Subgraph) *Let $G' = (V', E')$ be a subgraph of $G = (V, E)$. G' is said BCC-closed if, for each biconnected component of G, (V_i, E_i), either $E_i \cap E' = \emptyset$ or $E_i \subseteq E'$.*

3.1.2 Explicit vs Symbolic Solution to the BCC Problem

A picture similar to the one related to explicit SCC computation applies to the explicit algorithmic strategies developed for biconnected components graph decomposition.

In the early $70'$ Hopcroft and Tarjan developed an optimal $\Theta(V + E)$ algorithm to determine the biconnected components of a graph $G = \langle V, E \rangle$ [68, 106]. The biconnected components algorithm in [68, 106] relies on obtaining an ordering, to drive overall computation upon a DFS graph traversal.

To the knowledge of the author the BCC problem was never tackled before assuming a symbolic OBDD based data representation. Since DFS-based classical computational strategies for BCC decomposition can not be translated symbolically. Moreover, on the contrary of SCC problem, BCC problem seem not trivially solvable upon transitive closure. In the next sections we devise a symbolic strategy to compute biconnected components. Finally, we show how to take advantage of spine-sets (see section 2.2) to obtain BCCs within $\mathcal{O}(V)$ symbolic steps.

3.2 Spine-Sets and Biconnected Components

Given $G = (V, E)$, in this subsection we devise a number of relations connecting spine-sets, biconnected components, and the BCC-closed subgraphs introduced in Definition 3.2.1. More specifically Definition 3.2.1 , below, introduces a canonical way of splitting G into two BCC-closed subgraphs. Consider an articulation point a. By Lemma 3.1.3 there are at least two biconnected components containing a and the splitting takes place around one of them, say (V_*, E_*), with $a \in V_*$. As depicted in

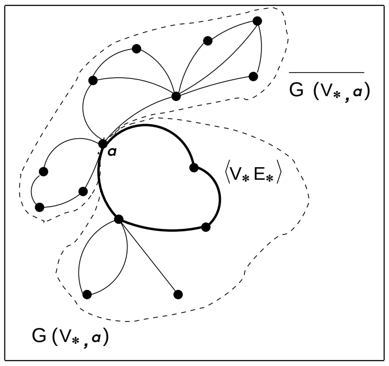


Figure 3.1: Graphs $G(V_*, a)$, $\overline{G(V_*, a)}$

Figure 3.1, one of the two subgraphs into which G gets split, contains (V_*, E_*) (and it is called $G(V_*, a)$). The other subgraph includes all the remaining biconnected components containing a (and is called $\overline{G(V_*, a)}$). We denote by $\overset{G}{\rightsquigarrow}$ (the nodes in) a simple path in the graph G.

Definition 3.2.1 *Let (V_*, E_*) be a biconnected component of $G = (V, E)$. If $a \in V_*$ is an articulation point of G, then $G(V_*, a)$ and $\overline{G(V_*, a)}$ are the subgraphs of G induced by the sets of nodes W and $(V \setminus W) \cup \{a\}$, respectively, where:*

$$W = \{a\} \cup \{v \mid \exists \overset{G}{\rightsquigarrow} (a \overset{G}{\rightsquigarrow} v \wedge \overset{G}{\rightsquigarrow} \cap (V_* \setminus \{a\}) \neq \emptyset\}$$

Lemma 3.2.2 *Consider a biconnected component (V_*, E_*) of $G = (V, E)$. If $a \in V_*$ is an articulation point of G, then:*

1. *The two subgraphs $G(V_*, a)$ and $\overline{G(V_*, a)}$ are BCC-closed;*
2. *(V_*, E_*) is a subgraph of $G(V_*, a)$.*

Proof. Let W be the vertex-set of $G(V_*, a)$. There can be no edge in G, (w, s), with $w \in W \setminus \{a\}$ and $s \in V \setminus W$, otherwise there would be a simple path traversing $V_* \setminus \{a\}$ from a to $s \in V \setminus W$. Hence, assume (by contradiction) that there exists a biconnected component in G, (V', E'), such that $V' \cap (W \setminus \{a\}) \neq \emptyset$ and $V' \cap (V \setminus W) \neq \emptyset$. As (V', E') is biconnected, for all z, t with $z \in V' \cap (W \setminus \{a\}), t \in V \setminus W$, there exists a path between z and t not containing a. We get to the contradiction that there exists an edge between a node in $W \setminus \{a\}$ and a vertex in $V \setminus W$. Point 2 follows directly from Definition 3.2.1. ■

Lemma 3.2.3 is the counterpart of Lemma 2.2.3 and Lemma 2.2.4, stated in Chapter 2 for strongly connected components.

Lemma 3.2.3 *Given $\overline{v_0 \dots v_p}$ in $G = (V, E)$, let V_* be the vertex-set of a biconnected component containing v_p. There is a minimum $0 \leq t \leq p$ such that:*

1. *$V_* \cap \{v_0, \ldots, v_p\} = \{v_t, \ldots, v_p\}$;*
2. *If v_t is an articulation point then $\overline{v_0 \ldots v_t}$ is a spine-set in $\overline{G(V_*, v_t)}$;*
3. *If $a \neq v_t$, $a \in V_*$ is an articulation point and V' is the vertex-set of $\overline{G(V_*, a)}$, then $V' \cap \{v_0, \ldots v_p\} \subseteq \{a\}$.*

Proof. Let us first prove item 1 of the lemma. By contradiction, suppose that there exists a sub-path of the chordless path $(v_1, \ldots, v_p)$, $q = (v_i, \ldots, v_j)$ with $|j - i| > 1$, in which the endpoints v_j and v_i belong to V_* and all the internal nodes do not belong to V_*. As V_* is connected, V_* contains a simple path from v_i to v_j, say q'. By concatenating the paths q and q' we obtain a simple cycle containing $v_{i+1}, \ldots, v_{j-1}$. This ensures that all the nodes in p belongs to V_* and contradicts our hypothesis. Item 2 follows immediately from the fact that each prefix of a chordless path is a chordless path. For Item 3 observe that, by Definition 3.2.1 and Item 1 in the Lemma, $\{v_0, \ldots, v_t\} \subseteq \overline{G(V_*, v_t)}$. This ensures that $\{v_0, \ldots, v_t\} \cap V' = \emptyset$ as the vertex-sets of $\overline{G(V_*, v_t)}$ and $\overline{G(V_*, a)}$, $a \neq v_t$, must be disjoint. By Item 1 in the Lemma $\{v_t, \ldots, v_p\} \subseteq V_*$. On this ground the thesis follows from the facts that $a \in V' \cap V_*$, V' contains at least one biconnected components other than that of V_* and two biconnected components can have no more than one vertex in common. ■

The *skeleton of* $\overline{G(V_*, a)}$, introduced below, is a particular spine-set in $\overline{G(V_*, a)}$ that will drive the computation in our symbolic BCC-algorithm.

Definition 3.2.4 (Skeleton of $\overline{G(V_*, a)}$) *Consider $G = (V, E)$, a biconnected component (V_*, E_*) in G and an articulation point $a \in V_*$. $\langle \mathcal{S}, u \rangle$ is a skeleton of $\overline{G(V_*, a)}$ iff u is a node in $\overline{G(V_*, a)}$ having maximum distance from a and $\mathcal{S}$ is the set of nodes on a shortest path from a to u.*

Lemma 3.2.5 *If $\langle \mathcal{S}, u \rangle$ is a skeleton of $\overline{G(V_*, a)}$, then $\langle \mathcal{S}, u \rangle$ is a spine-set in $\overline{G(V_*, a)}$.*

Proof. It follows immediately from the definition of spine-set. ■

3.3 From Spine-Sets to a Linear Symbolic Biconnected Components Algorithm

We focus here on biconnectivity. Relying on spine-sets, we ultimately outline a symbolic BCC-algorithm performing a linear number of symbolic steps. The algorithm we propose uses a rather simple strategy to find the nodes of each biconnected component in an OBDD-represented graph $G = (V, E)$. Yet, this strategy would have a quadratic performance (in the size of the vertex-set) if not properly combined with the notion of spine-set. Note how this is perfectly symmetric to what happens in the symbolic scc-algorithm in Section 2.2 of Chapter 2.

Given an edge $(u_*, v_*) \in E$, let V_* be the vertex-set of the biconnected component containing the edge $(u_*, v_*) \in E$. The strategy for building V_* relies on extending the vertex-set B, initialized as $B = \{u_*, v_*\}$, maintaining the invariant below:

$$\forall v \in B(v \in \{u_*, v_*\} \vee \exists \text{ a cycle linking } v, u_*, \text{ and } v_* \text{ in B}) \tag{3.3.1}$$

Invariant (3.3.1) ensures that $B \supseteq \{u_*, v_*\}$ induces a biconnected subgraph i.e. $B \subseteq V_*$. Under the above mentioned invariant, a safe increasing of B is obtained by adding to B all nodes on simple paths between two nodes of B. The search for these paths could naturally take place from a node $x \in B$ linked to some node outside B: a sort of *exploration point* for B.

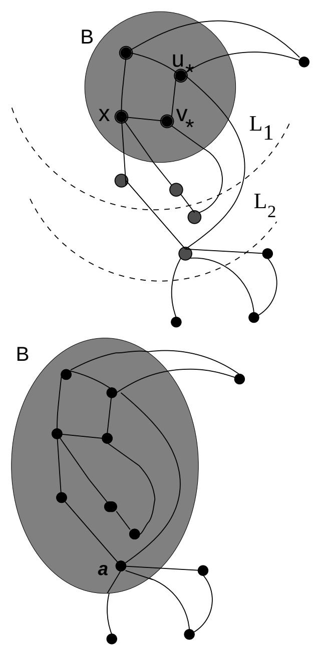


Figure 3.2: Augmenting B under invariant (1).

We depict in Figure 3.2 a successful attempt of augmenting $B \subseteq V_*$ looking for paths that source from the *exploration point* $x \in B$, cross $V \setminus B$, and terminate in a node of B other than x. The set L_1 containing the successors of $\{x\}$ in $V \setminus B$ is computed and then levels at increasing distance from L_1 are discovered. If a level L_p intersecting $B \setminus \{x\}$ is encountered, this guarantees the existence of some simple paths between two nodes of B.

If the overall process allows discovering only vertices outside B or equal to the exploration point, the attempt of growing $B \supseteq \{u_*, v_*\}$ fails. In this case, invariant (3.3.1) ensures that the *exploration point* involved in the visit is an articulation point of G. This situation is exemplified in Figure 3.2 on the bottom, if we choose the node a as the next *exploration point* for B. Always in Figure 3.2, the set of nodes that can be retrieved using a as exploration point is the set of vertices of the BCC-closed subgraph $\overline{G(V_*, a)}$. This set of nodes can be ignored while extending B to compute V^*, but it can be used used to localize the (recursive) computation of subsequent biconnected components.

Summarizing, the above ideas lead to a recursive procedure in which the process of building the vertex-set of each biconnected component (V_*, E_*) roughly results in subsequent breadth-first visits from a node of $B \subseteq V_*$. Each visit either augment $B \subseteq V_*$ or discovers a BCC-closed subgraph not containing (V_*, E_*) on which a recursive call can be made.

The problem, with the above approach, is that the symbolic steps performed to discover the subgraph $\overline{G(V_*, a)}$ make the algorithm quadratic in the number of symbolic steps.

Spine-sets allow to (piece-wise) drive in a depth-fashion the order in which biconnected components are discovered so that the global number of symbolic steps becomes linear. The way spines are involved in the strategy described above is the following. Whenever the visit from an exploration point a of B results in a set of p levels outside B, these levels are used to build a skeleton of the graph $\overline{G(V_*, a)}$.

This skeleton, say $\overline{av_1 \ldots v_p}$, keeps trace of the number of symbolic steps necessary to compute $\overline{G(V_*, a)}$. The order implicitly maintained in it can be used for subsequent computatation, so that the cost of computing $\overline{G(V_*, a)}$ is amortized onto the cost of assigning each node in $\overline{av_1 \ldots v_p}$ to its biconnected components. The linear performance of the full algorithm is a consequence of the following two facts.

1. The spines-driven order in which biconnected components get computed ensures that the global number of symbolic steps necessary to obtain the subgraphs for the recursive calls is $\mathcal{O}(|V|)$ (stated another way, the spine-sets generated during the entire algorithm collect at most $\mathcal{O}(|V|)$ nodes);

2. the remaining symbolic steps assign (at least) one node to its biconnected components.

3.3.1 The Linear Symbolic Biconnectivity Algorithm

The algorithm SYMBOLIC-BCC in Figure 3.3 implements the ideas outlined in the previous section and efficiently retrieves the vertex-set of each biconnected component in a symbolically represented graph.

SYMBOLIC-BCC is a recursive procedure that takes as input a graph, $G = (V, E)$, and a pair of sets of nodes $\langle S, N \rangle$. $\langle S, N \rangle$ is either a pair of empty sets or $S = \{v_0, \ldots, v_p\}$ and $N = \{v_p\}$, with $\overline{v_0 \ldots v_p}$ a spine in G. If $V \neq \emptyset$, lines (1)-(2) initialize the vertex-set of a biconnected component, B, to the endpoints of an edge, $\{v_*, u_*\}$. If the spine is not empty, then one of the endpoints of this edge is the spine-anchor. Line (3) updates the spine to $\overline{v_0 \ldots v_{p-1}}$ in the case in which $B = \{v_{p-1}, v_p\}$. This ensures the property that the spine represents a chordless path sharing with B only its last node (or is empty). Maintaining this property allows us to extend B in two phases, involving the nodes in the spine-set only to add them to B. The procedure VISIT in Figure 3.4 is involved in the first phase: the detection of paths reaching back $B \cup S$. The procedure EAT-SPINE in Figure 3.5 realizes the second phase.

VISIT gets as input the exploration point $X = \{x\}$ selected within line (5) in SYMBOLIC-BCC, the set B, the graph to explore $(V, E)$and its spine $\langle \mathcal{S}, N \rangle$ (collapsing in B). The procedure uses a stack of vertex-sets to keep trace of the levels at increasing distance from the set of nodes, outside $B \cup \mathcal{S}$, linked to the exploration point. Within each iteration of the loop in lines (6)-(8) of VISIT, a new level is pushed onto `stack` while either no new node can be discovered or a level intersecting $B \cup \mathcal{S}$ is detected. In the first case the vertex-set discovered is that of a BCC-closed subgraph that is used for the next recursive call. Lines (13)-(15) build a spine-set in such a subgraph by suitably selecting a node for each level popped out from `stack`. In the second case, the vertex-set discovered contains some simple path between the exploration point and a vertex in $B \cup \mathcal{S}$. The loop in lines (10) and (11) detects the vertex-sets of those paths containing exactly one node for each level pushed onto `stack` and whose last node belong to $B \cup \mathcal{S}$. These nodes are assigned to the set C that will be added to B in line (10) of the main algorithm.

The purpose of the subprocedure EAT-SPINE, in Figure 3.5, is that of augmenting B with the maximum spine-set that reaches back B. This ensures invariant (3.3.1) upon the termination of each SYMBOLIC-BCC loop iteration.

```
SYMBOLIC-BCC($V, E, \langle \mathcal{S}, N \rangle$)

– Initialize a BCC vertex-set with the endpoints of an edge –
(1)  if $(N \neq \emptyset)$ then $B \leftarrow N$ else $B \leftarrow \mathsf{pick}(V)$
(2)  $B \leftarrow \mathsf{pick}(\mathsf{img}(B)) \cup B$
– Ensure that $B \cap \mathcal{S} = N$ –
(3)  if $(B \setminus \mathcal{S} = \emptyset)$ then $\mathcal{S} \leftarrow \mathcal{S} \setminus N$; $N \leftarrow \mathsf{img}(N) \cap \mathcal{S}$

– Extend the vertex-set B –
(4)  while ($\exists$ an exploration-point other then the spine-anchor) do
          – Choose an exploration point other than the spine-anchor –
(5)       $X \leftarrow \mathsf{pick}(\mathsf{img}(\mathsf{img}(B) \cap (V \setminus B)) \cap (B \setminus N))$
          – Augment $B$ or isolate a BCC-closed subgraph and a –
          – spine-set for a new recursive call to SYMBOLIC-BCC –
(6)       $\langle C, V', \langle \mathcal{S}', N' \rangle \rangle \leftarrow$ VISIT($V, E, \langle \mathcal{S}, N \rangle, B, X$)
(7)       if $(V' \neq \emptyset)$
(8)           then SYMBOLIC-BCC($V', E \restriction V', \langle \mathcal{S}', N' \rangle$)
(9)                $V \leftarrow (V \setminus V') \cup X$;   $E \leftarrow E \restriction V$
(10)          else $B \leftarrow B \cup C$
          – Extend $B$ with a spine-suffix –
(11)      $\langle B, \langle \mathcal{S}, N \rangle \rangle \leftarrow$ EAT-SPINE($V, E, \langle \mathcal{S}, N \rangle, B$)

(12)  Return the vertex-set of the biconnected component built in $B$
– Recursive call in case $N$ contains (the unique) articulation point in $(V, E)$ –
(13)  if $(V \setminus B \neq \emptyset)$
(14)      then $V \leftarrow (V \setminus B) \cup N$;  $E \leftarrow E \restriction V$;  SYMBOLIC-BCC($V, E, \langle \mathcal{S}, N \rangle$)
```

Figure 3.3: The BCC-algorithm performing a linear number of symbolic steps.

```
VISIT($V, E, \langle \mathcal{S}, N \rangle, B, X$)

– Initialization –
(1)   Let stack be an empty stack of vertex-sets
(2)  $L \leftarrow \mathsf{img}(X) \cap (V \setminus (B \cup \mathcal{S}))$

(3)  if ($L = \emptyset$)
(4)     then return ($\emptyset, \emptyset, \langle \emptyset, \emptyset \rangle$)
(5)     else $U \leftarrow V \setminus (X \cup L)$;  $C \leftarrow \emptyset$;
             – Look for a level intersecting $(B \cup S) \setminus X$ –
(6)          while ($C = \emptyset \wedge L \neq \emptyset$) do
(7)               Push($L$, stack);    $L \leftarrow \mathsf{img}(L) \cap U$;    $U \leftarrow U \setminus L$;
(8)               $C \leftarrow L \cap ((B \cup \mathcal{S}) \setminus X))$
(9)          if ($C \neq \emptyset$)
                  – Collect in $C$ paths reaching back $B \cup \mathcal{S}$ –
(10)              then while (Notempty(stack)) do
(11)                        $C \leftarrow \mathsf{img}(C) \cap$ Pop(stack)
                  – Build spine in the BCC-closed subgraph discovered –
(12)              else $V' \leftarrow X \cup (V \setminus (B \cup U))$
(13)                   pick(Pop(stack))
(14)                   while (NotEmpty(stack)) do
(15)                        $\mathcal{S}' \cup$ pick(img($\mathcal{S}'$) $\cap$ (Pop(stack)))
(16)     then return ($C, V', \langle \mathcal{S}' \cup X, N' \rangle$)
```

Figure 3.4: The subprocedure VISIT used within the linear symbolic BCC-algorithm.

3.3.2 Correctness and Complexity Results

The soundness and completeness of the algorithm in Figure 3.3 are stated in Theorems 3.3.3 and 3.3.6, respectively. Throughout this section, we will use the notation V_c^l to indicate the variable V within the c-th recursive call to SYMBOLIC-SCC, on line l. V_c will denote the value of the parameter V to the c-th recursive call to SYMBOLIC-SCC. Whenever it will be also necessary to specify the loop iteration number, j, in SYMBOLIC-BCC, we will write $V^l_{\langle c,j \rangle}$. An analogous notation will be adopted for all the variables in SYMBOLIC-SCC.

The framework for the following lemma is the execution of the subprocedure VISIT($G, \langle \mathcal{S}, N \rangle, B, \{x\}$), where:

- $\langle \mathcal{S}, N \rangle$ is a spine in $G = (V, E)$;
- $B \subseteq V$ induces a biconnected subgraph of G such that $B \cap \mathcal{S} = N$;

```
EAT-SPINE($V, E, \langle \mathcal{S}, N \rangle, B$)

(1)  $C \leftarrow (\mathsf{img}(B \setminus N)) \cap \mathcal{S}$
– Update $B$ with maximum spine suffix reaching back $B$ –
(2)  while $(C \neq \emptyset)$ do
(3)            $B \leftarrow B \cup (\mathsf{img}(N) \cap \mathcal{S})$
(4)            $C \leftarrow C \setminus N$;   $\mathcal{S} \leftarrow \mathcal{S} \setminus N$;   $N \leftarrow \mathsf{img}(N) \cap \mathcal{S}$;
(5)  return($C, V, \langle \mathcal{S} \cup X, N \rangle$)
```

Figure 3.5: The subprocedure EAT-SPINE used within the linear symbolic BCC-algorithm.

- and $\{x\} \subset B$ is an exploration point of B.

Lemma 3.3.1 *Let $V', \langle \mathcal{S}', N' \rangle, C$ be the sets returned by* VISIT$(G, \langle \mathcal{S}, N \rangle, B, \{x\})$ *and assume that V_* is the vertex-set of the biconnected component containing B. Then, only one of the three cases is possible:*

- *x is an articulation point, $V' \neq \emptyset$ is the vertex set of $\overline{G(V_*, x)}$ and $\langle \mathcal{S}', N' \rangle$ is a spine-set in $\overline{G(V_*, x)}$*
- *$C \neq \emptyset$ and the nodes in C define some simple paths from X to $B \cup \mathcal{S}$ traversing $V \setminus (B \cup \mathcal{S})$*
- *$V' = C = \mathcal{S}' = N' = \emptyset$ and $\mathsf{img}(\{x\}) \subseteq (B \cup \mathcal{S})$*

Proof. Consider the pseudocode for VISIT in Figure 3.4. Line (2) initializes the set of vertices, say L_1, having distance 1 from x and which do not belong neither to B nor to the spine. If $L_0 = \emptyset$, then $\mathsf{img}(\{x\}) \subseteq (B \cup \mathcal{S})$ and VISIT returns only empty sets (line (4)). If $L_1 \neq \emptyset$, then the loop in lines (6)–(8) is executed. Such a loop discovers, level by level in a breadth first manner, nodes at increasing distance from L_1, without using the exploration point x. The loop stops when either from the last level of nodes discovered, say L_p, no new vertex can be reached, or it holds $L_p \cap (B \cup S) \neq \emptyset$. On the first condition, we obtain that all the nodes that can be discovered from L_1, without using x, belong to $V \setminus B$: thus x is an articulation point and the set of nodes discovered is the vertex-set of $\overline{G(V_*, x)}$. In this case, the else branch of the if statement in line (9) is executed and the just mentioned vertex-set is assigned to V' (line (12)). Lines (13)–(15) build a spine-set in $\overline{G(V_*, x)}$ using a (chordless) path which traverses each level discovered and reaches a node of maximal distance from x. Suppose instead that the loop in lines (6)–(8) finishes because it discovers a level of nodes L_p with $L_p \cap (\mathcal{S} \cup B) \neq \emptyset$. In such a case, C gets equal to $L_p \cap (B \cup \mathcal{S})$ on the end of the loop within the then branch of the if statement is executed. Lines (10)–(11) complete C whit the nodes on all the paths (of length p) from x to $L_p \cap (\mathcal{S} \cup B)$ which traverse all the level breadth first discovered. ∎

Relying on Lemma 3.3.1, Lemma 3.3.2 provides four invariants for SYMBOLIC-BCC which ensure its correctness.

Lemma 3.3.2 *In each recursive call,* SYMBOLIC-BCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$*, within the execution of* SYMBOLIC-BCC$(V, E, \langle \emptyset, \emptyset\rangle)$*, the following items hold on the entering of each iteration of the* while*-loop:*

1. (V_c^3, E_c^3) *is a connected and BCC-closed subgraph of* (V, E) *;*
2. $\{v_*, u_*\} \subseteq B_c^3 \subseteq V_* \subseteq V$ *and the subgraph induced by* B_c^3 *is biconnected;*
3. *either* $|B_c^3|$ *is increased or* $V_c^3 \setminus V_*$ *is decreased;*
4. $\langle \mathcal{S}_c^3, N_c^3\rangle$ *is a spine-set in* (V_c^3, E_c^3) *and, if* $\mathcal{S}_c^3 \neq \emptyset$*, then* $\mathcal{S}_c^3 \cap B_c^3 = N_c^3$*;*

where V_* *is the vertex-set of the biconnected component of* $(v^*, u^*) \in E$*.*

Proof. Consider the c-th recursive call within SYMBOLIC-BCC$(V, E, \langle \emptyset, \emptyset\rangle)$. In such a recursive call, consider the j-th checking of the while-loop guard. We use an inductive argument on the value of $\langle c, j\rangle$ to prove items (a)–(d) of the lemma. The base case is immediate, we sketch below the inductive step.

Inductive Step: Let's assume items (a)–(d) true for all (consistent) pairs $\langle 1, 1\rangle \leq \langle c', j'\rangle \leq \langle c, j\rangle$. There are two cases to be considered depending on which is the minimum consistent pair greater than $\langle c, j\rangle$.

In the first case $\langle c, j+1\rangle$ is the minimum consistent pair greater than $\langle c, j\rangle$ (i.e. the while-guard is checked more than j times in the c-th recursive call to SYMBOLIC-BCC). Within each iteration of the loop in SYMBOLIC-BCC the procedure VISIT is executed. We consider each of the three cases possible, by Lemma 3.3.1, for the triple returned by VISIT: $C, V', \langle \mathcal{S}', N'\rangle$. If all the sets are empty, then, by Lemma 3.3.1, $\mathsf{img}(X^7_{\langle c,j\rangle}) \subseteq \mathcal{S}^7_{\langle c,j\rangle} \cup B^7_{\langle c,j\rangle}$. By the choice of $X = \{x\}$ in line (5), some node of the spine-set other than the spine-anchor is linked to x. The procedure EAT-SPINE detects the least spine-set node, v_l, with the above property, subtract $\{v_l + 1, \ldots, v_p\}$ to the spine-set, and adds $\{v_l, \ldots, v_p\}$ to B. Hence, $B^4_{\langle c,j+1\rangle} = B \cup \{v_l \ldots v_p\}$ is biconnected, $B^4_{\langle c,j+1\rangle} \cap \mathcal{S}^4_{\langle c,j+1\rangle} = N^4_{\langle c,j+1\rangle}$ and items (a)–(d) are satisfied for $\langle c, j+1\rangle$ when VISIT returns only empty sets.

The case in which the set $V'_{\langle c,j\rangle}$ returned by VISIT is not empty is also easily dealt with. By Lemma 3.3.1 if $V'^7_{\langle c,j\rangle}$ is not empty then, $X^4_{\langle c,j\rangle} = \{x\}$ is an articulation point and $V'^7_{\langle c,j\rangle}$ is the vertex-set of $\overline{G(V_*, x)}$. Line (9) in SYMBOLIC-BCC subtracts this vertex-set from V. This way items 1.(a), 1.(c) get satisfied for $\langle c, j+1\rangle$. Items 1.(b), 1.(d) also hold since B and the spine-set can be modified only within EAT-SPINE which, as already discussed, maintains them. Finally, if VISIT returns only the set $C^7_{\langle c,j\rangle}$ not empty, line (10) in SYMBOLIC-BCC adds $C^7_{\langle c,j\rangle}$ to $B^4_{\langle c,j\rangle}$. Thus, B get enlarged and Lemma 3.3.1 ensures that B is added of the nodes on some simple paths from X to $B \cup S$ traversing $V \setminus (B \cup \mathcal{S})$. At this point, EAT-SPINE completes B with the maximal suffix of the chordless path defined by $\langle \mathcal{S}, N\rangle$ reaching back $B \setminus N$. It follows that $B^4_{\langle c,j+1\rangle}$ induces a biconnected subgraph and all the items in the lemma are satisfied.

In the second case the minimum consistent pair greater than $\langle c, j\rangle$ is $\langle c+1, 1\rangle$ and we must consider the c'-th recursive call to SYMBOLIC-BCC from which the c-th execution of SYMBOLIC-BCC was invocated ($c' \leq c$). If the just mentioned invocation was made within j'-th iteration in the loop of c'-th call to SYMBOLIC-BCC, then $V^4_{\langle c+1,1\rangle}$ and $\langle \mathcal{S}^4_{\langle c+1,1\rangle}, N^4_{\langle c+1,1\rangle}\rangle$ were built within the procedure VISIT. Thus, by Lemma 3.3.1, the graph induced by $V^4_{\langle c+1,1\rangle}$ is connected and BCC-closed and $\langle \mathcal{S}^4_{\langle c+1,1\rangle}, N^4_{\langle c+1,1\rangle}\rangle$ is a spine in it. Otherwise, the $c+1$-th recursive call to SYMBOLIC-BCC must be done in line 27 within the c'-th execution of SYMBOLIC-BCC. In this the validity of items (a)–(d) on $\langle c+1, 1\rangle$ follows easily from inductive hypothesis. ∎

Theorem 3.3.3 (Correctness) SYMBOLIC-BCC$(V, E, \langle\emptyset,\emptyset\rangle)$ *computes the vertex-sets of the biconnected components in* $G = (V, E)$.

Proof. It follows directly from Lemma 3.3.2. ∎

Theorem 3.3.6 states that SYMBOLIC-BCC$(V, E, \langle\emptyset,\emptyset\rangle)$ needs $\mathcal{O}(|V|)$ symbolic steps to compute $\{V_1, \ldots, V_n\}$, where $\{\langle V_1, E_1\rangle \ldots \langle V_n, E_n\rangle\}$ are the biconnected components in $G = (V, E)$. As already anticipated, the complexity result follows from the following considerations. First, the global number of symbolic steps spent to detect simple paths reaching back a partial biconnected component is $\mathcal{O}(|V_1| + \ldots + |V_n|)$. Moreover, the global number of symbolic steps spent for obtaining the subgraphs for the recursive calls is proportional to the number of insertions of nodes in a spine. By Lemma 3.3.5 also the global number of insertions of nodes into a spine is $\mathcal{O}(|V_1| + \ldots + |V_n|)$. Note that $|V_1| + \ldots + |V_n| \geq |V|$ because of the articulation points, however, by Lemma 3.3.4, given below, it holds that $(|V_1| + \ldots + |V_n|) = \mathcal{O}(|V|)$.

Lemma 3.3.4 *Consider the set A_G of articulation points of $G = (V, E)$. Given $a \in A_G$ let $m_G(a)$ (the* moltiplicity *of the articulation point a) be the number of biconnected components of G that contain a. Then*

$$\sum_{a\in A} m_G(a) \leq 2|V| - 4$$

Proof. By induction on the number of biconnected components of G, n. If $n = 1$, there are no articulation points and $2|V| - 4 \geq 2 * 2 - 4 = 0$. Otherwise, by Lemma 3.1.3 there is at least one articulation point in G. Let a be an articulation point of G and let $\langle V_*, E_*\rangle$ be a biconnected component of G such that $a \in V_*$. We use the inductive hypothesis on the two subgraphs $G_1 = \langle V_1, E_1\rangle = G(V_*, a)$ and $G_2 = \langle V_2, E_2\rangle = \overline{G(V_*, a)}$. By Definition 3.2.1, $V_1 \cap V_2 = \{a\}$, and by Lemma 3.2.2:

$$\forall w \in V_i \setminus \{a\}(w \in A_{G_i} \wedge m_{G_i}(w) = k) \Leftrightarrow (w \in A_G \wedge m_G(w) = k)$$

for $i = 1, 2$. By definition of $G(V_*, a) = G_1$, a can not be an articulation point in G_1. If a is an articulation point in G_2, $m_G(a) = m_{G_2}(a) + 1$. Otherwise $m_G(a) = 2$. Thus:

$$\begin{aligned}&\textstyle\sum_{a\in A} m_G(a) \leq \sum_{a\in A_{G_1}} m_{G_1}(a) + \sum_{a\in A_{G_2}} m_{G_2}(a) + 2 \leq \\ &\leq (2|V_1| - 4) + (2|V_2| - 4) + 2 = 2|V| + 2 - 8 + 2 = 2|V| - 4\end{aligned}$$

∎

Lemma 3.3.5 *Let v be a node belonging to m biconnected components of $G = (V, E)$. Overall* SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$*, v is inserted at most m times in a spine-set.*

Proof. Consider the c-th call to SYMBOLIC-BCC within SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$. By induction on the number of recursive calls, we prove that $\forall v \in V_c$, within the the first $c - 1$ (recursive) calls to SYMBOLIC-BCC:

- $v \in V_c$ has been inserted at least once in a spine if and only if $v \in \mathcal{S}_c$;
- if $v \in V_c$ has been inserted m times in a spine then v belongs to at least m distinct biconnected components. Moreover, in this case $m - 1$ biconnected component containing v have been already produced in output.

Base: Immediate as $\langle \mathcal{S}_1, N_1\rangle = \langle\emptyset, \emptyset\rangle$.

Inductive Step: Let $c > 1$ and $v \in V_c$. Assume that the c-th call to SYMBOLIC-BCC was done within the c'-th execution of SYMBOLIC-BCC, which builds the biconnected component vertex-set $V_{(*,c')}$. Let $a_1 \ldots, a_l$ be the articulation points of $(V_{c'}, E_{c'})$ that are contained in $V_{(*,c')}$. By Lemma 3.3.1 and Lemma 3.3.2, the subgraphs of $G_{c'} = (V_{c'}, E_{c'})$ involved in the recursive calls of SYMBOLIC-BCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'}\rangle)$, are $\overline{G_{c'}(V_{(*,c')}, a_k)}$, where $a_k \in \{a_1, \ldots, a_l\}$. These subgraphs have no vertex in common, thus $\forall c' < c'' < c$, $v \notin V_{c''}$ and we can safely avoid considering the recursive calls between the c'-th one and the c-th one for the inductive step. If the c recursive call to SYMBOLIC-BCC was made after the loop of the c'-th execution of SYMBOLIC-BCC, then $\langle \mathcal{S}_c, N_c\rangle$ is a "prefix" of $\langle \mathcal{S}_{c'}, N_{c'}\rangle$ and v was not involved in any operation of insertion into a spine-set within the c'-th execution of SYMBOLIC-BCC. In this case the inductive hypothesis (on c') ensures the inductive step.

Otherwise, the c-th recursive call to SYMBOLIC-BCC must be done within an iteration of the loop internal to the c'-th execution of SYMBOLIC-BCC. In this case $\langle \mathcal{S}_c, N_c\rangle$ must have been produced within VISIT, in the process of exploring from an articulation point of $V_{(*,c')}$. By Lemma 3.2.3, there is at most one vertex in $\mathcal{S}_{c'} \cap V_c$. Thus, we can conclude what follows on the ground of the inductive hypothesis on c'. If $v \in V_c \setminus \mathcal{S}_c$, then, when the c-th recursive call is done, v has not yet been inserted in any spine. If $v \in \mathcal{S}_c \wedge v \notin \mathcal{S}_{c'}$, then when the c-th recursive call is done v has been inserted only once in a spine-set. Finally, in the case that $v \in \mathcal{S}_{c'} \cap \mathcal{S}_c$ we have that $\{v\} = \mathcal{S}_{c'} \cap \mathcal{S}_c$: if v has been inserted m times in a spine-set upon the c'-th execution of SYMBOLIC-BCC, then v has been inserted $m+1$ times in a spine-set upon the c'-th execution of SYMBOLIC-BCC. Moreover, as the c'-th recursive call produces $V_{(*,c')} \supseteq \{v\}$ we have that m biconnected component containing v have been already produced in output upon the execution of SYMBOLIC-BCC$(V_c, E_c, \langle \mathcal{S}_c, N_c\rangle)$. ■

Theorem 3.3.6 (Complexity) *The algorithm* SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$ *runs in $\mathcal{O}(|V|)$ symbolic steps.*

Proof. The complexity of the algorithm can be computed by counting the global number of iterations of the loops in the subprocedures VISIT and EAT-SPINE. As EAT-SPINE is concerned, each time an iteration of its loop is performed a new node is assigned to a biconnected components. Thus, the global number of iterations of the

EAT-SPINE loop is $\mathcal{O}(|V_1|+\ldots+|V_n|)$, where $\{\langle V_1, E_1\rangle \ldots \langle V_n, E_n\rangle\}$ are the biconnected components of G.

Within each execution of VISIT the number of iterations of the first loop is the same as the number of iterations of the second loop. This number, say p, correspond to the number of disjoint levels of nodes pushed onto the stack used within VISIT. Upon the termination of VISIT, either, for each level in the stack, some nodes are assigned to a biconnected component, or, for each level in the stack, a vertex is inserted into a spine-set. It follows, by Lemma 3.3.5, that the global number of iterations of the VISIT loop is $\mathcal{O}(|V_1| + \ldots + |V_n|)$, where $\{\langle V_1, E_1\rangle \ldots \langle V_n, E_n\rangle\}$ are the biconnected components of G.

By Lemma 3.3.4 $\mathcal{O}(|V_1| + \ldots + |V_n|) = \mathcal{O}(|V|)$ ∎

II

Equivalence Based Graph Reductions

4

Bisimulation and Simulation Graph Reductions

Computing bisimulation and simulation equivalences, over a graph set of nodes, is an important task in many areas of Computer Science: from Modal Logic, to Concurrency Theory as well as Formal Verification, to cite only a few fields (see Subsection 4.2.1 and Subsection 4.2.2 for an overview).

With respect to simulation the notion of bisimulation enjoys, in the literature, a deeper understanding from both an algebraic and a computational point of view. One of the purposes of this part of the thesis is that of refining, to some extent, the above gap that we further discuss in this chapter, presenting bisimulation and simulation problems.

4.1 Preliminary Notions

Let $Q \subseteq T \times T$ be a binary relation over a set T:
- Q is a *quasi order* if and only if Q is reflexive and transitive;
- Q is a *partial order* if and only if Q is an antisymmetric quasi order;
- Q is *acyclic* if and only if its transitive closure is antisymmetric.
We will use Q^+ to refer to the transitive closure of Q and Q^* to refer to the reflexive and transitive closure of Q.

Definition 4.1.1 (Labelled Graphs) *A triple $G = (V, E, \Sigma)$ is said to be a* labelled graph *if and only if $G^- = (V, E)$ is a finite direct graph and Σ is a partition over V.*

Given a node $a \in V$ we denote by $[a]_\Sigma$ (or $[a]$, if Σ is clear from the context) the class of Σ to which a belongs. We say that two nodes $a, b \in V$ have the same *label* if they belong to the same class of Σ.

An equivalent way to define *labelled graphs* is to use a labelling function $\ell : V \rightarrow L$, where L is a finite set of labels (inducing a partition Σ_L on V). Given a partition Σ we will use the letters $\alpha, \beta, \gamma, \ldots$ to denote a generic element, i.e. a *block*, of Σ.

Definition 4.1.2 (Bisimulation) *Let $G = (V, E, \Sigma)$. A relation $\asymp \subseteq V \times V$ is a* bisimulation *over G if and only if:*

1. $a \asymp b \Rightarrow [a]_\Sigma = [b]_\Sigma$;
2. $(a \asymp b \wedge a \to c) \Rightarrow \exists d(c \asymp d \wedge b \to d)$;
3. $(a \asymp b \wedge b \to d) \Rightarrow \exists c(c \asymp d \wedge a \to c)$.

Two nodes a and b are bisimilar *($a \equiv_b b$) if there is a bisimulation $\asymp$ such that $a \asymp b$.*

A bisimulation can be neither reflexive nor transitive, and not even symmetric. However, the reader can easily verify that given a bisimulation, its reflexive, symmetric, and transitive closure is still a bisimulation. The proof of the following lemma can be found in [86].

Lemma 4.1.3 *Let $G = (V, E, \Sigma)$. The relation $\equiv_b$ is an equivalence relation over V and a bisimulation over G. Moreover, $\equiv_b$ is the maximum bisimulation, i.e., if $\asymp$ is a bisimulation over G, then $\asymp \subseteq \equiv_b$.*

Since $\equiv_b$ is an equivalence relation, it is possible to consider the quotient $B = V/\equiv_b$. We will use the notation $[a]_b$ to denote the equivalence class to which a belongs in B.

Definition 4.1.4 (Bisimulation Problem) *Given a labelled graph $G = (V, E, \Sigma)$ the* bisimulation problem *over G consists in computing the quotient $B = V/\equiv_b$, where $\equiv_b$ is the maximum bisimulation over G.*

Definition 4.1.5 (Simulation) *Let $G = (V, E, \Sigma)$. A relation $\leq \subseteq V \times V$ is said to be a* simulation *over G if and only if:*

1. $a \leq b \Rightarrow [a]_\Sigma = [b]_\Sigma$;
2. $(a \leq b \wedge a \to c) \Rightarrow \exists d(c \leq d \wedge b \to d)$.

In this case b simulates *a.*
Two nodes a and b are sim-equivalent *($a \equiv_s b$) if there exist two simulations $\leq_1$ and $\leq_2$, such that $a \leq_1 b$ and $b \leq_2 a$.*

Again, a simulation can be neither reflexive nor transitive (e.g., the empty relation is always a simulation), but given a simulation, its reflexive and transitive closure is still a simulation.

A simulation $\leq_s$ over G is said to be *maximal* if for all the simulations $\leq$ over G it holds $\leq \subseteq \leq_s$. The proves of the following results can be found in [81]

Lemma 4.1.6 *Given $G = (V, E, \Sigma)$ there always exists a unique maximal simulation $\leq_s$ over G. Moreover, $\leq_s$ is a quasi order.*

Corollary 4.1.7 *Let $G = (V, E, \Sigma)$. The relation $\equiv_s$ is an equivalence relation over V.*

Also in the case of simulation, since $\equiv_s$ is an equivalence relation, it is possible to consider the quotient $S = V/ \equiv_s$. We will use the notation $[a]_s$ to denote the equivalence class to which a belongs in S.

Definition 4.1.8 (Simulation Problem) *Given a labelled graph $G = (V, E, \Sigma)$, the* simulation problem *over G consists in computing the quotient $S = V/ \equiv_s$, where $\equiv_s$ is the sim-equivalence over G.*

4.2 State-of-the-art

4.2.1 Bisimulation

The notion of bisimulation, formally defined in Section 4.1, has been introduced with different purposes in many areas related to Computer Science. In Modal Logic it was introduced by van Benthem [4] as an equivalence principle between Kripke structures. In Concurrency Theory it was introduced by Park [86] for testing observational equivalence of the Calculus of Communicating Systems (CCS). In Set Theory, it was introduced by Forti and Honsell [50] as a natural principle replacing extensionality in the context of non well-founded sets. As far as Formal Verification is concerned [26, 99], several existing verification tools make use of bisimulation in order to minimize the state spaces of systems descriptions [29, 11, 46, 95], since bisimulation strongly preserves the whole μ-calculus. The reduction of the number of states is important both in compositional and in non-compositional model checking. Bisimulation serves also as a means of checking equivalence between transition systems. In the context of security many non interference properties are based on checking bisimulation between systems [49].

From a computational point of view, the main reason for the fortune of bisimulation and for its fast solution lies in the equivalence between the bisimulation problem and the problem of determining the *coarsest partition* of a set *stable* with respect to a given relation ([70]).

Definition 4.2.1 (Stability) *Let E be a binary relation on V, E^{-1} its inverse relation, and Σ a partition of V. Σ is said to be* stable *with respect to E iff for each pair α, γ of blocks of Σ, either $\alpha \subseteq E^{-1}(\gamma)$ or $\alpha \cap E^{-1}(\gamma) = \emptyset$.*

We say that a partition Π *refines* a partition Σ (Π is *finer* than Σ) if each block of Π is included in a block of Σ.

Definition 4.2.2 (Coarsest Stable Partition Problem) *Let E be a binary relation on V and Σ a partition over V. The* coarsest stable partition problem *is the problem of finding the coarsest partition B refining Σ stable with respect to E.*

This problem, that emerged also naturally in automata minimization [67], is equivalent to the bisimulation problem.

Proposition 4.2.3 *Let $G = (V, E, \Sigma)$ be a labelled graph.*

(i) *Let Π be a partition over V refining Σ and stable with respect to E. Then $\asymp_\Pi$, defined as $a \asymp_\Pi b$ iff $\exists \alpha \in \Pi\, (a \in \alpha \wedge b \in \alpha)$, is a bisimulation over G.*

(ii) *Let $\asymp$ be a bisimulation over G which is also an equivalence relation. Then $\Pi_\asymp = \{[a]_\asymp \mid a \in V\}$, where $[a]_\asymp = \{b \in V \mid a \asymp b\}$, is a partition stable with respect to E.*

Proof.

(i) We prove that $\asymp_\Pi$ is a bisimulation on G. Since Π refines Σ, it holds that if $a \asymp_\Pi b$, then a and b belong to the same class in Σ. Let a, b be such that $a \asymp_\Pi b$. This means that there is a class $\alpha \in \Pi$ such that $a \in \alpha$ and $b \in \alpha$. Assume there is $c \in V$ such that aEc. Let γ be the class such that $c \in \gamma$. Since Π is stable with respect to E and $E^{-1}(\gamma)$ is not empty (it contains a), this means that $\alpha \subseteq E^{-1}(\gamma)$. Thus, $b \in E^{-1}(\gamma)$, and this implies that there is $d \in \gamma$ such that bEd. By definition of $\asymp_\Pi$, $c \asymp_\Pi d$. Similarly we can prove that if $a \asymp_\Pi b$ and bEd, then there exists c such that $c \asymp_\Pi d$ and aEc.

(ii) By contradiction, assume that $\Pi_\asymp$ is not stable with respect to E. This means that there are blocks α and γ and two nodes a, b such that

$$a \in \alpha \setminus E^{-1}(\gamma) \wedge b \in \alpha \cap E^{-1}(\gamma)$$

This implies that there is a node $d \in \gamma$ such that bEd but no node c bisimilar to d (i.e., in γ) can be reached by an edge from a. Thus, $\neg a \asymp b$.

■

The next corollary follows immediately.

Corollary 4.2.4 *Let $G = (V, E, \Sigma)$. Computing the maximum bisimulation $\equiv_b$ on G or finding the coarsest stable partition of V refining Σ and stable with respect to E are equivalent problems.*

The first significant result related to the algorithmic solution of the bisimulation problem is in [67], where Hopcroft presents an algorithm for the minimization of the number of states in a given finite state automaton. Hopcroft's result has been subsequently clarified and improved in [59, 71]. The problem is equivalent to that of determining the coarsest partition of a set stable with respect to a finite set of functions. A variant of this problem is studied in [85], where it is shown how to solve it in linear time in case of a single function. In [69], Kanellakis and Smolka solved the problem for the general case (which is the same as solving the bisimulation problem) in which the stability requirement is relative to a relation E (on a set V). The algorithm by Kanellakis and Smolka requires $\mathcal{O}(|E||V|)$ steps and was outperformed by the $\mathcal{O}(|E| \log |V|)$ procedure in [84], realized by Paige and Tarjan. The main feature of the linear solution to the single function coarsest partition problem (cf. [85]), is the use of a *positive* strategy in the search for the coarsest partition: the starting partition is the partition with singleton classes and the output is built via a sequence of steps in which two or more classes are merged. Instead, Hopcroft's solution to the (more difficult) many functions coarsest partition problem is based on a (somehow more natural) *negative* strategy: the starting partition is the input partition and each step consists of the split of all those classes for which the stability constraint is not satisfied. The interesting feature of Hopcroft's algorithm lies in its use of a clever ordering (the so-called "process the smallest half" ordering) for processing classes that must be used in a split step. Starting from an adaptation of Hopcroft's idea to the relational coarsest partition problem, Paige and Tarjan succeeded in obtaining their fast solution [84]. The "process the smallest half" policy establishes that if a block α' is split into α and $\alpha' \setminus \alpha$, and α has less elements than $\alpha' \setminus \alpha$, then we can use (only) α as splitter and ignore $\alpha' \setminus \alpha$. In their generalization of this policy, Paige and Tarjan determine how to perform $|\alpha|$ operations even when it is necessary to use both α and $\alpha' \setminus \alpha$ as splitters.

In [70] Kannellakis and Smolka notice that the algorithm by Paige and Tarjan [84] for the relational coarsest partition problem can be used to determine the maximum bisimulation over a graph.

In [10] Bouajjani, Fernandez, and Halbwachs propose an algorithm for the relational coarsest partition problem tailored for the context of on-the-fly model checking, i.e., they stabilize only *reachable* blocks. In [75] Lee and Yannakakis improve this method by using only reachable blocks to stabilize the reachable blocks.

From a more abstract point of view, an interesting property of the notion of bisimulation is that the bisimulation problem over labelled graphs is equivalent to the one over unlabelled ones. Given a labelled graph $G = (V, E, \Sigma)$, it is possible to build a graph $G' = (V', E')$, with $V \subseteq V'$ and $E \subseteq E'$, such that for all $a, b \in v$ $a \equiv_b b$ over G if and only if $a \equiv_b b$ over G' (see [39, 38, 88]). As a matter of fact, in the context of non-well-founded sets, graphs are used to represent sets (see [1]), hence they have no labels and the notion of bisimulation defines set-equalities. The fact that the problem with labels can be reduced to the one without labels means that the set-theoretic formulation of the bisimulation problem is general enough to embed all the other bisimulation problems.

In [39, 38], exploiting the set-theoretic formulation of the bisimulation problem, an algorithm which optimizes the algorithm in [84] is presented. The worst case complexity of the algorithm described in [39, 38] is equal to the worst case complexity of the algorithm proposed in [84], but in a large number of cases it obtains better performances. In particular, this algorithm integrates positive and negative strategies and the combined strategy is driven by the set-theoretic notion of *rank* of a set. In the case of acyclic graphs the rank of a node a is nothing but the length of the longest path from a to a leaf.

Definition 4.2.5 *Let $G = (V, E)$ be an acyclic graph. The* rank *of a node a is* *recursively defined as follows:*

$$\begin{cases} \mathit{rank}(a) = 0 & \text{if } a \text{ is a leaf} \\ \mathit{rank}(a) = 1 + \max\{\mathit{rank}(c) \mid a \to c\} & \text{otherwise} \end{cases}$$

In the general case, the rank of a node a is the length of the longest path from a node c, reachable from a, to a leaf such that all the nodes involved in the path do not reach cycles. To give a formal definition we introduce the notion of graph of the strongly connected components.

Definition 4.2.6 (Strongly Connected Components) *Given the graph $G = (V, E)$, let $G^{scc} = (V^{scc}, E^{scc})$ be the graph obtained as follows:*

$$\begin{array}{lcl} V^{scc} & = & \{C \mid C \text{ is a strongly connected component in } G\} \\ E^{scc} & = & \{\langle C(a), C(c)\rangle \mid C(a) \neq C(c) \text{ and } a \to c\} \end{array}$$

Given a node $a \in V$, we refer to the node of G^{scc} associated to the strongly connected component of a as $C(a)$.

Observe that G^{scc} is acyclic.

We also need to distinguish between the well-founded part and the non-well-founded part of a graph G.

Definition 4.2.7 (Well-Founded Part) *Let $G = (V, E)$ and $a \in V$. $G(a) = (V(a), E \restriction V(a))$ is the subgraph*[1] *of G of the nodes reachable from a. $WF(G)$, the well-founded part of G, is $WF(G) = \{a \in V \mid G(a) \text{ is acyclic}\}$.*

Definition 4.2.8 (Rank) *Let $G = (V, E)$. The* rank *of a node a of G is defined as:*

$$\begin{cases} \mathit{rank}(a) = 0 & \text{if } a \text{ is a leaf in } G \\ \mathit{rank}(a) = -\infty & \text{if } C(a) \text{ is a leaf in } G^{scc} \text{ and } a \text{ is not a leaf in } G \\ \mathit{rank}(a) = \max(\{1 + \mathit{rank}(c) \mid C(a)E^{scc}C(c), c \in WF(G)\} \cup \\ \qquad\qquad \{\mathit{rank}(c) \mid C(a)E^{scc}C(c), c \notin WF(G)\}) & \text{otherwise} \end{cases}$$

Two important properties of the notions of rank, proved in [39, 38], suggest to exploit it in the bisimulation problem:

- if $a \equiv_b b$, then $\mathit{rank}(a) = \mathit{rank}(b)$;
- it is possible to compute $\equiv_b$ on the nodes at rank i manipulating *only* nodes having ranks less or equal to i.

Given a graph $G = (V, E)$, the rank vertex-set labeling can be computed in time $\mathcal{O}(|V| + |E|)$ [39, 38], using Tarjan's algorithm for the strongly connected components [106]. This means that using the ranks inside a bisimulation algorithm does not increase its asymptotic time complexity.

The algorithm in [39, 38] first splits the graph into ranks, then uses [85] and [84] as subroutines at subsequent levels, in increasing order of rank. The overall procedure

[1]We use $E \restriction V(a)$ to denote the restriction of E to $V(a)$.

terminates in linear time in many cases, for example when the input problem corresponds to a bisimulation problem on acyclic structures. It operates in linear time in other cases as well and, in any case, it runs at a complexity less than or equal to that of the algorithm by Paige and Tarjan [84]. Moreover, the partition imposed by the rank allows to process the input without storing the entire structure in main memory. This allows (potentially) to deal with larger graphs than those treatable using a Paige and Tarjan-like approach. A symbolic version of the algorithm in [39, 38] will be presented in Chapter 7 of this dissertation. In Chapter 7 we will discuss, also, about the use of the rank notion inside model checking procedures (i.e., not only to compute the bisimulation quotient, but also to evaluate the CTL formulae).

In [89] the bisimulation problem is tackled again from a set-theoretic point of view. It is shown how in the case of acyclic graphs a compact version of the Ackermann's encoding can be used to solve the bisimulation problem, then the encoding is extended to the general case.

In Section 4.1 we presented the bisimulation problem over a labelled graph G. It is possible to formulate the problem using two graphs G_1 and G_2 for which it is convenient to define *labelled graphs* using a labelling function $\ell : V \to L$, where L is a finite set of labels common for all the graphs.

Definition 4.2.9 (Two Graphs Bisimulation) *Let $G_1 = (V_1, E_1, \ell_1)$ and $G_2 = (V_2, E_2, \ell_2)$ be two labelled graphs. A relation $\asymp \subseteq V_1 \times V_2$ is a* bisimulation *between G_1 and G_2 if and only if:*

1. $a \asymp b \Rightarrow \ell_1(a) = \ell_2(b)$;
2. $(a \asymp b \wedge a \to c) \Rightarrow \exists d(c \asymp d \wedge b \to d)$;
3. $(a \asymp b \wedge b \to d) \Rightarrow \exists c(c \asymp d \wedge a \to c)$;
4. *for each $a \in V_1$ there exists $b \in V_2$ such that $a \asymp b$;*
5. *for each $b \in V_2$ there exists $a \in V_1$ such that $a \asymp b$;*

Two graphs G_1 and G_2 are bisimilar *if there exists a bisimulation between G_1 and G_2.*

Either definition of bisimulation allows us to quotient (reduce) G on the ground of the following property of $\equiv_b$ (and of all the bisimulations that are equivalence relations):

$$\forall a, b, c \; ((a \to c \wedge b \in [a]_b) \Rightarrow \exists d(d \in [c]_b \wedge b \to d)) \qquad (\flat)$$

The quotient graph $G/\equiv_b = (V_{\equiv_b}, E_{\equiv_b}, \ell_{\equiv_b})$ is defined as:

$$\begin{array}{rcl} V_{\equiv_b} & = & V/\equiv_b \\ [a]_b E_{\equiv_b} [c]_b & \Leftrightarrow & \exists c_1(c_1 \in [c]_b \wedge a \to c_1) \\ \ell_{\equiv_b}([a]_b) & = & \ell(a). \end{array}$$

Proposition 4.2.10 *Let $G = (V, E, \ell)$ be a labelled graph. The graph $G/\equiv_b$ is the minimum graph bisimilar to G.*

Proof. It can be easily proved that the relation $\forall a \in V(aB[a]_b)$ is a bisimulation between G and $G/\equiv_b$.

Let $G_1 = (V_1, E_1, \ell_1)$ be a graph and B_1 be a bisimulation between G and G_1. The relation B_1' defined as

$$\forall a, b \in V(aB_1'b \Leftrightarrow \exists k \in V_1(aB_1k \wedge bB_1k)).$$

is easily seen to be a bisimulation. Since B_1' is included in $\equiv_b$, it holds that $|V/\equiv_b| \leq V_1$. ■

4.2.2 Simulation

The notion of simulation, introduced in Section 4.1, first defined by Milner in [80] as a means to compare programs, is very close (less demanding, in fact) to the notion of bisimulation. Since the conditions in the definition of simulation are weaker than the ones in the definition of bisimulation, simulation provides, for example in the context of Formal Verification, a better space reduction tool than bisimulation. Nevertheless simulation is still adequate for the verification of all formulae of the branching temporal logic without quantifiers switches [34] (e.g., the formulae of ACTL*, see [60]). As explained in [63] "in many cases, neither trace equivalence nor bisimilarity, but similarity is the appropriate abstraction for computer-aided verification ...". In the case of finite-state systems the similarity quotients can be computed in polynomial time, while this is not the case for trace equivalence quotients. On infinite-state systems, finitely represented using hybrid automaton and other formalisms, the similarity quotients can be computed symbolically and in many cases the quotients are finite [63].

Several polynomial-time algorithms to solve the simulation problem on finite graphs have been proposed: the ones in [7], [28], and [30] achieve time complexities $\mathcal{O}(|V|^6|E|)$, $\mathcal{O}(|V|^4|E|)$, and $\mathcal{O}(|E|^2)$, respectively. A simulation procedure running in $\mathcal{O}(|V||E|)$ time was independently discovered in [63] and [8]. All of the algorithms just mentioned [7, 28, 30, 8, 63] obtain the similarity quotient $S = V/\equiv_s$ as a by-product of the computation of the entire similarity relation $\leq_s$ on the set of states V. Their space complexity is then limited from below by $\mathcal{O}(|V|^2)$.

Recently Bustan and Grumberg in [18] and Cleaveland and Tan in [31] improved the above results.

The procedure by Bustan and Grumberg [18] gives as output the quotient structure with respect to $\equiv_s$ and the simulation relation among classes of $S = V/_{\equiv_s}$ without computing the entire simulation on V. Hence, its space requirements (often critical, especially in the field of verification) depend on the size of S and are lower than the ones of the algorithms in [7, 28, 30, 8, 63]. In more detail, the algorithm described in [18] uses only $\mathcal{O}(|S|^2 + |V|\log(|S|))$ space whereas its time complexity is rather heavy: it is $\mathcal{O}(|S|^4(|E| + |S|^2) + |S|^2|V|(|V| + |S|^2))$.

The procedure in [31] combines the fix-point calculation techniques in [8] and [63] with the bisimulation-minimization algorithm in [84]. It is determined whether a system G_2 is capable of simulating the behavior of G_1, by interleaving the minimization via bisimulation of the two systems with the computation of the set of classes in G_2 able to simulate each class in G_1. The time complexity achieved is $\mathcal{O}(|B_1||B_2|+|E_1|\log(|V_1|)+|B_1||E_2|+|\varepsilon_1||B_2|)$, where ε_i and B_i represent the bisimulation reduced relation and state-space of T_i, respectively. Compared with the time complexities of [63] and [8], the latter expression has many occurrences of $|V_i|$ and $|E_i|$ replaced with $|B_i|$ and $|\varepsilon_i|$. Indeed, experimental results in [31] prove that the procedure by Cleaveland and Tan outperforms the ones in [63] and [8]. The space complexity of [31] depends on the product of the sizes of the two bisimulation quotients involved. Sinca bisimulation is an equivalence relation finer than simulation, such a space requirement may be more demanding than the one in [18].

As in the case of bisimulation, it is possible to formulate the simulation problem between two graphs. In particular, a "two graphs" formulation is used in [31].

Definition 4.2.11 (Two Graphs Simulation) *Let $G_1 = (V_1, E_1, \ell_1)$ and $G_2 = (V_2, E_2, \ell_2)$ be two labelled graphs. A relation $\leq \subseteq V_1 \times V_2$ is a* simulation *from G_1 to G_2 if and only if:*

1. *$a \leq b \Rightarrow \ell_1(a) = \ell_2(b)$;*
2. *$(a \leq b \wedge a \rightarrow c) \Rightarrow \exists d(c \leq d \wedge b \rightarrow d)$;*
3. *for each $a \in V_1$ there exists $b \in V_2$ such that $a \leq b$;*

A graph G_1 is simulated *by a graph G_2 (G_2 simulates G_1) if there exists a simulation from G_1 to G_2. Two graphs G_1 and G_2 are sim-equivalent if G_1 simulates G_2 and G_2 simulates G_1*

Hence, given a graph G it is possible to consider the problem of determining the minimum graph sim-equivalent to G. In the context of simulation, minimality is measured in terms of both the number of nodes and edges. It was only recently proved, in [18], that there always exists a unique smallest labelled graph that is sim-equivalent to G, i.e., there is a unique way to put a minimum number of edges between the elements of $S = V/\equiv_s$ in order to obtain a labelled graph sim-equivalent to G. In the case of bisimulation this was a trivial consequence of the property ($\flat$) (see Subsection 4.2.1), since if a node reaches with an edge an equivalence class, then all the nodes that are equivalent to it reach with an edge the same equivalence class. This is equivalent to say that

$$\forall a, c\ ([a]_b \cap E^{-1}([c]_b) \neq \emptyset \Rightarrow [a]_b \subseteq E^{-1}([c]_b)),$$

which is at the basis of the characterization of the bisimulation problem as (relational) coarsest partition problem. In the case of simulation, it can be the case that there exists a node a which reaches with an edge a node c, while a node which is sim-equivalent to a does not reach any node sim-equivalent to c. In Chapter 5 of this thesis we will prove that, when the above fact happens, we always have that there exists a node d such that a reaches d and $c \leq_s d$. Hence, the property which holds for the maximum simulation is that

$$\forall a, b, c \ ((a \to c \wedge b \in [a]_s) \Rightarrow \exists d(c \leq_s d \wedge b \to d)) \qquad (\sharp)$$

which in particular, since we are working on finite graphs, implies that

$$\forall a, c \ ([a]_s \cap E^{-1}([c]_s) \neq \emptyset \Rightarrow \exists [d]_s([c]_s \leq_s [d]_s \wedge [a]_s \subseteq E^{-1}([d]_s))).$$

This property, at the basis of the *generalized stability* condition that we will introduce in the second part of this thesis, provides a characterization of the simulation problem in terms of a partitioning problem. An immediate consequence is that there is a unique way to put a minimum number of edges among the elements of $V/\equiv_s$ in order to obtain a graph sim-equivalent to G: put an edge from $[a]_s$ to $[d]_s$ if and only if it holds $[a]_s \subseteq E^{-1}([d]_s)$ and there is no class $[c]_s$ such that $[a]_s \cap E^{-1}([c]_s) \neq \emptyset$ with $d \leq_s c$ (i.e., d is not a *little brother* of any c, in the terminology of [18]).

Example 4.2.12 Consider the two graphs in Figure 4.1. The graph on the right is the quotient with respect to the maximum simulation of the graph on the left. In the graph on the left there are no sim-equivalent nodes, hence all the classes in the quotient structure are singletons. In the graph on the left there is a node with label B which is a little brother, hence in the quotient structure we delete an edge. ■

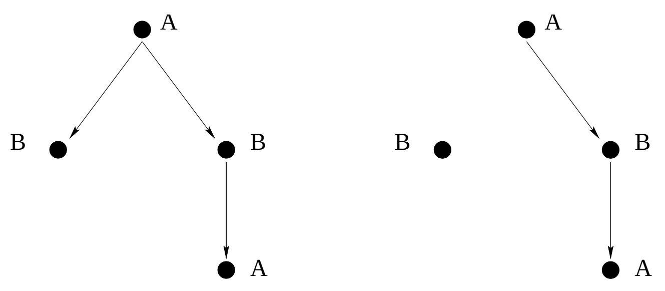


Figure 4.1: Example of little brother and of minimization in terns of edges.

A further important difference between bisimulation and simulation is that, while in the case of bisimulation working without labels is not a restriction (see Subsection 4.2.1), in the case of simulation this is not the case. This is a consequence of the following result whose proof is left to the reader.

Lemma 4.2.13 *Let $G = (V, E, \Sigma)$ be a graph such that $\Sigma = \{V\}$, i.e., all the nodes have the same label. Given a node $a \in V$ let $G(a)$ be the subgraph of G of the nodes reachable from a.*

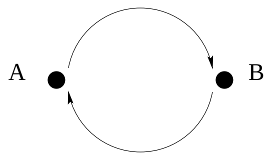
- *If $G(a)$ is acyclic, then for all $b \in V$*

$$b \leq_s a \qquad \textit{iff} \qquad G(b) \textit{ is acyclic and } \textit{rank}(b) \leq \textit{rank}(a).$$

- *If $G(a)$ is cyclic, then for all $b \in V$ it holds $b \leq_s a$.*

This means that in the case without labels (i.e., only one label) all the nodes which belong to the non-well-founded part of the graph are sim-equivalent.

Example 4.2.14 Consider the graph in the figure below on the left.



If we try to remove the labels A and B by adding new nodes and we do not want to use labels also on the new nodes, then, no matter how many new nodes and edges we add, the two nodes of the original graph become sim-equivalent. ■

The previous lemma has also strong implications on the use of the notion of rank in the computation of $\equiv_s$. It implies that

$$a \equiv_s b \quad \not\Rightarrow \quad \textit{rank}(a) = \textit{rank}(b),$$

since in a graph with only one label all the non-well-founded nodes are sim-equivalent even if they are at different ranks. Lemma 4.2.13 suggests that a reasonable definition of rank to be used in the computation of $\equiv_s$ could be

$$\textit{rank}^*(a) \quad = \quad \begin{cases} \max(\{1 + \textit{rank}(c) \mid C(a)E^{scc}C(c)\} & \text{if } a \in WF(G) \\ +\infty & \text{otherwise} \end{cases}$$

i.e., on the well-founded part of the graph we use the usual notion of rank, while all the nodes in the non-well-founded part have rank $+\infty$. Using such a definition we obtain

$$\begin{aligned} a \equiv_s b \quad &\Rightarrow \quad \textit{rank}^*(a) = \textit{rank}^*(b); \\ a \leq_s b \quad &\Rightarrow \quad \textit{rank}^*(a) \leq \textit{rank}^*(b). \end{aligned}$$

This implies that, whenever we use a technique which needs to compute $\leq_s$ in order to determine $\equiv_s$, when we process nodes at rank i we have to keep into account also all the nodes at rank less than i.

A second disadvantage of the use of *rank** is that it is reasonable to believe that at least one half of the nodes have rank $+\infty$, which means that it does not split enough the computation.

For this last reason in [54], in order to exploit a notion of rank in the simulation computation, the opposite approach is taken, i.e., a notion of rank which splits the graph as much as possible is introduced admitting to have sim-equivalent nodes with different ranks. In any case, what is required to the notion of rank is the property that the information at rank i is enough to compute the similarity relations among the nodes of rank at most i. In particular, we want to avoid the situation in which

a and b are two nodes whose rank is at most i, at the end of the i-th iteration we "believe" that $a \equiv_s b$ (resp. $a \not\equiv_s b$) and at the end of the $(i+k)$-th iteration we discover that $a \not\equiv_s b$ (resp. $a \equiv_s b$). A necessary and sufficient condition to obtain the above property is that:

if $a \rightarrow b$, then the rank of b is not greater than the rank of a.

The following notion of rank turns out to satisfy the above condition and splits the graph as much as possible:

Definition 4.2.15 (sim-Rank) *Let $G = (V, E, \Sigma)$, the $rank_s$ of a node is recursively defined as follows:*

$$rank_s(a) \quad = \quad \begin{cases} 0 & \textit{if } C(a) \textit{ is a leaf in } G^{scc} \\ \max\{1 + rank_s(c) \mid C(a) E^{scc} C(c)\} & \textit{otherwise} \end{cases}$$

In [54] the simulation problem is algorithmically analyzed within a constraint logic programming framework, and the above notion of rank is used to give an incremental solution to the simulation problem.

5 From Bisimulation to Simulation: Coarsest Partition Problems

In this chapter we present a new result in the form of a novel algorithm for simulation, based on the mapping of the notion on a suitable coarsest partition problem. Encoding the simulation problem with a *generalized coarsest partition problem* establishes a new fil rouge connecting simulation and bisimulation, whose casting onto a coarsest partition problem is well known in the literature (see Subsection 4.2.1). Such an encoding proved to be a central tool for both a deeper understanding of the algebraic bisimulation properties, and for the development of space/time efficient bisimulation algorithms. To some extent, we show that a similar pattern applies to our simulation encoding.

For the sake of readability and clearness, we choose to collect a number of technical proofs, involved in the following of this chapter, within Appendix A.

5.1 Simulations as Partitioning Problems

We introduce in this section the *Generalized Coarsest Partition Problem* (GCPP) which is the central notion in our approach. The term *generalized* is used since we are not only going to deal with partitions to be refined, as in the classical coarsest partition problems (see Subsection 4.2.1), but with *pairs* in which we have a partition and a binary relation over the partition. The equivalence between the simulation problem and the GCPP will be proved at the end of this section.

Definition 5.1.1 (Partition pair) *Let $G = (V, E)$ be a finite graph. A* partition pair *over G is a pair $\langle \Sigma, P \rangle$ in which Σ is a partition over V, and $P \subseteq \Sigma^2$ is a reflexive and acyclic relation over Σ.*

Notice that a labelled graph $G = (V, E, \Sigma)$ can be seen as a graph $G' = (V, E)$ together with the partition pair $\langle \Sigma, I \rangle$, where I is the identity relation over Σ. Given a graph $G = (V, E, \Sigma)$ we will start considering the partition pair $\langle \Sigma, I \rangle$ and we will modify it in order to obtain a partition pair $\langle S, \preceq\rangle$, where S is the simulation quotient $V/\equiv_s$, while $\preceq$ is the maximal simulation over $V/\equiv_s$.

Given two partitions Π and Σ, such that Π is finer than Σ (i.e., each block of Π is included in a block of Σ), and a relation P over Σ, we use the notation $P(\Pi)$ to refer to the relation *induced* on Π by P, i.e.:

$$\forall \alpha, \beta \in \Pi((\alpha, \beta) \in P(\Pi) \Leftrightarrow \exists \alpha', \beta'((\alpha', \beta') \in P \wedge \alpha \subseteq \alpha' \wedge \beta \subseteq \beta')).$$

Denoting by $\mathcal{P}(G)$ the set of partition pairs over G below, we introduce the partial order over which we will define the notion of coarsest partition pair.

Definition 5.1.2 *Let $\langle \Sigma, P\rangle, \langle \Pi, Q\rangle \in \mathcal{P}(G)$:*

$$\langle \Pi, Q\rangle \sqsubseteq \langle \Sigma, P\rangle \quad \textit{iff} \quad \Pi \textit{ is finer than } \Sigma \textit{ and } Q \subseteq P(\Pi).$$

The search for the coarsest partition pair will proceed using the following transition relations:

Definition 5.1.3 ($E_\exists, E_\forall$) *Let $G = (V, E)$, and Π be a partition of V. The* $\exists$-transition *on Π is the relation*

$$\alpha E_\exists \gamma \quad \textit{iff} \quad \exists a \exists c(a \in \alpha \wedge c \in \gamma \wedge aEc).$$

The $\forall$-transition *on Π relation*

$$\alpha E_\forall \gamma \quad \textit{iff} \quad \forall a(a \in \alpha \Rightarrow \exists c(c \in \gamma \wedge aEc)).$$

Hence, $\alpha E_\forall \gamma$ is a shorthand for $\alpha \subseteq E^{-1}(\gamma)$, while $\alpha E_\exists \gamma$ stands for $\alpha \cap E^{-1}(\gamma) \neq \emptyset$. Similar notations (called *relation transformers* and *abstract transition relations*) were introduced in [35] in order to combine model checking and Abstract Interpretation.

We are now ready to introduce the fundamental notion in the generalized coarsest partition problems, the notion of stability of a partition pair with respect to a relation.

Definition 5.1.4 (Stability) *Let $G = (V, E)$. A partition pair $\langle \Sigma, P\rangle$ over G is* stable *with respect to the relation E if and only if*

$$\forall \alpha, \beta, \gamma \in \Sigma((\alpha, \beta) \in P \wedge \alpha E_\exists \gamma \Rightarrow \exists \delta \in \Sigma((\gamma, \delta) \in P \wedge \beta E_\forall \delta)).$$

The condition in the definition of stability is equivalent to:

$$\begin{aligned}\forall \alpha, \beta, \gamma \in \Sigma((\alpha, \beta) \in P \wedge \alpha \cap E^{-1}(\gamma) \neq \emptyset \Rightarrow \\ \exists \delta \in \Sigma((\gamma, \delta) \in P \wedge \beta \subseteq E^{-1}(\delta))).\end{aligned}$$

As it will be proved (see Theorem 5.1.10) the stability condition is exactly the condition holding between two classes of $V/\equiv_s$: if $\alpha, \beta \in V/\equiv_s$ with $\alpha \leq_s \beta$ (i.e., all the elements of α are simulated by all the elements of β) and an element a in α reaches an element c in γ, then all the elements b of β must reach at least one element d which simulates c. In particular, considering all the $\leq_s$-maximal elements d simulating c reached by elements in β, we have that all the elements in β reach a class δ which simulates c and, hence, which simulates γ.

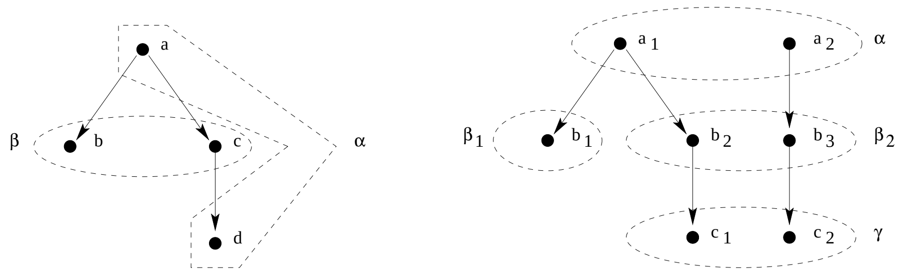


Figure 5.1: G and Σ on the left, G' and Π on the right.

Example 5.1.5 Consider the graph G depicted on the left in Figure 5.1 and the partition pair $\langle \Sigma, I\rangle$, where $\Sigma = \{\alpha, \beta\}$, with α and β shown in Figure 5.1, and I is the identity relation. The partition pair $\langle \Sigma, I\rangle$ is not stable since: $\beta E_{\exists}\alpha$, α is the only class in Σ such that $(\alpha, \alpha) \in I$, and it does not hold $\beta E_{\forall}\alpha$.

Consider now the graph G' depicted on the right in Figure 5.1 and the partition pair $\langle \Pi, P\rangle$, where $\Pi = \{\alpha, \beta_1, \beta_2, \gamma\}$ is shown in Figure 5.1, while $P = \{(\beta_1, \beta_2)\} \cup I$, with I identity relation. It is easy to prove that $\langle \Pi, P\rangle$ is stable.

If the partition $\{\{a_1, a_2\}, \{b_1, b_2, b_3\}, \{c_1, c_2\}\}$ represents the labelling on G', then Π and Pcorrespond to the simulation equivalence and the maximal simulation on G', respectively. ■

We use the notion of stability of a partition pair with respect to a relation in the definition of generalized coarsest partition problem, in the same way as the notion of stability of a partition with respect to a relation is used in the definition of coarsest partition problem.

Definition 5.1.6 (Generalized Coarsest Partition Problem) *Given a graph G and partition pair $\langle \Sigma, P\rangle$ over G the* generalized coarsest partition problem *consists in finding a partition pair $\langle S, \preceq\rangle$ such that:*

(a) $\langle S, \preceq\rangle \sqsubseteq \langle \Sigma, P^+\rangle$;

(b) $\langle S, \preceq\rangle$ is stable with respect to E;

(c) $\langle S, \preceq\rangle$ is $\sqsubseteq$-maximal satisfying (a) and (b).

The partition pair $\langle S, \preceq\rangle$ is a stable refinement *of $\langle \Sigma, P\rangle$.*

Notice that in the above definition $\langle S, \preceq\rangle$ is a refinement of $\langle \Sigma, P^+\rangle$, while it can be the case that it is not a refinement of $\langle \Sigma, P\rangle$. This ensures that $\preceq$ is transitive. In general, the solution of the GCPP can always be found by a suitable sequence of splits (of classes) and adequate completion of the relation in order to take the newly generated classes into account.

Remark 5.1.7 Notice that it is important that $\preceq$ is reflexive (which is the case since $\langle S, \preceq\rangle$ is a partition pair). This is necessary in order to prove that there is always a unique maximal solution. Consider the graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(a, a), (a, b), (b, c)\}$, and the partition pair $\langle \Sigma, P \rangle$ with $\Sigma = \{V\}$ and $P = \{(V, V)\}$. Assume we would not require, in the solution to the GCPP, to have a reflexive relation on the final partition. Then, we would have the following four maximal solution to the GCPP applied to $\langle \Sigma, P \rangle$. Note that only the first pair enjoys reflexivity and is a partition pair.

- $\langle \Pi = \{\alpha, \beta, \gamma\}, Q = \{(\gamma, \beta), (\beta, \alpha), (\alpha, \alpha), (\beta, \beta), (\gamma, \gamma)\} \rangle$,
  where $\alpha = \{a\}, \beta = \{b\}, \gamma = \{c\}$
- $\langle \Pi' = \{\alpha', \beta'\}, Q' = \{(\beta', \alpha'), (\alpha', \alpha')\rangle$,
  where $\alpha' = \{a\}, \beta' = \{b, c\}$
- $\langle \Pi'' = \{\alpha'', \beta''\}, Q'' = \{(\beta'', \beta''), (\beta'', \alpha'')\} \rangle$,
  where $\alpha'' = \{a, b\}, \beta'' = \{c\}$
- $\langle \Sigma, \emptyset \rangle$

■

Our next task is to prove that a given GCPP has always a unique solution. To this end we exploit a connection between the similation problems and the GCPP's.

Theorem 5.1.8 (Existence and Uniqueness) *Given $G = (V, E)$ and a partition pair $\langle \Sigma, P \rangle$ over G, the GCPP over G and $\langle \Sigma, P \rangle$ has always a unique solution.*

Proof. See Appendix A. ■

Again by exploiting the connection between the simulation problems and the GCPP's we can prove the following result.

Corollary 5.1.9 *If $\langle S, \preceq \rangle$ is the solution of the GCPP over G and $\langle \Sigma, P \rangle$, then $\preceq$ is a partial order over S.*

Proof. See Appendix A. ■

Hence, each generalized coarsest partition problem has a unique solution which can be determined by solving a simulation problem. The following easily proved result states that the converse also holds.

Theorem 5.1.10 (Simulation as GCPP) *Let $G = (V, E, \Sigma)$ and let $\langle S, \preceq \rangle$ be the solution of the GCPP over $G^- = (V, E)$ and $\langle \Sigma, I \rangle$, where I is the identity relation. S is the simulation quotient of G, i.e., $S = V/\equiv_s$, and $\leq_s$ defined as*

$$a \leq_s b \quad \textit{iff} \quad [a]_s \preceq [b]_s$$

is the maximal simulation over G.

Proof. See Appendix A. ■

By the above theorem, in order to solve the problem of determining the simulation quotient of a labelled graph $G = (V, E, \Sigma)$ we can equivalently solve the generalized coarsest partition problem over (V, E) and $\langle \Sigma, I \rangle$.

5.1.1 Solving the Generalized Coarsest Partition Problems

To give an operational content to the results in the previous section, we now introduce an operator σ, mapping partition-pairs into partition-pairs, which will turn out to be the engine of our simulation algorithm, in Section 5.2. A procedure which computes σ will be used to solve GCPP's and, hence, to compute similarity quotients: it is only necessary to iterate the computation of σ at most $|S|$ times.

The operator σ is defined in such a way that it refines the partition-pair $\langle \Sigma, P \rangle$ obtaining a partition-pair $\langle \Pi, Q \rangle$ which is *more stable* than $\langle \Sigma, P \rangle$ and is never finer than the solution of the GCPP over $\langle \Sigma, P \rangle$.

σ is specified by three conditions and, in the first one, we simply impose to split those classes of Σ which do not respect the stability condition with respect to themselves: if a class α is such that $\alpha E_{\exists} \gamma$ and it does not exists a class δ such that $(\gamma, \delta) \in P$ and $\alpha E_{\forall} \delta$, then the pair (α, α) does not respect the stability condition, hence we must split α. The first condition is used to build Π, while the second and the third conditions in σ are used to define Q on Π. Intuitively, the second and the third conditions allow to obtain Q from P by starting from $P(\Pi)$ and removing the maximum number of pairs which contradict stability.

Definition 5.1.11 (Operator σ) *Let $G = (V, E)$ and $\langle \Sigma, P \rangle$ be a partition pair over G. The partition pair $\langle \Pi, Q \rangle = \sigma(\langle \Sigma, P \rangle)$ is defined as:*
(1σ) Π *is the coarsest partition finer than* Σ *such that*

$$(a) \quad \forall \alpha \in \Pi \, \forall \gamma \in \Sigma (\alpha E_{\exists} \gamma \Rightarrow \exists \delta \in \Sigma ((\gamma, \delta) \in P \wedge \alpha E_{\forall} \delta));$$

(2σ) *Q is maximal such that $Q \subseteq P(\Pi)$ and if $(\alpha, \beta) \in Q$, then:*

$$\begin{array}{ll} (b) \quad \forall \gamma \in \Sigma (\alpha E_{\forall} \gamma \Rightarrow \exists \gamma' \in \Sigma ((\gamma, \gamma') \in P \wedge \beta E_{\exists} \gamma')) & \quad \textit{and} \\ (c) \quad \forall \gamma \in \Pi (\alpha E_{\forall} \gamma \Rightarrow \exists \gamma' \in \Pi ((\gamma, \gamma') \in Q \wedge \beta E_{\exists} \gamma')). & \end{array}$$

By abuse of notation we use $E_{\exists}$ and $E_{\forall}$ also when the classes belong to different partitions.

Condition (a) in (1σ) imposes to suitably split the classes of the partition Σ: these splits are forced by the fact that we are looking for a partition-pair *stable* with respect to the relation of the given graph and we know that in each partition-pair $\langle \Sigma, P \rangle$ the second component is reflexive. Using condition (b) in (2σ), together with condition (a) in (1σ) and exploiting the fact that P is acyclic, it is possible to prove that Q is acyclic: the acyclicity of P ensures that a cycle could arise only among classes of Π which are all contained in a unique class of Σ, then using condition (a) in (1σ) and (b) in (2σ) we obtain a contradiction. Condition (c) is fundamental, together with condition (a), in order to obtain the result in Theorem 5.1.14: if it holds that $\alpha E_{\forall} \gamma$, then *no matter how we split* α one of the subclasses generated from α has a chance (in the solution of GCPP) to be in relation with at least one of the subclasses generated from β only if $\beta E_{\exists} \gamma'$ and γ is in relation with γ'. The following results guarantee the correctness of σ and, hence, of our approach.

Lemma 5.1.12 (Existence) *Let $G = (V, E)$ and $\langle \Sigma, P \rangle$ be a partition pair over G. There exists at least one partition pair $\langle \Pi, Q \rangle$ which satisfies the conditions in Definition 5.1.11, i.e., σ is always defined.*

Proof. Consider the partition Π_1 defined as $\Pi_1 = \{\{a\} \mid a \in V\}$. Clearly Π_1 is finer than Σ.
Π_1 satisfies all the conditions in $(1)\sigma$ (but it can be the case that it is not a coarsest), since P is reflexive.
Let Π be a partition such that:

- Π is finer than Σ;
- Π_1 is finer than Π;
- Π satisfies all the conditions in $(1)\sigma$;
- if Π is finer than Π_2 and Π_2 is finer than Σ, then Π_2 does not satisfy the conditions in $(1)\sigma$.

Notice that there exists at least a partition Π which satisfies all these conditions because there are only a finite number of partitions.

Consider $Q_1 = \{(\alpha, \alpha) \mid \alpha \in \Pi\}$ over Π.
Clearly Q_1 satisfies all the conditions in $(2)\sigma$, but it can be the case that it is not maximal.
Let Q be a relation over Π such that:

- $Q_1 \subseteq Q \subseteq P(\Pi)$;
- Q_1 is acyclic;
- Q satisfies all the conditions in $(2)\sigma$;
- if Q_2 is such that $Q \subseteq Q_2 \subseteq P(\Pi)$, then Q_2 does not satisfy the conditions in $(2)\sigma$.

There exists at least a relation Q which satisfies all these conditions because there are only a finite number of relations. ■

Theorem 5.1.13 (Uniqueness) *Let $G = (V, E)$ and $\langle \Sigma, P \rangle$ be a partition pair over G. There exists a unique maximal pair $\langle \Pi, Q \rangle$ which satisfies the conditions in Definition 5.1.11, i.e., σ is a function.*

Proof. The previous lemma states that there always exists a maximal pair $\langle \Pi, Q \rangle$ which satisfies the conditions in Definition 5.1.11; we have to prove that this maximal pair is unique.
Let Sort be a reverse topological sorting of the elements in Σ with respect to P. Consider Π defined in Figure 5.2.

Notice that Π does not depend on the topological sorting we have chosen.
It is clear that Π satisfies (1σ).

```
$\Pi := \Sigma$;
while $\mathsf{Sort} \neq \emptyset$ do
        $\gamma := dequeue(\mathsf{Sort})$;
        for each $\alpha \in \Pi$ do
                if $\alpha E_\exists \gamma \wedge \forall \delta((\gamma, \delta) \in P \Rightarrow \neg \alpha E_\forall \delta)$ then
                        replace $\alpha$ with $\alpha \cap E^{-1}(\gamma)$ and $\alpha \setminus E^{-1}(\gamma)$ in $\Pi$
```

Figure 5.2:

Let us assume by contradiction that there exists Π_1 such that Π_1 satisfies (1σ) and Π_1 is not finer than Π.
This means that there exist $a, b \in V$, $\alpha \in \Pi_1$ and $\alpha_a, \alpha_b \in \Pi$ such that $a, b \in \alpha$, $a \in \alpha_a$, $b \in \alpha_b$ and $\alpha_a \neq \alpha_b$.
Consider the class α' which was obtained during the construction of Π and which was the smallest class obtained during the construction such that $\alpha \subseteq \alpha'$.
Let γ be the class of Σ which first split α' into two parts.
Notice that for all δ such that $(\gamma, \delta) \in P$ and $\delta \neq \gamma$ it holds $\neg \alpha' E_\exists \delta$ (since all these δ's have already been extracted from Sort).
Hence it holds that for all δ such that $(\gamma, \delta) \in P$ and $\delta \neq \gamma$ $\neg \alpha E_\forall \delta$, moreover, since α is not a subset of one of the two part of α', $\neg \alpha E_\forall \gamma$ and $\alpha E_\exists \gamma$. This is in contradiction with the fact that Π_1 satisfies (1σ).

Similarly it is possible to prove that there exists a unique maximal relation Q over Π which satisfies (2σ).

It is possible to obtain Q from $P(\Pi)$ by removing all the pairs which do not fulfill the condition, until a fix point is reached. It is immediate to prove that such a fix point is reflexive. In fact, being $\langle \Sigma, P \rangle$ a partition pair over G, we have that P and hence $P(\Pi)$ are reflexive. As each pair belonging to $\{(\alpha, \alpha) \mid \alpha \in \Pi\}$ respect both condition (b) in (2σ) and condition (c) in (2σ) we can conclude that Q is reflexive.

As far as the acyclicity of Q is concerned consider the relation Q_b on Π obtained from $P(\Pi)$ by removing those pairs which don't respect condition (b) in (2σ); we will prove that Q_b is acyclic. Being Q contained in Q_b we will obtain, as a byproduct, the acyclicity of Q.

Assume by contradiction that Q_b is cyclic, i.e., there exist $\alpha_1, \alpha_2, .., \alpha_n \in \Pi$ with $\alpha_i \neq \alpha_j$ for $1 \leq i, j \leq n, i \neq j$ such that $(\alpha_1, \alpha_2) \in Q_b \wedge .. \wedge ..(\alpha_{n-1}, \alpha_n) \in Q_b \wedge (\alpha_n, \alpha_1) \in Q_b$. Since P is acyclic and $Q \subseteq P(\Pi)$ there must exists a class $\alpha \in \Sigma$ such that $\alpha_1, \ldots, \alpha_n \subseteq \alpha$.
Let α be the class in Σ such that $\alpha_i \subseteq \alpha$ for $1 \leq i \leq n$. We have just proved that Π i.e the coarsest partition finer than Σ which respect condition (a) in (2σ) can be obtained using the procedure in Figure 5.2, where Sort is a reverse topological sorting of the elements in Σ with respect to P. As, for $1 \leq i, j \leq n \wedge i \neq j$, we have $\alpha_i \neq \alpha_j$ and $\alpha_i \subseteq \alpha \in \Sigma$, by the above pseudo-code we deduce that there exist $\gamma \in \Sigma$, a permutation of $\alpha_1, .., \alpha_n$, say $\alpha'_1, .., \alpha'_n$, and an index $1 \leq k \leq n-1$ such that

$$\alpha'_1 E_\forall \gamma \wedge .. \wedge \alpha'_k E_\forall \gamma \wedge \nexists \delta \in \Sigma((\gamma, \delta) \in P \wedge (\alpha'_{k+1} E_\exists \delta \vee .. \vee \alpha'_n E_\exists \delta)) \qquad (1)$$

Let's use the letters S and V to refer respectively to $\alpha'_1 \cup .. \cup \alpha'_k$ and $\alpha'_{k+1} \cup .. \cup \alpha'_n$. As $\alpha'_1, .., \alpha'_n$ is a permutation of $\alpha_1, .., \alpha_n$ and we have supposed that $(\alpha_1, \alpha_2) \in Q_b \wedge .. \wedge ..(\alpha_{n-1}, \alpha_n) \in Q_b \wedge (\alpha_n, \alpha_1) \in Q_b$ we have that there exist two indexes $1 \leq t, s \leq n$ such that $s = t+1 \vee (t = n \wedge s = 1)$ and $\alpha_t \in S \wedge \alpha_s \in V$. From $\alpha_t \in S$ we obtain that $\alpha_t E_\forall \gamma$; since $(\alpha_t, \alpha_s) \in Q_b$ and Q_b respect condition (b) in (2σ) we have that $\alpha_s E_\exists \delta \wedge (\gamma, \delta) \in P$ and $\alpha_s \in V$ which contradicts the condition in Equation (1). ■

Fix points of the σ operator can be used to compute the solution of the GCPP.

Theorem 5.1.14 (Fix Point) *Let $G = (V, E)$ and $\langle \Sigma, P \rangle$ a partition pair over G with P transitive. Let $\langle S, \preceq \rangle$ be the solution of the GCPP over G and $\langle \Sigma, P \rangle$. If n is such that $\sigma^n(\langle \Sigma, P \rangle) = \sigma^{n+1}(\langle \Sigma, P \rangle)$, then $\sigma^n(\langle \Sigma, P \rangle) = \langle S, \preceq \rangle$.*

Proof. See Appendix A. ■

Estimating how large is the index n in the worst case allows us to point out another important property of the operator σ. When we iteratively apply the operator σ until we reach a fix point, at each iteration we refine a partition and we remove pairs from a relation. What we prove in the following theorem is that at each iteration in which we do refine the partition. In a certain sense this means that the two conditions we have given in (2σ) to remove pairs are *optimal*.

Theorem 5.1.15 (Complexity) *Let $G = (V, E)$ and $\langle \Sigma, P \rangle$ be a partition pair over G with P transitive. Let $\langle S, \preceq \rangle$ be the solution of the GCPP over G and $\langle \Sigma, P \rangle$. If i is such that $\sigma^i(\langle \Sigma, P \rangle) = \langle \Sigma_i, \mathtt{P}_i \rangle$ and $\sigma^{i+1}(\langle \Sigma, P \rangle) = \langle \Sigma_i, \mathtt{P}_{i+1} \rangle$, then $\mathtt{P}_{i+1} = \mathtt{P}_i$ and $\langle \Sigma_i, \mathtt{P}_i \rangle = \langle S, \preceq \rangle$.*

Proof. Let $\langle S, \preceq \rangle$ be the solution of the GCPP on G and $\langle \Sigma, P \rangle$. Using the same reasoning-schema of Theorem 5.1.14 we obtain that $\langle S, \preceq \rangle \sqsubseteq \langle \Sigma_i, \mathtt{P}_{i+1} \rangle \sqsubseteq \langle \Sigma_i, \mathtt{P}_i \rangle$. If we prove that $\langle \Sigma_i, \mathtt{P}_i^+ \rangle$ is stable we can conclude by Lemma A.0.4 that $\langle S, \preceq \rangle \sqsupseteq \langle \Sigma_i, \mathtt{P}_i^+ \rangle$ and hence $\langle S, \preceq \rangle = \langle \Sigma_i, \mathtt{P}_i \rangle$ and $\mathtt{P}_{i+1} = \mathtt{P}_i$.
Let $\alpha, \beta, \gamma \in \Sigma_i$ be such that $(\alpha, \beta) \in \mathtt{P}_i^+$ and $\alpha E_\exists \gamma$. From the fact that $(\alpha, \beta) \in \mathtt{P}_i^+$ we have that there exists $(\alpha, \beta_1), \ldots, (\beta_n, \beta) \in \mathtt{P}_i$. Since $\alpha E_\exists \gamma$ and $\Sigma_{i+1} = \Sigma_i$, we have that there exists γ' such that $(\gamma, \gamma') \in \mathtt{P}_i$ and $\alpha E_\forall \gamma'$. Since $\mathtt{P}_i$ is obtained from $\sigma^i(\langle \Sigma, P \rangle)$ we obtain that there exists δ'_1 such that $(\gamma', \delta'_1) \in \mathtt{P}_i$ and $\beta_1 E_\exists \delta'_1$. Using again the fact that $\Sigma_{i+1} = \Sigma_i$ we obtain that there exists δ_1 such that $(\delta'_1, \delta_1) \in \mathtt{P}_1$ and $\beta_1 E_\forall \delta_1$. By induction on n we arrive to the conclusion that there exists δ such that $(\gamma, \delta) \in \mathtt{P}_i^+$ and $\beta E_\forall \delta$. ■

Corollary 5.1.16 *The solution $\langle S, \preceq \rangle$ of the GCPP over a graph G and a partition pair $\langle \Sigma, P \rangle$ can be computed iterating σ at most $|S|$ times.*

The characterizations obtained in this section allow us to conclude that if we are able to define a procedure which, given a partition-pair $\langle \Sigma, P \rangle$, computes $\sigma(\langle \Sigma, P \rangle)$, then we can use it to solve GCPP's and hence to compute similarity quotients. In particular, we recall that given a similarity quotient problem over a labelled graph $G = (V, E, \Sigma)$ in order to solve it it is sufficient solve the GCPP over G and $\langle \Sigma, I \rangle$ (notice that I is trivially transitive).

Moreover, the last corollary ensures that it will be necessary to iterate the procedure which computes σ at most $|S|$ times. This is a first improvement on time complexity with respect to the algorithm presented in [18]: in a certain sense [18] computes an operator which refines the partition-pair less than σ, and hence it is possible that the computation has to be iterated up to $|S|^2$ times.

In the next section we present the procedure which computes σ.

5.2 The Partitioning Algorithm

In this section we propose an algorithm which solves the generalized coarsest partition problem we have presented in the previous section. The operator σ, also introduced in previous section, will be the engine of our procedure. The PARTITIONING ALGORITHM (see Figure 5.3) takes as input a graph $G = (V, E)$ and a partition pair $\langle \Sigma, P \rangle$, with P transitive, calls the two functions REFINE (see Figure 5.4) and UPDATE (see Figure 5.5) until a fix point is reached, and returns the partition pair $\langle S, \preceq \rangle$ which is the solution of the GCPP over G and $\langle \Sigma, P \rangle$. In order to solve the GCPP over G and $\langle \Sigma, P \rangle$, with P not transitive, it is sufficient to first compute P^+, the cost of this operation is $\mathcal{O}(\Sigma^3)$, and hence it does not affect the global cost of the algorithm. The function REFINE takes as input a partition pair $\langle \Sigma_i, P_i \rangle$ and it returns the partition Σ_{i+1} which is the coarsest that satisfies the condition in (1σ) of Definition 5.1.11. The function UPDATE takes as input a partition pair $\langle \Sigma_i, P_i \rangle$ and the refinement Σ_{i+1} and it produces the acyclic and reflexive relation over Σ_{i+1} which is the greatest satisfying the conditions in (2σ) of Definition 5.1.11. The quotient structures defined below, will be used.

Definition 5.2.1 (Quotient Structures) *Let $G = (V, E)$ and Π be a partition of V. Let $E_\exists$ and $E_\forall$ be the relations in Definition 5.1.3.*
The $\exists$-quotient structure *over Π is the graph $\Pi_\exists = (\Pi, E_\exists)$.*
The $\forall$-quotient structure *over Π is the graph $\Pi_\forall = (\Pi, E_\forall)$.*
The $\exists\forall$-quotient structure *over Π is the structure $\Pi_{\exists\forall} = (\Pi, E_\exists, E_\forall)$.*

Definition 5.2.2 (Induced Structure) *Let $G = (V, E)$. Let Σ and Π be two partitions of V with Π finer than Σ the* $\exists\forall$-induced quotient structure *over Π is the structure $\Sigma_{\exists\forall}(\Pi) = (\Pi, E_\exists^\Sigma(\Pi), E_\forall^\Sigma(\Pi)))$, where:*

$$\begin{array}{lcc} \alpha E_\exists^\Sigma(\Pi)\gamma & \textit{iff} & \alpha \cap E^{-1}(\gamma) \neq \emptyset \\ \alpha E_\forall^\Sigma(\Pi)\gamma & \textit{iff} & \alpha E_\exists^\Sigma(\Pi)\gamma \wedge \alpha \subseteq E^{-1}(\gamma') \wedge \gamma \subseteq \gamma' \in \Sigma \end{array}$$

with $\alpha, \beta \in \Pi$.

At the end of each iteration of the while-loop in the PARTITIONINGALGORITHM we have that $\langle \Sigma_{i+1}, P_{i+1} \rangle = \sigma(\langle \Sigma_i, P_i \rangle)$.

```
PartitioningAlgorithm($(V,E)$, $\langle\Sigma, P\rangle$)

(1) change := $\top$;
(2) $i := 0; \Sigma_0 := \Sigma; \mathtt{P}_0 := P$;

– Refine the partition pair (quotient graph) until stability with respect to $E$ –
(3) while (change) do
(4)     change := $\bot$;
(5)     $\Sigma_{i+1}$ := refine($\Sigma_i, \mathtt{P}_i$, change);
(6)     $\mathtt{P}_{i+1}$ := update($\Sigma_i, \mathtt{P}_i, \Sigma_{i+1}$);
(7)     $i := i+1$;
```

Figure 5.3: The PartitioningAlgorithm.

It is immediate to see that the refine function works exactly as described in the proof of Theorem 5.1.13, and hence it produces the partition which is the first element of $\sigma(\Sigma_i, \mathtt{P}_i)$.

Corollary 5.2.3 *If $\sigma(\langle\Sigma_i, \mathtt{P}_i\rangle) = \langle\Pi, Q\rangle$, then $\Pi = \Sigma_{i+1}$.*

update removes pairs from $\mathtt{P}_i(\Sigma_{i+1}) = \mathtt{Ind}_{i+1}$ in order to obtain the relation $\mathtt{P}_{i+1}$ which satisfies the two conditions in (2σ) of Definition 5.1.11. We start with the relation $\mathtt{Ind}_{i+1}$, which is nothing but the relation induced on Σ_{i+1} by $\mathtt{P}_i$, and the $\exists\forall$-induced structure on Σ_{i+1}. We obtain $\mathtt{Ref}_{i+1}$ by removing from $\mathtt{Ind}_{i+1}$ all the pairs (α, β) such that there exists $\gamma \in \Sigma_i$ such that $\alpha E_\forall \gamma$ and there does not exists $\gamma' \in \Sigma_i$ such that $\beta E_\exists \gamma'$. Hence, $\mathtt{Ref}_{i+1}$ satisfies condition *(b)*, but not necessarily condition *(c)*, of (2σ). In order to guarantee also condition *(c)* we build the quotient structure $\Sigma_{(i+1)\exists\forall}$ and we remove all pairs not satisfying condition *(c)* from $\mathtt{Ref}_{i+1}$. Hence, $\mathtt{P}_{i+1}$ satisfies (2σ).

The deletion of "wrong" pairs is performed by new_HHK (see Figure 5.6), which is a version of [63] adapted to our purposes here.

Notice that the space complexity of the calls to the adapted version of [63] remains limited since they are made on quotient structures. This function is based on the use of the two structures $\Sigma_{i\exists\forall}(\Sigma_{i+1})$ (cf. Definition 5.2.2) and $\Sigma_{i+1\exists\forall}$ (cf. Definition 5.2.1), and on the following equivalent formulation of the second condition in (2σ).

Proposition 5.2.4 *Let $G = (V, E)$, $\langle\Sigma, P\rangle$ be a partition pair and Π be a partition finer than Σ. Q satisfies (2σ) of Definition 5.1.11 if and only if Q is the maximal relation over Π such that $Q \subseteq P(\Pi)$ and if $(\alpha, \beta) \in Q$, then:*

$$\forall\gamma \in \Pi(\alpha E^{\Sigma}_{\forall}(\Pi)\gamma \Rightarrow \exists\gamma' \in \Pi((\gamma, \gamma') \in P(\Pi) \wedge \beta E^{\Sigma}_{\exists}(\Pi)\gamma'))$$
$$\forall\gamma \in \Pi(\alpha E_\forall \gamma \Rightarrow \exists\gamma' \in \Pi((\gamma, \gamma') \in Q \wedge \beta E_\exists \gamma')),$$

```
$\textsc{Refine}(\Sigma_i, \mathtt{P}_i, \mathsf{change})$

(1)  $\Sigma_{i+1} := \Sigma_i$;
(2)  for each $\alpha \in \Sigma_{i+1}$ do $\mathtt{Stable}(\alpha) := \emptyset$;
(3)  for each $\gamma \in \Sigma_i$ do $\mathtt{Row}(\gamma) := \{\gamma' \mid (\gamma, \gamma') \in \mathtt{P}_i)\}$;
(4)  Let $\mathtt{Sort}$ be a reverse topological sorting of $\langle \Sigma_i, \mathtt{P}_i \rangle$;

– Split the classes in $\Sigma_i$ using maximal classes with respect to $\mathtt{P}_i$ –
(5)  while $(\mathtt{Sort} \neq \emptyset)$ do
(6)      $\gamma := dequeue(\mathtt{Sort})$;
(7)      $A := \emptyset$;
(8)      for each $(\alpha \in \Sigma_{i+1}, \alpha E_\exists \gamma, \mathtt{Stable}(\alpha) \cap \mathtt{Row}(\gamma) = \emptyset)$ do
(9)          $\alpha_1 := \alpha \cap E^{-1}(\gamma)$;
(10)         $\alpha_2 := \alpha \setminus \alpha_1$;
(11)         if $\alpha_2 \neq \emptyset$ then $\mathtt{change} := \top$;
(12)         $\Sigma_{i+1} := \Sigma_{i+1} \setminus \{\alpha\}$;
(13)         $A := A \cup \{\alpha_1, \alpha_2\}$;
(14)         $\mathtt{Stable}(\alpha_1) := \mathtt{Stable}(\alpha) \cup \{\gamma\}$;
(15)         $\mathtt{Stable}(\alpha_2) := \mathtt{Stable}(\alpha)$;
(16)     $\Sigma_{i+1} := \Sigma_{i+1} \cup A$;
(17)     $\mathtt{Sort} := \mathtt{Sort} \setminus \{\gamma\}$;
(18)  return $\Sigma_{i+1}$
```

Figure 5.4: The Refine procedure.

```
$\textsc{Update}(\Sigma_i, \mathtt{P}_i, \Sigma_{i+1})$

(1)  $\mathtt{Ind}_{i+1} := \{(\alpha_1, \beta_1) \mid \alpha_1, \beta_1 \in \Sigma_{i+1}, \alpha_1 \subseteq \alpha, \beta_1 \subseteq \beta(\alpha, \beta) \in \mathtt{P}_i\}$;
(2)  $\Sigma_{i\exists\forall}(\Sigma_{i+1}) := \langle \Sigma_{i+1}, E_\exists^{\Sigma_i}(\Sigma_{i+1}), E_\forall^{\Sigma_i}(\Sigma_{i+1}) \rangle$;
(3)  $\mathtt{Ref}_{i+1} := \textsc{NewHHK}(\Sigma_{i\exists\forall}(\Sigma_{i+1}), \mathtt{Ind}_{i+1}, \bot)$;
(4)  $\Sigma_{(i+1)\exists\forall} := \langle \Sigma_{i+1}, E_\exists, E_\forall \rangle$;
(5)  $\mathtt{P}_{i+1} := \textsc{NewHHK}(\Sigma_{(i+1)\exists\forall}, \mathtt{Ref}_{i+1}, \top)$;
(6)  return $\mathtt{P}_{i+1}$
```

Figure 5.5: The Update function.

where $E_\forall^\Sigma(\Pi)$ and $E_\exists^\Sigma(\Pi)$ are the edges of the $\exists\forall$-induced quotient structure, while $E_\exists$ and $E_\forall$ are the edges of the $\exists\forall$ quotient structure.

```
NewHHK(T, R_1, R_2), K, U)

(1)  $P := K$;
(2)  for each $c \in T$do
(3)      $sim(c) := \{e \mid (c,e) \in K\}$;
(4)      $rem(c) := T \setminus pre_1(sim(c))$;
(5)  while $\{c \mid rem(c) \neq \emptyset\} \neq \emptyset$do
(6)      let $c \in \{c \mid rem(c) \neq \emptyset\}$;
(7)      while $rem(c) \neq \emptyset$ do ;
(8)          let $b \in rem(c)$;
(9)          $rem(c) := rem(c) \setminus \{b\}$;
(10)          for each $c \in T$ do
(11)          if $b \in sim(a)$then
(12)              $sim(a) := sim(a) \setminus \{b\}$;
(13)              $P := P \setminus \{(a,b)\}$;
(14)              if $(U)$ then for each $b_1 \in pre_1(b)$do
(15)                  if $post_1(b_1) \cap sim(a) = \emptyset$then
(16)                      $rem(a) := rem(a) \cup \{b_1\}$;
(17)   return $P$
```

Figure 5.6: The NewHHK function.

Proof. In order to have our thesis we only need to prove that

$$\forall \gamma \in \Pi(\alpha E^{\Sigma}_{\forall}(\Pi)\gamma \Rightarrow \exists \gamma' \in \Pi((\gamma,\gamma') \in P(\Pi) \wedge \beta E^{\Sigma}_{\exists}(\Pi)\gamma')) \Leftrightarrow$$

$$\forall \gamma \in \Sigma(\alpha E_{\forall}\gamma \Rightarrow \exists \gamma' \in \Sigma((\gamma,\gamma') \in P \wedge \beta E_{\exists}\gamma)).$$

($\Rightarrow$) Assume, by contradiction, that

$$\forall \gamma \in \Pi(\alpha E^{\Sigma}_{\forall}(\Pi)\gamma \Rightarrow \exists \gamma' \in \Pi((\gamma,\gamma') \in P(\Pi) \wedge \beta E^{\Sigma}_{\exists}(\Pi)\gamma'))\wedge$$

$$\exists \gamma \in \Sigma(\alpha E_{\forall}\gamma \wedge \forall \gamma' \in \Sigma((\gamma,\gamma') \in P \Rightarrow \neg\beta E_{\exists}\gamma)). \tag{1}$$

Let $\gamma \in \Sigma$ be the class such that $(\alpha E_{\forall}\gamma \wedge \forall \gamma' \in \Sigma((\gamma,\gamma') \in P \Rightarrow \neg\beta E_{\exists}\gamma))$. As $\alpha E_{\forall}\gamma$ we can find a class of Π, γ^1 such that $\alpha E_{\exists}\gamma^1 \wedge \gamma^1 \subseteq \gamma$.

By construction of $\Sigma_{i\exists\forall}(\Sigma_{i+1})$ we have that $\alpha E^{\Sigma}_{\forall}(\Pi)\gamma^1$ and hence (by equation 1) that there exists a class in Π, say δ^1, such that $\beta E^{\Sigma}_{\exists}(\Pi)\delta^1 \wedge (\gamma^1,\delta^1) \in P(\Pi)$.
Let δ be the class of Σ such that $\delta^1 \subseteq \delta$. As $\beta E^{\Sigma}_{\exists}(\Pi)\delta^1 \wedge (\gamma^1,\delta^1) \in P(\Pi)$ we deduce, by definition of $P(\Pi)$ and by construction of $\Sigma_{i\exists\forall}(\Sigma_{i+1})$, that $\beta E_{\exists}\delta \wedge (\gamma,\delta) \in P$ which contradicts our assumption. ($\Leftarrow$) Assume by contradiction that

$$\forall \gamma \in \Sigma(\alpha E_{\forall}\gamma \Rightarrow \exists \gamma' \in \Sigma((\gamma,\gamma') \in P \wedge \beta E_{\exists}\gamma))\wedge$$

$$\exists \gamma \in \Pi(\alpha E^{\Sigma}_{\forall}(\Pi)\gamma \wedge \forall \gamma' \in \Pi((\gamma,\gamma') \in P(\Pi) \Rightarrow \neg\beta E^{\Sigma}_{\exists}(\Pi)\gamma')) \tag{2}$$

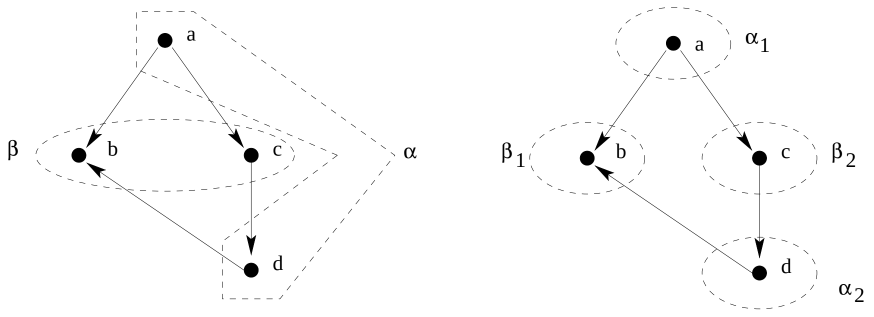


Figure 5.7: The conditions in (2σ) on the quotient structures.

Let γ^1 be the class of Π such that $\alpha E^{\Sigma}_{\forall}(\Pi)\gamma^1 \wedge \forall \delta^1 \in \Pi((\gamma^1, \delta^1) \in P(\Pi) \Rightarrow \neg \beta E^{\Sigma}_{\exists}(\Pi)\delta^1)$. Consider $\gamma \in \Sigma$ such that $\gamma^1 \subseteq \gamma$. By construction of $\Sigma_{i\exists\forall}(\Sigma_{i+1})$ (as $\alpha E^{\Sigma}_{\forall}(\Pi)\gamma^1$) we have that $\alpha E_{\forall}\gamma$ and hence, by equation (2), that $\exists \delta \in \Sigma$ such that $(\gamma, \delta) \in P \wedge \beta E_{\exists}\delta$. Let $\delta^1 \in \Pi$ be such that $\delta^1 \subseteq \delta$. As $\beta E_{\exists}\delta \wedge (\gamma, \delta) \in P$ we deduce, by definition of $P(\Pi)$ and by construction of $\Sigma_{i\exists\forall}(\Sigma_{i+1})$, that $\beta E^{\Sigma}_{\exists}(\Pi)\delta^1 \wedge (\gamma^1, \delta^1) \in P$ which contradicts our assumption. ■

We present the conditions in Proposition 5.2.4 describing one iteration of the PARTITIONING ALGORITHM on a simple example.

Example 5.2.5 Consider the graph G and the partition $\Sigma = \{\alpha, \beta\}$ shown on the left in Figure 5.7.

The partition Σ_1 we obtain by applying REFINE$(\Sigma, I, \top)$ is the partition $\{\alpha_1, \alpha_2, \beta_1, \beta_2\}$ shown on the right in Figure 5.7.

The relation $\texttt{Ind}_1$ is formed by the pairs (α_i, α_j), (β_i, β_j), with $i, j \in \{1, 2\}$.

The pair (β_2, β_1) does not satisfies condition *(b)* in (2σ). In fact, $\beta_2 E_{\forall}\alpha$, while β_1 does not reach existentially any class of Σ. Using the conditions in Proposition 5.2.4 this corresponds to the fact that $\beta_2 E^{\Sigma}_{\forall}(\Sigma_1)\alpha_2$, while β_1 does not reach through $E^{\Sigma}_{\exists}(\Sigma_1)$ any class.

By computing NEW_HHK$(\Sigma_{0\exists\forall}(\Sigma_1), \texttt{Ind}_1, \perp)$ we obtain

$$\texttt{Ref}_1 = \{(\alpha_1, \alpha_2), (\alpha_2, \alpha_1), (\beta_1, \beta_2)\} \cup I$$

The pair (α_2, α_1) does not satisfies condition *(c)* in (2σ). In fact, $\alpha_2 E_{\exists}\beta_2$, while α_1 does not reach existentially any class δ such that $(\beta_2, \delta) \in \texttt{Ref}_1$. This is equivalent to what is stated in the second condition of Proposition 5.2.4.

By computing NEW_HHK$(\Sigma_{1\exists\forall}, \texttt{Ref}_1, \top)$ we obtain

$$\texttt{P}_1 = \{(\alpha_1, \alpha_2), (\beta_1, \beta_2)\} \cup I,$$

i.e., the maximal relation satisfying the conditions in Proposition 5.2.4. ■

The computation performed by the function UPDATE corresponds to determine the largest relation included in $\texttt{P}_i$ and satisfying conditions (2σ), thereby getting us closer to stability. The correctness of UPDATE is based on some technical lemmas reported in Appendix A. Hence we can prove the following invariant relative to the functions UPDATE and REFINE.

Theorem 5.2.6 (Refine-Update invariant) *The following holds:*

$$\langle \Sigma_{i+1}, \mathtt{P}_{i+1} \rangle = \sigma(\langle \Sigma_i, \mathtt{P}_i \rangle).$$

Proof. See Appendix A. ■

As a consequence of Theorem 5.1.15 and of Corollary 5.1.16, since the PARTITIONING ALGORITHM terminates whenever $\Sigma_{i+1} = \Sigma_i$, we can conclude that the PARTITIONING ALGORITHM computes the solution $\langle S, \preceq \rangle$ of the GCPP over G and $\langle \Sigma, P \rangle$, with P transitive, performing at most $|S|$ iterations of the while-loop.

Theorem 5.2.7 (Time complexity) *Given a graph $G = (V, E)$ and a partition pair $\langle \Sigma, P \rangle$, with P transitive, the algorithm* PARTITIONING ALGORITHM *computes the solution $\langle S, \preceq \rangle$ of the GCPP over them in time $\mathcal{O}(|S|^2|E|)$.*

Proof. From Corollary 5.1.16 we have that at most $|S|$ iterations of the while-loop are performed. We now prove that, in each iteration of the PARTITIONING ALGORITHM, the REFINE function takes $\mathcal{O}(|\Sigma_i||E|) = \mathcal{O}(|S||E|)$ time. The cost of the refining steps over the entire PARTITIONING ALGORITHM is then $\mathcal{O}(|S|^2|E|)$.

The initialization phase (the instructions before the while-loop) in REFINE takes time $\mathcal{O}(|\Sigma_i|^2)$. During each iteration of the while-loop an element γ is taken out of Sort and it is never reinserted in it: as there are $|\Sigma_i|$ classes in Sort, the while-loop is iterated at most $|\Sigma_i|$ times. Once γ has been dequeued, the set $\mathsf{Split}(\gamma) = \{\alpha \mid \mathsf{E}^{-1}(\gamma) \cap \alpha \neq \emptyset\}$ can be computed in $\mathcal{O}(|E^{-1}(\gamma)|)$ time. Each instruction in the while-loop, different from the for-loop, costs $\mathcal{O}(1)$; without considering the innermost for-loop, the global cost of the while-loop, in a refining step, is then $\mathcal{O}(E^{-1}(\gamma_1) + \ldots + E^{-1}(\gamma_{|\Sigma_i|})) + \mathcal{O}(|\Sigma_i|) = \mathcal{O}(|E|)$.
$\mathsf{Split}(\gamma)$ contains at most $|E^{-1}(\gamma)|$ elements and hence the number of for-loop's iterations is bounded by $|E^{-1}(\gamma)|$. Assuming to have P_i represented as an $\Sigma_i \times \Sigma_i$ adjacency matrix, the check $\mathsf{Stable}(\alpha) \cap \mathsf{Row}(\gamma) = \emptyset$ can be implemented in $\mathcal{O}(|\mathsf{Stable}(\alpha)|) = \mathcal{O}(|\Sigma_i|)$. As far as the remaining operations in the for-loop are concerned, we observe that, for each class $\alpha \in \Sigma_{i+1}$ with $E^{-1}(\gamma) \cap \alpha \neq \emptyset$, the sets $\alpha_1 = E^{-1}(\gamma) \cap \alpha$ and $\alpha_2 = \alpha \setminus \alpha_1$ can be provided while computing (i.e., at the cost of computing) $\mathsf{Split}(\gamma)$: strategies similar to the ones suggested in [84] can be used to this purpose. For-loop's instructions involving the updating of Σ_{i+1} and the setting of the Stable sets (relative to the new Σ_{i+1} classes) can be implemented in $\mathcal{O}(1)$. Thus the global cost of the for-loop in a refining step turn out to be $\mathcal{O}(E^{-1}(\gamma_1)|\Sigma_i| + \ldots + E^{-1}(\gamma_{|\Sigma_i|})|\Sigma_i|) = \mathcal{O}(|E||\Sigma_i|)$. We get that the complexity of the REFINE function is $\mathcal{O}(|S|^2 + |S||E|) = \mathcal{O}(|S||E|)$ [1].

[1]We assume that $|V| = \mathcal{O}(|E|)$ in the graph in input: note that in the context of model checking, Kripke structures modeling the finite systems to validate always satisfy the above assumptions.

In UPDATE the cost of the initialization of $\mathsf{Ind_{i+1}}$ is $\mathcal{O}(\Sigma_{i+1}^2)$. As far as the initialization of the $\exists\forall$-quotient structure and the $\exists\forall$-induced quotient structure are concerned, the following procedure builds $\Sigma_{(i+1)\exists\forall} = \langle \Sigma_{i+1}, E_\exists, E_\forall \rangle$ and $\Sigma_{i\exists\forall}(\Sigma_{i+1}) = \langle \Sigma_{i+1}, E_\exists^{\Sigma_i}(\Sigma_{i+1}), E_\exists^{\Sigma_i}(\Sigma_{i+1}) \rangle$ in $\mathcal{O}(|\Sigma_{i+1}|^2 + |E|)$ time (each iteration of the innermost for-loop can be easily implemented at the cost of computing $E^{-1}(\alpha')$):

```
1. for each $\alpha \in \Sigma_i$ do
2.     compute $E^{-1}(\alpha)$ and sign each $\beta' \in \Sigma_{i+1}, \beta' \subseteq E^{-1}(\alpha)$;
3.     for each $\alpha' \in \Sigma_{i+1}, \alpha' \subseteq \alpha$ do
4.         $E_\exists := \{\langle \alpha'\beta' \rangle \mid \beta' \in \Sigma_{i+1}, \beta' \cap E^{-1}(\alpha') \neq \emptyset\}$;
5.         $E_\exists^{\Sigma_i}(\Sigma_{i+1}) := \{\langle \alpha'\beta' \rangle \mid \beta' \in \Sigma_{i+1}, \beta' \cap E^{-1}(\alpha') \neq \emptyset\}$;
6.         $E_\forall := \{\langle \alpha'\beta' \rangle \mid \beta' \in \Sigma_{i+1}, \beta' \subseteq E^{-1}(\alpha')\}$;
7.         $E_\exists := \{\langle \alpha'\beta' \rangle \mid \beta' \in \Sigma_i + 1$ has been signed in step 2,
                   $\beta' \cap E^{-1}(\alpha') \neq \emptyset\}$.
```

Without considering the two calls to the NEW_HHK function, the complexity of the updating steps overall the PARTITIONING ALGORITHM is then $\mathcal{O}(|S|(|E| + |S|^2)) = \mathcal{O}(|S|^2|E|)$. In order to evaluate the cost of performing the entire set of calls to the function NEW_HHK, we can either directly apply the results in [63] to a single procedure's call or use a global argument. We could apply directly the results in [63] since both the calls to NEW_HHK, relative to a single UPDATE step, correspond to a call to the function in [63] on a graph having Σ_i nodes linked by two types of edges: $\forall$-edges and $\exists$-edges (moreover the first call is stopped after only one iteration). The only substantial difference between the function in [63] and NEW_HHK is that some of the operations of predecessor's set's computation are specialized for the $\forall$-edges. As two nodes are linked by a $\forall$-edge only if they are linked by an $\exists$-edge and there are $\mathcal{O}(|E|)$ $\exists$-edges, we can conclude, using the results in [63], that each call to NEW_HHK costs $\mathcal{O}(|S||E|)$.

However, the results in [63] are based on the use of a special counter table requiring $\mathcal{O}(|S|^2 \log(|S|))$ space. This counter table allows to check the innermost if guard in NEW_HHK in constant time (it keeps track, for each couple of classes $\langle \alpha, \beta \rangle$, of the value $|post(\alpha) \cap\ sim(\beta)|$). Using a global argument to evaluate the cost of the entire set of calls to the function NEW_HHK, we can obtain the same time complexity avoiding to maintain the counter tables. In particular, the innermost if statement of NEW_HHK overall the PARTITIONING ALGORITHM is $\mathcal{O}(|S|^2|E|)$, without any counter table. In fact, the innermost if statement in NEW_HHK is executed only after a class, say β, is removed from the set of classes simulating another class, say α; its cost, without counter table, is easily proved to be $\mathcal{O}(|S||E^{-1}(\beta)|)$. Let α_k be a class in Σ_k and consider, for each iteration of the PARTITIONING ALGORITHM i, $\alpha_i \supset \alpha_k$ and the classes $\beta_i^1, .., \beta_i^{m_i}$ removed from $sim(\alpha_i)$ while executing the innermost for-loop in a NEW_HHK call. By Definition 5.1.11, Corollary A.0.10, and Lemma A.0.11 the classes

$\beta_1^1, .., \beta_1^{m_1}, .., \beta_k^1, .., \beta_k^{m_k}$ are mutually disjoint; hence the cost of executing the innermost if statement of NEW_HHK, involving a class just recognized to be not able to simulate $\alpha_i \supset \alpha_k$, is $(|E^{-1}(\beta_1^1)|+..+|E^{-1}(\beta_1^{m_1})|+..+|E^{-1}(\beta_k^1)|+..+|E^{-1}(\beta_k^{m_k})|)|S| = \mathcal{O}(|E||S|)$. As there are $|S|$ classes in Σ_k, the innermost if statement of the function NEW_HHK takes, overall the entire PARTITIONING ALGORITHM, $\mathcal{O}(|S|^2|E|)$.

■

Theorem 5.2.8 (Space complexity) *Given a graph $G = (V, E)$ and a partition pair $\langle\Sigma, P\rangle$, with P transitive, the algorithm* PARTITIONING ALGORITHM *computes the solution $\langle S, \preceq\rangle$ of the GCPP in space $\mathcal{O}(|S|^2 + |V|\log(|S|))$.*

Proof. During each iteration of the algorithm it is necessary to consider: the relation E; the relation P_i (at most $\mathcal{O}(|\Sigma_i|^2)$ space); the relation which maps each node in V into the class of Σ_i to which it belongs (space $\mathcal{O}(|V|\log|\Sigma_i|)$).

As observed in [63] it is not really necessary to keep the relation E in memory: we can use it only when it is necessary to provide the set of successors and predecessors of a given node. Hence the space complexity is $\mathcal{O}(|S|^2 + |V|\log(|S|)))$. ■

5.2.1 Implementation and Tests

To asses the performance of the PARTITIONING ALGORITHM we have implemented it in Standard ML and interfaced it with the Concurrency Workbench of the New Century (CWB-NC) (see [29]). The CWB-NC release incorporates both the simulation algorithm by Paige and Bloom [8] and the simulation algorithm by Cleaveland and Tan [31]. We tested our routine and the latter mentioned procedure on some toy examples as well as on case studies included in the CWB-NC.

The CWB-NC analysis routines work on transitions systems having labelled edges. Thus, we adapted our algorithm following an approach similar to the one used in [31]. The REFINE step is performed once with respect to each action, while in the UPDATE step an action parameter is employed.

In the sequel we describe some of the data structures used giving several implementation details. Then, some experimental results will be presented. The tests have been executed on a Pentium III, 400 MHz PC, 256MB RAM, OS Linux Red Hat 6.2.

We make use of two modifiable arrays of records (called coarser_partition_table and finer_partition_table, respectively) to represent, in each iteration i of the algorithm, the partition Σ_{i-1} (to be refined) and the partition Σ_i (result of the refinement step). Each record in the finer_partition_table corresponds to a block of Σ_i and we associate to it the following fields:

- states: a doubly linked list of states representing the set of states belonging to the block;
- touched_states: a doubly linked list of states used to keep trace of the states touched while scanning the elements having transitions into a Σ_{i-1} class;

- superclass: the index in the coarser_partition_table of the Σ_{i-1} class containing the block;
- stable_blocks: a list of indexes in the coarser_partition_table allowing to represent Stable(α') within the REFINE step.

Each record in the coarser_partition_table corresponds to a block of Σ_{i-1} and we keep the following information associated to it:

- splitted_blocks: a list of indexes in the finer_partition_table corresponding to the blocks $\alpha' \in \Sigma_i$ such that $\alpha' \subseteq \alpha$;
- greater_blocks: a list of indexes in the coarser_partition_table corresponding to the blocks $\beta \in \Sigma_{i-1}$ such that $(\alpha, \beta) \in P_{i-1}$.

We do not represent explicitly the set of states of each class $\alpha \in \Sigma_{i-1}$: this set can be retrieved by combining the doubly linked list of states of each Σ_i class contained in α. The states are integers ranging over [1..num_states]. Thus, they are used to index the table states_info maintaining, for each state, a pointer to the position in the unique doubly linked list (states or touched_states) they belong to. The relation E^{-1} is maintained, by means of an adjacency list. This allows to retrieve, for each node a, the set $E^{-1}(a)$ in time proportional to its size.

At the beginning of each refinement step i the coarser_partition_table and the finer_partition_table represent the same partition Σ_{i-1}. After the refinement has been performed the finer_partition_table keeps Σ_i and the splitted_blocks lists in the coarser_partition_table allow to retrieve the correspondence between Σ_{i-1} and Σ_i. Hence, it is possible to construct $\exists\forall$ quotient structure and the $\exists\forall$ induced quotient structure. These are represented by means of two pairs of adjacency lists. In the adjacency list representing $\exists$ quotient structure, for example, we associate to each Σ_i class, α, the list of Σ_i classes β such that $\alpha E^{\exists} \beta$. This allows us to retrieve, given a block α, the set of blocks β with $\alpha E^{\exists} \beta$ in time proportional to its size. Beside the construction of the quotient structures some other operations are necessary before entering the i step of update. In particular, the partition Σ_i is copied in the coarser_partition_table. Meanwhile the relation relation induced over Σ_i by P_{i-1} is computed and stored in the greater_blocks fields of the coarser_partition_table. The remove sets used in NEW_KKH are globally represented as an array of lists of indexes in the coarser_partition_table.

Now we present the results of some tests performed to compare our implementation of the PARTITIONING ALGORITHM with the implementation of the algorithm by Cleaveland and Tan ([31]) available inside CWB-NC[2].

We start with some tests on toy examples built to point out the differences between the two algorithms. Figure 5.8 shows the structures of two examples. In this two cases the PARTITIONING ALGORITHM takes advantage of the fact that its time and space

[2]The implementation of the Cleaveland and Tan algorithm we use is the one with ALT table and without Path Compression (see [31]).

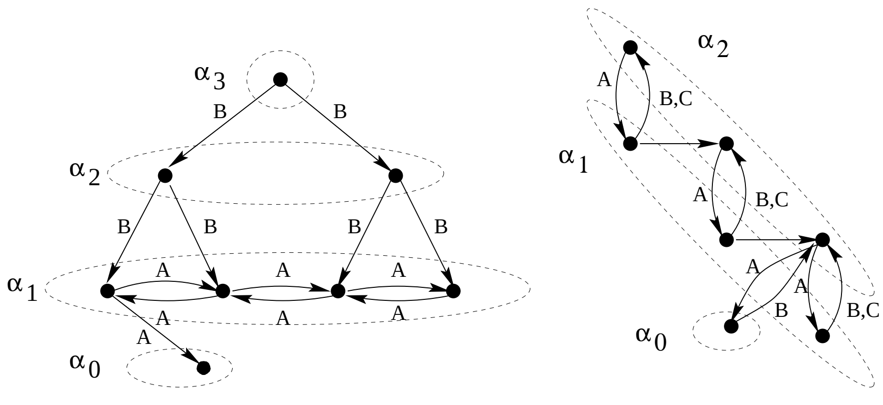


Figure 5.8: The Tree and the 3-classes examples.

Tree (Fig. 5.8 left)			3-classes (Fig. 5.8 right)			A variant of Tree		
$\|V\|$	**CT**	**PA**	States	**CT**	**PA**	States	**CT**	**PA**
256	0.54	0.13	500	0.14	0.01	256	0.06	0.13
512	2.19	0.28	1000	12.56	0.08	512	0.16	0.3
1024	8.41	0.61	1500	27.65	0.25	1024	0.34	0.71
2048	–	1.42	2000	–	0.32	2048	1.16	1.54
8192	–	8.53	10000	–	2.18	8192	5.98	9.01

Table 5.1: Results of the tests on Tree, 3-classes, and a variant of Tree .

complexities depend on the size of the simulation quotient. In both examples all the states are not bisimilar. In the example on the left of Figure 5.8, we call it Tree, each level of the tree corresponds to a simulation class and the block α_0 is simulated by all the other blocks. Hence, the size of the simulation quotient is logarithmic with respect to the state space. The example on the right of Figure 5.8, we call it 3-classes, has three classes of simulation, α_0, α_1, and α_2, and α_0 is simulated by α_1. Hence, the size of the simulation quotient of 3-classes is three, while the size of the bisimulation quotient is equal to the size of the state space.

Table 5.1 shows the results of running the two simulation procedures over graphs having the structure of Tree and 3-classes with increasing state space sizes. In the tables we use **CT** to denote the algorithm by Cleveland and Tan and **PA** to denote our PARTITIONING ALGORITHM. Moreover, we use – in the cases in which we run out of memory and we express the times in seconds. As expected, since in both examples the simulation reduction is larger than the bisimulation one, the PARTITIONING ALGORITHM uses less time and space than the procedure by Cleaveland and Tan.

Notice that if in the Tree example we remove the node in the class α_0 we have that the bisimulation and the simulation quotients coincide and they are given by the

	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert B\rvert$	$\lvert S\rvert$	**CT**	**PA**
ABP-lossy	57	130	15	14	0.05	0.04
ABP-safe	49	74	18	17	0.04	0.03
Two-link-netw	1589	6819	197	147	7.53	5.16
Three-link-netw	44431	280456	2745	1470	–	1336.24

Table 5.2: Tests on four bit-alternating-protocol models included in the CWB-NC.

levels of the tree. In Table 5.1 we show also the results of the tests performed on this variant of the Tree. In this case the Cleaveland and Tan algorithm takes advantage of the large (maximal) bisimulation reduction and performs better than our algorithm.

In Table 5.2 we report the results of tests performed using a benchmark taken from CWB-NC. In particular, Table 5.2 shows both some information about the structures of different models of the alternating-bit protocol included in the CWB-NC release and the times resulting from minimizing the systems. This last example shows the space efficiency of the PARTITIONING ALGORITHM on a more concrete example. The time performances of the PARTITIONING ALGORITHM may be surprising unless one considers that the $\mathcal{O}(|S^2||E|)$ time complexity is reached only in cases in which in each iteration only few splits are performed and $|S|$ iterations are needed to get to the simulation quotient.

III

Applications to Model Checking

6
Overview on Model Checking

In this chapter we briefly present the formal verification technique of model checking, within which most of the material introduced in this dissertation finds applications. For a complete survey, we suggest [26, 78, 99].

Model checking is a formal method to check the correctness of finite states concurrent systems, against a given specification. The first key ingredient of the above technique is the use of *temporal logic* as a specification language: this application of temporal logic was originally suggested by Amir Pnueli [90, 91], who was consequently awarded of the Turing Award for this idea, in 1996. Temporal logics allow to describe the temporal evolution of the computation, expressing properties that are crucial for the correctness of systems continuously interacting with the environment (reactive systems). The second fundamental ingredient, independently suggested in [92, 21], is that of reducing the verification problem to that of checking the satisfiability of a temporal formula, ϕ, (representing the system specification) against a *specific model*, $\mathcal{M}$, (representing the system behavior): $\mathcal{M} \models \phi$. Solving a *model checking* problem, rather than a *validity checking* one, as originally suggested in [91], is exactly the engine for having an automatic verification algorithm which is linear with respect to the model size and, for crucial fragments of temporal logics, with respect to the formula size, also. In model checking, the structures used to define the semantics of temporal logic formulas are called *Kripke structures.* Kripke structures are simply labeled transition systems, whose nodes represent possible global states of the modeling system and whose edges encode possible system transitions. Labels are used to define the set of properties satisfied by each global state of the system.

Definition 6.0.9 (Kripke Structure) *Let AP be a set of atomic propositions. A* Kripke structure $\mathcal{M}$*, over AP is a tuple $\mathcal{M} = (S, S_0, R, L)$ where:*

- *S is a finite set of states;*
- *$S_0 \subseteq S$ is the set of initial states;*
- *$R \subseteq S \times S$ is the transition relation;*
- *$L : S \mapsto 2^{AP}$ is a function that labels each state with the atomic propositions true in that state.*

A path π, from a state $s \in S$ in a Kripke structure, $\mathcal{M} = (S, S_0, R, L)$, is an infinite sequence of states $\pi = \langle s_0 s_1 \ldots \rangle$ such that $s_0 = s$ and $\forall i > 0(s_i R s_{i+1})$. Given $\pi = \langle s_0 s_1 \ldots \rangle$, we will use the notation π^k to denote $\pi^k = \langle s_k s_{k+1} \ldots \rangle$.

Beside the above mentioned characteristics of being completely automatic and time efficient, model checking has the advantage of producing, in case the verification is not successful, a "bad" trace of the system (*counterexample*) witnessing its lack of adherence to the specification. Moreover, fairness constraints are easily embedded in the overall procedure. The main challenge in model checking is dealing with the notorious *state explosion problem*: modeled systems have tipically many components that interact in parallel with each other and, consequently, the number of global states increase exponentially with respect to the system components.

6.1 Temporal Logics

In model checking the temporal logic language is used to describe the time evolution of the computation, within the systems undergoing verification. Two possible views on the nature of time induce two types of temporal logics. In *linear time* temporal logic, the time is viewed as a linear chain of events. In *branching time* temporal logic, instead, each point of time is characterized by many immediate future events: in this context, concurrent computational paths can be modeled.

6.1.1 Branching Time Temporal Logics

We focus here on powerful and widely used, in the context of model checking, branching time logics called CTL* and CTL [21, 22]. Formulae in CTL* are built, as in Definition 6.1.1, below, using classical boolean operators, the existential and the universal path quantifiers (A and E), and the temporal operators X (neXt state), U (Until), and R (Release).

Definition 6.1.1 (CTL* Syntax) *Let AP be a set of atomic propositions and $p \in AP$.* Path formulae *and* state formulae *of CTL* are inductively defined as follows:*

$$\begin{array}{lll} \textit{state formulae} & \phi := & p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid A\psi \mid E\psi \\ \textit{path formulae} & \psi := & \phi \mid \psi \vee \psi \mid \psi \wedge \psi \mid X\psi \mid \psi U \psi \mid \psi R \psi \end{array}$$

The logic CTL* is defined as the set of state formulae defined as above.

Definition 6.1.2 (CTL* Semantics) *Consider a Kripke structure over the set of atomic propositions AP, $\mathcal{M} = (S, S_0, R, L)$. Let $s \in S$, and π be a path in $\mathcal{M}$. Given a* state formula *ϕ and a path formula ψ, the relations $\mathcal{M}, \pi \models \psi$ and $\mathcal{M}, s \models \phi$ are* *inductively defined as follows:*

$$\begin{array}{lcl}
\mathcal{M}, s \models p & \Leftrightarrow & p \in L(s) \\
\mathcal{M}, s \models \neg p & \Leftrightarrow & \{p\} \cap L(s) = \emptyset \\
\mathcal{M}, s \models \phi_1 \vee \phi_2 & \Leftrightarrow & \mathcal{M}, s \models \phi_1 \vee \mathcal{M}, s \models \phi_2 \\
\mathcal{M}, s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \mathcal{M}, s \models \phi_1 \wedge \mathcal{M}, s \models \phi_2 \\
\mathcal{M}, s \models E\psi & \Leftrightarrow & \textit{there is a path } \pi \textit{ from } s \textit{ such that } \mathcal{M}, \pi \models \psi \\
\mathcal{M}, s \models A\psi & \Leftrightarrow & \textit{each path } \pi \textit{ from } s \textit{ is such that } \mathcal{M}, \pi \models \psi \\
\mathcal{M}, \pi \models \phi & \Leftrightarrow & s \textit{ is the first state of } \pi \textit{ and } \mathcal{M}, s \models \phi \\
\mathcal{M}, \pi \models \psi_1 \vee \psi_2 & \Leftrightarrow & \mathcal{M}, \pi \models \psi_1 \vee \mathcal{M}, \pi \models \psi_2 \\
\mathcal{M}, \pi \models \psi_1 \wedge \psi_2 & \Leftrightarrow & \mathcal{M}, \pi \models \psi_1 \wedge \mathcal{M}, \pi \models \psi_2 \\
\mathcal{M}, \pi \models X\psi & \Leftrightarrow & \mathcal{M}, \pi^1 \models \psi \\
\mathcal{M}, \pi \models \psi_1 U \psi_2 & \Leftrightarrow & \textit{there exists } k \geq 0 \textit{ such that } \mathcal{M}, \pi^k \models \psi_2 \\
 & & \textit{and for all } 0 \leq j < k, \textit{ it holds } \mathcal{M}, \pi^j \models \psi_1 \\
\mathcal{M}, \pi \models \psi_1 R \psi_2 & \Leftrightarrow & \textit{for all } j \geq 0, \textit{ if for every } i < j,\ \mathcal{M}, \pi^i \not\models \psi_1, \\
 & & \textit{then } \mathcal{M}, \pi^j \models \psi_2
\end{array}$$

Typical derived CTL* temporal operators are the operators G (Globally) and F (Finally), defined as follows:

$$F\phi \equiv True U \phi$$
$$G\phi \equiv \neg F \neg \phi$$

The Computation Tree Logic (CTL), is the subset of CTL* defined by the following grammar:

Definition 6.1.3 (CTL Syntax) *Let AP be a set of atomic propositions and $p \in AP$. The logic CTL is defined by the following set of (state) formulae:*

$$\phi := p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid AX\phi \mid EX\phi \mid A(\phi U \phi) \mid E(\phi U \phi) \mid A(\phi R \phi) \mid E(\phi R \phi)$$

Some typical CTL formulae, that can arise in verifying a finite state concurrent system are given below [26]:

- $EF(start \wedge \neg ready)$: it is possible to get to a state where *start* holds but *ready* does not hold;
- $AG(req \rightarrow AF ack)$: If a request occurs, then it will be eventually acknowledged;
- $AG(EF\, restart)$: from any state it is possible to get to a *restart* state.

Both the universal and existential fragments of CTL and CTL* (called ACTL, ACTL* and ECTL, ECTL*, respectively), in which only one path quantifier is admitted, are often used within powerful methods to deal with state explosion problem, as equivalence reduction and compositional reasoning [34, 33, 18, 26, 99].

6.1.2 Linear Time Temporal Logic

The Linear Temporal Logic LTL [52] is the fragment of CTL* logic consisting of formulae of type $A\psi$, where ψ is a path formula inductively defined as follows:

$$\psi := p \mid \neg p \mid \psi \vee \psi \mid \psi \wedge \psi \mid X\psi \mid \psi U \psi \mid \psi R \psi.$$

The CTL* sublogics CTL and LTL have not comparable expressing power [111, 26, 99]. An example of CTL formula that can not be expressed in the LTL language is the formula $AGEF\psi$, stating that from all states it is possible to reach a state in which ψ holds. Conversely, most of the properties expressing constraints imposing a fair use of resources (*fairness* constraints), such as the LTL formula $AFG(\neg en \vee ack)$ (if an event is continuously enabled it will occur infinitely often), can not be expressed in CTL.

6.1.3 Fairness

Fairness is a crucial assumption in the context of correctness analysis of many systems. For example, when verifying a communication protocol over reliable channels only fair execution paths, in which no message happens to be continuously sent but never received, should be considered. Fairness constraints are usually classified into *strong fairness* constraints and *weak fairness* constraints [52, 51]. Weak fairness guarantees that no enabled transition is postponed forever, as in the above mentioned example. Strong fairness, instead, is used, for example, in the analysis of synchronous interactions. Strong fairness imposes that if an action is enabled infinitely often, then it will be taken infinitely often. Both weak and strong fairness can be expressed as LTL properties:

$$\text{strong fairness} \quad A(GF\, en \rightarrow GF\, ack)$$

$$\text{weak fairness} \quad AGF(\neg en \vee ack)$$

To consider fairness constraints in the framework of CTL logic, in which neither weak nor strong fairness properties can be expressed, *Weak Fair Kripke Structures* (or *generalized Büchi automata*) and *Strong Fair Kripke Structures* (or *Streett Automata*) are used to evaluate formulae.

Definition 6.1.4 (Weak Fair Kripke Structure) *A weak fair Kripke structure, $\mathcal{M}_{\mathcal{F}} = (S, S_0, R, L, F)$, is a Kripke structure with an* acceptance condition, *$F = \langle F_1 \subseteq 2^S, \ldots, F_n \subseteq 2^S \rangle$. A fair path of $\mathcal{M}_F$ is a path, π, such for each F_i in F, $inf(\pi) \cap F_i \neq \emptyset$, where $inf(\pi)$ is the set of states comparing infinitely often in π.*

Definition 6.1.5 (Strong Fair Kripke Structure) *A strong fair Kripke structure, $\mathcal{M}_{\mathcal{F}} = (S, S_0, R, L, F)$, is a kripke structure with an* acceptance condition, *$F = \langle (P_1 \subseteq 2^S, Q_1 \subseteq 2^S) \ldots, (P_n \subseteq 2^S, Q_n \subseteq 2^S) \rangle$. A fair path of $\mathcal{M}_F$ is a path, π, such for each (P_i, Q_i) in F, $inf(\pi) \cap P_i \neq \emptyset \rightarrow inf(\pi) \cap Q_i \neq \emptyset$.*

The semantic of CTL* formulae with respect to (weak/strong) fair Kripke structures simply requires that only (weak/strong) fair paths are taken into consideration.

6.2 Explicit Model Checking

Given a Kripke structure $\mathcal{M} = (S, S_0, R, L)$ and a temporal formula ϕ, the *model checking problem* consists in determining the set of states $\{s \in S_0 \mid \mathcal{M}, s \models \phi\}$.

6.2.1 CTL Model Checking

In CTL model checking, the formulae given as input are expressed in CTL. The first algorithms for CTL model checking were defined in [21, 92] and had a polynomial complexity with respect to the size of both the model to verify and the specification formula. In [22], a linear CTL model checking algorithm was given. The problem of LTL model checking was shown to be PSPACE complete [103]. Later, Lichtenstein and Pnueli proved that LTL model checking can be solved in time exponential with respect to the size of the input formula, but linear with respect to the size of the model [76]. Finally, the authors of [41] proved that CTL* model checking problem has essentially the same complexity of LTL model checking problem.

The linear CTL model checking algorithm consists of a number of iterations, corresponding to the number of nested operators in the input CTL formula ϕ: in each iteration, i, the procedure labels each state in the set $\{s \in S \mid \mathcal{M}, s \models \phi_i\}$, where ϕ_i is the subformula of ϕ in which $i - 1$ nested operators are considered. Considering any of the CTL operators, within a given iteration of the CTL model checking algorithm, has cost $\mathcal{O}(|R| + |S|)$. Consequently, the complexity overall CTL model checking procedure, $\mathcal{O}(|\phi|(|R| + |S|))$, that is linear in the sizes of both the input formula and the modeling Kripke structure.

The subprocedures processing the set of complete CTL operators $\neg, \vee, EX, EU, EG$ are simply graph algorithms. Consider the formula $\neg\phi$: assuming to have labeled all states $\{s \in S \mid \mathcal{M}, s \models \phi\}$, it is simply necessary to mark all states in $S \setminus \{s \in S \mid \mathcal{M}, s \models \phi\} = \{s \in S \mid \mathcal{M}, s \models \neg\phi\}$. The other boolean operators are similarly dealt with. For formulae of type $EX\phi$, every state having some successor labeled ϕ need to be labeled with $EX\phi$. To handle formulae of type $E(\phi_1 U \phi_2)$, the relation R^{-1} is used to visit the graph from the set of states labeled ϕ_2. All states reachable backwards from $\{s \in S \mid \mathcal{M}, s \models \phi_2\}$, trough a path whose states are all labeled with ϕ_1, must be labeled with $E(\phi_1 U \phi_2)$. Finally, the procedure to process formulae of type $EG\phi$ can be based on the computation of the strongly connected components of $\mathcal{M}$ [26, 99].

Fairness, Witnesses and Counterexamples

Recall from Subsection 6.1.3 that dealing with fairness constraints, in the CTL model checking problem, can be accomplished by evaluating CTL formulae on fair Kripke structures $\mathcal{M}_{\mathcal{F}} = (S, S_0, R, L, \mathcal{F})$. Both strong and weak fair CTL model checking problems can be computationally solved with an overall complexity of $\mathcal{O}(|\phi|(|S| + |R|)|\mathcal{F}|)$, where $|\mathcal{F}|$ represents the number of components in the final acceptance condition [26, 99]. We briefly outline, below, the algorithmic solution of the weak fair CTL model checking problem, and we refer to [26, 99] for strong fair CTL model checking.

Consider a weak fair Kripke structure $\mathcal{M}_{\mathcal{F}} = (S, S_0, R, L, \mathcal{F} = \langle F_1, \ldots, F_n \rangle)$. The weak fair CTL algorithm proceeds, similarly to CTL procedure, analyzing bottom up the input formula and labeling those states that satisfy the considered subformulae. To deal with the operator EG, assuming a weak fair semantic, an $\mathcal{O}((|S| + |R|)|\mathcal{F}|)$ subprocedure can be defined upon Lemma 6.2.2, below.

Definition 6.2.1

Lemma 6.2.2 $\mathcal{M}_{\mathcal{F}}, s \models EG\phi$ *if and only if the following two conditions are satisfied:*

- $s \in S' = \{s \in S \mid \mathcal{M}_{\mathcal{F}}, s \models \phi\}$*;*
- *there exists a path in* S' *leading from* s *to a not trivial strongly connected component of* $\langle S', R|_{S'} \rangle$, $\mathcal{C}$, *such that* $\forall i = 1 \ldots n \, (\mathcal{C} \cap F_i \neq \emptyset)$

Roughly, a fair CTL subprocedure to deal with EG operator, relies on a strongly connected components graph decomposition, followed by a check that some not trivial strongly connected component exists, containing at least one state for each $F_i \subseteq S$ in the acceptance condition. By definition, a fair path starts from a state s if and only if $\mathcal{M}_{\mathcal{F}}, s \models EG\,true$. As fair CTL semantics is equal to CTL one, apart from the fact that only fair path are taken into consideration, it follows that CTL operators EX and EU can be dealt with, assuming weak fairness, as follows:

- Assume to have labeled with label *fair* each state in $\{s \in S \,|\, \mathcal{M}_{\mathcal{F}}, s \models EG\,true\}$;
- Use standard CTL procedures for the operators $\wedge$ and EX to label states in $\{s \in S \mid \mathcal{M}_{\mathcal{F}}, s \models EX\phi\}$, since $\mathcal{M}_{\mathcal{F}}, s \models EX\phi$ if and only if $\mathcal{M}, s \models EX(\phi \wedge fair)$;
- Use standard CTL procedures for the operators $\wedge$ and EU to label states in $\{s \in S \mid \mathcal{M}_{\mathcal{F}}, s \models E(\phi U \psi)\}$, since $\mathcal{M}_{\mathcal{F}}, s \models E(\phi U \psi)$ if and only if $\mathcal{M}, s \models E(\phi U(\psi \wedge fair))$.

An important feature of model checking technique is the ability to find *counterexamples* and *witnesses*. A counterexample for an universal formula $A\phi$ is a path in the Kripke structure modeling the path formula $\neg\phi$. Conversely a witness of the existential formula $E\psi$ is a path in the Kripke structure modeling the path formula ψ. Integrating, within the CTL algorithms described so far, the ability of retrieving counterexamples and witnesses, requires a few technical efforts does not modify the complexity of the overall algorithm. We refer to [26, 99] for a complete description of the argument.

6.2.2 LTL Model Checking

As already observed, LTL and CTL have not comparable expressing power. Hence, there are important classes of correctness properties (as the already discussed fairness properties) that can be expressed in LTL but not in CTL. Besides, the constraints imposed on the syntax of CTL formulae make the language further more unnatural than LTL: specifying properties within LTL is simpler than within the CTL framework [111, 110, 58]. Even if exponential in conmplexity with respect of the size of the formula in input, the LTL model checking problem have be shown to be solvable in time linear with respect to the size of the model [76]. Moreover, the LTL problem can be naturally translated into a Büchi automata language containment one, whose algorithmic solution can benefits of space saving powerful heuristic, as the technique of *on the fly* analysis [58, 110, 26, 99]. For these reasons, many model checkers exists that consider LTL expressed properties [64, 19].

The technique of (automata based) LTL model checking assumes that both the system to be analyzed, and the negation of the property to be checked, are represented by means of Büchi automata, whose definition is given below.

Definition 6.2.3 (Büchi Automata) *A* Büchi automaton *is defined as a tuple* $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$*, where:*

- Σ *is a finite alphabet;*
- Q *is a finite set of states;*
- $\delta : Q \times \Sigma \times Q$ *is the transition relation;*
- Q_0 *is the set of initial states;*
- $F \subseteq Q$ *is a set of nodes that represents the acceptance condition.*

A representation of a system in terms of a Kripke structure, $\mathcal{M} = (S, S_0, R, L : S \mapsto 2^{AP})$, is easily translated into a Bčhi automaton $\mathcal{A} = \langle \Sigma, S \cup \{\iota\}, \delta, \{\iota\}, S \cup \{\iota\} \rangle$, where $\Sigma = 2^{AP}$. In $\mathcal{A}$, the transition relation is such that $(s, a, s') \in \delta$ if and only if $((s, s') \in R \wedge L(s') = a) \vee (s = \iota, s' \in S_0 \wedge L(s') = a)$. LTL specifications can also be given as a Büchi automata over the language $\Sigma = 2^{AP}$: we do not enter in the details regarding the translation of LTL formulae and we redirect to [26, 99, 58] for a complete description. We only stress the fact that the resulting automata can have exponential size with respect to the input formula. Given these premises, LTL model checking automata-based algorithm is easily described in tree phases:

- build the Büchi representation of both the system to be verified, $\mathcal{A}$, and of the *negation* of the LTL property asserting correctness, $\mathcal{A}_{\neg\phi}$;
- build the product automaton $\mathcal{A} \times \mathcal{A}_{\neg\phi}$;
- The system satisfies the given specification if and only if the product automaton, $A \times \mathcal{A}_{\neg\phi}$, does not accept any words. This means, in fact, that there are no traces of the systems witnessing satisfiability of $\neg\phi$.

The language of a Büchi automaton, $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$, is not empty if and only if there exists a cycle of states that is reachable from an initial state and contains a final state [108]. On the ground of this characterization, it is possible to solve the Büchi language emptiness problem in linear time with respect to the input automaton's size: an algorithm that decomposes the (reachable) state space of the automaton in its strongly connected components and, then, checks if some final state belongs to a non trivial strongly connected component, can be used for this purpose. In most model checkers, a special algorithm called Double DFS [32, 66] is indeed used to solve the Büchi emptiness problem. Double DFS avoids preprocessing the entire graph by strongly connected components decomposition, and performs two nested depth first search visits to detect final states in cycles as soon as possible. Moreover, the Double DFS procedure can be adapted to merge the tasks of computing the parallel composition of $\mathcal{A}$ and $\mathcal{A}_{\neg\phi}$, with the task of analyzing such a composed automaton for language emptiness. More precisely, the above task merging results in the construction of only those states, in $\mathcal{A} \times \mathcal{A}_{\neg\phi}$, visited within Double DFS. In the literature, the above outlined space saving approach to LTL model checking is called *on the fly model checking* [58, 66, 65, 26, 99].

6.3 State Explosion Problem

The research conducted in the last ten years resulted in a number of techniques to deal with the often huge size of structures involved in modeling many concurrent components on large input sets. We briefly list, in this section, some of the most used techniques to face the state explosion problem, with their relative benefits. The approach (to state explosion problem) called *symbolic model checking*, whose introduction had dramatical consequences on standard model sizes in verification, will be presented with more details in the next section.

A large part of methods to face the state explosion problem in the literature is based on *abstraction*: model checking is conducted on abstracted (i.e. reduced in size) modeling structures. In this context, *strong preserving abstractions* usually minimize the modeling transition system using a notion of equivalence relation on states that can be computed in a completely automatic way. The *strong preserving* issue refers to the fact that no formula (of an appropriate fragment of temporal logic) distinguishes between the original(concrete) system and the abstracted one. Naturally, the more expressive the temporal logic preserved, the higher the price to pay in terms of power of reduction. Hence, while bisimulation equivalence preserves CTL* and CTL logics, its abstracting potential is less than the one of simulation equivalence, strongly preserving both ACTL and ACTL*. Another equivalence relation used in this context is *stuttering bisimulation*, preserving the fragment of CTL* without the *next* (X) operator [33, 87, 26, 99]. Techniques that result in computing stuttering bisimulation reductions are also called *partial order reductions techniques.*

Weak preserving abstractions [33, 36] obtain a reduced modeling system $\mathcal{M}_{\mathcal{A}}$, that preserves only the formulas true if interpreted on the concrete model $\mathcal{M}$ (i.e. if $\mathcal{M}, s \models \phi$, then $\mathcal{M}_{\mathcal{A}}, s \models \phi$). Usually, the methods are based on *abstract interpretation* [33] and the *abstract domains* involved are defined manually depending on properties to be analyzed. Though *weak preserving abstraction* often need human intervention, they allow to deal with significative reduced structures.

Compositional reasoning [60, 26, 27] exploits modular structure of many systems. In compositional reasoning components of a complex system are verified separately and then results are composed to infer overall correctness. When there are mutual dependencies between components, a strategy called *assume guarantee reasoning* [60, 26] is used. With this approach properties of each component, in a given time step, are verified on the ground of assumptions on the behavior of other components, in the previous time step.

Finally *symmetry reductions* techniques have been considered by various authors in [24, 43, 23]. Symmetry reduction methods aim at eliminating redundancy, in the form of symmetric structures, that naturally occur in protocols and hardware design.

6.4 Symbolic Model Checking

In his Ph.D. thesis Ken McMillan [78] proposed an approach to state explosion problem, in model checking, on the ground of a symbolic OBDD based data representation. The work in [78] presented a number of convincing examples of symbolic verification, addressing the dramatic gap between standard sizes in explicit (10^4,10^5 states) and in symbolic (beyond 10^{20} states) model checking. *symbolic model checking* is nowadays recognized as a milestone in the journey that lead, from automatic verification life of academic, toy examples, to the integration of model checking techniques in the real development of systems (companies as IBM, Motorola, NASA and many other use model checking tools in their industrial process).

Symbolic model checking algorithms take as input a temporal formula and a Kripke structure, $\mathcal{M} = (S, S_0, R, L)$, in which both the relation and the set of (initial) states are represented by means of OBDDs (see Section 1.2); the labelling function is given symbolically, also, in the form of $|AP|$ OBDDs of sets of states, one for each set $S_P = \{s \in S \mid \mathcal{M}, s \models p \in AP\}$.

As in the explicit case, symbolic CTL model checking algorithms proceed analyzing bottom up the input CTL formula: in each iteration the set of states satisfying an appropriate subformula is detected. Subformulae in which the outmost operator is a boolean one are easily dealt with, symbolically. Given the OBDDs representing the sets $\{s \in S \,|\, \mathcal{M}, s \models f_1\}$ and $\{s \in S \,|\, \mathcal{M}, s \models f_2\}$, say $f_1(v)$ and $f_2(v)$, the OBDD representing $\{s \in S|\mathcal{M}, s \models f_1 \wedge f_2\}$ is simply obtained by computing the appropriate OBDD boolean composition $f_1(v) \wedge f_2(v)$ (see Section 1.1.2). Subformulae in which the outermost CTL operator is EX are dealt with using a relational product: the OBDD representing the set of states $\{s \in S \,|\, \mathcal{M}, s \models EXf\}$ is symbolically computed as $\exists v(f(v') \wedge R(v, v'))$. Finally, the remaining CTL operators are characterized in terms of maximal or minimumn *fixpoints operators* on sets of states: on the ground of fixpoints characterizations, checking satisfiability for formulae involving these operators, consists in a process of *refining* or *coarsening* a symbolic represented set of states. More precisely, denoting by $\mu Z.[\tau]$ ($\nu Z.[\tau]$), the minimum (maximum) fixpoint of the set-transformer operator τ, the following equivalences hold [78, 26, 99]:

- $E(f_1 U f_2) = \mu Z.[f_2 \vee (f_1 \wedge EXZ)]$;

- $EFf = \mu Z.[f \vee EXZ]$;
- $EGf = \nu Z.[f \wedge EXZ]$.

The symbolic procedures for computing both the maximum and the minimum fix-point of a (continuous, monotone) predicate transformer are based on well known results by Tarski in [107], and ultimately consist in a sequence of set refinements or coarsening. In the CTL context, each refinement (coarsening) involves a constant number of OBDD compositions, and there are at most $\mathcal{O}(|S|)$ refinement (coarsening) steps. Hence the complexity of each iteration of the main symbolic CTL algorithm, dealing with a subformula of the input formulae ϕ, consists of $\mathcal{O}(|S|)$ symbolic steps. The overall complexity of the CTL symbolic algorithm outlined is $\mathcal{O}(|\phi||S|)$ symbolic steps.

Passing from the symbolic representation of Kripke structure to the symbolic encoding of (generalized) Büchi, or Street automaton, is an easy step, involving only the need of encoding the acceptance conditions (sequences of sets of states). The task of deciding the language emptiness of a Büchi automata, $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$, that in previous sections we showed equivalent to the LTL model checking problem, is done symbolically with a procedure due to Emerson and Lei [42]. Such a procedure involves the computation of the two nested fix points (a maximum and a minimum fix points) below:

$$\nu Y.[\mathsf{pre}(\mu Z.[Y \cap (F \cup \mathsf{pre}(Z))])]$$

Being based on two nested maximal and minimum fix points the algorithm by Emerson and Lei requires $\Theta(|S^2|)$ symbolic steps in the worst case. It is possible to design a slightly modified version of the procedure by Emerson and Lie, to compute generalized Büchi, as well as Street, automata language emptiness [42, 5].

6.5 Tools for Model Checking

A number of tools have been developed in the literature for symbolic model checking. As part of his Ph.D. dissertation Ken McMillan and some Carnegie Mellon researchers developed a tool called SMV, performing CTL symbolic model checking. The algorithms outlined in the previous section and described in [78] are implemented in the SMV systems. The model checker NuSMV [19] is a reimplementation and extension of SMV, jointly developed by Carnegie Mellon University, the ITC-IRST, and the University of Trento, Italy. The NuSMV tool permits both CTL symbolic model checking (admitting weak and strong fairness constraints) and LTL symbolic model checking. Besides, NuSMV integrates symbolic OBDD based model checking with *bounded model checking* [20], which is a verification technique relying on encoding the model checking problem on a propositional satisfiability problem, and on the use of efficient SAT solvers. Another popular symbolic model checking tool is VIS, developed jointly by the University of California, at Berkeley, the University of Colorado at Boulder, and the University of Texas, Austin. VIS performs design verification using symbolic fair CTL model checking and language emptiness.

In the context of explicit model checking, LTL on the fly model checking proved its "on the fly" space (and time) saving resource to be powerful enough to deal with a number of real systems. The state-of-the-art model checker for LTL on the fly model checking is the tool SPIN, developed at Bells labs. The SPIN model checkers uses the Double DFS algorithm in [32, 66] to check Büchi emptiness and relies on partial order reduction techniques [65, 87], to enhance over the state explosion problem.

7 Rank-Based Symbolic Bisimulation Reduction (and Model Checking)

The problem of computing, within a symbolic framework, bisimulation and simulation reductions attracted the attention of a number of researchers in model checking [47, 48, 12, 26, 99]. Merging techniques and tools to face the space explosion problem is, in fact, a natural framework to achieve better standard models dimensions in model checking. Recently, the authors of [39] left open the problem of giving a symbolic version of their original bisimulation algorithm, saving both time and space exploiting the original notion of *rank*. Broadly, such notion allows to layer the input graph and to localize bisimulation computation. The main obstacle to jump, within such translation, was recognized in the definition of a symbolic subprocedure to obtain the graph layers (ranks). Given such subunit, in fact, it is possible to easily design a symbolic version of the mentioned algorithm, that manipulate sets of nodes with at most $\mathcal{O}(V)$ symbolic steps. On the ground of the symbolic SCC procedure defined in Chapter 2, an $\mathcal{O}(V)$ symbolic steps algorithm computing ranks can be designed, as well. With these preliminaries, we discuss in this chapter the problem of defining a symbolic rank-based algorithm as well as that of enhancing model checking technique using the notion of rank.

7.1 Symbolic Bisimulation and Simulation Procedures

The first symbolic bisimulation algorithm was defined by Bouali and de Simone in [12]. The procedure in [12] implements the naïve negative strategy refining identity equivalence relation, until a fixpoint is reached. More precisely, given the bisimulation approximation $B_n(s, s')$, the operator involved in the fixed point computation, refines $B_n(s, s')$ to $B_{n+1}(s, s')$ as follows:

$$
\begin{gathered}
B_{n+1}(s, s') \Leftrightarrow B_n(s, s') \wedge \\
\wedge \forall s_1 (B_n(s, s_1) \rightarrow \exists s'_1 (E(s', s'_1) \wedge B_n(s_1, s'_1)) \wedge \\
\wedge \forall s'_1 (B_n(s', s'_1) \rightarrow \exists s_1 (E(s, s_1) \wedge B_n(s, s_1))
\end{gathered}
$$

The authors of [12] work on a symbolic representation of the subgraph maintaining only the nodes reachable from the initial vertex and presents a number of experimental results. In [12] there is no further discussion on the overall complexity, however the number of OBDD operations is easily bounded by $\mathcal{O}(V^2)$, since in the worst case $\mathcal{O}(V^2)$ iterations of the operator are needed to obtain a fixpoint. Since a relation on the graph nodes is manipulated, $3n$ variables are needed in the symbolic encoding by OBDDs in [12], where $n = \log(V)$.

In [47] Fisler and Vardi analyze the complexity of the symbolic versions of the bisimulation algorithm by Paige and Tarjan [84], the one by Bouajjani, Fernandez, and Halbwachs [10], and the procedure by Lee and Yannakakis [75]. All these algorithms manipulate a partition over the graph vertex-set, rather than a relation on nodes. Hence, as noted in [47] a lower number of OBDD variables is needed overall the computation. Moreover, for all algorithms the number of OBDD operations is asymptotically bounded by $\mathcal{O}(V)$ in the worst case, since a partition on V is refined. A deeper experimental comparison among these symbolic bisimulation procedures is conducted in [47] concluding that the symbolic version of the algorithm in [84] performs better than the other two algorithms, since it gains from the *right* choice of the splitters.

Symbolic simulation algorithms that refine the identity relation using an operator which implements the definition of simulation, similarly to the symbolic bisimulation algorithm in [12], can be easily obtained and are discussed in [26, 99]. Even if they do not present any implementation, the authors of [18] discuss the problem of giving a symbolic version of their simulation algorithm: such translation is easily realizable since the overall algorithm essentially works refining sets of nodes as well as an order relation on nodes classes. The same pattern applies to our simulation algorithm in Chapter 5.

7.2 Explicit Rank-Based Bisimulation Reduction

The main idea behind the explicit bisimulation algorithm presented in [38] is the use of the notion of *rank* to both initialize the partition and drive the order of the splitting operations. The notion of *rank* is given in [39] on the ground of the *component graph*, G^{scc}, recalled in Definition 7.2.1, below.

Definition 7.2.1 *Given a graph $G = \langle V, E\rangle$, let the* component graph *of G, $G^{scc} = \langle V^{scc}, E^{scc}\rangle$, to be the graph obtained as follows:*

$$
\begin{array}{rcl}
V^{scc} & = & \{c : c \textit{ is a strongly connected component in } G\} \\
E^{scc} & = & \{\langle c_1, c_2\rangle : c_1 \neq c_2 \textit{ and } (\exists n_1 \in c_1)(\exists n_2 \in c_2)(\langle n_1, n_2\rangle \in E)\}
\end{array}
$$

Given a node $n \in V$, we refer to the node of G^{scc} associated to the strongly connected component of n as $c(n)$.

We need to distinguish the acyclic (well-founded) part of a graph G from its cyclic (non-well-founded) part.

Definition 7.2.2 *Let $G = \langle V, E \rangle$ and $n \in V$. $G(n) = \langle V(n), E \restriction V(n) \rangle$ is the subgraph of G of the nodes reachable from n. $WF(G)$, the well-founded part of G, is $WF(G) = \{n \in V : G(n)$ is acyclic$\}$.*

The nodes in $WF(G)$ are said to be well-founded, while the other nodes are said to be non-well-founded.

Definition 7.2.3 *Let $G = \langle V, E \rangle$. The* rank *of a node n of G is defined as:*

$$\begin{cases} rank(n) = 0 & \text{if } n \text{ is a leaf in } G \\ rank(n) = -\infty & \text{if } c(n) \text{ is a leaf in } G^{scc} \text{ and } n \text{ is not a leaf in } G \\ rank(n) = \max(\{1 + rank(m) : \langle c(n), c(m) \rangle \in E^{scc}, m \in WF(G)\} \cup & \\ \qquad\qquad \{rank(m) : \langle c(n), c(m) \rangle \in E^{scc}, m \notin WF(G)\}) & \text{otherwise} \end{cases}$$

In Figure 7.1, below, we report the rank-based algorithm presented in [38]. As a first step, such algorithm determines the rank-layers, say $B_{-\infty}, B_0 \dots B_\rho$, of the input graph, that are used to initialize the bisimulation partition (lines (1)–(5) of RankBisimulation). The rank layer $B_{-\infty}$ collects all nodes having no outgoing edges, and represents a definitive bisimulation class: hence, $B_{-\infty}$ is used to split opportunely the other partition elements i.e. the other rank layers (lines (6)–(8)). Finally, the algorithm processes the rank layers $B_0 \dots B_\rho$ in *increasing* order. For each rank layer, B_i, a classical efficient bisimulation algorithm (as, for example, Paige and Tarjan algorithm [84]) is used, to obtain the final bisimulation splitting local to B_i. Once obtained such final partition over B_i, its classes are used to split opportunely higher rank layers. These layers are further processed, in turn, with the chosen classical efficient bisimulation procedure (lines (9)–(17) of RankBisimulation). The correctness of the rank-based bisimulation algorithm in [38] is based, ultimately, on the two following facts. First, each graph edge, (u, v), is such that $rank(u) \geq rank(v)$ (this allow to process ranks in increasing order), secondly, the rank layering is a partition coarser than bisimulation. We address to [38, 39] for further details about correctness of RankBisimulation algorithm.

The complexity of RankBisiulation easily follows from the following considerations. Since G^{scc} can be computed using Tarjan's classical algorithm [106] in time $\mathcal{O}(|V| + |E|)$, step (1) can be performed in time $\mathcal{O}(|V| + |E|)$. Given a graph $G = \langle V, E \rangle$, let $\rho = \max\{rank(n) : n \in V\}$ and, for $i \in \{-\infty, 0, \dots, \rho\}$, let $B_i = \{n \in V : rank(n) = i\}$. First, the algorithm initializes the partition P to be refined using the blocks B_i. For each rank i a bisimulation procedure Bisim is called on $G_i = \langle B_i, E \restriction B_i \rangle$ and the partition P is updated on nodes of rank greater than i. The procedure presented in [84] can be used with a cost $\mathcal{O}(|E \restriction B_i| \log |B_i|)$, while [85] would cost $\mathcal{O}(|E \restriction B_i| + |B_i|)$. In this way all the edges connecting nodes at different ranks are used only once. Hence, the algorithm correctly computes the

```
RankBisimulation ($G = \langle V, E\rangle$)

(1)  for $n \in V$ do compute $rank(n)$;                  — compute the ranks
(2)  $\rho := \max\{rank(n) : n \in V\}$;
(3)  for $i = -\infty, 0, \dots, \rho$ do $B_i := \{n \in V : rank(n) = i\}$;
(4)  $P := \{B_i : i = -\infty, 0, \dots, \rho\}$;        — partition initialized with the $B_i$'s
(5)  $G :=$ collapse$(G, B_{-\infty})$;                    — collapse all the nodes of rank $-\infty$
(6)  for $n \in V \cap B_{-\infty}$ do                     — refine blocks at higher ranks
(7)      for $C \in P \wedge C \neq B_{-\infty}$ do
(8)          $P := (P \setminus \{C\}) \cup \{\{m \in C : \langle m, n\rangle \in E\}, \{m \in C : \langle m, n\rangle \notin E\}\}$;
(9)  for $i = 0, \dots, \rho$ do
(10)     $D_i := \{X \in P : X \subseteq B_i\}$;           — find blocks currently at rank $i$
(11)     $G_i := \langle B_i, E \restriction B_i\rangle$;  — isolate the subgraph of rank $i$
(12)     $D_i :=$ Bisim$(G_i, D_i)$;   — process rank $i$ calling either [84] or [85]
(13)     for $X \in D_i$ do
(14)         $G :=$ collapse$(G, X)$;                      — collapse nodes at rank $i$
(15)     for $n \in V \cap B_i$ do                         — refine blocks at higher ranks
(16)         for $C \in P \wedge C \subseteq B_{i+1} \cup \ldots \cup B_\rho$ do
(17)             $P := (P \setminus \{C\}) \cup \{\{m \in C : \langle m, n\rangle \in E\}, \{m \in C : \langle m, n\rangle \notin E\}\}$;
```

Figure 7.1: The algorithm in [38].

bisimulation quotient of the input graph G and can be implemented so as to run with a worst case complexity of $\mathcal{O}(|E| \log |V|)$. Moreover, if G is an acyclic graph, the **for**-loop in line (9) is superfluous and the overall cost is $\mathcal{O}(|V| + |E|)$. In [38] further optimizations of the above algorithm are presented allowing a linear time complexity also in other cases.

We conclude this section by observing that the space requirement of the above algorithm can be reduced to the size of the largest G_i to be processed.

7.3 Symbolic Rank Computation

In order to define a symbolic version of the algorithm proposed in [38] (see Figure 7.1) we mainly need to find a symbolic efficient strategy to compute the rank-partition of the graph. As we explain further, the remaining steps are easily realizable in a symbolic setting (the symbolic bisimulation algorithms discussed in [47, 48] contain such steps) :

- the operations in the **for**-loops at steps (6) and (15) are standard splitting operations, i.e. they replace C with $C \cap \mathsf{pre}(X)$ and $C \setminus \mathsf{pre}(X)$;

- the extraction of the subgraph G_i at line (11) corresponds to the boolean operation $E \upharpoonright B_i(\bar{x}, \bar{y}) = E(\bar{x}, \bar{y}) \wedge (B_i(\bar{x}) \wedge B_i(\bar{y}))$;
- the operation $\textsc{Bisim}(G_i, D_i)$ at line (12) can be performed by using a symbolic bisimulation algorithm.
- $\textsc{collapse}(G, X)$ (steps (5) and (14)) means that all the nodes in X are bisimilar and we do not have to further process them;

For the above reasons, in this section we concentrate our efforts on the rank computation.

In the explicit case Tarjan's algorithm [106] identifies, in $\mathcal{O}(|V| + |E|)$ steps, all the strongly connected components of G. Once the graph G^{scc} has been computed, it is possible to assign to each node of G its rank, accordingly to Definition 7.2.3, through a visit of G. Such a two-step procedure is applicable also symbolically: if the symbolic SCC algorithm defined in Chapter 2 is used, an overall $\mathcal{O}(V)$ symbolic steps bisimulation procedure is obtained. However, as we will see in the rest of this section, it is possible to obtain the graph rank layering even directly i.e. without passing through the finer SCC partition. This way, nodes sharing their codes in the same OBDD need not to be separated because they belong to different SCC components; rather, nodes are assigned to distinct OBDDs only if they belong to distinct ranks. The above outlined symbolic rank procedure relies on a special rephrasing of Definition 7.2.3, exploiting a different characterization of the notion of rank. Definition 7.3.1, below, introduces precisely such new characterization, leading us to the definition of a procedure performing the rank-layering of a graph in $\mathcal{O}(|V|)$ symbolic steps, and avoiding the computation of G^{scc}.

Definition 7.3.1 *Let $G = \langle V, E\rangle$. For each node $n \in V$ let $\overline{rank}(n)$ be defined as follows:*

$$\begin{cases} \overline{rank}(n) = 0 & \text{if } n \text{ is a leaf of } G \\ \overline{rank}(n) = \max(\{1 + \overline{rank}(m) : \langle n, m\rangle \in E\}) & \text{if } n \in WF(G) \text{ is not a leaf} \\ \overline{rank}(n) = \max(\{-\infty\} \cup \{1 + \overline{rank}(m) : \\ \qquad m \in WF(G) \;\wedge\; path(n, m)\}) & \text{if } n \notin WF(G) \end{cases}$$

where $path(n, m)$ is true iff there is a path connecting n to m in G.

The following lemma states the equivalence between the above definition and Definition 7.2.3.

Lemma 7.3.2 *Let $G = \langle V, E\rangle$. For each node $n \in V$ it holds:*

$$rank(n) = \overline{rank}(n).$$

Proof. Consider $G^{scc} = \langle V^{scc}, E^{scc}\rangle$. We start by observing that if $n, m \in V$ belong to the same strongly connected component, then by Definition 7.2.3 it holds that $rank(n) = rank(m)$. Since two nodes in the same strongly connected component reach exactly the same nodes, it also holds, by Definition 7.3.1, that $\overline{rank}(n) = \overline{rank}(m)$.

With the above consideration in mind we will proceed in our proof by induction on the height of G^{scc}.

For the base case, let $n \in V$ be such that $c(n)$ is a leaf in G^{scc}. Then, either n is a leaf of G or there is no path from n to any node in $WF(G)$. Hence, by Definition 7.2.3, either $\mathit{rank}(n) = \overline{\mathit{rank}}(n) = 0$, or $\mathit{rank}(n) = \overline{\mathit{rank}}(n) = -\infty$.

For the inductive step, let $n \in V$ be such that $c(n)$ has height $h+1$ in G^{scc}. If $n \in WF(G)$ then $\langle n, m\rangle \in E$ iff $\langle c(n), c(m)\rangle \in E^{scc}$. Moreover, if $\langle c(n), c(m)\rangle$ is an edge of G^{scc}, then m is a well-founded node. Hence, exploiting the inductive hypothesis together with Definition 7.2.3 and Definition 7.3.1 it holds that:

$$\max(\{1 + \overline{\mathit{rank}}(m) : \langle n, m\rangle \in E\}) = \max(\{1 + \mathit{rank}(m) : \langle c(n), c(m)\rangle \in E^{scc}\})$$

and $\mathit{rank}(n) = \overline{\mathit{rank}}(n)$.

If $n \notin WF(G)$, consider the set $S = \{m \mid \langle c(n), c(m)\rangle \in E^{scc}\}$. Since a well-founded node is reachable from n iff it is reachable from some $m \in S$, it holds that $\overline{\mathit{rank}}(n)$ is:

$$\max(\{\overline{\mathit{rank}}(m) : m \in S \cap WF(G)\} \cup \{\overline{\mathit{rank}}(m) : m \in S \setminus WF(G)\}) \cup \{-\infty\}.$$

The inductive hypothesis and the definition of *rank* allow us to easily get the thesis.

■

By Definition 7.3.1, the rank of a well-founded node is the maximum length of a path starting from it, while the rank of a non-well-founded node is 1 plus the maximum length of a path starting from one of its well-founded descendants (or $-\infty$ if such a path does not exist). The symbolic rank-layering algorithm in Figure 7.2 proceeds as follows: it identifies the well-founded nodes, starting from rank 0 up to rank p; then, it uses the well-founded nodes to compute the ranks of the non-well-founded ones. In particular, it first uses the well-founded nodes at rank p to determine the non-well-founded nodes at rank $p+1$, then it uses the well-founded rank $p-1$ to determine the non-well-founded rank p, and so on. The linear complexity of the procedure follows from the fact that each pre-image computation discovers at least one new node of the graph. Hence, the number of symbolic steps is linear in the number of nodes of the graph. Theorems 7.3.3 and 7.3.4 state the correctness and the complexity of the proposed algorithm.

Theorem 7.3.3 *Let* $G = \langle V, E\rangle$ *be a graph. The algorithm* SymbolicRank *always terminates and the classes of the partition over* V *induced by the rank are* $\{B_{-\infty}, B_0, \ldots, B_\rho\}$.

Proof. Consider the set of nodes SET in the algorithm SymbolicRank. Such a set is initialized in step (2) to V. Then, whenever it is modified, some nodes are removed from it and no node is added. In particular, each iteration of the first while-loop assigns to SET its pre-image. Such a pre-image is always a subset of SET. Each iteration of the second while-loop removes from SET the subset $SET \cap FRONT$ which is not empty (guard of the loop). The above considerations ensure the termination of

SymbolicRank ($G = \langle V, E\rangle$)

```
(1)  $i := 0$;
(2)  $SET := V$;                                  — SET is the set of not-ranked nodes
(3)  $PRESET := \mathsf{pre}(SET)$;               — PRESET = preimage of not-ranked nodes
(4)  while ($SET \neq PRESET$) do
(5)      $B_i := SET \setminus PRESET$;           — $B_i$ = well-founded nodes of rank i
(6)      $SET := PRESET$;                         — remove well-founded nodes of rank i from SET
(7)      $PRESET := \mathsf{pre}(SET); i := i+1$; — update PRESET

— SET now contains only not well-founded nodes —
(8)  for $j = i$ down to 1 do
(9)      $FRONT := B_{j-1}$;                      — put in FRONT well-founded nodes of rank $j-1$
(10)      while $\mathsf{pre}(FRONT) \cap SET \neq \emptyset$ do
(11)          $FRONT := \mathsf{pre}(FRONT) \cap SET$;    — discover new nodes
(12)          $SET := SET \setminus FRONT$;               — remove from SET the new nodes
(13)          $B_j := B_j \cup FRONT$;                    — assign rank j to the nodes discovered
(14)      if $SET \neq \emptyset$ then $B_{-\infty} := SET$;  — rank $-\infty$ to the nodes still in SET
(15)      return $\{B_{-\infty}, B_0, \ldots, B_\rho\}$
```

Figure 7.2: The Symbolic Rank algorithm.

the two while-loop as well as of the algorithm SymbolicRank. Moreover, as soon as a subset has been removed from SET it is inserted in one of the B_i (lines (5) and (7)), while $B_{-\infty}$ (line (14)) collects whatever remain in SET. Thus $\{B_{-\infty}, B_0, \ldots, B_\rho\}$ is a partition over V. We will now prove that each $n \in WF(G)$ is put in the right rank-set ($B_{rank(n)}$), during the $rank(n)+1$-th iteration of the first while-loop. Let us proceed by induction on the *rank* of $n \in WF(G)$. The first iteration of the loop in step (4) puts in B_0 all nodes in $V \setminus \mathsf{pre}(V)$ and it is entered only if such a set is not empty. Thus, if there are not well-founded nodes ($V \setminus \mathsf{pre}(V)$), the first while-loop is not executed. Otherwise, all the leaves of the graph are put in B_0 during its first iteration. For the inductive step, note that lines (6)–(7) of the code ensure that, as soon as a vertex is assigned to a rank, it is removed from SET. Hence, at the beginning of the $j+1$-th iteration, with $j+1 \leq max\{rank(n)+1 | n \in WF(G)\}$, of the first loop, SET is V deprived of all well-founded nodes having height less then j. If SET is equal to its preimage ($PRESET$) we have that $SET = V \setminus WF(G)$ and the loop is not entered. Otherwise B_j is equipped of all well-founded nodes having height j. Now, consider the for-loop (line (8)) and let $\gamma = max\{rank(n) | n \in WF(G)\}$. We have just proved that, on the entering to such a loop, SET contains all non-well-founded nodes of V. The first for-loop iteration is executed only if $i \geq 1$ (i.e. only if some well-founded rank has been generated) and inserts in $B_{\gamma+1}$ all nodes having some descendent in B_γ. Moreover, line (12) removes from SET all nodes just assigned to a rank. Thus, an inductive argument can be again used to prove that the j-th iteration, with $j \in \{1, \dots, \gamma+1\}$, puts in $B_{\gamma+2-j}$, all non-well-founded nodes whose maximal-height well-founded descendent has *rank* $\gamma+1-j$. Hence, when line (14) is executed, SET contains all nodes having no well-founded descendent which are put in B_∞. We can conclude that $\{B_{-\infty}, B_1, \dots, B_i\}$ are the classes of the partition over V induced by the *rank*. ■

Theorem 7.3.4 *Let $G = \langle V, E\rangle$ be a graph. The* **Symbolic Rank** *algorithm performs $\mathcal{O}(|V|)$ symbolic steps to produce the partition $\{B_{-\infty}, B_0, \dots, B_\rho\}$ over V.*

Proof. Let γ be the maximum rank of a well-founded node and $M = V \setminus WF(G)$. We will prove that the algorithm in Figure 7.2 performs at most $\mathcal{O}(\gamma+|M|)$ symbolic steps. Trivially $\gamma \leq |WF(G)|$, hence it holds $\mathcal{O}(\gamma+|M|) = \mathcal{O}(|V|)$. The j-th iteration of the first while-loop discovers exactly those well-founded nodes having rank $j-1$ performing only a constant of symbolic steps. Hence, to execute lines (1)–(7) we perform at most $\mathcal{O}(\gamma)$ symbolic steps. As stated by Theorem 7.3.3, before entering in the for-loop the set SET is $V \setminus WF(G) = M$. During each iteration of the innermost while-loop at least one node is removed from SET (line (12)), since $FRONT \cap SET \neq \emptyset$ because of the while-guard. Moreover, SET is never augmented during the computation. Since during each iteration of the innermost while-loop only one pre-image operation is executed the global cost of lines (8)–(15) is $\mathcal{O}(|M|)$ symbolic steps and we have the thesis. ■

Note that also the number of set-differences, intersections and unions involved in the procedure is $\mathcal{O}(|V|)$.

7.4 Symbolic Rank-Based Bisimulation Reduction

As we said in previous sections, in [47] Fisler and Vardi analyzed the symbolic cost of three symbolic bisimulation algorithms. In particular, they prove that for the symbolic version of the Paige and Tarjan algorithm the overall complexity depends on $\alpha(2M + D + I + Q)$, where α is the number of iterations necessary to reach the fixpoint, M is the cost of an image or preimage operation and D, I, and Q are the costs of one operation of difference, intersection, and equality test, respectively.

A symbolic version of the Paige and Tarjan algorithm can be used in step (7.a) of our symbolic algorithm. The differences between using directly the symbolic version of the Paige and Tarjan algorithm and using it inside our routine are similar to the differences that arises in the explicit case [38]. In particular, we start with an initial partition, the rank-partition, which is closer to bisimulation than the one used in Paige and Tarjan algorithm, hence, in general, our computation requires less iterations to converge to a fixpoint. Moreover, we use the edges which connect nodes at different ranks only once, while it is possible that in the Paige and Tarjan algorithm these edges are used more times.

The rank notion can be used to enhanced the computation in Bouali and de Simone bisimulation algorithm [12], as well. The Bouali and de Simone algorithm starts with the total relation $R_0 = \{\langle n, m\rangle : n, m \in R\}$, where R is the subset of V of reachable nodes, and refines it until bisimulation relation is reached. The correctness of the Bouali and de Simone algorithm remains valid whenever the starting relation R_0 contains the maximum bisimulation relation $\equiv$, i.e. $\equiv \subseteq R_0$. Moreover, the more the relation R_0 approximates the relation $\equiv$, the less iterations are necessary to compute $\equiv$. Hence, once the rank has been symbolically computed, we can exploit it to speed up the Bouali and de Simone algorithm by starting with the relation

$$R_0 = \{\langle n, m\rangle : \mathit{rank}(n) = \mathit{rank}(m)\}.$$

The Ordered Binary Decision Diagram of R_0 can be immediately built from the OBDD's for $B_{-\infty}, B_0, \ldots, B_\rho$, since the characteristic function of R_0 is

$$(B_{-\infty}(\bar{x}) \wedge B_{-\infty}(\bar{y})) \vee \bigvee_{i=0}^{\rho} B_i(\bar{x}) \wedge B_i(\bar{y}).$$

7.5 Rank-based Model Checking

In the previous sections the notion of rank turns out to be at the basis of the development of an efficient bisimulation algorithm. In this section we briefly discuss how the same notion could be exploited to define optimization heuristics for model checking procedures. We deal with both the explicit and the symbolic model checking for CTL temporal logic [26, 99].

7.5.1 The Rank in Explicit CTL Model Checking

In the explicit setting, the core of the linear CTL model checking procedure described in [22, 26, 99], and outlined in Section 6.2, is constituted of four subroutines. Each subroutine is aimed at processing one operator in the complete set of CTL operators $\{\neg, EX, EG, EU\}$. These subroutines, and the overall linear CTL explicit model checking algorithm, require to keep in main memory the entire system. This could be avoided if we were able to partition the input model in layers over which the computation could be *localized*. In this subsection we shortly discuss exactly this issue.

A first obvious observation is that the CTL formulas whose outermost operator is a boolean one could be locally processed using any arbitrary graph partition. In other words, consider an arbitrary partition, $\langle L_1 \ldots L_k\rangle$, on the states of a Kripke structure: if L_i^ϕ is the set of states labeled with ϕ (i.e. satisfying ϕ) in the class L_i, then it is possible to label with $\neg\phi$ all states in $L_i \setminus L_i^\phi$, independently from considering the other blocks in the partition.

More attention has to be paid in the definition of a partition on the states of a Kripke structure, if we require to localize the processing of subformulae whose outermost operator is one of EX, EU, and EG. A condition on states partition, easy to prove sufficient for the above localization purpose, is the following: it is possible to define a complete order on the classes of the partition, $\langle L_1 \dots L_k \rangle$, such that for each path, $\pi = \langle s_0 \dots \rangle$, in the Kripke structure, it holds that $\forall i \geq 0(s_i \in L_j \wedge s_{i+1} \in L_{j'} \rightarrow j \geq j')$. In other words, the classes of the partition are ordered layers, such that each Kripke structure path traverses layers in decreasing order. A layering as above permits to use CTL standard procedures for EX, EU, and EG, to locally deal with CTL subformulae on each layer. It is only necessary to pay attention to correctly initialize the labelling of states in a given layer L_i, provided the labelling on the lower layers $L_{i-1}, \dots, L_0$ is properly defined. For instance consider the CTL formula $E(hUg)$ and assume to have already determined (i.e. labelled) states in layers $L_0, \dots, L_{i-1}$ satisfying it. Before applying the classical CTL subroutine for the case EU on L_i, we simply have to label with $E(hUg)$ those states in L_i having some successors in a lower layer labelled with $E(hUg)$. Note that this step requires to keep in memory at most two layers of the model at a time.

The notion of rank in Definition 7.2.3 allows to partition the model in layers over which localizing CTL model checking as described above. In order to use a localized CTL model checking an alternative notion of rank can be the height of the strongly connected component to which the node belongs. Such a rank definition is suitable for CTL model checking since it induces a model stratification such that the layers successively encountered by a path are of decreasing height. This notion of rank performs better than the one in Definition 7.2.3 relatively to CTL model checking. In fact, it induces a finer partition on the model and can be determined with the same time complexity using Tarjan explicit algorithm in [106].

7.5.2 The Rank in Symbolic CTL Model Checking

In this section we briefly discuss how to localize the computation of CTL symbolic model checking algorithm (see [78, 26, 99] or Section 6.4) on Kripke structure layers. Layers of states can be represented using OBDDs and need to satisfy the same property required in the explicit case, i.e. they are entered by Kripke structure paths in a decreasing order. As pointed out in Section 7.5.1, the notion of rank in Definition 7.2.3 induces a graph partition with the above property. Moreover, in Section 7.3 we proved that such a partition can be computed in a linear number of symbolic steps.

Let $B_{-\infty}, B_0(\bar{x}), \dots, B_\rho(\bar{x})$ be the OBDDs representing the nodes having ranks $-\infty, 0, \dots, \rho$, respectively and $E(\bar{x}, \bar{y})$ be the OBDD representing the relation of the graph. It is possible to partition $E(\bar{x}, \bar{y})$ in the OBDDs:

- $E_{-\infty}(\bar{x}, \bar{y}), E_0(\bar{x}, \bar{y}), \dots, E_\rho(\bar{x}, \bar{y})$ representing sets of edges among nodes having the same rank, i.e. $E_i(\bar{x}, \bar{y}) = E(\bar{x}, \bar{y}) \wedge B_i(\bar{x}) \wedge B_i(\bar{y})$;

- $E\downarrow_1(\bar{x}, \bar{y}), \dots, E\downarrow_\rho(\bar{x}, \bar{y})$ representing sets of edges connecting states in a given rank to states in lower ranks, i.e. $E\downarrow_i(\bar{x}, \bar{y}) = E(\bar{x}, \bar{y}) \wedge B_i(\bar{x}) \wedge \bigvee_{j<i} B_j(\bar{y})$.

It is possible to build the OBDDs of states satisfying the CTL formula AGg by ranks: at each rank, i, the OBDDs $B_i(\bar{x}), E_i(\bar{x}, \bar{y})$, and $E\downarrow_i(\bar{x}, \bar{y})$ are considered.

The OBDD of the states at rank 0 and satisfying AGg can be built by computing:

$$\nu Z.(B_0 \wedge g \wedge AXZ).$$

Let $AG^* = AG_0 \vee \ldots \vee AG_i$ be the OBDD of all states having rank less than $i+1$ and satisfying AGg. The OBDD of states at rank $i+1$ satisfying AGg can be obtained by computing:

$$K = AX(AG^* \vee B_{i+1}) \qquad \text{and} \qquad \nu Z.(K \wedge g \wedge B_i \wedge AXZ).$$

In the computation of K it is only necessary to use the OBDD $E \downarrow_{i+1} (\bar{x}, \bar{y})$ to implement the AX operation. The AX operation involved in the fixpoint can be instead implemented by using $E_{i+1}(\bar{x}, \bar{y})$.

Similar arguments apply to the other CTL operators showing how the notion of *rank* provides a method to distribute the computation on successive layers. Such a layering, on the one hand speeds up the convergence of the fixpoint computations. On the other hand, it allows to use the OBDDs representing the *ranks* which could be smaller than the OBDD for the entire model.

8

Symbolic SCC Analysis, Bad Cycle Detection and Fairness

The aim of this chapter is that of collecting a number of tasks, in symbolic model checking, whose computation is enhanced on the ground of the symbolic SCC algorithm introduced in Chapter 2.

The computation of strongly connected components is an important preprocessing step in explicit model checking. In fact, it allows to answer in linear time (with respect to the system model and the checking formula sizes) if $\mathcal{M}$ is a model for the CTL formulas involving EG [26, 99]. SCC computation is used to solve the Büchi emptiness problem, to which the LTL model checking problem can be reduced [110, 26, 99]; besides, SCC analysis is involved in CTL fair model checking [26, 99].

Within symbolic model checking, special algorithms, not computing upon SCC analysis, were developed for the above problems: in fact, due to the complexity of SCC symbolic algorithms in the literature to date, relying on SCC computation resulted in a computational bottle neck, rather than a speedup. The development of the $\mathcal{O}(V \log(V))$ symbolic steps SCC algorithm in [5] allowed to design SCC-based algorithms, for some of the cited problems, performing in $\mathcal{O}(V \log(V))$, rather than $\mathcal{O}(V^2)$ symbolic steps [5, 113, 6, 93]. The SCC symbolic algorithm in Chapter 2 finally cut down from $\mathcal{O}(V \log(V))$ to $\mathcal{O}(V)$) the number of symbolic steps needed for performning the same tasks.

8.1 Büchi Language Emptiness, Double DFS and Spine-Sets

Within the automata based approach to LTL model checking [110] (see Section 6.2), both the system model and the negation of the correctness requirement, $\neg\phi$, are translated into Büchi automata, $\mathcal{A}$ and $\mathcal{A}_{\neg\phi}$. Then, the language emptiness of the automaton $\mathcal{A} \times \mathcal{A}_{\neg\phi}$ is checked: in fact, the composed automaton accepts an input if and only if the system does not satisfy the correctness specification. The Büchi emptiness problem is in turn equivalent to the detection of a cycle, reachable from the initial states and containing a *final* state: for this reason, in the above context the algorithmic task of determining the language emptiness for $\mathcal{A} \times \mathcal{A}_{\neg\phi}$ is called *bad cycle detection*.

More precisely, (the language of) a Büchi automaton is defined as follows:

Definition 8.1.1 (Büchi Automata) *A* Büchi automaton*, $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$, is a tuple in which:*

- *Σ is a finite alphabet;*
- *V is a finite set of states;*
- *$\Delta : Q \times \Sigma \times Q$ is the transition relation;*
- *V_0 is the set of initial states;*
- *$F \subseteq Q$ is a set of nodes that represents the acceptance condition.*

A *run* of $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$, on the word $\alpha \in \Sigma^\omega$, is a mapping, $\rho : \{0, 1, \ldots, \omega\} \mapsto V$, such that $\rho(0) \in V_0 \wedge \forall i \geq 0(\rho(i), \alpha[i], \rho(i+1)) \in \Delta)$. In a Büchi automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, Q_0, F \rangle$ a run is *accepting* if and only if $inf(\rho) \cap F \neq \emptyset)$, where $inf(\rho)$ represents the set of states comparing an infinite number of times in the codomain of ρ.

The *language* of $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$ consists of those words $\alpha \in \Sigma^\omega$ for which there exists an accepting run.

Lemma 8.1.2 ([108]) *The language of $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$ is not empty if and only if the graph, $\langle V, E = \cup_{a \in \Sigma} \Delta(a) \rangle$, contains a cycle of states that can be reached through a path starting from an initial state $q \in V_0$.*

The characterization in Lemma 8.1.2 allows to use SCC decomposition to design an explicit Büchi emptiness language algorithm, linear in the size of $\mathcal{A} \times \mathcal{A}_{\neg\phi}$. Let V' be the set of states reachable from V_0 using the relation $E = \cup_{a \in \Sigma} \Delta(a)$. Once obtained the SCCs in $\langle V', E \rangle$, using linear Tarjan algorithm [106], it is only necessary to check if some strongly connected component is not trivial and contains a final state. By combining a similar strategy with the SYMBOLIC-SCC algorithm in Chapter 2, it is possible to design a Büchi emptiness algorithm performing $\mathcal{O}(V)$ symbolic steps. The Büchi emptiness algorithms used in the symbolic model checkers VIS and SMV [94, 78], are based on a procedure by Emerson and Lei [42], that uses a nested alternate fix-point computation, and performs $\Theta(V^2)$ symbolic steps in the worst case [5]. More precisely, it is possible to obtain a linear symbolic Büchi emptiness algorithm by simply substituting line (7) of SYMBOLIC-SCC, with the following instruction, that checks if the outlined strongly connected component is not trivial and contains a final state:

```
(7')  if (post($F \cap SCC$) $\cap$ $SCC \neq \emptyset$) then return "language NOT EMPTY"
```

Line (7′) performs a constant number of symbolic steps and thus, clearly, it does not change the complexity of SYMBOLIC-SCC.

Explicit model checkers, as SPIN [64], indeed solve the bad cycle detection problem, avoiding to compute each strongly connected component. A linear procedure called NestedDFS [32, 66] is used. NestedDFS proceeds by discovering nodes, in a depth first search manner, as soon as a final node, $v \in F$, is detected; at that point a new nested DFS visit is called from v. On this ground a bad cycle can be recognized by the presence of the same element on the stacks maintained by nested DFS executions. This greedy strategy drives computation around final states, allowing to recognize as soon as possible bad cycles. In the following, we aim at recovering some of these ideas in the symbolic framework: in particular, we will make use of the spine-set technique to design a symbolic bad cycle detection algorithm performing $\mathcal{O}(V)$ symbolic steps, and avoiding to build unnecessary SCCs (i.e. SCCs not containing any final node).

More precisely, the procedure SymNestedSearch, depicted in Figure 8.1, uses the following ingredients, common to NestedDFS explicit procedure:

- the process of graph visiting, from a final node, is stopped as soon as a new final node is discovered. At that point, such a new final node is used as the initial vertex for a new graph visiting. While in the explicit case graph visiting is made in a depth first search manner, in the symbolic case, breadth first visit is used.
- In the explicit case, the use of stacks of states allows us to trace paths passing from final nodes, and to recognize the case in which such paths become cycles. In this case, the presence of a bad cycle is detected. In the symbolic case, we use nodes in a spine-set to witness paths passing from final nodes. if, within the computation, some node in spine is rediscovered, this means that a bad cycle has been detected.

In more details, each iteration of SymNestedSearch (see Figure 8.1), grows the forward-set, $FW(v)$, of a final node, as soon as one of the three events occurs:

1. a node in $\mathcal{S}^*$ is touched. $\mathcal{S}^*$ is a set maintaining *some nodes* on a spine, $\mathcal{S}$, anchored in the final node v: hence, this event witnesses the presence of a bad cycle and the computation is stopped (line (6));
2. no final node is detected in $FW(v)$. In this case, it is unnecessary to perform SCC decomposition of $FW(v)$ and this set of states is discarded (lines (13)–(14)).
3. a final node $u \in F$ is discovered. In this case, a stack of set of nodes, `stackFW`, is used to take trace of the partial forward-set computed and the discovering restarts from u. Moreover, the spine-set is augmented with u and a skeleton of the partial forward-set computed (lines (7)–(12)).

```
SymNestedSearch($V, E, \langle \mathcal{S}^*, N^* \rangle, F,$ stackFW, guard)

– Initialize the forward-set of a final node –
(1)  if $(N^* \neq \emptyset)$ then $L \leftarrow N^*$ else $L \leftarrow$ pick$(F \cap V)$;
(2)  $FW \leftarrow L$;  if guard then $\mathcal{S}^* \leftarrow \mathcal{S}^* \setminus ($post$(N^*)$‘$\cap N^*)$

– Compute the forward-set of a final node. Stop if you –
– either detect a cycle or another final node –
(3)  Repeat
(4)    $L \leftarrow$ post$(L) \setminus FW; FW \leftarrow L \cup FW$;
(5)  until $(L = \emptyset \vee L \cap \mathcal{S}^* \neq \emptyset \vee L \cap F \neq \emptyset)$

– Case 1: a cycle containing a final node has been detected –
(6)  if $(L \cap \mathcal{S}^* \neq \emptyset)$ then return "language NOT EMPTY"

– Case 2: computation restarts from a new final node –
(7)  if $(L \cap F \neq \emptyset)$ then
(8)    set $N^*$ to be a singleton in $F \cap L; \mathcal{S}^* \leftarrow \mathcal{S}^* \cup N^*$;
(9)    if (guard) then
(10)       add to $\mathcal{S}^*$ the nodes on a skeleton of $FW$, ending at $N^*$
(11)    guard $\leftarrow true$; push(stackFW, $FW$)
(12)    SymNestedSearch($V, E, \langle \mathcal{S}^*, N^* \rangle, F,$ stackFW, guard)

– Case 3: discard forward-set discovered, not containing any
– not trivial SCC such that $SCC \cap F \neq \emptyset$ –
(13)  $V \leftarrow V \setminus FW; \mathcal{S}^* \leftarrow \mathcal{S}^* \setminus FW; N^* \leftarrow$ pop(stackFW); guard $\leftarrow false$;
(14)  SymNestedSearch($V, E, \langle \mathcal{S}^*, N^* \rangle, F,$ stackFW, guard)
```

Figure 8.1: The symbolic algorithm SymNestedSearch for Büchi language emptiness detection.

As a technical detail, we maintain boolean guard, `guard`: when `guard` is true the set N^* represents the anchor of the spine-sets partly traced in $\mathcal{S}^*$; when `guard` is false the set N^* maintains the last partial forward-set in the stack, that will be augmented in the next iteration. Moreover, `guard` prevents skeleton computation, when a partial forward-set is resorted and augmented: this is way $\mathcal{S}^*$ is a set maintaining only some nodes in a path (a spine-set); relaxing this constraint would change the complexity of SymNestedSearch. More precisely, consider Figure 8.2, where $\mathcal{S}^*$, and the partial forward-sets in `stackFW` are depicted for a given iteration of SymNestedSearch, in which $FW(v)$ is going to be built. The black nodes represented the nodes maintained in $\mathcal{S}^*$, among them, double outlined nodes denote final nodes in $\mathcal{S}^*$. By adding to $\mathcal{S}^*$ the gray nodes on dotted paths, we would obtain a spine-set,

$\langle \mathcal{S}, v\rangle$, anchored in v. The nodes on the dotted paths, represent the skeleton of a (partial) forward-set built in *more than one* iteration: the explicit construction of such a skeleton would require to analyze, again and again in each of such iterations, the whole partial built forward-set. Instead, the implicit *enlarging spine-set* $\langle \mathcal{S}, v\rangle$, is useful to analyze the complexity of our algorithm. Note that, for each iteration of SymNestedSearch$(V, E, \langle \mathcal{S}^*, N^*\rangle, \ldots)$, there always exists an enlarging spine-set $\langle \mathcal{S}, v\rangle$ maintaining all the nodes in $\mathcal{S}^*$: such a spine-set would be built by omitting the **if** statement checking `guard` in line (9) and maintains (the nodes on) a skeleton for each element of `stackFW`.

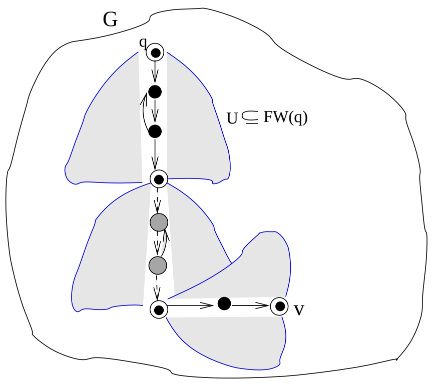


Figure 8.2: Pictorial view of spine-set used within the algorithm NestedSymSearch

In Theorem 8.1.3 we show that the global number of iterations of **repeat**-loop in lines (3) − (5) is accounted by the lengths of some disjoint enlarging spine-sets. Note that, in order to obtain the complexity of SymNestedSearch, it is only sufficient to count the global number of executions the first **repeat**-loop. In fact, the construction of a skeleton in line (10), is enabled only if the current iteration does not enlarge an already partially built forward-set, $FW(v)$: thus the cost of computing a skeleton of $FW(v)$ is equal to the cost of obtaining $FW(v)$ (in the loop). Moreover, each other line in SymNestedSearch requires a constant number of symbolic steps. In the following Lemma, fixed an iteration SymNestedSearch$(V, E, \langle \mathcal{S}^*, N^*\rangle, \ldots)$, we denote by $\langle \mathcal{S}, v\rangle$, the spine-set that would be built in place of $\langle \mathcal{S}^*, N^*\rangle$, if we would omit the **if**-statement of line (10).

Theorem 8.1.3 *The global number of **repeat**-loop performed upon each recursive call to* SymNestedSearch$(V_n, E_n, \langle \mathcal{S}^*{}_n, N^*_n\rangle, \ldots)$*, within the execution of the algorithm* SymNestedSearch$(V, E, \langle \emptyset, \emptyset\rangle, \ldots)$*, is at most* $|\mathcal{S}| + |V \setminus V^n|$.

Proof. By induction on the number of iterations. Consider the $n+1$-th recursive call, SymNestedSearch$(V_{n+1}, E_{n+1}, \langle \mathcal{S}^*{}_{n+1}, N^*_{n+1}\rangle, \ldots)$. We consider two cases. In the first case, we assume that the $n+1$-th call to the algorithm was done while executing lines (13) − (15). Hence, the n-th call to the algorithm discarded the forward set discovered, and executed exactly $|\mathcal{S}_n| \setminus |\mathcal{S}_{n+1}|$ iterations of the **repeat**-loop. The thesis follows from these considerations and from the inductive hypothesis. In the second case, we assume that the $n+1$-th call to the algorithm was done while executing lines (17) – (12). Hence, the n-th call to the algorithm discovered a new final node $u \notin \mathcal{S}^*$, while enlarging $FW(v)$. Consider a skeleton, ending at u, $\langle \mathcal{S}', u\rangle$, in the partial forward-set discovered. Assume, by contradiction, that such a skeleton contains a node, z, preceding v in the path represented by the spine-set $\langle \mathcal{S}'', v\rangle$, where $\mathcal{S}^{''} = \mathcal{S}^n$ in case `guard`=*false*, otherwise $\mathcal{S}^{''} = \mathcal{S}^n \setminus (\mathsf{post}(N^*) \cap N^*)$. Then z must not be final. Let t be the first final node following z in $\langle \mathcal{S}^{''}, v\rangle$ (t must exist, because $v \neq z$ is final). Then z allows to reach the final nodes u and t: if t can be reached from z before u, then a previous recursive call would stop (building a forward-set) encountering t, instead of u. Otherwise, the loop of the n-th recursive call would rediscover the final node t in $\mathcal{S}^*$. It follows that $\langle \mathcal{S}^{''} \cup \mathcal{S}', u\rangle$ is a spine-set: the thesis follows using inductive hypothesis and the fact that there are at most $|\mathcal{S}'|$ **repeat**-loop iterations within the n recursive call to the algorithm. ■

8.2 Fairness Constraints and Spine-Sets

In Section 6.2, we briefly discuss the importance of fairness assumptions in the context of correctness analysis of many systems. For example, when verifying a communication protocol over reliable channels only fair execution paths, in which no message happens to be continuously sent but never received, should be considered. Fairness constraints are usually classified into *strong fairness* constraints and *weak fairness* constraints [52, 51]. Weak fairness guarantees that no enabled transition is postponed forever, as in the above mentioned example. Strong fairness, instead, is used, for example, in the analysis of synchronous interactions. Strong fairness imposes that if an action is enabled infinitely often, then it will be taken infinitely often. While it is possible to express fairness constraints in LTL logic, CTL model checking consider fairness by modelling systems with a variants of Büchi automata. Namely, weak fairness constraints are introduced by means of a family of sets of states $\mathcal{F} = \langle F_1, \ldots, F_p\rangle$, that represent the acceptance condition of a generalized Büchi automaton $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, \mathcal{F} = \langle F_1, \ldots, F_k\rangle\rangle$. A run, in a generalized Büchi automaton, is accepting if and only if $\forall i = 1 \ldots k(inf(\rho) \cap F_i \neq \emptyset)$. Restricting the evaluation of CTL formulas to weak fair paths, corresponds to solve a generalized Büchi emptiness problem. In turn, non-emptiness language of a generalized Büchi automaton can be characterized as the presence of a reachable cycle of states, containing at least one element for each $F_i \in \mathcal{F}$. Symbolic CTL model checkers as VIS and SMV [94, 78], use, again, the Emerson and Lie [42] procedure to solve the above problem within $\mathcal{O}(V^2)$ symbolic steps. Assuming that the number of fairness constraints is a constant with respect to the automaton states space, it is possible adapt the algorithm Symbolic-SCC, introduced in Chapter 2, to the task of checking Büchi language emptiness in a linear number of symbolic steps. More precisely, it is only necessary to substitute line (7) of the Symbolic-SCC procedure with the following group of instructions:

```
(7'')  if (for all $F_i \in \mathcal{F}$ ($F_i \cap SCC \neq \emptyset$)) $\wedge$(post($SCC$) $\cap$ $SCC \neq \emptyset$)
           then return "language not empty"
```

With our assumptions on the number of fairness constraints, line (7′′) performs a constant number of symbolic steps and does not change the complexity of SYMBOLIC-SCC.

Passing from weak to strong fairness constraints, in CTL model checking, corresponds to solve a language emptiness problem for Streett automata. A Streett automaton differs from a (generalized) Büchi one only in the acceptance condition $\mathcal{F} = \langle (P_1, Q_1), \dots, (P_n, Q_n) \rangle$, constituted of a family of couples of vertices sets. A run of a Streett automaton, ρ, is *accepting* if for each pair of subsets in the acceptance condition, (P_i, Q_i), it holds $inf(\rho) \cap P_i \neq \emptyset \rightarrow inf(\rho) \cap Q_i \neq \emptyset$.The language of a Streett automaton, $\mathcal{A}$, is not empty if and only if $\mathcal{A}$ contains a cycle, $C = \langle s_1 \dots s_n \rangle$, such that for each pair of subsets in the acceptance condition, (P_i, Q_i), $(\{s_1, \dots, s_n\} \cap P_1 \neq \emptyset \rightarrow \{s_1, \dots, s_n\} \cap Q_1 \neq \emptyset)$.

On this ground, once determined a strongly connected component of a Streett automaton, determining if it witnesses language emptiness is a bit more tricky than in the case of Büchi automata. In fact, given a pair of subset in the acceptance condition, (P_i, Q_i), if a strongly connected component, S, does intersect only P_i it still possible that it witnesses language not emptiness: the reason is that a cycle not intersecting neither P_i nor Q_i can be contained in such strongly connected component. If present, such cycle should be searched for, recursively, in the strongly connected components of $S \setminus P_i$. The authors of [5] combined exactly the above ideas with their strategies to obtain SCC decomposition in $\mathcal{O}(V \log(V))$ symbolic steps: the result is a symbolic algorithm for Streett automata language detection performing $\mathcal{O}(V \log(V))$ symbolic steps. Adapting the overall approach to deal with our SCC symbolic algorithm allows to design the procedure SYMSA (see Figure 8.3), which solves the same problem in a linear number of symbolic steps. More precisely, the SYMSA algorithm is obtained by simply substituting line (7) of the SYMBOLIC-SCC procedure described in Chapter 2, with a call to the subprocedure depicted in Figure 8.4, that implements the recursive SCC analysis outlined above.

Assuming that the number of fairness constraints is a constant with respect to the automaton states space, algorithm SYMSA performs $\mathcal{O}(V)$ symbolic steps since:

- each line in the subroutine CHECKSCC, but line 6 requires at most a $\mathcal{O}(k)$ symbolic steps, where k is the number of fairness constraints;
- given a strongly connected component of the automaton, SCC, line 6 is executed at most $\mathcal{O}(k)$ times on (a subset) of it: in fact, each recursive call subtract at least one vertex from it (within line 5). Thus, given a strongly connected component of the automaton, SCC, line 6 requires globally $\mathcal{O}(SCC)$ symbolic steps.

```
SymSA($V, \Delta, \mathcal{F} = \langle (P_1, Q_1), \ldots, (P_k, Q_k) \rangle \rangle, \langle \mathcal{S}, N \rangle$)

(1)  if $V = \varnothing$ then return;

– Determine the node for which the scc is computed –
(2)  if $\mathcal{S} = \varnothing$ then N $\leftarrow$ $Pick$(V);

– Compute the forward-set of the the singleton N and a skeleton –
(3)  $\langle FW, New\mathcal{S}, NewN \rangle \leftarrow$ skel-forward($V, \Delta, N$);

– Determine the scc containing N –
(4)  $SCC \leftarrow N$;
(5)  while $((\mathsf{pre}(SCC) \cap FW) \setminus SCC) \neq \varnothing$ do
(6)          $SCC \leftarrow SCC \cup (\mathsf{pre}(SCC) \cap FW)$;

– Check if the scc witness that the automaton language is not empty –
(7)      CheckSCC($SCC, V, \Delta, \mathcal{F}$);

– First recursive call: computation of the scc's in $V \setminus FW$ –
(8)  $V' \leftarrow V \setminus FW$;  $\Delta' \leftarrow \Delta \restriction V'$;
(9)  $\mathcal{S}' \leftarrow \mathcal{S} \setminus SCC$;  $N' \leftarrow \mathsf{pre}(SCC \cap \mathcal{S}) \cap (\mathcal{S} \setminus SCC)$;
(10) SymBE($V', \Delta', \mathcal{F}, \langle \mathcal{S}', N' \rangle$)

– Second recursive call: computation of the sccs in $FW \setminus SCC$ –
(11) $V' \leftarrow FW \setminus SCC$;  $\Delta' \leftarrow \Delta \restriction V'$;
(12) $\mathcal{S}' \leftarrow New\mathcal{S} \setminus SCC$;  $N' \leftarrow NewN \setminus SCC$;
(13) SymBE($V', \Delta', \mathcal{F}, \langle \mathcal{S}', N' \rangle$)
```

Figure 8.3: The symbolic algorithm for checking language emptiness for a Streett automaton in $\mathcal{O}(|V|)$ symbolic steps.

```
CheckSCC($SCC, Q, \Delta, \mathcal{F} = \langle (P_1, Q_1), \ldots, (P_k, Q_k) \rangle$)

(1) if (for all$(P_i, Q_i) \in \mathcal{F}$ $(SCC \cap P_i \neq \emptyset \rightarrow SCC \cap Q_i \neq \emptyset)$)
(2)     then return "language not empty"
(3) $C \leftarrow SCC$;
(4) foreach $(P_i, Q_i) \in \mathcal{F}(P_i \cap SCC \neq \emptyset \wedge Q_i \cap SCC = \emptyset)$ do
(5)     $C \leftarrow C \setminus L_i$;
(6) if $(C \neq \emptyset)$ then SymbolicSCC$(C, \Delta \restriction C, \mathcal{F}, \langle \emptyset, \emptyset \rangle)$;
```

Figure 8.4: The subroutine CheckSCC, checking if a strongly connected component witnesses not language emptiness

Conclusions

Dealing with algorithm design and analysis is a fascinating and paying activity for reasons that are well outlined in the words of the Turing award Donald Knuth:

"People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically..."

– D. E. Knuth –

Within the research field of model checking, the "double happiness" concerning enjoying the elegancy of strong mathematical frameworks and their practical utility, is certainly in store for both algorithmists and logicians. With this dissertation, we aimed at sharing at least a bit of this pleasure: indeed, we do not claim any originality in this, and we can again recognize our objective in the following words of D.E. Knuth:

"... pleasure has probably been the main goal all along. But I hesitate to admit it, because computer scientists want to maintain their image as hard-working individuals who deserve high salaries. Sooner or later society will realise that certain kinds of hard work are in fact admirable even though they are more fun than just about anything else."

– D. E. Knuth –

More precisely, in this dissertation the above "attitude to pleasure", the state explosion problem in model checking, the algorithmic task challenging, and the mathematical modeling issue, meet in the coarse research themes below:

1. *Symbolic Graph Algorithms*, concerning the definition and the analysis of graph algorithms assuming a succinct representation of graphs by means of OBDDs;

2. *Equivalence Graph Reductions*, concerning the study of algorithmic, as well as algebraic properties of (temporal logics strong preserving) graph reductions, as bisimulation and simulation.

We clearly focus our attention only on some specific problems within the above general themes, leaving open a number of questions that deserve, in our opinion, further investigation.

Looking Back and Ahead this Thesis

As far as symbolic graph algorithms is concerned, our symbolic procedures for strong connectivity and biconnectivity, as well as the spine-set symbolic algorithmic strategy defined, can be viewed as a first attempt in the direction of defining general techniques, for interfacing and translating symbolic and explicit algorithms. To this purpose, we address the following questions, that still remains on the way:

- A better assessment of theoretic tools to compare relative merits of symbolic procedures. These tools could have the form of new parameters to be considered in comparing symbolic algorithms or that of a classification of graph topologies, supporting specific characteristics of graph algorithms: for example, it could be interesting to know why, experimentally, in model checking computing SCCs on the ground of transitive closure is considered infeasible because of OBDDs explosion, and for which graphs transitive closure (and iterative squaring) is a speed up, rather than a bottle neck. A first attempt in this direction have been done in [116], were it is recognized that very regular graphs as grids, well support symbolic transitive closure computation.

- A great deal of experimental work has been done, with symbolic algorithms, in the framework of model checking. However graph examples from other fields were space parameter is a primary one (WWW analysis, bioinformatics, mobile communication modelling, social networks ...), have not be considered in experimenting symbolic graph algorithms. This could be a good starting point to have a feeling of the impact of symbolic graph algoritms beyond model checking.

- The relationship between symbolic and explicit algorithms, and the merits of symbolic solutions should be further analyzed, in our opinion. It would be nice, for example, to devise classes of graph problems whose algorithmic solution on the one hand has the same worst case complexity, both in the explicit and in the symbolic case, and, on the other hand, it is time/space sublinear in many cases within the symbolic framework.

Another interesting way to proceed on, in our opinion, is that of considering OBDD-like structures integrating quantitative information. In the literature, for example, there exist variants of OBDDs representing multivalued functions, rather than boolean ones; further, there exist OBDD-like data structures for representing a first-order Boolean logic over inequalities of the form $x - y < d$ and $x - y \leq d$ (Difference Decision Diagrams [82]). However, these representations are either not canonical or imposes strict constraints on the range of represented functions. Reasoning on such data structures, from an algorithmic point of view, could have applications in modeling and reasoning on weight graphs and, consequently, in network analysis, in model checking with stochastic parameters, or in bioinformatica modeling.

A final consideration, closer to the model checking application field, concerns the the need of integrating our SCC analysis algorithm within existing model checking tools, in order to assess the practical effects of our procedure.

As far as ongoing work and open questions concerning the second part of this dissertation, dealing with graph equivalence based reductions, we pose the problem of extending our work to the notion of *fair simulation*. Fair simulation notions have been recently compared and analyzed, within the model checking framework, in [17, 6, 61, 44]. We redirect to the future, finally, new breakthrough to make cleaner and more elegant our simulation algorithm, as well as algorithms defined throughout this work. The importance that we deserve to the “outword form” and to the intuitiveness of algorithms is well presented in the word of R. Tarjan, that we use to conclude this thesis work.

“. . . if you come up with a clever algorithm that is clean and elegant you have a much better chance of people using it. ”

– R. Tarjan –

A

Some Technical Proofs

We collect in this appendix a number of technical proofs of the results stated in Chapter 5.

Proof of Theorems 5.1.8 and 5.1.10

In order to prove Theorem 5.1.8 we introduce the notion of generalized unlabelling. The generalized unlabelling of $G = (V, E)$ and $\langle \Sigma, I \rangle$, where I is the identity relation over Σ, is a labelled graph, $G = (V', E', \Sigma')$, where all the nodes of V belong to the same block of Σ'. The nodes in $V' \setminus V$ have new labels and they are used to keep trace of the blocks of Σ. When the partition is based on a relation P different from I, the nodes in $V' \setminus V$ keep trace also of P.

Definition A.0.1 (Generalized unlabelling) *Let $G = (V, E)$ and let $\langle \Sigma, P \rangle$ be a partition pair over G. Consider the labelled graph $G' = (V', E', \Sigma')$ defined as*[1]*:*

$$\begin{array}{rcl} L & = & \{\ell_\alpha \mid \alpha \in \Sigma\} \\ V' & = & V \uplus L \\ E' & = & E \cup \{\langle a, \ell_\alpha \rangle \mid a \in \beta \in \Sigma \wedge (\alpha, \beta) \in P^+\} \\ \Sigma' & = & \{V\} \cup \{\{\ell_\alpha\} \mid \ell_\alpha \in L\}. \end{array}$$

We call the labelled graph G' the generalized unlabelling *of G and $\langle \Sigma, P \rangle$.*

The idea is to prove that there is a correspondence between the simulations on G' and the stable refinements of $\langle \Sigma, P^+ \rangle$. We show the correspondence we have in mind on the following example.

Example A.0.2 Consider the graph G in Figure A.1 together with the partition pair $\langle \Sigma, I \rangle$, where $\Sigma = \{\alpha, \beta\}$ and I is the identity relation.

The partition pair $\langle \Sigma, I \rangle$ is not stable. For instance, $\beta E_\exists \alpha$ and there is not a class δ greater than or equal to α such that $\beta E_\forall \delta$.

A stable partition pair which refines $\langle \Sigma, I \rangle$ is $\langle \Pi, P \rangle$, where $\Pi = \{\alpha_1, \alpha_2, \beta_1, \beta_2\}$ is shown in Figure A.2, while $P = \{(\beta_1, \beta_2)\} \cup I$.

The generalized unlabelling G' of G and $\langle \Sigma, I \rangle$ is shown in Figure A.3.

[1]We use $\uplus$ to denote the disjoint union.

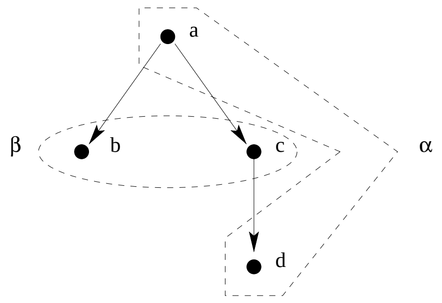


Figure A.1: The graph G and the partition Σ.

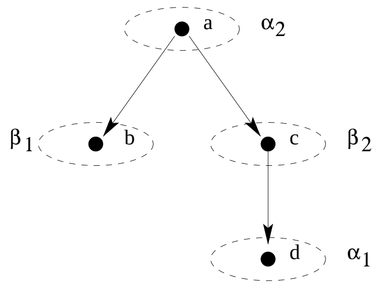


Figure A.2: The graph G and the partition Π refining Σ.

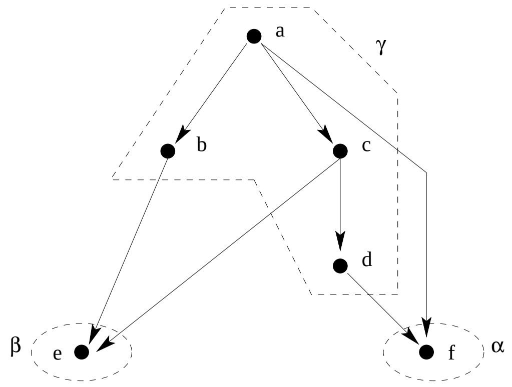


Figure A.3: The generalized unlabelling G'.

The stable refinement $\langle \Pi, P\rangle$ on G corresponds to the simulation $S = \{(b,c)\} \cup I$ on G'. ■

Now we prove the strong connection between stable refinements of $\langle \Sigma, P\rangle$ and simulations over G'.

Definition A.0.3 *Let $G = (V,E)$ be a graph, $\langle \Sigma, P\rangle$ a partition pair over G, and G' the generalized unlabelling of G and $\langle \Sigma, P\rangle$. Consider the set S_{GP} of the stable (with respect to E) refinements of $\langle \Sigma, P\rangle$ and the set S_{sim} of the simulations over G'. We define the two functions $f_1 : S_{GP} \mapsto S_{sim}$ and $f_2 : S_{sim} \mapsto S_{GP}$ as follows*[2]*:*

$$\begin{gathered} f_1(\langle \Pi, Q\rangle) = \leq_{\langle \Pi, Q\rangle} \\ \forall a, b \in V(a \leq_{\langle \Pi, Q\rangle} b \textit{ iff } a \in \alpha \in \Pi \wedge b \in \beta \in \Pi \wedge (\alpha, \beta) \in Q) \\ \forall \ell_\alpha \in L(\ell_\alpha \leq_{\langle \Pi, Q\rangle} \ell_\alpha) \end{gathered}$$

$$\begin{gathered} f_2(\leq) = \langle \Pi_s, Q_s\rangle \\ \forall a \in V([a] = \{b \mid a \leq^* b \leq^* a\} \in \Pi_s) \\ (\alpha, \beta) \in Q_s \textit{ iff } \exists a \in \alpha \exists b \in \beta (a \leq^* b) \end{gathered}$$

Notice that the condition $\exists a \in \alpha \exists b \in \beta(a \leq^* b)$ is equivalent to $\forall a \in \alpha \forall b \in \beta(a \leq^* b)$, since we are using the (reflexive and) transitive closure of the simulation $\leq$. Notice also that it is necessary to prove that the codomains of f_1 and f_2 are correctly defined. We prove this fact and some properties of f_1 and f_2 in the following Lemma.

Lemma A.0.4 *The function f_1 and f_2 are such that:*

1. *if $\langle \Pi, Q\rangle \in S_{GP}$, then $f_1(\langle \Pi, Q\rangle) \in S_{sim}$;*
2. *if $\leq \in S_{sim}$, then $f_2(\leq) \in S_{GP}$;*
3. *if $\langle \Pi_1, Q_1\rangle \sqsubseteq \langle \Pi_2, Q_2\rangle$, then $f_1(\langle \Pi_1, Q_1\rangle) \subseteq f_1(\langle \Pi_2, Q_2\rangle)$;*
4. *if $\leq^1 \subseteq \leq^2$, then $f_2(\leq^1)) \sqsubseteq f_2(\leq^2)$;*
5. *$\leq \subseteq f_1(f_2(\leq))$;*
6. *$\langle \Pi, Q\rangle \sqsubseteq f_2(f_1(\langle \Pi, Q\rangle))$.*

Proof. (1). We prove that if $\langle \Pi, Q\rangle$ belongs to S_{GP}, then $\leq_{\langle \Pi, Q\rangle}$ is a simulation over G'.
If $a \leq_{\langle \Pi, Q\rangle} b$, then only two cases are possible:

- a and b are elements of V;
- $a = b$ and a is an element of L.

[2] $\leq^*$ is the reflexive and transitive closure of $\leq$.

Hence in both cases we have that they belongs to the same class in Σ'.
The second case is trivial. In the first case if c is such that $aE'c$, then it can either be $c \in L$ or $c \in V$.
If $c \in L$, from $aE'c$ we have that $a \in \alpha \in \Sigma$, $c = \ell_\gamma$ and $(\gamma, \alpha) \in P^+$. From the fact that $a \leq_{\langle \Pi, Q \rangle} b$ we have that $a \in \alpha' \in \Pi$ and $b \in \beta' \in \Pi$ and $(\alpha', \beta') \in Q$. Hence we have that $b \in \beta \in \Sigma$ and $(\alpha, \beta) \in P^+$. From $(\gamma, \alpha) \in P^+$ and $(\alpha, \beta) \in P^+$ we obtain $(\gamma, \beta) \in P^+$, hence $c \leq_{\langle \Pi, Q \rangle} c$ and $bE'c$.
If $c \in V$, then we have that $a \in \alpha' \in \Pi$, $c \in \gamma' \in \Pi$, $b \in \beta' \in \Pi$, $\alpha' E_\exists \gamma'$, and $(\alpha', \beta') \in Q$ (since $a \leq_{\langle \Pi, Q \rangle} b$). Hence, since $\langle \Pi, Q \rangle$ is stable, there exists $\delta' \in \Pi$ such that $(\gamma', \delta') \in Q$ and $\beta' E_\forall \delta'$. Let $d \in \delta'$ be such that bEd. Since $c \leq_{\langle \Pi, Q \rangle} d$, the thesis follows.

It is immediate to prove that if the relation $\leq$ is a simulation over G', then $\langle \Pi_s, Q_s \rangle$ is a partition pair, i.e., Q_s is reflexive and acyclic.

(2). We prove that if $\leq$ is a simulation over G', then $\langle \Pi_s, Q_s \rangle$ is a stable refinement of $\langle \Sigma, P \rangle$.
Observe that Q_s is acyclic.
Let α', β', γ' in Π_s be such that $\alpha' E_\exists \gamma'$ and $(\alpha', \beta') \in Q_s$. We have to prove that there exists $\delta' \in \Pi_s$ such that $(\gamma', \delta') \in Q_s$ and $\beta' E_\forall \delta'$.
From $\alpha' E_\exists \gamma'$ we obtain that there exist $a \in \alpha'$ and $c \in \gamma'$ such that $aE'c$. From $(\alpha', \beta') \in Q_s$ we obtain that $a \leq^* b$ for all $a \in \alpha'$ and $b \in \beta'$. Hence, since $\leq^*$ is a simulation, we have that for all $b \in \beta'$ there exists d_b such that $bE'd_b$ and $c \leq^* d_b$ (hence $d_b \in V$). Let $b_1, \ldots, b_t$ be an ordering of the elements of β'.

We inductively define as follows the lists $L^1, .., L^n, ..$ of elements related through E' with $b_1, \ldots, b_t$:

- $L^1 = d_1^1, \ldots, d_t^1$
  where d_1^1 is such that $b_1 E' d_1^1$ and $c \leq^* d_1^1$ and, for $j = 2, \ldots, t$, d_j^1 is such that $b_j E' d_j^1$ and $d_{j-1}^1 \leq^* d_j^1$. Notice that L^1 is correctly defined (i.e $d_1^1, \ldots, d_t^1$ always exist) since $\leq^*$ is a simulation, $a \leq^* b \wedge aE'c$ and, for $j = 2, \ldots, t$, $b_{j-1} \leq^* b_j$.
- $L^{h+1} = d_1^{h+1}, \ldots, d_t^{h+1}$
  where d_1^{h+1} is such that $b_1 E' d_1^{h+1}$ and $d_t^h \leq^* d_1^{h+1}$ and, for $j = 2, \ldots, t$, d_j^{h+1} is such that $b_j E' d_j^{h+1}$ and $d_{j-1}^{h+1} \leq^* d_j^{h+1}$.
  L^{h+1} is correctly defined (i.e $d_1^{h+1}, \ldots, d_t^{h+1}$ always exist) since $b_1, \ldots, b_t$ is an ordering of the elements of the class β' and $\leq^*$ is a simulation.

By concatenation of the lists $L^1, L^2, \ldots$ we obtain an infinite chain of elements $d_1^1, \ldots, d_t^1, d_1^2, \ldots, d_t^2, .., d_1^n, \ldots, d_t^n, ..$ such that

1. for all $k > 0$ and for $j = 1, \ldots, t$ $b_j^k E' d_j^k$
2. for all $k > 0$ and for $j = 2, \ldots, t$, $d_{j-1}^k \leq^* d_j^k$
3. $c \leq^* d_1^1$ and for all $k > 0$ $d_t^{k-1} \leq^* d_1^k$

Since the set of the successors of b_1 is a finite set we have that there exist v, u such that $d_1^v = d_1^u = d_1^*$ and $v < u$. Hence we obtain a prefix of the chain of the form $d_1^1 \ldots \leq^* d_1^* \leq^* \ldots \leq^* d_1^*$ with $c \leq^* d_1^1$. This means that all the elements of this succession between the two occurrences of d_1^* belongs to the same class $\delta' \in \Pi$. Moreover all the elements β' $(b_1, \ldots, b_t)$ have a successor in δ' and obviously $(\gamma', \delta') \in Q_s$, i.e., the thesis.

(3). Let $\leq_1 = f_1(\langle \Pi_1, Q_1 \rangle)$ and $\leq_2 = f_1(\langle \Pi_2, Q_2 \rangle)$. We assume that $\langle \Pi_1, Q_1 \rangle \sqsubseteq \langle \Pi_2, Q_2 \rangle$ and we have to prove that $\leq_1 \subseteq \leq_2$.
If $a \leq_1 b$, then $a \in \alpha_1 \in \Pi_1$, $b \in \beta_1 \in \Pi_1$ and $(\alpha_1, \beta_1) \in Q_1$, hence $a \in \alpha_2 \in \Pi_2$, $b \in \beta_2 \in \Pi_2$ and $(\alpha_2, \beta_2) \in Q_2$, from which $a \leq_1 b$.

(4). Let $\langle \Pi_1, Q_1 \rangle = f_2(\leq_1)$ and $\langle \Pi_2, Q_2 \rangle = f_2(\leq_2)$. We assume that $\leq_1 \subseteq \leq_2$ and we prove that $\langle \Pi_1, Q_1 \rangle \sqsubseteq \langle \Pi_2, Q_2 \rangle$.
From $\leq_1 \subseteq \leq_2$ we have $\leq_1^* \subseteq \leq_2^*$.
If $a, b \in \alpha_1 \in \Pi_1$, then $a \leq_1^* b \leq_1^* a$, hence $a \leq_2^* b \leq_2^* a$, from which $a, b \in \alpha_2 \in \Pi_2$, i.e., Π_1 is finer than Π_2.
If $(\alpha_1, \beta_1) \in Q_1$, then since Π_1 is finer than Π_2 there exists $\alpha_2, \beta_2 \in \Pi_2$ such that $\alpha_1 \subseteq \alpha_2$ and $\beta_1 \subseteq \beta_2$. We have to prove that $(\alpha_2, \beta_2) \in Q_2$. Let $a_1 \in \alpha_1$ and $b_1 \in \beta_1$ be such that $a_1 \leq_1^* b_1$. It holds that $a_1 \leq_2^* b_1$, $a_1 \in \alpha_2$ and $b_1 \in \beta_2$, hence $(\alpha_2, \beta_2) \in Q_2$.

(5). If $a \leq b$, then $a \in \alpha \in \Pi_s$, $b \in \beta \in \Pi_s$ and $(\alpha, \beta) \in Q_s$. Hence we obtain $a \leq_{\langle \Pi_s, Q_s \rangle} b$.

(6). If $a, b \in \alpha \in \Pi$, then, since Q is reflexive, it holds $a \leq_{\langle \Pi, Q \rangle} b \leq_{\langle \Pi, Q \rangle} a$, hence $a, b \in \alpha' \in \Pi_{\langle \Pi, Q \rangle}$, i.e., Π is finer than $\Pi_{\langle \Pi, Q \rangle}$. Let $(\alpha, \beta) \in Q$, consider $\alpha', \beta' \in \Pi_{\langle \Pi, Q \rangle}$ such that $\alpha \subseteq \alpha'$ and $\beta \subseteq \beta'$. Let $a_1 \in \alpha$ and $b_1 \in \beta$. It holds that $a_1 \leq_{\langle \Pi, Q \rangle} b_1$, $a_1 \in \alpha'$, and $b_1 \in \beta'$, hence $(\alpha', \beta') \in Q_{\langle \Pi, Q \rangle}$. ■

We are now ready to prove the existence and uniqueness of the solution of a GCPP.

Theorem 5.1.8 Given $G = (V, E)$ and a partition pair $\langle \Sigma, P \rangle$ over G, the GCPP over G and $\langle \Sigma, P \rangle$ has always a unique solution.

Proof. Let $\leq_s$ be the maximal simulation over G'. From Lemma A.0.4(2) we have that $\langle \Pi_m, Q_m \rangle = f_2(\leq_s)$ is a stable refinement of $\langle \Sigma, P \rangle$. We prove that if $\langle \Pi, Q \rangle$ is another stable refinement, then $\langle \Pi, Q \rangle \sqsubseteq \langle \Pi_m, Q_m \rangle$. From Lemma A.0.4(6) we know that $\langle \Pi, Q \rangle \sqsubseteq f_2(f_1(\langle \Pi, Q \rangle))$. Since $f_1(\langle \Pi, Q \rangle)$ is a simulation over G' we also know that $f_1(\langle \Pi, Q \rangle) \subseteq \leq_s^m$. Hence, using Lemma A.0.4(4) we obtain that $\langle \Pi, Q \rangle \sqsubseteq f_2(f_1(\langle \Pi, Q \rangle)) \sqsubseteq f_2(\leq_s^m) = \langle \Pi_m, Q_m \rangle$. As for the uniqueness, assume $\langle \Pi', Q' \rangle \neq \langle \Pi_m, Q_m \rangle$ is another solution. Then, $\langle \Pi_m, Q_m \rangle \sqsubseteq \langle \Pi', Q' \rangle$. Using Definition A.0.3 of f_1 we immediately deduce $f_1(\langle \Pi', Q' \rangle) \neq f_1(\langle \Pi_m, Q_m \rangle)$. By Lemma A.0.4(3) and A.0.4(5) we then obtain $\leq_s = f_1(f_2(\leq_s)) \subset f_1(\langle \Pi', Q' \rangle)$ which is a contradiction. ■

Corollary 5.1.9 If $\langle S, \preceq \rangle$ is the solution of the GCPP over G and $\langle \Sigma, P \rangle$, then $\preceq$ is a partial order over S.

Proof. We have to prove that $\preceq$ is transitive. This is an immediate consequence of the fact that $\langle S, \preceq \rangle = f_2(\leq_s)$, where $\leq_s$ is the maximal simulation over G' (generalized unlabelling). ■

Theorem 5.1.10 [Simulation as GCPP] Let $G = (V, E, \Sigma)$ and let $\langle S, \preceq \rangle$ be the solution of the GCPP over $G^- = (V, E)$ and $\langle \Sigma, I \rangle$, where I is the identity relation. S is the simulation quotient of G, i.e., $S = V/\equiv_s$, and $\leq_s$ defined as

$$a \leq_s b \quad \text{iff} \quad [a]_s \preceq [b]_s$$

is the maximal simulation over G.

Proof. Let G' be the generalized unlabelling of G^- and $\langle \Sigma, I \rangle$. The restriction of the maximum simulation over G' to the nodes of G is the maximum simulation over G. Hence, from Lemma A.0.4 we immediately obtain the thesis ■

proof of Theorem 5.1.14

In order to prove Theorem 5.1.14 we introduce the following lemmas.

Lemma A.0.5 *Let $\langle S, \preceq \rangle$ be the solution of the GCPP over G and $\langle \Sigma, P \rangle$. If $\langle S, \preceq \rangle \sqsubseteq \langle \Pi, Q \rangle$, then $\langle S, \preceq \rangle \sqsubseteq \sigma(\langle \Pi, Q \rangle)$.*

Proof. From Corollary 5.1.9 we have that $\preceq$ is transitive.
We have to prove that if $\langle S, \preceq \rangle \sqsubseteq \langle \Pi, Q \rangle$, then $\langle S, \preceq \rangle \sqsubseteq \sigma(\langle \Pi, Q \rangle) = \langle \Pi', Q' \rangle$ i.e that

1. S is a partition finer than Π'
2. The partial order (on S) $\preceq$ is included in the relation induced on S by Q' (which is an acyclic and reflexive relation on Π')

1. If we prove that $\langle S, \preceq \rangle$ satisfies $(1\sigma)(a)$ (over $\langle \Pi, Q \rangle$) we can then conclude, by theorem 5.1.13, that S is comparable with with Π' and finer then Π' i.e our thesis. So let's start proving that $\langle S, \preceq \rangle$ satisfies $(1\sigma)(a)$ (over $\langle \Pi, Q \rangle$).
   If $\beta \in S$ and $\alpha \in \Pi$ are such that $\beta E_\exists \alpha$, then there exists $\alpha' \in S$ such that $\alpha' \subseteq \alpha$ and $\beta E_\exists \alpha'$. Since $\langle S, \preceq \rangle$ is the solution of a GCPP it holds that there exists $\delta' \in S$ such that $(\alpha', \delta') \in \preceq$ and $\beta E_\forall \delta'$. Hence since $\preceq \subseteq Q(S)$ it holds that $\delta' \subseteq \delta \in \Pi$, $(\alpha, \delta) \in Q$, and $\beta E_\forall \delta$, i.e., (1σ) is satisfied.

2. we have to prove that the partial order (on S) $\preceq$ is included in the relation induced on S by Q', where $\langle \Pi', Q' \rangle = \sigma(\langle \Pi, Q \rangle)$. Notice that as a by-product of 1. we have that the partition S is finer than Π' which is finer than Π.
   We will use the notation α_P, β_P.. to denote the elements of a partition P.
   Suppose by contradiction that there exists $\overline{\alpha}_S, \overline{\beta}_S, \overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'}$ such that $\overline{\alpha}_S \subseteq \overline{\alpha}_{\Pi'}, \overline{\beta}_S \subseteq \overline{\beta}_{\Pi'}$ and $(\overline{\alpha}_S \preceq \overline{\beta}_S \wedge (\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'}) \notin Q')$. Without losing generality we can assume that the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ is the first one which is assumed not to bee in Q' and is such that there exist $\overline{\alpha}_S \subseteq \overline{\alpha}_{\Pi'}, \overline{\beta}_S \subseteq \overline{\beta}_{\Pi'}$ with $\overline{\alpha}_S \preceq \overline{\beta}_S$. We have to consider 3 cases:
   (a) the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ is not in the relation induced on Π' by Q;

(b) the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ has been removed from the relation induced on Π' by Q in order to satisfy condition (b) in (2σ);

(c) the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ has been removed from Q' in order to satisfy condition (c) in (2σ).

(a) Consider the classes of Π, $\overline{\alpha}_{\Pi}$ and $\overline{\beta}_{\Pi}$ such that $\overline{\alpha}_{\Pi} \supseteq \overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi} \supseteq \overline{\beta}_{\Pi'}$; we immediately deduce that $\overline{\alpha}_S \supseteq \overline{\alpha}_{\Pi}, \overline{\beta}_S \supseteq \overline{\beta}_{\Pi}$ and $(\overline{\alpha}_S \preceq \overline{\beta}_S \ \wedge\ (\overline{\alpha}_{\Pi}, \overline{\beta}_{\Pi}) \notin Q,)$ which contradicts our assumption that $\langle S, \preceq\rangle \sqsubseteq \langle \Pi, Q\rangle$.

(b) Recall that since $(\langle S, \preceq\rangle)$ is the solution of the GCPP it holds that $(\langle S, \preceq\rangle)$ is stable i.e

$$\forall \alpha_S, \beta_S, \gamma_S(\alpha_S \preceq \beta_S \wedge \alpha_S\, E_\exists\, \gamma_S \Rightarrow \exists \delta_S(\gamma_S \preceq \delta_S \wedge \beta_S\, E_\forall\, \delta_S)).$$

If the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ is assumed not to bee in Q' because of condition (b) in (2σ), then there exists a class of Π, γ_Π, such that:

$$\overline{\alpha}_{\Pi'}\, E_\forall\, \gamma_\Pi \ \wedge\ \neg\exists \delta_\Pi((\gamma_\Pi, \delta_\Pi) \in Q \wedge \overline{\beta}_{\Pi'}\, E_\exists\, \delta_\Pi) \tag{1}$$

From $\overline{\alpha}_{\Pi'}\, E_\forall\, \gamma_\Pi$ we obtain that that for every class of S, α_S which is included[3] in $\alpha_{\Pi'}$ there exists a class of S included in γ_Π, γ_S, such that $\alpha_S\, E_\exists\, \gamma_S$. Hence we have that there is a class of S, say γ^*_S, included in γ_Π and such that $\overline{\alpha}_S\, E_\exists\, \gamma^*_S$. From the second conjunct in assertion (1), $\gamma^*_S \subseteq \gamma_\Pi$, $\overline{\beta}_S \subseteq \overline{\beta}_{\Pi'}$ and from our assumption that $\preceq$ is included in the relation on S induced by Q we deduce that

$$\neg\exists \delta_S(\gamma^*_S \preceq \delta_S \wedge \overline{\beta}_S\, E_\exists\, \delta_S)$$

and hence that

$$\overline{\alpha}_S \preceq \overline{\beta}_S \ \wedge\ \overline{\alpha}_S\, E_\exists\, \gamma^*_S \ \wedge\ \neg\exists \delta_S(\gamma^*_S \preceq \delta_S \wedge \overline{\beta}_S\, E_\forall\, \delta_S)$$

i.e $\langle S, \preceq\rangle$ is not stable, which is a contradiction.

(c) Recall that we assumed that the pair $(\overline{\alpha}_{\Pi'}, \overline{\beta}_{\Pi'})$ is the first one which is not in Q' and is such that there exist $\overline{\alpha}_S \subseteq \overline{\alpha}_{\Pi'}, \overline{\beta}_S \subseteq \overline{\beta}_{\Pi'}$ with $\overline{\alpha}_S \preceq \overline{\beta}_S$. Exploiting this assumption and using the same reasoning schema used in (b) we deduce the absurd that $\langle S, \preceq\rangle$ is not stable.

■

Lemma A.0.6 *Let $G = (V, E)$ be a graph and $\langle \Pi, Q\rangle$ be a partition pair over G. Let $\langle S, \preceq\rangle$ be the solution of the GCPP over G and $\langle \Pi, Q\rangle$. If $\sigma(\langle \Pi, Q\rangle) = \langle \Pi, Q\rangle$, then $\langle \Pi, Q^+\rangle = \langle S, \preceq\rangle$.*

[3]From 1. we have that the partition S is finer than Π' which is finer then Π.

Proof. We know that, by definition, $\langle S, \preceq\rangle \sqsubseteq \langle \Pi, Q^+\rangle$. Hence we only have to prove that Q^+ is stable with respect to G. If $\alpha, \beta, \gamma \in \Pi$ are such that $(\alpha, \beta) \in Q^+$ and $\alpha E_\exists \gamma$, then from the condition in (1σ) applied to α and γ we have that there exists $\gamma_1' \in \Pi$ such that $(\gamma, \gamma_1') \in Q$ and $\alpha E_\forall \gamma_1'$. Let $\beta_1, \ldots, \beta_n$ be such that $(\alpha, \beta_1) \in Q$, $(\beta_i, \beta_{i+1}) \in Q$ and $\beta_n = \beta$. Using the second condition of (2σ) on (α, β_1) and γ_1' we obtain that there exists γ_1'' such that $(\gamma_1', \gamma_1'') \in Q$ and $\beta_1 E_\exists \gamma_1''$. Now applying (1σ) to β_1 and γ_1'' we obtain that there exists γ_1''' such that $(\gamma_1'', \gamma_1''') \in Q$ and $\beta_1 E_\forall \gamma_1'''$. Similarly by induction on n we obtain a chain of classes of Q of the form $\gamma, \gamma_1', \gamma_1'', \gamma_1''', \ldots, \gamma_n', \gamma_n'', \gamma_n'''$ such that $\beta E_\forall \gamma_n'''$. Hence $(\gamma, \gamma_n''') \in Q^+$ and $\beta E_\forall \gamma_n'''$, i.e., our thesis. ∎

The following lemma is an immediate consequence of the definition of GCPP.

Lemma A.0.7 *Let $G = (V, E)$ be a graph and $\langle \Pi, Q\rangle \sqsubseteq \langle \Sigma, P\rangle$ be two partition pairs. Let $\langle S, \preceq\rangle$ be the solution of the GCPP over G and $\langle \Sigma, P\rangle$. If $\langle S, \preceq\rangle \sqsubseteq \langle \Pi, Q^+\rangle$, then $\langle S, \preceq\rangle$ is also the solution of the GCPP over G and $\langle \Pi, Q\rangle$.*

Theorem 5.1.14 [Fix Point] Let $G = (V, E)$ and $\langle \Sigma, P\rangle$ a partition pair over G with P transitive. Let $\langle S, \preceq\rangle$ be the solution of the GCPP over G and $\langle \Sigma, P\rangle$. If n is such that $\sigma^n(\langle \Sigma, P\rangle) = \sigma^{n+1}(\langle \Sigma, P\rangle)$, then $\sigma^n(\langle \Sigma, P\rangle) = \langle S, \preceq\rangle$.

Proof. Since P is transitive we have that $\langle S, \preceq\rangle \sqsubseteq \langle \Sigma, P\rangle$. Hence applying Lemma A.0.5 we obtain that $\langle S, \preceq\rangle \sqsubseteq \sigma(\langle \Sigma, P\rangle)$. Assume that we have proved that $\langle S, \preceq\rangle \sqsubseteq \sigma^i(\langle \Sigma, P\rangle)$ applying Lemma A.0.5 we obtain that $\langle S, \preceq\rangle \sqsubseteq \sigma^{i+1}(\langle \Sigma, P\rangle)$.
Hence we have obtained that for all $i \in \mathbb{V}$ it holds that

$$\langle S, \preceq\rangle \sqsubseteq \sigma^i(\langle \Sigma, P\rangle) \sqsubseteq \langle \Sigma, P\rangle.$$

Moreover from the fact that P is transitive we obtain that

$$\sigma^i(\langle \Sigma, P\rangle)^+ \sqsubseteq \langle \Sigma, P\rangle,$$

where $\sigma^i(\langle \Sigma, P\rangle)^+$ is the transitive closure of the relation in $\sigma^i(\langle \Sigma, P\rangle)$. Hence we have that

$$\langle S, \preceq\rangle \sqsubseteq \sigma^i(\langle \Sigma, P\rangle)^+,$$

so applying Lemma A.0.7 we obtain that for all i it holds that $\langle S, \preceq\rangle$ is also the solution of the GCPP over $\sigma^i(\langle \Sigma, P\rangle)$.
Applying Lemma A.0.6 to $\sigma^n(\langle \Sigma, P\rangle)$ we obtain $\langle S, \preceq\rangle = \sigma^n(\langle \Sigma, P\rangle)^+$ and we already have that $\langle S, \preceq\rangle \sqsubseteq \sigma^n(\langle \Sigma, P\rangle)$, hence it must be $\langle S, \preceq\rangle = \sigma^n(\langle \Sigma, P\rangle)$. ∎

Proof of Theorem 5.2.6

In order to prove Theorem 5.2.6 we introduce the following operator which characterizes the computations of the function NEWHHK.

Definition A.0.8 *Let $D = (T, R_1, R_2)$ be such that $R_2 \subseteq R_1 \subseteq T \times T$, and $K \subseteq T \times T$ a relation over T. We define $\tau_D(K)$ as:*

$$\tau_D(K) = K \setminus \{(a,b) \mid \exists c(aR_2c \wedge \forall d(bR_1d \Rightarrow (c,d) \notin K))\}.$$

$Fix(\tau_D)(K)$ denotes the greatest fix point of τ_D smaller than K.

Example A.0.9 Consider the relations in Example 5.2.5. In particular, let $T = \Sigma_1$, $R_1 = E^{\Sigma}_{\exists}(\Sigma_1)$ and $R_2 = E^{\Sigma}_{\forall}(\Sigma_1)$, and $K = \texttt{Ind}_1$. In this case we obtain that $\tau_D(K) = \texttt{Ref}_1$. If we instantiate again our framework with $T = \Sigma_1$, $R_1 = E_{\exists}$, $R_2 = E_{\forall}$, and $K = \texttt{Ref}_1$ we have that $Fix(\tau_D)(K) = \texttt{P}_1$. ■

From the definition of τ and Proposition 5.2.4 we immediately get the following result.

Corollary A.0.10 *Let $G = (V, E)$ be a graph, and $\langle \Sigma, P \rangle$ be a partition pair. If $\sigma(\langle \Sigma, P \rangle) = \langle \Pi, Q \rangle$, $Q = Fix(\tau_{\Pi_{\exists\forall}})(\tau_{\Sigma_{\exists\forall}(\Pi)}(P(\Pi)))$.*

We now establish a connection between the operator τ and the function NEWHHK presented in Figure 5.6.

Lemma A.0.11 *It holds that:*

$$\begin{array}{rcl} \text{NEWHHK}(D, K, \bot) & = & \tau_D(K) \\ \text{NEWHHK}(D, K, \top) & = & Fix(\tau_D)(K). \end{array}$$

Proof. Let $P_1 = \text{NEWHHK}(D, K, \bot)$. Let $(a,b) \notin P_1$, we have to prove that $(a,b) \notin \tau_D(K)$. If $(a,b) \notin K$, then the result is trivial. If $(a,b) \in K$, from $(a,b) \notin P_1$ we have that there exists c such that $a \in pre_2(c), b \in rem(c)$ and $b \in sim(a)$. From $a \in pre_2(c)$ we have that aR_2c. From $b \in rem(c)$, since we are in the case in which $U = \bot$, we have that $b \in T \setminus pre_1(\{e \mid (c,e) \in K\})$ (notice that $rem(c)$ is never modified and when it is build the initial definition of $sim(c)$ is $\{e \mid (c,e) \in K\}$). This means that $\forall e((c,e) \in K \Rightarrow \neg bR_1e)$, i.e., $\forall d(bR_1d \Rightarrow (c,d) \notin K)$. Hence we have obtained aR_2c and $\forall d(bR_1d \Rightarrow (c,d) \notin K)$, which implies that (a,b) is not in $\tau_D(K)$. Now we assume that $(a,b) \notin \tau_D(K)$ and we prove that $(a,b) \notin P_1$. If $(a,b) \notin K$ is trivial. If $(a,b) \in K$, then from the definition of τ_D, we have that there exists c such that aR_2c and $\forall d(bR_1d \Rightarrow (c,d) \notin K)$. This implies that $a \in pre_2(c)$ and $b \in rem(c)$. From the fact that $(a,b) \in K$ we also have in the initialization it holds $b \in sim(a)$. We can assume that c is the first such that $rem(c) \neq \emptyset$ and $b \in rem(c), a \in pre_2(c)$. This means that also $b \in sim(a)$ still holds. Hence, at this step (a,b) is removed from P_1, i.e., the thesis.

Let $P_2 = \text{NEWHHK}(D, K, \top)$. Let $(a,b) \notin P_2$ we have to prove that $(a,b) \notin Fix(\tau_D)(K)$. The case in which $(a,b) \notin K$ is trivial. Let k be the number of extractions performed on $LS = \{c \mid rem(c) \neq \emptyset\}$ at the iteration in which (a,b) is removed from P_2. We proceed by induction on k. If $k = 1$, then c is the first element we extract from LS, hence $rem(c) := T \setminus pre_1(\{e \mid (c, e) \in K\})$, i.e., $rem(c) = \{b \mid \forall d(bR_1d \Rightarrow (c, d) \notin K\}$. Since we remove (a, b) from P_2 this means that aR_2c, $b \in rem(c)$ and $b \in sim(a)$, and using the fact that $rem(c) = \{b \mid \forall d(bR_1d \Rightarrow (c, d) \notin K\}$, this implies that $(a, b) \notin \tau_D(K)$, hence $(a, b) \notin Fix(\tau_D)(K)$. Let us assume we have proved the thesis in the case $k \leq h$, and (a, b) is such that it is removed from P_2 at the $(h+1)$-th extraction from LS. This means that there exists c such that aR_2c and $b \in rem(c)$. If b was in $rem(c)$ in the initialization then, as in the base case, we can prove that $(a, b) \notin \tau_D(K)$. If b has been added to $rem(c)$ after the initialization, then there exists $h' \leq h$ such that at the h'-th extraction from LS we have the following situation: a pair (c, d) has been removed from P_2, $b \in pre_1(d)$ and $post_1(b) \cap sim(c) = \emptyset$. It is immediate to prove that the following invariant holds:

$$\forall c \in T(sim(c) = \{e \mid (c, e) \in P\}).$$

Hence, we have that the situation at the h'-th can be rewritten as (c, d) has been removed from P_2, $b \in pre_1(d)$ and $\forall d'(bR_1d' \Rightarrow (c, d') \notin P_2)$. By inductive hypothesis we have that $(c, d) \notin Fix(\tau_D)(K)$ and the same for each d' such that bR_1d', i.e., there exists i such that $(c, d) \notin \tau_D^i(K)$ and $\forall d'(bR_1d' \Rightarrow (c, d') \notin \tau_D^i(K))$. We have obtained that aR_2c and $\forall d'(bR_1d' \Rightarrow (c, d') \notin \tau_D^i(K))$, hence $(a, b) \notin \tau_D^{i+1}(K)$. We now prove that if $(a, b) \notin Fix(\tau_D(K))$, then $(a, b) \notin P_2$. If $(a, b) \notin K$, then is trivial. Let k be the number such that $(a, b) \in \tau_D^{k-1}(K)$ and $(a, b) \notin \tau_D^k(K)$. We proceed by induction on k and we prove that $(a, b) \notin \tau_D^k(K)$ implies $(a, b) \notin P_2$. If $k = 1$, then this is equivalent to say that there exists c such that aR_2c and $\forall d(bR_1d \Rightarrow (c, d) \notin K)$. This implies that initially $b \in rem(c)$. Since we are assuming that $(a, b) \in K$ we also have that initially $b \in sim(a)$. Hence, when c is extracted for the first time from LS (since T is finite we are sure that if $sim(c) \neq \emptyset$, then there exists an iteration in which c is extracted from LS), if (a, b) has not yet been removed from P_2 we have that aR_2c, $b \in rem(c)$, and $b \in sim(a)$. Hence, at this point (a, b) is removed from P_2. Let us assume we have proved the thesis in the case $k \leq h$ and we prove the thesis for $k + 1$. Our hypothesis is that $(a, b) \in \tau_D^k(K)$ and $(a, b) \notin \tau_D^{k+1}(K)$. This means that there exists c such that aR_2c and $\forall d(bR_1d \Rightarrow (c, d) \notin \tau_D^k(K))$. Applying the inductive hypothesis we obtain that $\forall d(bR_1d \Rightarrow (c, d) \notin P_2)$. When we remove from P_2 the last pair (c, d') such that bR_1d' we have that $b \in prec_1(d')$ and $post_1(b) \cap sim(c) = \emptyset$. This implies that b is added to $rem(c)$. Let us consider the first iteration after the one in which we add b to $rem(c)$ in which c is extracted from LS. We have that aR_2c, $b \in rem(c)$, and, assuming that (a, b) has not yet been removed from P_2, $b \in sim(a)$, hence at this point (a, b) is removed from P_2. ■

Theorem 5.2.6 The following holds:

$$\langle \Sigma_{i+1}, \mathtt{P}_{i+1} \rangle = \sigma(\langle \Sigma_i, \mathtt{P}_i \rangle).$$

Proof. This is an immediate consequence of Corollary 5.2.3, of Corollary A.0.10, and of Lemma A.0.11. ■

List of Figures

List of Tables

Bibliography

[1] P. Aczel. *Non-well-founded sets*, volume 14 of *CSLI Lecture Notes*. Stanford University Press, 1988.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[3] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[4] J. van Benthem. *Modal Correspondence Theory*. PhD thesis, Universiteit van Amsterdam, Instituut voor Logica en Grondslagenonderzoek van Exacte Wetenschappen, 1976.

[5] R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 37–54. Springer, 2000.

[6] R.P. Bloem. *Search Techniques and Automata for Symbolic Model Checking*. Cs, University of Colorado, 2001.

[7] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-like Languages*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 1989.

[8] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, June 1995.

[9] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[10] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In E. Clarke and R. Kurshan, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 197–203. Springer, 1990.

[11] A. Bouali. XEVE, an ESTEREL verification environment. In A. J. Hu and M. Y. Vardi, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 500–504. Springer, 1998.

[12] A. Bouali and R. de Simone. Symbolic bisimulation minimization. In G. von Bochmann and D. K. Probst, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'92)*, volume 663 of *LNCS*, pages 96–108. Springer, 1992.

[13] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, S. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *AThe International Journal of Computer and Telecommunications Networking*, 33(1-6):309–320, 2000.

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

[15] R.E. Bryant. Symbolic boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[16] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[17] D. Bustan. *Equivalence-Based Reductions and Checking for Preorders.* PhD thesis, Technion Haifa, 2002.

[18] D. Bustan and O. Grumberg. Simulation based minimization. In D.A. McAllester, editor, *Proc. of Int. Conference on Automated Deduction (CADE'00)*, volume 1831 of *LNCS*, pages 255–270. Springer, 2000.

[19] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv short overview. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.

[20] A. Cimatti, E. Giunchiglia, M. Roveri, M. Pistore, R. Sebastiani, and A. Tacchella. Integrating bdd-based and sat-based symbolic model checking. In *Proceeding of 4th International Workshop on Frontiers of Combining Systems (FroCoS'2002)*, volume 2309 of *LNCS*, pages 49–56. Springer, 2002.

[21] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[22] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Proceeding of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983.

[23] E.M. Clarke, E.A. Emerson, S.Jha, and A.P. Sistla. Symmetry reductions in model checking. In *Proceeding of the 10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998.

[24] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proceeding of the 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.

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

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

[27] E.M. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceeding of the 1Fourth Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Press, 1989.

[28] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[29] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In R. Alur and T. A. Henzinger, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 394–397. Springer, 1996.

[30] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In K.G. Larsen and A. Skou, editors, *Proc of Int. Conference on Computer Aided Verification (CAV'91)*, volume 575 of *LNCS*, pages 48–58. Springer, 1992.

[31] R. Cleaveland and L. Tan. Simulation revised. In T. Margaria and W. Yi, editors, *Proc. of Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 480–495. Springer, 2001.

[32] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[33] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, 1996.

[34] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In C. Courcoubetis, editor, *Proc. of Int. Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 479–490. Springer, 1993.

[35] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[36] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages Systems*, 19(2):253–291, 1997.

[37] A. Dovier, R. Gentilini, C. Piazza, and A. Policriti. Rank-based symbolic bisimulation (and model checking). In Ruy J. Guerra B. de Queiroz, editor, *Proc. of Workshop on Language, Logic, Information, and Computation (Wollic'02)*, volume 67 of *ENTCS*, pages 167–184. Elsevier Science, 2002.

[38] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 79–90. Springer, 2001.

[39] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. *Journal of Theoretical Computer Science*, 311(1–3):221–256, 2004.

[40] R. Drechsler and D. Sieling. Binary decision diagrams in theory and practice. *Software Tools for Technology Transfer*, 3:103–112, 2001.

[41] E.A. Emerson and C. Lei. Modalities for model checking: Branching time strikes back. In *Proceeding of the 12th ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, 1985.

[42] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceeding of IEEE Symposium on Logic in Comuter Science*, pages 267–278. Computer Society Press, 1986.

[43] F.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[44] K. Etessami, T. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for bchi automata. In *Proc. of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, volume 2076 of *LNCS*, pages 694–707. Springer, 2001.

[45] J. Feigenbaum, S. Kannan, M.Y. Vardi, and M. Viswanathan. The complexity of problems on graphs represented as OBDDs. *Chicago Journal of Theoretical Computer Science*, 47:1–25, 1999.

[46] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. of Int. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer, 1996.

[47] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In L. Pierre and T. Kropf, editors, *Proc. of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 338–341. Springer, 1999.

[48] K. Fisler and M.Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[49] R. Focardi and R. Gorrieri. The compositional security checker: a tool for the verification of information flow security properties. *IEEE Transaction on Software Engineering*, 23(9):550–571, 1997.

[50] M. Forti and F. Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore di Pisa, Cl. Sc.*, IV(10):493–522, 1983.

[51] N. Francez. *Fairness*. Springer-Verlag, 1986.

[52] D.M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In *Proceeding of the 7th ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1980.

[53] R. Gentilini, C. Piazza, and A. Policriti. Simulation as coarsest partition problem. In J. P. Katoen and P. Stevens, editors, *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 415–430. Springer, 2002.

[54] R. Gentilini, C. Piazza, and A. Policriti. Simulation reduction as constraint. In Marco Comini and Moreno Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.

[55] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *Proc. of Int. Symposium on Discrete Algorithms (SODA'03)*, ACM, pages 573–582, 2003.

[56] R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

[57] R. Gentilini and A. Policriti. Biconnectivity on symbolically represented graphs: A linear solution. In *Proc. of Int. Symposium on Algorithms and Computation (ISAAC'03)*, volume 2906 of *LNCS*, pages 554–564. Springer, 2003.

[58] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceeding of the Fifteenth International Symposium on Protocol Specification, Testing and Verification*, pages 353–362, 1995.

[59] D. Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2:97–109, 1973.

[60] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and systems*, 16(3):843–871, May 1994.

[61] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization.

[62] G.D. Hachtel and F. Somenzi. A symbolic algorithms for maximum flow in 0-1 networks. *Formal Methods in System Design*, 10(2/3):207–219, 1997.

[63] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of Symposium on Foundations of Computer Science (FOCS'95)*, pages 453–462. IEEE Computer Society Press, 1995.

[64] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[65] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 197–211, 1994.

[66] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *The Spin Verification System (Proc. of the 2nd Spin Workshop.)*, pages 23–32. American Mathematical Society, 1996.

[67] J.E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations, Ed. by Zvi Kohavi and Azaria Paz*, pages 189–196. Academic Press, 1971.

[68] J.E. Hopcroft and R.E. Tarjan. Efficient algorithms for graph manipulation [h] (algorithm 447). *Communication of ACM*, 16(6):372–378, 1973.

[69] C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 228–240, 1983.

[70] P.C. Kannellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[71] T. Knuutila. Re-describing an algorithm by hopcroft. *Theoretical Computer Science*, 250:333–363, 2001.

[72] D. Kozen. Results on the propositional μ-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359. Springer-Verlag, 1982.

[73] M. Lbbing and I. Wegener. The number of knight's tours equals 33,439,123,484,294 - counting with Binary Decision Diagrams. *Electronic Colloquium on Computational Complexity (ECCC)*, 47(2), 1995.

[74] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[75] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. of ACM Symposium on Theory of Computing (STOC'92)*, pages 264–274. ACM Press, 1992.

[76] O. Lichtenstein and A. Pneuli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceeding of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[77] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[78] K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[79] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design OBDD - Foundations and Applications*. Springer-Verlag, 1998.

[80] R. Milner. An algebraic definition of simulation between programs. In *Proc. of Int. Joint Conference on Artificial Intelligence (IJCAI'71)*, pages 481–489. Morgan Kaufmann, 1971.

[81] R. Milner. A calculus of communicating systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes on Computer Science*, volume 92. Springer, 1980.

[82] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Workshop on Symbolic Model Checking*, volume 23, The IT University of Copenhagen, Denmark, June 1999.

[83] B.M.E. Moret. Decision trees and diagrams. *ACM Computing Surveys*, 14(4):593–623, 1982.

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

[85] R. Paige, R.E. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40(1):67–84, 1985.

[86] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. of Int. Conference on Theorical Computer Science (TCS'81)*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[87] D. Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.

[88] C. Piazza. *Computing in Non Standard Set Theories*. Cs, Department of Computer Science, University of Udine, 2002.

[89] C. Piazza and A. Policriti. Ackermann encoding, bisimulations, and OBDD's. In M. Leuschel, A. Podelski, C.R. Ramakrishnan, and U. Ultes-Nitsche, editors, *Proc. of Int. Workshop on Verification and Computational Logic (VCL'01)*, volume DSSE-TR-2001-3 of *Southampton University Technical Report*, pages 43–53. ACM Digital Library, 2001.

[90] A. Pnueli. The temporal logic of programs. In *Proceeding of the 18th Annual Symposium on Fondations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[91] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[92] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[93] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.

[94] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[95] W.R. Roscoe. *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter Model Checking CSP. Prentice Hall, 1994.

[96] J.V. Sanghavi, R.K. Ranjan, R.K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *Proc. of ACM/IEEE Design Automation Conference*, 1996.

[97] D. Sawitzki. Implicit flow maximization by iterative squaring. In *Proc. of Int. Conference on Current Trends in Theory and Practice of Computer Science SOFSEM2004)*, volume 2932 of *LNCS*, pages 301–313. Springer, 2004.

[98] D. Sawitzki. A symbolic approach to the all-pairs shortest-paths problem. In *Proc. of Int. Conference on Graph-Theoretic Concepts in Computer Science (WG 2004)*, 2004. To Appear.

[99] K. Schneider. *Verification of Reactive Systems*. Springer Verlag, 2004.

[100] D. Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, 2002.

[101] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48(2):139–144, 1993.

[102] A. Prasad Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theor. Comput. Sci.*, 49(2-3):217–237, 1987.

[103] A.P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of ACM*, 32(3):733–749, 1985.

[104] F. Somenzi. *Calculational System Design*, volume 173 of *Nato Science Series F: Computer and Systems Sciences*, chapter Binary Decision Diagrams, pages 303–366. IOS Press, 1999.

[105] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1.* 2001. Available at http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html.

[106] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[107] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[108] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, B(5):133–191, 1990.

[109] J.H. Touati, R.K. Brayton, and R.P Kurshan. Testing language containment for omega-automata using BDD's. *Information and Computation*, 118(1):101–109, 1995.

[110] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[111] M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proceeding of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[112] J.S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Survey*, 33(2):209–271, 2001.

[113] C. Wang, R. Bloem, G.D. Hachtel, K. Ravi, and F. Somenzi. Divide and compose: Scc refinement for language emptiness. In *Proc. of International Conference on Concurrency Theory (CONCUR01)*, volume 2154 of *LNCS*, pages 456–671. Springer, 2001.

[114] I. Wegener. *Branching Programs and Binary Decision Diagrams.* SIAM, 2000.

[115] I. Wegener. BDDs design, analysis, complexity, and applications. *Discrete Applied Mathematics*, 138:229–251, 2004.

[116] R. Woelfel. Symbolic topological sorting with OBDDs. *Journal of Discrete Algorithms*, 2004. To Appear.

[117] A. Xie and P. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1225–1230, 2000.