Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XII Ciclo – 1999

Advances in Planning as Model Checking

Marco Daniele

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XII Ciclo - 1999

Marco Daniele

Advances in Planning as Model Checking

Thesis Committee		Reviewers
Prof. Fausto Giunchiglia	(Advisor)	Dott. Limor Fix
Prof.ssa Luigia Carlucci Aiello		Prof. Moshe Y. Vardi
Prof. Marco Schaerf		

Author's address:
Marco Daniele
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: daniele@dis.uniroma1.it

To Simona

Preface

This thesis inspects the planning as model checking paradigm, bringing together several contributions in the fields of formalization and efficiency.

This planning paradigm has been recently proposed and seems to be very promising to develop formally clear and efficient planners dealing with expressive planning problems, that is, dealing with nondeterministic actions, partial observability of world states, and temporally extended goals.

Joining formal clearness, efficiency, and expressiveness becomes possible by exploiting the large amount of research carried out in the field of model checking, a very successful formal verification technique able to automatically check finite-state systems with respect to temporal specifications.

More in detail, a planning domain is looked at as a semantic structure, properties of planning domains are expressed in some temporal logic, and planning amounts to checking whether temporal formulas are true in the semantic structures, that is, amounts to *model checking*.

Previous work in this field uses model checking techniques for addressing nondeterministic planning domains, but does not cast them as model checking problems, loosing the formal clearness. Other work exploits the model checking framework in order to deal with temporally extended goals and partial observability.

In this thesis, we show how the former approach can be enhanced and formalized as a model checking problem, and improve the model checking techniques required by the latter.

Acknowledgements

I am very grateful to all the people I have met during these three years.

In particular, I want to thank my advisor Fausto Giunchiglia, Paolo Traverso, and Moshe Y. Vardi.

I want to thank Adolfo Villafiorita Monteleone, Piergiorgio Bertoli, Luciano Serafini, Alessandro Cimatti, Marco Roveri, Stefano Aguzzoli, Gerhard Wickler, Roberto Sebastiani, Andrea Morichetti, Marco Pistore, Enrico Blanzieri, Gianluca Mameli, and Michele Nori.

I want to thank Limor Fix, Avner Landver, Roy Armoni, Boris Ginsburg, Gila Kamhi, Ranan Fraer, Elad Shankar, Yaacov Nissim, Sitvanit Ruah, and Robert Krauthgamer.

I want to thank Bernd Finkbeiner and Henny Sipma.

Last, but first, I will thank myself, if I have been able to steal a little bit from all of you.

Contents

Preface iii

Acknowledgements v

Introduction 1

1 Planning 7

1.1 Classical Planning . . . 7
1.2 Planning Problem Representation . . . 8
1.2.1 STRIPS . . . 8
1.2.2 ADL . . . 11
1.3 Approaches to Classical Planning . . . 12
1.3.1 Planning as State-Space Search . . . 12
1.3.2 Planning as Plan-Space Search . . . 15
1.3.3 Deductive Planning . . . 17
1.3.4 Planning Graph Analysis . . . 19
1.3.5 Planning as Satisfiability . . . 22
1.4 Nonclassical Planning . . . 26

2 Temporal Logics 29

2.1 The Computational Tree Logic CTL* . . . 30
2.2 CTL and LTL . . . 33
2.3 Fairness . . . 36

3 Model Checking 37

3.1 CTL Model Checking . . . 37
3.1.1 Fair CTL Model Checking . . . 42
3.2 LTL Model Checking . . . 42
3.2.1 Graph-based LTL Model Checking . . . 43
3.2.2 Automata-based LTL Model Checking . . . 47

4 Symbolic Model Checking **53**
4.1 Ordered Binary Decision Diagrams 53
4.2 Symbolic Kripke structures 58
4.3 Fix point characterization of CTL operators 59
4.4 CTL symbolic Model checking 61
4.4.1 Symbolic Fair CTL Model Checking 65
4.5 LTL symbolic model checking 67

5 Planning as Model Checking **69**
5.1 Symbolic Approach to Planning 69
5.1.1 Planning Problems 70
5.1.2 Plans as State-Action Tables 72
5.1.3 Weak Planning 73
5.1.4 Strong Planning 76
5.1.5 Strong Cyclic Planning 78
5.2 Automata-based Approach to Planning 87
5.2.1 Planning Problems 87
5.2.2 Planning with Complete Information 88
5.2.3 Conformant Planning with Incomplete Information . . . 89
5.2.4 Conditional Planning with Incomplete Information . . . 90
5.2.5 Improved Automata Generation for Linear Temporal Logic 92

6 Conclusions and Related Work **115**

Bibliography **119**

Introduction

Planning [38, 34, 62, 5, 43, 15] the course of actions to achieve a goal has been one of the first and most challenging interests of Artificial Intelligence, motivated and since then strictly related to robotic applications. So far, the research in planning has mainly followed two divergent directions. From one end, it has heavily limited the expressiveness to yield efficient planners and, from the other end, has developed very expressive frameworks in which planning is a very hard task. Several practical planners [62, 5, 41] are available for the so called *classical planning*, which makes some fundamental assumptions: the planner has complete information about the initial state of the world, complete observability of the world states, the goals are goals of attainment, and the effects of execution of actions are deterministic. These assumptions are unrealistic in several practical domains. For instance, in a realistic robotic application, the action "pick-up a block" can not be simply described through the deterministic effect "the block is at hand". "Pick-up a block" might result either in a success or a failure, and the result can not be known a priori of the execution. On the other hand, expressiveness has been targeted in deductive planning [38, 77, 78], where planning amounts to prove a theorem, or in theory of actions [56, 52, 53]. Unfortunately, while the automatic generation of plans in deductive planning is still an open problem, the framework based on theory of action does not even deal with the problem of plan generation.

Temporal logic [67, 30] was introduced by philosophers for providing a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. In temporal logic, time is not mentioned explicitly. Instead, a formula might specify that *eventually* some designate property is satisfied, or that another property is *never* satisfied. Temporal logics come in two ways, according to the time structure: in *linear time temporal logic* [30], each instant of time has a unique successor, while in *branching time temporal logic* [30] each instant of time can have many successors.

Model checking [22, 69, 51, 9, 84] is a formal technique used for checking that a *finite-state* system satisfies its specifications. Model checking has raised a lot of attention during the last ten years, since it succeeded in making formal verification applicable in practice, allowing for early discovery of subtle

logical errors in real designs. In this approach the system to verify is modeled as state-transition systems, while its specifications are expressed in temporal logic. An efficient search procedure is then used to *check* whether the state-transition system is a *model* of the specifications. Most important, such check is completely automatic and, when failing, provides a counterexample showing why the system does not satisfies its specifications. On the other hand, the algorithmic nature of model checking makes it very sensitive to the size of the system. This problem—known as state-space explosion problem—is the major limitation of the approach. The most important discovery to face such a problem has been *symbolic model checking* [9, 60], which exploits a new data structure, namely, the *ordered binary decision diagrams* [6], to concisely represent state-transition systems and to efficiently manipulate them. An alternative approach relies on automata on infinite words [8, 81, 82], and exploits the close relationship existing between them and temporal logics [84, 85, 35].

Planning as model checking [15, 37, 2, 19, 18, 26, 16, 27] is a new planning paradigm that seems to be very promising in re-setting the focus of the research on planning towards more balanced objectives, that is, towards building planners that deal with more realistic planning problems and have good performances. This approach has been proposed by Cimatti et al. [15], who first used symbolic techniques to solve planning problems. The main idea underlying this paradigm is that, as in model checking, planning problems are faced model-theoretically. That is, planning domains are formalized as semantic models, properties of planning domains are formalized as temporal formulas, and planning is carried out by verifying whether semantic models satisfy temporal formulas. Looking at planning from this perspective introduces many new important features:

- The approach is *well-founded*: Planning problems are given a clear and intuitive formalization.

- The approach is *general*: The same framework can be used to naturally tackle many different aspects of planning, e.g., many initial states, partial observability, nondeterministic domains, and extended goal, that is, not only goals of attainment.

- The approach is *practical*: By exploiting the large amount of techniques developed for model checking, it is possible to devise efficient algorithms that generate plans automatically and that can deal with large size problems.

Beyond [15], many other works have then extended the approach to deal with nonclassical planning in several ways [19, 18, 26, 16]. More in detail, [15] introduces *weak plans*, that is, plans that may achieve the goal but, because of

nondeterminism and because plans are sequences of actions, are not guaranteed to do so. Indeed, nondeterminism has to be tackled by planning conditional behaviors, which depend on the information that can be gathered at execution time, e.g., try to pick up the block again if the execution of "pick-up a block" has failed. [19] then introduces *strong plans*, namely, plans that are guaranteed to achieve the goal in spite of nondeterminism. However, most often, a conditional plan is not enough: plans encoding iterative trial-and-error strategies like "pick up the block until succeed" are the only acceptable solutions. Indeed, in several realistic domains, a certain effect, say action success, can never be guaranteed a priori of execution and, in principle, iterative plans might loop forever, under an infinite sequence of failure. The planner, however, should generate iterative plans whose executions always have a possibility of terminating and, if they do, they achieve the goal. [18, 26] deals with *strong cyclic plans*, whose aim is to encode such iterative trial-and-error strategies. On the other hand, Vardi and DeGiacomo [27] have shown how to cope with *temporally extended goals* and *partial observability* in deterministic domains by exploiting the automata-based approach. Here, both the planning domain and the goal are looked at as automata on infinite words, and are then suitably combined in order to select the paths in the planning domain that are compatible with the goal. The close relationship between the Linear Temporal Logic LTL [30] and automata on infinite words makes then more comfortable to express goals as LTL formulas, like "eventually G" for goals of attainment or "always eventually G" for going infinitely often through the goal.

In this thesis, we inspect planning as model checking, dealing with efficiency, formalization, and expressiveness aspects. In particular, the thesis builds on [15, 19, 18, 27, 35], bringing the following contributes:

1. We provide a formal definition of strong cyclic plans based on the branching time Computational Tree Logic CTL [30]. The idea is that a strong cyclic plan is a solution such that "*for each* possible execution, *always* during the execution, there *exists* the possibility of *eventually* achieving the goal". The formalization is obtained by exploiting the universal and existential path quantifiers of CTL, as well as the "always" and "eventually" temporal connectives.

   We define a new algorithm for strong cyclic planning. Our algorithm is correct and complete, i.e., it generates strong cyclic plans according to the formal definition while, if no strong cyclic solutions exist, it terminates with failure. The algorithm in [18] did not satisfy the formal specifications. Indeed, it could generate plans that could get stuck in loops with no possibility of terminating.

   We improve the quality of strong cyclic solutions by eliminating nonrelevant actions.

We show how to extend the above formalization to weak (*there exists* an execution that *eventually* achieves the goal) and strong (*all* the executions *eventually* achieve the goal) plans.

2. We improve the algorithm for translating LTL formulas into automata. The algorithm is used to produced the goal automaton starting from the LTL formula representing the goal. Since this translation is PSPACE-complete [23] and since the goal automaton has to be composed with the usually huge automaton for the planning domain, it is highly desirable to keep the goal automaton as small as possible.

   We propose a test methodology to test the above translation. Moreover, our methodology can be also used for evaluating LTL deciders and its underlying concepts, basically targeting a uniform coverage of the formula space, can be exported to other logics.

Part of the material included in this thesis has already been published in the following papers:

- Daniele, M., Traverso, P., Vardi, M. Y., Strong Cyclic Planning Revisited. In *Proceeding of the 2nd European Conference on Planning* (ECP99).

- Cesta, A., Riccucci, P., Daniele, M., Traverso, P., Giunchiglia, E., Piaggio, M., and Shaerf, M., Jerry: a system for the automatic generation and execution of plans for robotic devices - the case study of the Spider arm. In *Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space* (ISAIRAS99).

- Daniele, M., Giunchiglia, F., and Vardi, M. Y. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer-Aided Verification* (CAV99).

The thesis consists of two parts. The first part (Chapters 1–4) deals with preliminaries, while the second one (Chapter 5) introduces the new material. More in detail,

- Chapter 1 introduces planning, discussing plan representation and planning approaches.

- Chapter 2 deals with temporal logics, presenting the logics LTL and CTL as sublogics of the more powerful logic CTL*.

- Chapter 3 presents model checking algorithms for both CTL and LTL specifications.

- Chapter 4 introduces the ordered binary decision diagrams and shows how to exploit them in order to yield symbolic model checking algorithms for both CTL and LTL specifications.

- Chapter 5 is the core of the thesis. It presents a formal framework giving semantics to weak, strong, and strong cyclic plans, and introduces symbolic algorithms for weak, strong, and strong cyclic planning. Moreover, it discusses the automata-based approach to planning and presents, and describes experiments with, an algorithm for generating automata corresponding to LTL goals.

Chapter 1

Planning

In this chapter we present the fundamental concepts about planning. More in detail, we start by discussing classical planning in Section 1.1, introduce the most important representations of planning problems in Section 1.2, describe the most relevant approaches to classical planning in Section 1.3, and conclude with Section 1.4 by surveying some solutions considered in nonclassical planning.

1.1 Classical Planning

The field of AI planning seeks to build control algorithms that enable an agent to synthesize a program of actions, whose execution from some initial state satisfies the goal. More in detail, a *planning domain* is a finite set of *world states*, and a finite set of *actions*, which are responsible for transforming the state of the world. A *planning problem* is a *planning domain*, plus a description of the *initial states* and a description of the desired behavior, i.e., of the *goal*. A solution for a planning problem, i.e., a *plan*, is a program of actions whose execution starting from some initial state satisfies the goal.

Note that the above formulation of planning is rather abstract, since the plan is a program with no further structure, actions can be nondeterministic, that is, the result of their executions cannot be known a priori, and the goal has to be (somehow) satisfied, rather than reached. However, most of the research in planning deals with *classical planning*, by making the following assumptions:

Deterministic effects: The effect of executing an action is a deterministic function of the action and the state of the world when the action is executed.

Omniscience: The agent has complete knowledge about the initial state of

the world, that is, the initial state is unique, and has complete observability of each state of the world.

Sole cause of change: The only way the worlds changes is by agent's own actions. There are no other agents and the world is static by default.

Goals of attainment: Goals are described by the state of the world the agent wants to achieve after the execution of the plan. No attention is kept to the way in which the goal state is reached.

Note that, in classical planning, due to the deterministic nature of actions, the unique initial state, and the simple nature of the goals, plans have a very simple structure, namely, a *sequence* of actions.

1.2 Planning Problem Representation

In this section, we describe the two most popular languages for planning problem representation, namely, STRIPS and ADL.

1.2.1 STRIPS

STRIPS [34] is one of the earliest representation language for planning problems[1] and, due to its simplicity, one of the most popular. This representation models actions as operation on a database, which records the current state of the world.

A STRIPS description is a pair $\langle L, O \rangle$ where L is a subset of a first-order language for describing states and O is a set of actions.

More in detail, the alphabet of L consists of a finite set of *constant* symbols c_i, a finite set of *variable* symbols x_i, a finite set of *predicate* symbols p_i with arity $a(p_i)$, and the negation $\neg$. A constant or a variable is also called a *term*. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a n-ary predicate and the t_i are terms. A *literal* is an atom or its negation and, as usual, $\neg\neg A$ is assumed to be A. A *ground* literal is a literal without variables. A ground atom is also referred to as *fluent*.

State descriptions and goals can be constructed from the above fragment of the first-order logic. As an example, consider the robot hand and initial configuration of blocks shown in Figure 1.1 (left). This situation can be represented by the set of literals $\{ON(A, TABLE), ON(C, A), ON(B, TABLE), CLEAR(C), CLEAR(B), HANDEMPTY\}$. Here, the constant symbols are A,

[1] Actually, STRIPS was a planner obtained by adapting a problem solver. Indeed, the name STRIPS stands for STanford Research Institute Problem Solver. Due to the success of the description language, the acronym STRIPS has been since then used to denote the language.

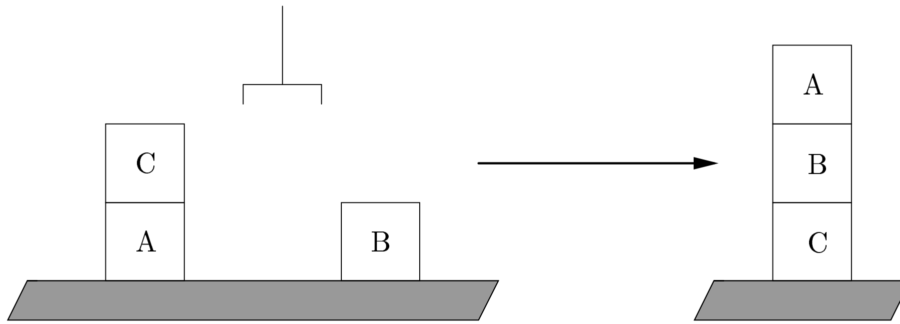

Figure 1.1: Initial and goal states for the "Sussman Anomaly" problem in the Block World.

B, and C, while the predicate symbols are *CLEAR*, *HANDEMPTY*, and *ON*. The literal CLEAR(B) means that block B has a clear top, that is, no other block is on it. The *ON* predicate is used to describe which block are directly on other blocks. The predicate *HANDEMPTY* is true just when the robot hand is empty, as in the situation depicted.

Since we require the initial state to be unique, all literals not explicitly listed in the description are assumed to be false. This is called the "Closed World Assumption" [70]. This means that, for instance, $\neg ON(A,C)$ and $\neg CLEAR(A)$ are implicitly in the initial state description.

Goal descriptions can be expressed as a set of literals too. For example, if we want the robot to construct a stack of blocks in which, as in Figure 1.1 (right), the block B is on the block C and the block A is on the block B, we might describe the goal as $\{ON(B,C), ON(A,B), ON(C,TABLE)\}$. Figure 1.1 yields a simple block-stacking challenge called the "Sussman Anomaly"[2].

For goal expressions, we allow set of literals, and any variables in goal expressions are assumed to be existentially quantified. For initial and intermediate state descriptions, we allows only set of ground literals.

Action description consists of three sets of positive literals that are called *precondition*, *delete list*, and *add list* respectively. As an alternative representation, actions can be seen as two sets, the precondition and the *effect*. As before, the precondition is a set of positive literals, while the effect is simply a set of literals, where the negated ones represent the above delete list.

To understand how the execution of an action changes the current state of the world, it is necessary to introduce the concept of *unification* among literals. Given a set of literals, the target is to compute a *substitution* of terms for the

[2]The problem was discovered at MIT in 1973 by Allen Brown who noticed that the HACKER problem solver, developed by Sussman, had problems dealing with it.

variables to make all the literals identical. Substitutions are represented as set of pairs $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable and each t_i is a term in which x_i does not occur. For example, the literals $ON(x, C)$ and $ON(A, C)$ are unified by the substitution $\{x/A\}$. A unifying substitution for the set E of literals is called a *unifier* of E. The *composition* of two substitutions s_1 and s_2, denoted as $s_1 s_2$, is that substitution obtained by applying s_2 to the terms of s_1 and then adding those pairs of s_2 having variables not occurring among the variables of s_1. Thus, for example, $\{z/g(x,y)\}\{x/A, y/B, w/C, z/D\}$ is $\{z/g(A,B), x/A, y/B, w/C\}$. A unifier g of a set E of literal is called the *most general unifier* of E if, for every other unifier s of E, there exists a substitution s' such that s can be obtained by composing g and s'.

An action is *executable* in a state if there exists a most general unifier unifying each one of the literals in the preconditions with some ground literal in the state description. We call such unifier the *match substitution*. When an action is executable and is executed in a state description, the first step is to apply the match substitution to both the add and the delete lists. We assume that all the variables occurring in such lists also occur in the precondition. Second, the ground literals from the delete list are removed from the old state description, while the ground literals from the add list are added to this latter state description to produce the new state description.

As an example, we could model the action of moving a block x from the source y to the target z as follows

MOVE(x, y, z):

Precondition: $\{ON(\text{x,y}), CLEAR(x), CLEAR(z), x \neq y, x \neq z,$
$z \neq TABLE\}$
Delete list: $\{ON(x, y), CLEAR(z)\}$
Add list: $\{ON(x, z), CLEAR(y)\}$

or, alternatively, as

MOVE(x, y, z):

Precondition: $\{ON(\text{x,y}), CLEAR(x), CLEAR(z), x \neq y, x \neq z,$
$z \neq TABLE\}$
Effect: $\{\neg ON(x, y), \neg CLEAR(z), ON(x, z), CLEAR(y)\}$

The above action is executable in the state depicted in Figure 1.1 when for example x is substituted with C, y with A, and z with B. When the action is executed, the new state description becomes $\{CLEAR(A), CLEAR(C), ON(C, B),$ $ON(A, TABLE), ON(B, TABLE)\}$.

Note that our definition is restricted so that a block can not be moved to the table. This is necessary because the action's effects are different when the destination is the table. Specifically, the intuition for the table is that it is always clear, and this clashes with the effect making the destination not clear.

1.2.2 ADL

Another interesting language for planning problem representation is the Action Description Language (ADL) [62], which introduces *conditional effects* and *universal quantification*.

Conditional effects are useful to relax the annoying aspect of the *MOVE* operator defined above, that is, the restriction that the destination can not be the table. Due to this restriction, to describe the possible movement actions, it is necessary to augment *MOVE* with an additional *MOVE-TO-TABLE*. This is irritating for both the user, software engineering, and efficiency. Indeed, for example, a planner has to commit whether the destination is the table or some other block, even if the movement action is required to deal with part of the goal that has nothing to do with the destination. This problem is solved by allowing action definitions to use conditional effects. The basic idea is simple: we allow a special *when* clause in the syntax of action effect. *when* takes two arguments, an antecedent and a consequent. Both the antecedent and the consequent are a set of literals, but their interpretation is very different. The antecedent refers to the state *before* the action is executed, while the consequent refers to the state *after* the execution. The interpretation is that the execution of the action will have the consequent's effect just in the case that the antecedent is true immediately before the execution. For example, we can extend the *MOVE* definition in order to release the constraint on the destination as follows

$MOVE(x, y, z)$:

Precondition: $\{ON(x,y), CLEAR(x), CLEAR(z), x \neq y, x \neq z, z \neq TABLE\}$
Effect: $\{\neg ON(x,y), ON(x,z), CLEAR(y), (\{\neg CLEAR(z)\}\ when\ \{z \neq TABLE\})$

Universal quantification is very handy to express actions in a concise and clear way. For example, one could implement the *CLEAR* predicate by universally quantifying over the *ON* predicate. Another interesting case is mixing universal quantification with conditional effects that, for example, allows for the specification of objects like briefcases where moving the briefcase causes *all* objects inside to move as well:

MOVE(x, y, z):

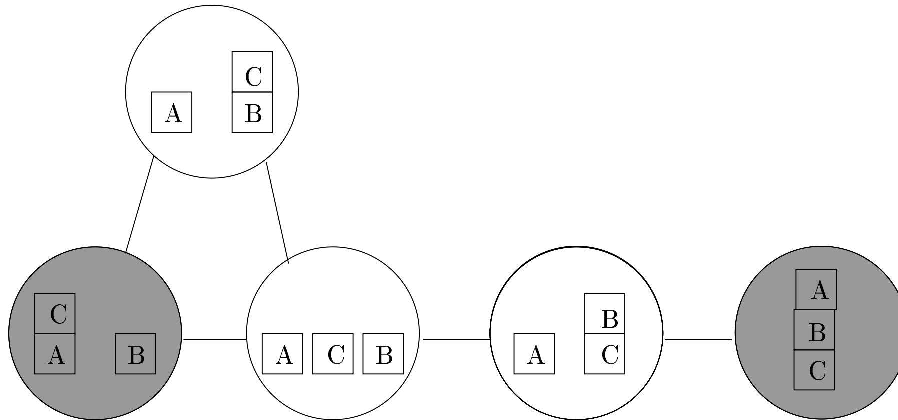

Figure 1.2: A fragment of the state space in the block world.

Precondition: $\{BRIEFCASE(x), AT(y), y \neq z\}$
Effect: $\{AT(x,z), \neg AT(x,y),$
$(\forall k)(\{AT(k,z), \neg AT(k,y)\}\ when\ \{IN(k,x)\})\}$

Finally, note that, while universal quantification is basically syntactic sugar, conditional effects strictly increase the expressive power of STRIPS.

1.3 Approaches to Classical Planning

In this section we try to classify the solutions adopted by some of the most relevant classical planning systems. More in detail, we consider state-space search in Subsection 1.3.1, and plan-space search in Subsection 1.3.2. In Subsection 1.3.3 we introduce deductive planning. Finally, we consider the last advances in classical planning, namely, planning as graph analysis in Subsection 1.3.4 and as satisfiability in Subsection 1.3.5. Such a classification is not a partition since, for example, searching in the state-space is a particular case of searching in the plan-space, and deductive planning, due to its generality, can encode all the other approaches.

1.3.1 Planning as State-Space Search

The simplest way to build a planner is to cast the planning problem as search through the space of world states. Figure 1.2 shows a fragment of such a space for the world of blocks. Each node in the graph denotes a state of the world, and edges connect worlds that can be reached by executing a single action. In general, arcs are directed, but in our model of the block world all

Algorithm REGRESSION(*initial-state*, *current-goal*, *actions*, *path*)

1. **Termination:** If *initial-state* satisfies the *current-goal* then return *path*.
2. **Action selection:** Let *act* = CHOOSE from *actions* an action whose effects matches at least one literal in *current-goal*.
3. **Goal regression:** Let *new-current-goal* be the result of regressing *current-goal* through *act*, and *new-path* be the result of concatenating *act* and *path*.
4. **Failure:** If no choice for *act* is possible or *new-current-goal* is undefined or contains *current-goal* then return failure.
5. **Recursive invocation:** Return REGRESSION(*init-state*, *new-current-goal*, *actions*, *new-path*)

Figure 1.3: A regressive, state-based planner. The initial call should set *path* to the null sequence.

the actions are reversible, so that we have replaced two directed edges with a single undirected one to increase readability. Note that the initial and the goal states of the Sussman anomaly are highlighted in grey. When phrased in this manner, the solution to a planning problem is a path through the state space.

The advantage of casting planning as a simple search problem is the immediate applicability of all the familiar brute force and heuristic search algorithms [47]. For example, one could use depth-first, breadth-first, or iterative deepening A* search starting from the initial state until the goal is located. Alternatively, more sophisticated memory bounded algorithms could be used [71, 48].

A handy way to describe search algorithms is to specify them nondeterministically by using a nondeterministic CHOOSE primitive. CHOOSE takes a set of possible options and "magically" selects the right one. In real planners, CHOOSE can be implemented through any exhaustive search method or approximated with some aggressive search strategy.

In Figure 1.3, we describe a nondeterministic regressive planning algorithm that operates by searching backwards from the goal until the initial state is found.

When REGRESSION is called on the Sussman anomaly, *current-goal* is initially set to $\{ON(A,B), ON(B,C)\}$, and *path* is set to the null sequence of actions and, since this situation does not satisfy the initial state, CHOOSE demands an action whose effect contains a literal in *current-goal*. Magically, the

action *MOVE-A-FROM-TABLE-TO-B* is returned.

The next step, that is, *goal regression*, forms the core of the algorithm. *new-current-goal* is assigned the result of regressing the set *current-goal* through the action *act*. The result of this regression is another set of literals that encodes the weakest precondition that must be true before *act* is executed in order to assure that *current-goal* will be true after *act* is executed. This is simply the union of *act*'s precondition with all the literals in the current goal, except those provided by the effects of *act*, that is,

$$\textit{new-current-goal} = \text{precondition}(\textit{act}) \cup (\textit{current-goal} \setminus \text{add-list}(\textit{act}))$$

In our example, suppose that *MOVE-A-FROM-TABLE-TO-B* is defined as

Precondition: $\{ON(A, TABLE), CLEAR(A), CLEAR(B)\}$
Delete-list: $\{ON(A, TABLE), CLEAR(B)\}$
Add-list: $\{ON(A, B)\}$

the effect of regressing $\{ON(A, B),\ ON(B, C)\}$ is the set $\{ON(A, TABLE),$ $CLEAR(A),\ CLEAR(B),\ ON(B, C)\}$. Since *act* does not affect *ON(B,C)*, it remains part of the weakest preconditions.

If the selection of the action is not possible, or the regression step fails, the algorithm returns failure. More in detail:

- If no action has an effect containing a literal matching one of the literal in *current-goal*, then no action if profitable.
- If the effect of *act* conflicts with *current-goal*, the result of regressing *current-goal* through *act* is undefined. Indeed, no matter what is true before *act* is executed, its execution will ruin things.
- If *current-goal* is contained in its regression, each state satisfying the regression satisfies *current-goal* as well. Thus, there is no point in considering such an *act* because any successful plan that might result could be improved by eliminating *act* from *path*.

Otherwise, if both the selection and the regression steps are successful, the selected action is appended to the current partially-specified path, and the algorithm is invoked recursively.

STRIPS is a classic planner searching the state space through a regressive algorithm. Moreover, it uses the *means-ends strategy* to direct the search process, that is, actions are selected in order to minimize the difference between the current state and the initial one. Due to this strategy, STRIPS produces straightforward solutions to many problems, but there are problems for which it produces nonoptimal solutions, that is, solutions longer than necessary, and problems for which it cannot find any solution at all.

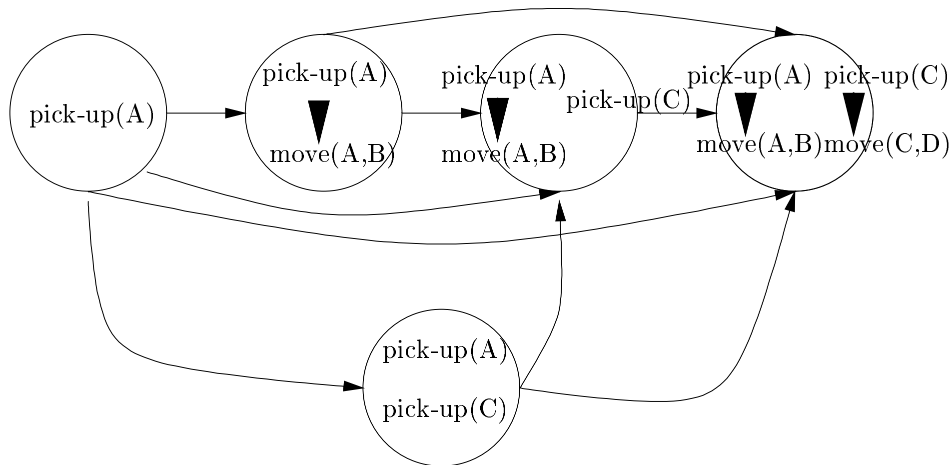

Figure 1.4: A fragment of the plan space in the block world.

1.3.2 Planning as Plan-Space Search

In 1974, Earl Sacerdoti built a planner, called NOAH [72], with many novel features among which the most innovative was the reformulation of planning. Instead of searching the space of states, in which edges denote action execution, Sacerdoti phrased planning as search through the space of plans. In this space, nodes represent partially-specified plans and edges denote plan refinements. A plan refinement is obtained by adding an action and defining its execution order with respect to the actions already present in the plan. Figure 1.4 illustrates a fragment of such a space in the block world. In such case, the ordering among actions is given by the arrows relating them. Note that the ordering among actions defining the partial plans labeling the nodes is a partial ordering, rather than the total order one obtains by searching the state space. For this reasons, planner working in this environment are also called *partial order planners*. This means that the resulting plan is indeed a set of totally ordered set of actions, namely, the ones compatible with the partial order plan. Partial order planners are also called *least commitment* planners, since they allow for deferring decisions about action ordering until this is really required.

While visiting the state space has been seen being (conceptually) simple, searching the plan space is much more complex and requires introducing some background work. A *partial plan* is a 3-tuple $\langle A, O, L\rangle$ where A is a set of *actions*, O is a set of *partial ordering constraints* over A, and L is the set of *causal links*, which we are about to explain below. As a partial order planner refines a plan, it must do constraint satisfaction to ensure the consistency of O, that is, to ensure that from O can be extracted at least one sequence of

actions whose order is compatible with the one stated in O.

A key aspect that partial order plans have to take into account is keeping track of past decisions and the reasons for those decisions. For example, if one purchases plane tickets to satisfy the goal of boarding the plane, then one should be sure to take them at the airport. If another goal, say having one's hands free to open the taxi door, causes one to drop the tickets, one should be sure to pick them up again. A good way of ensuring that the different actions introduced for different goals will not interfere each other is to record the dependencies among actions. To record these dependencies, we use causal links. A causal link consists of three parts: two actions, namely the link's producer A_p and its consumer A_c, and a literal Q, which is an effect of the first action and a precondition of the second one. We write such a causal link as $A_p \overset{Q}{\rightarrow} A_c$, and say that Q is *supported* by A_p in A_c.

Causal links are used to detect whether a newly introduced action interferes with past decisions. We call such an action a *threat*. More precisely, suppose that $\langle A, O, L \rangle$ is a partial plan, $A_p \overset{Q}{\rightarrow} A_c$ is a causal link in L, and let A_t be a different action in A. We say that A_t threatens $A_p \overset{Q}{\rightarrow} A_c$ if the followings are satisfied:

- $O \cup \{A_p < A_t < A_c\}$ is consistent,
- A_t has $\neg Q$ as effect.

For example, if A_p asserts $Q =$*ON(A,B)*, which is a precondition for A_c, and the plan contains $A_p \overset{Q}{\rightarrow} A_c$, then A_t would be considered a threat if it moved A off B and the ordering constraints did not prevent A_t from being executed between A_p and A_c.

When a plan contains a threat, there is a danger that the plan will not work as anticipated. To prevent this from happening, the planning algorithm must check for threats and take evasive countermeasures. For example, the algorithm could add an additional ordering constraint to ensure that A_t is executed before A_p. This particular threat protection method is called *demotion*. adding a symmetric constraint $A_c < A_t$ is called *promotion*.

The totally undefined plan, or *null plan*, can be represented as the tuple $\langle A = \{A_0, A_\infty\}, O = \{A_0 < A_\infty\}, \{\}\rangle$, where A_0 is a new "start" action with no precondition and add-list defining the initial state, and A_∞ is a new "end" action with no effects and precondition defining the goal. For example, the null plan corresponding to the Sussman anomaly is

A_0:

Preconditions:
Effect: $\{ON(A, TABLE), ON(C, A), ON(B, TABLE),$
$CLEAR(C), CLEAR(B)\}$

A_∞:

Preconditions: $\{ON(B,C), ON(A,B), ON(C,TABLE),$
$CLEAR(A)\}$

In Figure 1.5 we describe a simple regressive, partial order plan algorithm. POP maintains the set of currently unsupported literals in *agenda* as set of tuples $\langle Q_{ij}, A_i \rangle$. Thus, *agenda* is initially set to the goal, that is, to $\{\langle Q_i, A_\infty \rangle : Q_i$ is a literal of the goal$\}$. POP starts with the null plan for the planning problem at hand, and makes nondeterministic choices until all literals of every action's precondition have been supported by causal links and all threatened links have been protected from possible interferences.

The most important results in planning as plan-space search is UCPOP [62], a sound and complete partial order planner for ADL.

1.3.3 Deductive Planning

Deductive planning [38, 77, 78] consists of formulating the planning problem as a problem of deduction in such a way that a theorem prover can solve it and, by solving it, exhibits a proof from which a plan can be extracted.

For example, the Green's formulation [38], which is considered to be one of the first attempts to solve planning problems, is based on a resolution theorem prover, involves one set of assertions that describe the initial state and another set that describe the effects of the various actions. To keep track of which facts are true in which state, a "state" or "situation" variable is included in each predicate. This idea, often referred to as the *situation calculus*, goes back to McCarthy [58, 59]. The goal condition is then described by a formula with an existentially quantified state variable. That is, the system would attempt to prove that there exists a state in which certain condition are true. A constructive proof method can then be used to produce the set of actions that generate the desired state.

Suppose that we have the initial situation depicted in Figure 1.1 (left). Suppose we name this initial state SO. Then we denote the fact that block x is on some other block y (or on the table) in situation SO by the literal $ON(x, y, SO)$ $(ON(x, TABLE, SO))$. The state name is made an explicit argument of the predicates. The complete configuration of blocks in the initial state is then given by

$ON(C, A, SO)$
$ON(A, TABLE, SO)$
$ON(B, TABLE, SO)$
$CLEAR(C, SO)$
$CLEAR(B, SO)$

Algorithm POP($\langle A, O, L\rangle$, *agenda*, *actions*)

1. **Termination:** If *agenda* is empty return $\langle A, O, L\rangle$.

2. **Goal selection:** Let $\langle Q, A_{need}\rangle$ = CHOOSE a pair from the *agenda*.

3. **Action selection:** Let A_{add} = CHOOSE an action that adds Q. A_{add} can be a newly instantiated action or an action already in A, which can be consistently ordered prior to A_{need}. If A_{add} is newly instantiated then let

   - $A' = A \cup \{A_{add}\}$.
   - $L' = L \cup \{A_{add} \stackrel{Q}{\rightarrow} A_{need}\}$
   - $O' = O \cup \{A_0 < A_{add} < A_{need} < A_\infty\}$.

   else let

   - $A' = A$
   - $L' = L \cup \{A_{add} \stackrel{Q}{\rightarrow} A_{need}\}$
   - $O' = O \cup \{A_{add} < A_{need}\}$

4. **Failure:** If no action can be chosen then return failure.

5. **Update goal set:** Let $agenda' = agenda \backslash \{\langle Q, A_{need}\rangle\}$. If A_{add} is newly instantiated, then for each literal Q_i of its preconditions, add $\langle Q, A_{need}\rangle$ to $agenda'$.

6. **Causal link protection:** For every action A_t that might threaten a causal link $A_p \stackrel{Q}{\rightarrow} A_c \in L'$, CHOOSE a consistent ordering constraint between:

   (a) **Demotion:** Add $A_t < A_p$ to O'

   (b) **Promotion:** Add $A_c < A_t$ to O'

   If neither constraint is consistent, then return failure.

7. **Recursive invocation:** POP($\langle A', O', L'\rangle$,$agenda'$, *actions*)

Figure 1.5: A regressive partial order planner. The initial call must set $\langle A, O, L\rangle$ to the null plan for the planning problem, and *agenda* to the goal.

Now we need a way to express the effects that various actions might have on the states. The natural way of doing this is through implications. Moreover, when an action is executed in one state, we use the special term "$do(action, state)$" to denote the new state. Thus, if the action $MOVE(x, y, z)$ moving the block x from the position y to the position z is executed in SO, the new state is represented by the term $do(MOVE(x, y, z), SO)$. A possible formulation of $MOVE(x, y, z)$ follows:

$MOVE(x, y, z)$:

$$(CLEAR(x, S) \wedge CLEAR(z, S) \wedge ON(x, y, S) \wedge (x \neq z)) \Rightarrow$$
$$(ON(x, z, do(MOVE(x, y, z), S)) \wedge CLEAR(x, do(MOVE(x, y, z), S))$$
$$\wedge\ CLEAR(y, do(MOVE(x, y, z), S)))$$

However, the above formulation does not completely specify the effects of the action. We must also state through further assertions that certain features are *unaffected* by the action. Such assertions are called the *frame assertions* [59]. For example, the following assertion expresses that the blocks that are not moved stay in the same position:

$$ON(v, w, S) \wedge (v \neq x) \Rightarrow$$
$$ON(v, w, do(MOVE(x, y, Z), S))$$

Finally, suppose that we want to achieve the simple goal depicted in Figure 1.1 (right). This goal would be expressed as

$$(\exists S) ON(A, B, S) \wedge ON(B, C, S) \wedge ON(C, TABLE, S) \wedge CLEAR(A, S)$$

The problem can now be solved by finding a constructive proof of the goal formula, where the existential witness of the form

$$do(a_1, (\ldots, do(a_n, SO) \ldots)$$

represents the plan $a_1 \ldots s_n$.

1.3.4 Planning Graph Analysis

This approach [5, 46, 54] is based on constructing and analyzing a compact structure called the *planning graph*. The approach combines aspects of both state-space search and partial-order planners. Indeed, it makes strong commitments while constructing the planning graph, but generates partially ordered plans. The approach alternates between two phases: *graph expansion* and *solution extraction*. The graph expansion phase extends the planning graph forward in "time" until it has achieved a necessary, but possibly insufficient, condition for plan existence. The solution extraction phase then performs a backward-chaining search on the graph, looking for a plan that solves the problem. If no solution is found, the cycle repeats by further expanding the planning graph.

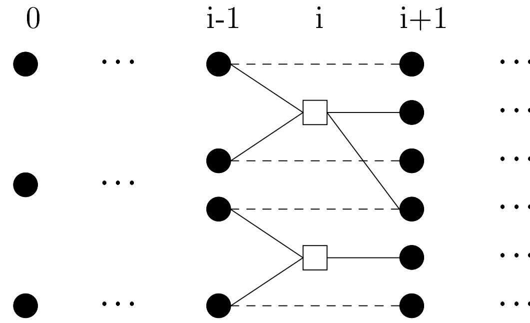

Figure 1.6: A fragment of a planning graph.

Expanding the Planning Graph

The planning graph contains two types of nodes, proposition nodes and action nodes, arranged into levels as shown in Figure 1.6. The planning graph alternates proposition (circle) and action (square) layers. Horizontal dashed lines between propositions layers represent "maintenance actions", which encode the possibility that unaffected propositions will persist until the next layer. Even-numbered levels contain proposition nodes, that is, ground literals, and the zeroth level consists precisely of the propositions that are true in the initial state of the planning problem at hand. Nodes in odd-numbered levels correspond to action instances. There is one such node for each action instance whose preconditions are present and are mutually consistent at the previous level. Edges connect proposition nodes to the action instances at the next level whose preconditions mention those propositions, and additional edges connect from action nodes to subsequent propositions made true by the action's effects.

Note that the planning graph represents "parallel" actions at each action level. However, just because two actions are included in the planning graph at some level does not mean that it is possible to execute both at once. Central to the efficiency of this approach is inference regarding a binary mutual exclusion relation (from now on *mutex*) among nodes at the same level. We define this relation recursively as follows

- Two action instances at level i are mutex if either
  - *Inconsistent effects:* The effect of one action is the negation of another action's effect, or
  - *Interference:* One action deletes the precondition of another, or
  - *Competing Needs:* The actions have preconditions that are mutually exclusive at level $i-1$.

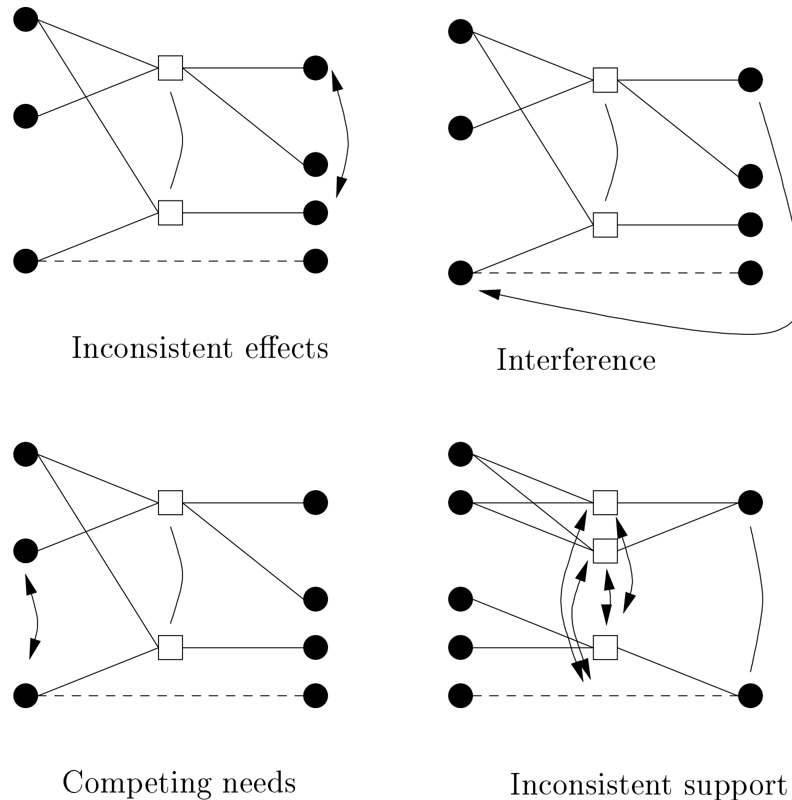

Figure 1.7: Graphical depiction of mutex definition. Curved lines represent mutex relations, curved lines with arrows represent mutex causes.

- Two propositions at level i are mutex if either
  - Inconsistency: One is the negation of the other, or
  - Inconsistent support: All the ways of achieving the propositions, that is, actions at level $i-1$, are pairwise mutex.

In Figure 1.7, a graphical representation of the above conditions is given. While expanding the plan, one has also to take into account to propagate the mutex relationship from one layer to the next one.

Solution extraction

The second phase starts when the planning graph has been extended to an even level i in which all the goal literals are present and none are pairwise mutex. This is a necessary condition for plan existence, but it does not ensure that a

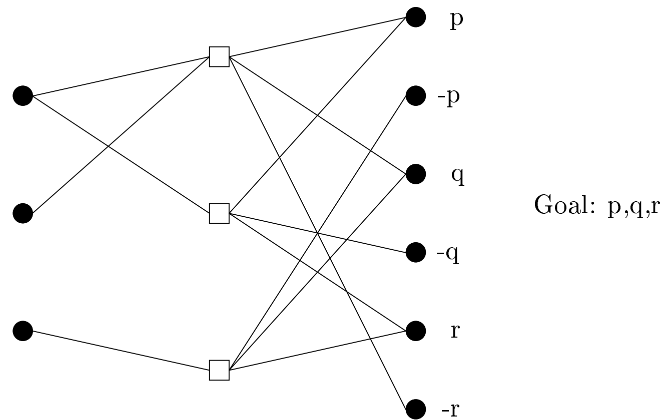

Figure 1.8: Mutex is not a sufficient condition for planning existence.

plan does exist, as shown in Figure 1.8. The point is that the propagation of the mutex conditions finds many incompatibilities, but not all.

Solution extraction searches for a plan by considering each of the literals in turn. For each such literal at level i, again magically, CHOOSE selects an action a at level $i-1$ that achieves the literal and is nonmutex with those already gathered. If no such choice is possible, then failure is returned.

Once a consistent set of actions for level i has been found, we have to consider their preconditions. Therefore, if $i = 1$ one has to check that the preconditions hold in the initial state, otherwise such preconditions are recursively analyzed.

1.3.5 Planning as Satisfiability

Planning as propositional satisfiability [43, 44, 42, 41] has gained a lot of attention during the last years, due to recent advances in propositional satisfiability methods [75, 4, 57, 88]. Indeed, the first attempt of facing planning in such a manner were rather unremarkable [43].

Figure 1.9 shows the typical architecture of a planning system based on a propositional decider. The *compiler* takes a planning problem as input, guesses a plan length, and generates a propositional formula, usually in conjunctive normal form, that is satisfiable if and only if a plan of such a length exists. During this step, a symbol table records the correspondence between propositional variables and the planning instance. The *simplifier* shrinks the formula removing possible redundancies, and the *solver* uses systematic or stochastic methods to find a satisfying assignment that the *decoder* translates, by using the symbol table, into a solution plan. Similarly to the graph-based approach,

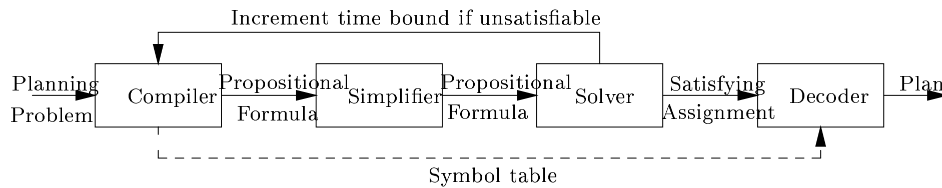

Figure 1.9: The structure of a typical planner via satisfiability.

if the solver fails in finding a satisfying assignment, the compiler generates a new encoding reflecting a longer length.

Beside the solver, a critical component of the above architecture is the compiler, which should produce quickly a "good" formula. However, this translation is complicated by the fact that a propositional formula can be measured in terms of the number of variables, the number of clauses, or the total number of literals summed over all clauses. Moreover, often a decrease of one parameter will increase another. In what follows, we present a parameterized space of possibilities, developed in [33], with two dimensions: *action representation* and *frame axioms.*

Each of the encodings we are about to introduce resembles the structure of a plan in the graph-based approach: fluents occur at even-numbered times and actions at odd-numbered times. Moreover, all such encodings use the following set of *universal axioms*

- *init:* The initial state is completely specified at time zero, including all properties presumed false by the closed-world assumption.

- *goal:* In order to test for a plan of length n, all desired goal properties are asserted to be true at time $2n$.

- *precondition-effect implication:* Actions imply their preconditions and effects. For each odd time t between 1 and $2n-1$ and for each consistent ground action, an axiom asserts that execution of the action at time t implies its effects hold at time $t+1$ and its preconditions hold at time $t-1$.

Action Representation

The first major encoding choice is how to represent actions. This choice specifies the correspondence between propositional variables and ground actions and works out a tradeoff between the number of propositional variables and the number of clauses.

In the *regular* representation, each ground action is represented by a different propositional variable, for a total of $n \cdot |Act| \cdot |Dom|^{A_{Act}}$, where n is the number of odd-time steps in plan, $|Act|$ is the number of action schemata, $|Dom|$ is the number of constants in the domain, and A_{Act} is the maximum arity of actions schemata. Therefore, the factor $|Act| \cdot |Dom|^{A_{Act}}$ represents the number of ground actions. Since systematic solvers take time exponential in the number of variables, and large number of variables also slow stochastic solvers down, we would like to reduce this number.

In order to do this, in [44] was introduced the *simple operator splitting*, which replaces each n-ary action with n unary fluents throughout the encoding. For example, $MOVE(A, B, C, t)$[3] is replaced with the conjunction of $MOVE\text{-}ARG1(A, t)$, $MOVE\text{-}ARG2(B, t)$, and $MOVE\text{-}ARG3(C, t)$. Doing this for all fully-instantiated actions reduces the number of variables needed to represent all actions to $n \cdot |Act| \cdot |Dom| \cdot A_{Act}$.

In simple splitting, only instances of the same action share propositional variables. An alternative is *overloaded splitting*. Overloaded splitting replaces $MOVE(A, B, C, t)$ by conjuncting $ACT(MOVE, t)$, $ARG1(A, t)$, $ARG2(B, t)$, and $ARG3(C, t)$, while a different action $PAINT(A, RED, t)$ is replaced by the conjunction $ACT(PAINT, t)$, $ARG1(A, t)$, $ARG2(RED, t)$. This technique further reduces the number of variables to $n(|Act| + |Dom|A_{Act})$.

Finally, the *bitwise* representation shrinks the number of variables even more, by representing the actions with only $n \log_2\lceil(|Act| \cdot |Dom|^{A_{Act}})\rceil$ variables. More in detail, this is carried out by numbering the ground actions from 0 to $|Act| \cdot |Dom|^{A_{Act}} - 1$. The number encoded by the bit symbols determines the ground action that executes at each odd time step. For example, if there were 4 ground actions, $\neg bit1_t \wedge \neg bit2_t$ replaces the first action, $\neg bit1_t \wedge bit2_t$ replaces the second one, and so on.

Frame Axioms

The other important encoding choice is related to the frame axioms, that is, how to constrain unaffected fluents when an action occurs.

Classical frame axioms [59] state which fluents are left unchanged by a given action. For example, a classical frame axiom for the $MOVE$ action, stating that moving a block A from B to C leaves D's clearness unchanged, can be encoded as $CLEAR(D, t-1) \wedge MOVE(A, B, C, t) \Rightarrow CLEAR(D, t+1)$.

Adding classical frame axioms for each action and each odd time to the universal axioms almost produces a valid encoding of the planning problem. However, if no action occurs at time t, the axioms of the encoding can infer

[3] Note that we are using nonstandard notation here in order to emphasize the combinatorics. Indeed, when writing $MOVE(A, B, C, t)$ we denote a *propositional variable*. A more clear but heavier notation would have been *MOVE-A-FROM-B-TO-C-AT-TIME-t*.

nothing about the truth value of fluents at time $t+1$, which can therefore take arbitrary values. The solution is to add an *at-least-one* axiom for each time step, that is, a disjunction of every possible ground action ensuring that some action occurs at each odd time step. However, as shown in [33], this approach has a huge effect on the size of these axioms. The resulting plan consists then of a totally-ordered sequence of actions. Indeed, classical frame and at-least-one axioms do not force that exactly one action occurs at each odd time step. However, when combined with the precondition-effect implication axioms and considering that the initial state is completely defined they ensure that any two actions occurring at odd time t lead to an identical state at time $t+1$. Therefore, the linear ordering can be obtained by randomly selecting, for each odd time t, an action among those occurring at time t.

Explanatory frame axioms [39] enumerate the set of actions that could have occurred in order to account for a state change. For example, an explanatory axiom would say which actions could have caused D's clearness status to change from true to false as

$$\begin{aligned}&CLEAR(D,t-1) \wedge \neg CLEAR(D,t+1)\\&\Rightarrow (MOVE(A,B,D,t) \vee MOVE(A,C,D,t) \vee \ldots\end{aligned}$$

As a supplement to the universal axioms, explanatory frame axioms must be added for each ground fluent and each odd time t to produce a correct encoding. With explanatory frame axioms, a change in a fluent's truth value implies that some action occurs, so (contrapositively) if no action occur at an odd time step, this will be correctly treated as a no-operation. Therefore, no at-least-one axioms are required.

Explanatory frame axioms brings an important benefit. Since they do not explicitly force the fluents unaffected by action execution to remain unchanged, they permit parallelism. Specifically, any actions whose preconditions are satisfied at time t and whose effects do not contradict each other might be executed in parallel. This kind of parallelism is, however, problematic, since it might give rise to valid plans from which no totally-ordered sequence of actions can be extracted. For example, suppose that the action a_1 has precondition X and effect Y, while the action a_2 as precondition $\neg Y$ and effect $\neg X$, and suppose that $X \wedge \neg Y$ and $\neg X \wedge Y$ are satisfiable. While these actions might be executed in parallel since neither their preconditions nor their effects are inconsistent, there is no legal total ordering of the two actions. Hence, one must explicitly rule out this type of pathological behavior with the *exclusion* axioms. *Complete exclusion* ensures that only one action can occur at each odd-time step and, for each ground action a_1 and a_2 and odd-time t, adds clauses encoding that either a_1 or a_2 cannot occur at time t. *Conflict exclusion* results in partially-ordered plans from which totally-ordered plans can be extracted by introducing new clauses only for the conflicting actions.

Finally, experience [33, 44] shows that explanatory frame axioms are clearly superior to classical frame axioms in almost every case.

1.4 Nonclassical Planning

Classical planning makes some fundamental assumptions: the planner has complete information about the initial state of the world, complete observability on the world states, effects of the execution of actions are deterministic, and therefore the solution to the planning problem can be expressed as a sequence of actions. These assumptions are unrealistic in several practical domains, like robotics, scheduling, and control. The initial state of a planning problem may be not unique, some features of the world may be not observable, and the effect of actions may have several effects. Moreover, in case of nondeterminism, plans as sequences of actions are bound to failure. Indeed, nondeterminism must be tackled by planning conditional behaviors, which depend on the information that can be gathered at execution time. For instance, in a realistic robotic application, the action "pick-up a block" cannot be simply described as a STRIPS operator whose effect is "the block is at hand". "Pick-up a block" might result either in a success or a failure, and the result cannot be known *a priori* of execution. A useful plan, depending on the action outcome, should execute different actions, e.g., try to pick-up the block again if the action execution has failed.

Most often, a conditional plan is not enough: plans encoding iterative trial-and-error strategies, like "pick up a block until succeed", are the only acceptable solutions. In several realistic domains, a certain effect (e.g., action success) might never be guaranteed a priori of execution and, in principle, iterative plans might loop forever, under an infinite sequence of failures. The planner, however, should generate iterative plans whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal.

However, even though classical planning suffers from the lacks we have described, most of the work in planning is focused on it. Only in the last couple of years, some works have extended classical planners to *contingent* planners [87, 86, 63, 68, 19], which generate plans with conditionals, or to *conformant* planners [76, 17, 16], which unrealistically try to find solutions to nondeterministic planning problem as sequences of actions. Note that the first problem one has to solve when dealing with nondeterministic planning domains is that neither STRIPS nor ADL are expressive enough. Indeed, in order to express nondeterminism one needs languages that allows us to express the fact that an action has multiple outcomes or, in other words, disjunctive effects.

Deductive planning frameworks [77, 78] can be used to specify desired plans in nonclassical frameworks. Nevertheless, the automatic generation of plans in these deductive frameworks is still an open problem. [13] proposes a framework based on process algebra and mu-calculus for reasoning about nondeterministic and concurrent actions. The framework is rather expressive, but it does not deal with the problem of plan generation. In planning based on Markov Decision Processes [28, 11, 40], nondeterministic environments are dealt with through stochastic automata, where actions induce transitions with an associated probability, and states have an associated reward. The planning task is then reduced to look for optimal executions with respect to rewards and probability distributions. Planning as model checking [15, 37, 19, 2, 18, 26, 16, 27], which is the subject of this thesis, joins expressiveness with the possibility of devising automatic practical planners. In particular, [18, 26] deal with iterative plans, while [27] handles incomplete information and temporally extended goals in deterministic domains.

Chapter 2

Temporal Logics

Temporal logics [67, 30] were introduced by philosophers for providing a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. Later, in a landmark paper [64], Pnueli argued that temporal logic could be a useful formalism for specifying and verifying correctness of finite-state computer programs. More generally, looking at finite-state systems as temporal logic semantic structures, temporal logics can be used to describe properties of such systems.

A finite-state system such a planning domain, a hardware controller, or a communication protocol can be described abstractly by a structure consisting of a finite set of the possible states of the system and a set of the legal transitions between states. For example, in a planning problem in the block world, states may differ on the location of blocks, while in a communication protocol some states might represent situations in which some input buffer is full, and some other states might represent situations in which the buffer is only partially filled. In addition, we also need a way to describe properties of such states. To this end, we label states with symbols from some set to represent such properties. These symbols are called *atomic propositions*. A tuple consisting of a set of states, a transition relation, and a labeling of states by atomic propositions is called a *state transition graph*, or a *Kripke structure* [30].

In temporal logic, time is not mentioned explicitly. Instead, a formula might specify that *eventually* some designate property is satisfied, or that another property is *never* satisfied. These operators can also be combined with Boolean connectives or nested arbitrarily.

The temporal logics we are going to consider can be uniformly introduced as fragments of the more powerful logic called CTL* [30].

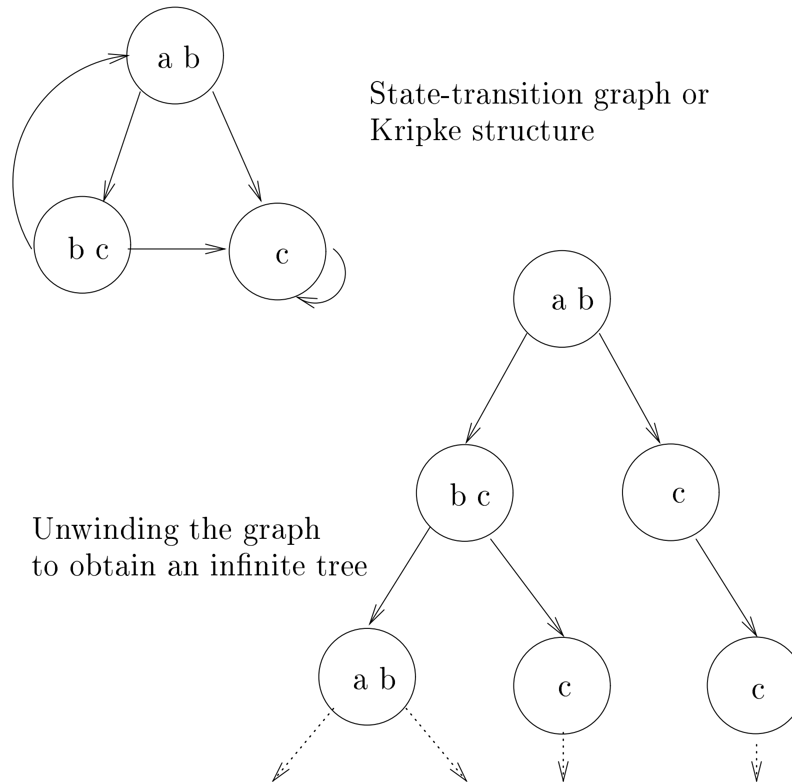

Figure 2.1: Computation trees.

2.1 The Computational Tree Logic CTL*

Conceptually, CTL* formulas describe properties of *computation trees.* The tree is formed by designating a state in a Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state as the root, as illustrated in Figure 2.1. The computation tree shows all the possible executions starting from the initial state.

In CTL*, formulas are composed of *path quantifiers* and *temporal operators.* The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers: **A** ("for all the computation paths") and **E** ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five such operators:

- **X** ("next time") requires that a property hold in the second state of the path.

- The **F** ("eventually" or "in the future") operator is used to assert that a property holds at some state on the path.
- **G** ("always" or "globally") specifies that a property holds at every state on the path.
- The **U** ("until") operator is a bit more complicated, since it is used to combine two properties. It holds if there is a state on the path where the second property holds and, at every preceding state on the path, the first property holds.
- **V** ("releases") also combines two properties, and is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds.

The remainder of this section contains a precise description of the syntax and semantics of CTL*. There are two types of formulas in CTL*: *state formulas*, which are true in a specific state, and *path formulas*, which are true along a specific path. Let $\mathcal{P}$ be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in \mathcal{P}$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are state formulas.
- If f is a path formula, then $\mathbf{E}(f)$ and $\mathbf{A}(f)$ are state formulas.

Two additional rules are needed to specify the syntax of path formulas:

- If f is a state formulas, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$, and $f\mathbf{V}g$ are path formulas.

The *length* of a path or state formula f, denoted as $|f|$, is defined inductively as follows:

- If $f \in \mathcal{P}$, then $|f| = 0$.
- If f is $\neg f_1$, then $|f| = |f_1| + 1$.
- If f is $f_1 \wedge f_2$ or $f_1 \wedge f_2$, then $|f| = |f_1| + |f_2| + 1$.
- If f is $\mathbf{E}f_1$ or $\mathbf{A}f_1$, then $|f| = |f_1| + 1$.
- If f is $\mathbf{X}f_1$, then $|f| = |f_1| + 1$.
- If f is $\mathbf{F}f_1$, then $|f| = |f_1| + 1$.

- If f is $\mathbf{G} f_1$, then $|f| = |f_1| + 1$.
- If f is $f_1 \mathbf{U} f_2$, then $|f| = |f_1| + |f_2| + 1$.
- If f is $f_1 \mathbf{V} f_2$, then $|f| = |f_1| + |f_2| + 1$.

We define the semantics of CTL* with respect to a Kripke structure $M = \langle S, R, L \rangle$, where S is the set of states; $R \subseteq S \times S$ is the *total* transition relation, i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$; and $L : S \rightarrow 2^{\mathcal{P}}$ is a function that labels each state with the set of atomic propositions true in that state. Unless otherwise stated, all of our results apply only to *finite* Kripke structures.

A *path* π in M is an infinite sequence of states $s_0 s_1 \ldots$ such that, for every $i \geq 0$, we have that $(s_i, s_{i+1}) \in R$. We use π_i to denote the *suffix* of π starting at s_i. The state labeling can be extended to paths, that is, the labeling of a path $s_0 s_1 \ldots$ is the sequence $L(s_0) L(s_1) \ldots$. If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along the path π in the Kripke structure M. When the Kripke structure M is clear from the context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

- $M, s \models p$ iff $p \in L(s)$, for $p \in \mathcal{P}$.
- $M, s \models \neg f_1$ iff $M, s \not\models f_1$.
- $M, s \models f_1 \vee f_2$ iff $M, s \models f_1$ or $M, s \models f_2$.
- $M, s \models f_1 \wedge f_2$ iff $M, s \models f_1$ and $M, s \models f_2$.
- $M, s \models \mathbf{E}(g_1)$ iff there is a path π from s such that $M, \pi \models g_1$.
- $M, s \models \mathbf{A}(g_1)$ iff for all path π from s we have that $M, \pi \models g_1$.
- $M, \pi \models f_1$ iff s is the first state of π and $M, s \models f_1$.
- $M, \pi \models \neg g_1$ iff $M, \pi \not\models g_1$.
- $M, \pi \models g_1 \vee g_2$ iff $M, \pi \models g_1$ or $M, \pi \models g_2$.
- $M, \pi \models g_1 \wedge g_2$ iff $M, \pi \models g_1$ and $M, \pi \models g_2$.
- $M, \pi \models \mathbf{X} g_1$ iff $M, \pi_1 \models g_1$.
- $M, \pi \models \mathbf{F} g_1$ iff there exists $k \geq 0$ such that $M, \pi_k \models g_1$.
- $M, \pi \models \mathbf{G} g_1$ iff for all $k \geq 0$ we have that $M, \pi_k \models g_1$.

- $M, \pi \models g_1 \mathbf{U} g_2$ iff there exist $k \geq 0$ such that $M, \pi_k \models g_2$ and, for all $0 \leq j < k$, we have $M, \pi_j \models g_1$.
- $M, \pi \models g_1 \mathbf{V} g_2$ iff for all $k \geq 0$, either $M, \pi_k \models g_2$ or for some $0 \leq j < k$ we have that $M, \pi_j \models g_1$.

It is easy to see that the operators $\vee$, $\neg$, $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{E}$ are sufficient to express any other CTL* formula:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $f \mathbf{V} g \equiv \neg(\neg f \mathbf{U} \neg g)$
- $\mathbf{F} f \equiv (\text{TRUE} \mathbf{U} f)$
- $\mathbf{G} f \equiv \neg \mathbf{F} \neg f$
- $\mathbf{A}(f) \equiv \neg \mathbf{E}(\neg f)$

2.2 CTL and LTL

In this section, we consider two useful sublogics of CTL*: one is a *branching-time* logic and one is a *linear-time* logic. The distinction between the two is how they handle branching in the underlying computation tree. In branching-time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

Computation Tree Logic (CTL) [30] is a restricted subset of CTL* that permits only branching-time operators: each of the temporal operators $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, $\mathbf{U}$, and $\mathbf{V}$ must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL* that is obtained if the following rule is used to specify the syntax of path formulas.

- If f and g are state formulas, then $\mathbf{X} f$, $\mathbf{F} f$, $\mathbf{G} f$, $f \mathbf{U} g$, and $f \mathbf{V} g$ are path formulas.

Linear Temporal Logic (LTL) [30], on the other hand, consists of formulas that have the form $\mathbf{A} f$ where f is a *restricted* path formula, i.e., one in which the only state formulas permitted are atomic propositions. More precisely, a path formulas is now defined as

- If $p \in \mathcal{P}$, then p is a path formula
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X} f$, $\mathbf{F} f$, $\mathbf{G} f$, $f \mathbf{U} g$, and $f \mathbf{V} g$ are path formulas.

It can be shown [29, 31, 50] that the three logics that we have discussed so far have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}p)$. Likewise, there is no LTL formulas that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}p)$. The disjunction of these two formulas $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$ is a CTL* formula that is not expressible in either CTL or LTL.

Because of its structure, CTL is often proposed starting by the following ten operators:

- **AX** and **EX**.
- **AF** and **EF**.
- **AG** and **EG**.
- **AU** and **EU**.
- **AV** and **EV**.

that, in turn, can be expressed in terms of the three operators **EX**, **EG**, and **EU**:

- $\mathbf{AX}f \equiv \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f \equiv \mathbf{E}(\text{TRUE}\,\mathbf{U}f)$
- $\mathbf{AG}f \equiv \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f \equiv \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}(f\mathbf{U}g) \equiv \neg\mathbf{E}(\neg g\mathbf{U}(\neg f \wedge \neg g)) \wedge \neg\mathbf{EG}(\neg g)$
- $\mathbf{A}(f\mathbf{V}g) \equiv \neg\mathbf{E}(\neg f\mathbf{U}\neg g)$
- $\mathbf{E}(f\mathbf{V}g) \equiv \mathbf{E}(\neg(\neg f\mathbf{U}\neg g))$

The four operators that are used most widely are illustrated in Figure 2.2. Each computation tree has the state s_0 as its root.

In turn, LTL is often proposed by omitting the leading path quantifier **A**, that is, dealing only with restricted path formulas. As a consequence, LTL formulas are now assigned semantics with respect to paths, highlighting time linearity, and not with respect to Kripke structures anymore.

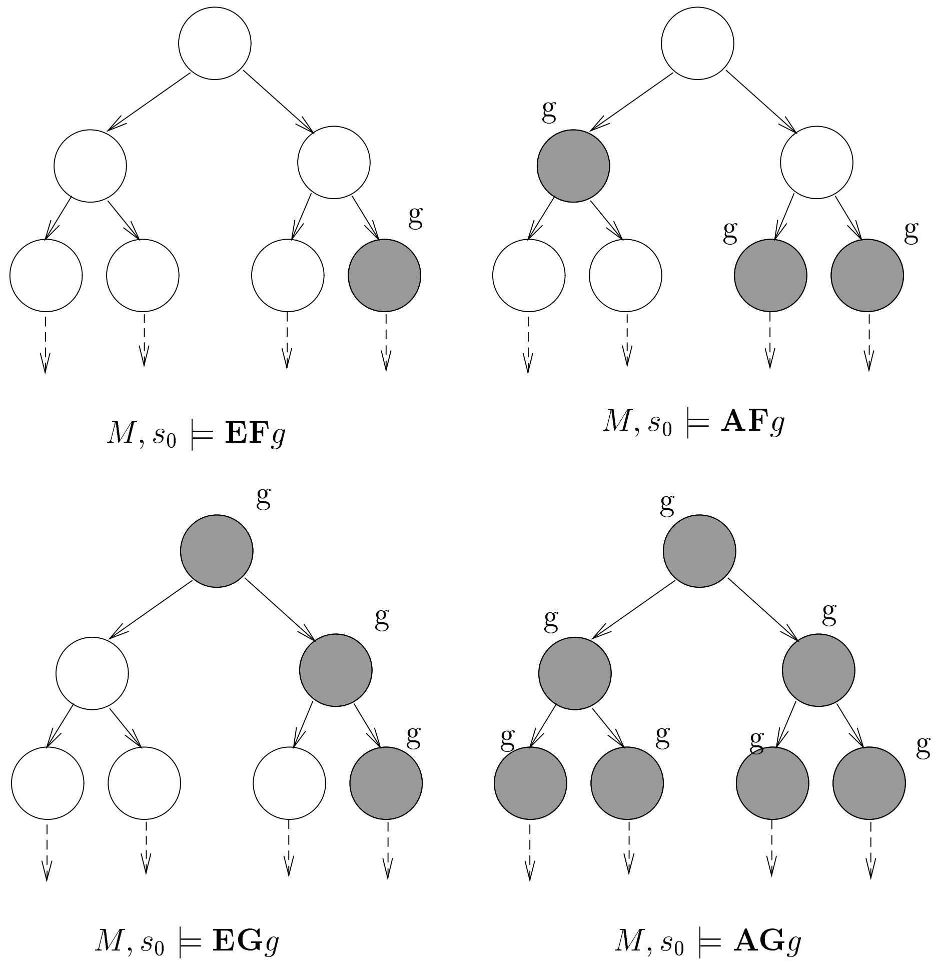

Figure 2.2: Basic CTL operators.

2.3 Fairness

A very important issue is the one of *fairness*. Indeed, in many cases we are only interested in stating properties along fair computation paths. For example, consider a communication protocol that operates over reliable channels having the property that no message is ever continuously transmitted but never received. Such property, which can be expressed in LTL as $\mathbf{G}(\text{msg_snd}) \rightarrow \mathbf{F}(\text{msg_rcv})$, cannot be directly expressed in CTL [29, 31]. In order to deal with fairness, it is necessary to modify CTL semantics. Such a semantics is called *fair semantics*. A *fairness constraint* is an arbitrary set of sets of states. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The fair semantics is then obtained by restricting the path quantifiers to fair paths.

Formally, a *fair Kripke structure* is a tuple $M = \langle S, L, R, F \rangle$ where S, L, and R are defined as before and $F \subseteq 2^S$ is the set of fairness constraints. A path is fair if for every fairness constraint set there exists a state that occurs in the path infinitely often. The fair CTL* semantics is then defined by replacing the semantic definitions of the path quantifiers. In what follows, $M, s \models_F f$ and $M, \pi \models_F g$ denote the fair semantics with respect to the fair constraints F.

- $M, s \models_F \mathbf{E}g_1$ iff there exists a fair path π from s such that $M, \pi \models_F g_1$.
- $M, s \models_F \mathbf{A}g_1$ iff for all the fair paths π from s we have that $M, \pi \models_F g_1$.

Chapter 3

Model Checking

Model checking [22, 69, 51, 9, 84] is a formal technique for verifying finite-state systems with respect to their specifications. Specifications are expressed in some temporal logic, while the system to be checked is looked at as a semantic structure in such logic. The verification process is then carried out by *checking* that the system induces indeed a *model* of the specifications, or by producing a counter-example. In particular, in CTL and LTL model checking, the finite-state systems are represented as Kripke structures and often comes along with a set of initial states. Model checking can then be reformulated as checking that each initial state satisfies the specifications. In the rest of the chapter, we introduce the model checking algorithm for CTL and LTL.

3.1 CTL Model Checking

CTL model checking [22, 69] has been the first experience with this formal verification technique. Let $M = \langle S, R, L \rangle$ be a Kripke structure and f be a CTL formula. In order to check whether some designated initial states satisfy f, we first compute the set of states that satisfy f and then check that the initial states are there contained.

The algorithm is depicted in Figure 3.1 and works by labeling the states of M with the set of f's subformulas they satisfy. Being s a state, the above labeling will be denoted as $label(s)$. Subformulas are analyzed according to an order making g_1 preceding g_2 whenever g_1 is a subformula of g_2. As we have seen before, we can restrict to formulas containing only the following operators: $\neg$, $\vee$, **EX**, **EG**, and **EU**.

The basic case are propositions, which are solved by looking at M's labeling, i.e., for each $s \in S$ we will have $label(s) = L(s)$.

When dealing with formulas of the form $g = \neg f_1$, we label with g those states that are not labeled with f_1. To handle formulas of the type $g = f_1 \vee f_2$,

```
procedure CHECK(f)
begin
  case f of
    proposition:
      for each s ∈ S such that f ∈ L(s) do
        label(s) = label(s) ∪ {f}
    ¬f1:
      CHECK(f1)
      for each s ∈ S such that f1 ∉ label(s) do
        label(s) = label(s) ∪ {f}
    f1 ∨ f2:
      CHECK(f1)
      CHECK(f2)
      for each s ∈ S such that f1 ∈ label(s) or f2 ∈ label(s) do
        label(s) = label(s) ∪ {f}
    EXf1:
      CHECK(f1)
      for each s ∈ S such that ∃s' ∈ S, R(s, s'), and f1 ∈ label(s') do
        label(s) = label(s) ∪ {f}
    E(f1Uf2):
      CHECK(f1)
      CHECK(f2)
      CHECKEU(f1, f2)
    EG(f1):
      CHECK(f1)
      CHECKEG(f1)
  end
end
```

Figure 3.1: Procedure for CTL model checking. Before calling the procedure, the label sets have to be assigned the empty set.

```
procedure CheckEU(f1, f2)
begin
  T := {s : f2 ∈ label(s)}
  for every s ∈ T do label(s) := label(s) ∪ {E(f1 U f2)}
  while T ≠ ∅ do
    select s ∈ T
    T := T \ {s}
    for every t such that R(t, s) do
    begin
      if E(f1 U f2) ∉ label(t) and f1 ∈ label(t) then
      begin
        label(t) := label(t) ∪ {E(f1 U f2)}
        T := T ∪ {t}
      end
    end
  end
end
```

Figure 3.2: Procedure for labeling the states satisfying $\mathbf{E}(f_1 \mathbf{U} f_2)$.

we label with g states that are labeled either with f_1 or with f_2.

The case $g = \mathbf{EX} f_1$ is handled by labeling with g states that are connected through R to some state labeled with f_1.

For formulas of the form $g = \mathbf{E}(f_1 \mathbf{U} f_2)$, we first label with g states that are labeled with f_2. Then we work backwards, labeling with g those states that are labeled with f_1 and are connected through R to some state labeled with g. The algorithm implementing this backward labeling is depicted in Figure 3.2, and takes time $\mathcal{O}(|S| + |R|)$.

The case in which $g = \mathbf{EG} f_1$ is slightly more complicated. First, we restrict our attention on the graph $G = \langle S', R' \rangle$ obtained by deleting from S the states that are not labeled with f_1 and restricting R accordingly, i.e., $R' = \{(s_1, s_2) : s_1, s_2 \in S' \text{ and } R(s_1, s_2)\}$. Second, G is decomposed in nontrivial maximal strongly connected components. A *strongly connected component* (SCC) C is a subgraph such that each node in C can be reached from each other node in C through paths entirely contained in C. C is *nontrivial* if it contains at least one edge. C is *maximal* if there does not exist a strictly larger SCC C' containing C. All states belonging to some SCC are then labeled with g. Finally, we work backward, as we have done for $\mathbf{E}(f_1 \mathbf{U} f_2)$, and label with g all the states in G from which some SCC can be reached. The algorithm for labeling states with $\mathbf{EG} f_1$ is depicted in Figure 3.3. This algorithm also takes time $\mathcal{O}(|S| + |R|)$ and, therefore, the overall complexity is $\mathcal{O}(|f| \cdot (|S| + |R|))$.

procedure $\text{CHECKEG}(f_1)$
begin
  $S' := \{s \in S : f_1 \in \mathit{label}(s)\}$
  $R' := \{(s_1, s_2) \in R : s_1, s_2 \in S'\}$
  $SCC := \{C : C \text{ is a maximal non trivial SCC in } \langle S', R' \rangle\}$
  $T := \cup_{C \in SCC}\{s : s \in C\}$
  for every $s \in T$ **do** $\mathit{label}(s) := \mathit{label}(s) \cup \{\mathbf{EG} f_1\}$
  while $T \neq \emptyset$ **do**
  begin
    $T := T \setminus \{s\}$
    for every t **such that** $t \in S'$, $R'(t, s)$, **and** $\mathbf{EG}(f_1) \notin \mathit{label}(t)$ **do**
    begin
      $\mathit{label}(t) := \mathit{label}(t) \cup \mathbf{EG} f_1$
      $T := T \cup \{t\}$
    end
  end
end

Figure 3.3: Procedure for labeling states satisfying $\mathbf{EG} f_1$.

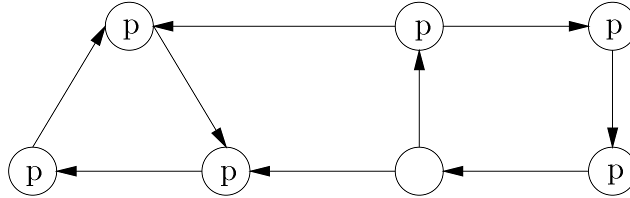

Figure 3.4: A Kripke structure.

As an example, consider to compute $\mathbf{EG}p$ in the Kripke structure shown in Figure 3.4. After the first step, only the nodes already labeled with p are kept. These nodes are shown in Figure 3.5. The next step is then to decompose into maximal nontrivial SCCs the graph obtained in the previous step. The only SCC is greyed in Figure 3.6. All these nodes have to be labeled with $\mathbf{EG}p$. Finally, we work backward to label with $\mathbf{EG}p$ all the nodes, actually just one, from which the SCC is reachable. The result of this computation is then depicted in Figure 3.7.

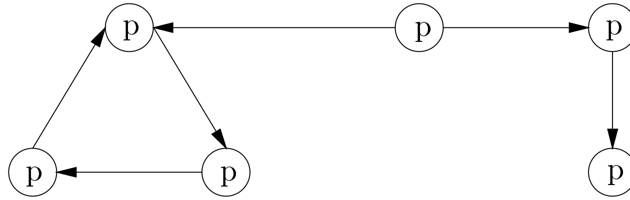

Figure 3.5: Computing $\mathbf{EG}p$, pruning nodes not satisfying p.

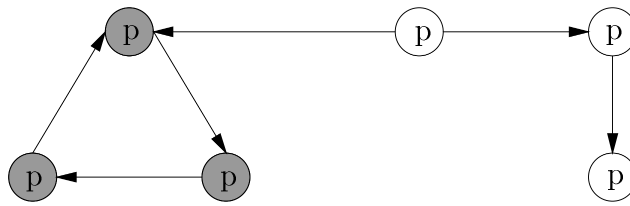

Figure 3.6: Computing $\mathbf{EG}p$, decomposition in SCCs.

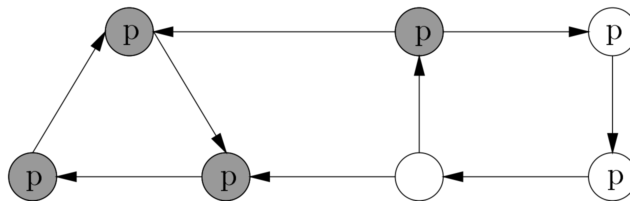

Figure 3.7: Computing $\mathbf{EG}p$, backward traversal from the SCCs.

3.1.1 Fair CTL Model Checking

In this section we show how the CTL model checking algorithm can be extended to deal with fair Kripke structures $M = \langle S, R, L, F \rangle$, where $F = \{F_1, \ldots, F_n\}$.

The key point of this extension is being able to deal with formulas of the form $\mathbf{EG} f_1$, which is carried out through the procedure FAIRCHECKEG. The algorithm resembles the one given for regular Kripke structures, from which it differs only in the partitioning into SCCs, which are now are also required to be fair. An SCC C is *fair* if, for each $F_i \in F$, we have that $C \cap F_i \neq \emptyset$. All the states belonging to such SCCs are then labeled with $\mathbf{EG} f_1$ and, finally, all the states that are labeled with f_1 and from which there exists a path leading to some fair maximal non trivial SCC are labeled with $\mathbf{EG} f_1$. The complexity of the above step is $\mathcal{O}((|S| + |R|) \cdot |F|)$, since it is necessary to check whether the SCCs are fair or not.

Leaning on the above algorithm, one can handle the remaining cases. First of all, the new proposition *ExistsFairPath* is introduced and all the states satisfying $\mathbf{EG}(\text{TRUE})$ according to the fair semantics, that is, those from which a fair path departs, are labeled with *ExistsFairPath*. Then, when dealing with a proposition p, one has to label with p all the states s such that $p \in L(s)$ and $\mathit{ExistsFairPath} \in \mathit{label}(s)$. The cases for dealing with the propositional operators $\neg$ and $\vee$ are the same as before. The remaining temporal operators are computed as

- FAIRCHECKEX(f_1) = CHECKEX($f_1 \wedge \mathit{ExistsFairPath}$)
- FAIRCHECKEU(f_1) = CHECKEU($f_1, f_2 \wedge \mathit{ExistsFairPath}$)

All of the previous computations have complexity $\mathcal{O}((|S| + |R|) \cdot |F|)$ and, therefore, the overall complexity is $\mathcal{O}((|S| + |R|) \cdot |F| \cdot |f|)$.

3.2 LTL Model Checking

LTL model checking [51, 84] requires a completely different approach with respect to the one we have just described for CTL. First, it requires the construction of a graph, called the *tableau*, whose paths encode all the models of the *negation* of the LTL specifications. The tableau and the Kripke structure are then searched for compatible paths: finding one means that the Kripke structure does not satisfy the specifications. In what follows, we describe two instances of the above approach. In the first one, the tableau is exactly a graph, while in the second one it is an automaton on infinite words.

3.2.1 Graph-based LTL Model Checking

Let $M = \langle S, R, L\rangle$ be a Kripke structure and f be the LTL specifications, that is, a restricted path formula. Unless otherwise stated, in this chapter we deal with restricted path formulas built starting from atomic propositions and the operators $\neg, \vee, \mathbf{X}$, and $\mathbf{U}$. Below we describe the approach considered in [55].

The definition of the tableau requires some background work. The *closure* of a formula g, denoted as $\overline{g}$, is the smallest set of formulas such that

- $g \in \overline{g}$.
- For every $g_1 \in \overline{g}$, then $\neg g_1 \in \overline{g}$.
- For every $g_1 \vee g_2 \in \overline{g}$, then $g_1, g_2 \in \overline{g}$.
- For every $g_1\mathbf{U}g_2 \in \overline{g}$, then $g_1, g_2, \mathbf{X}(g_1\mathbf{U}g_2) \in \overline{g}$.

In the above, to keep the closure finite, we identify $\neg\neg g_1$ with g_1. An *atom* for g is a subset $A \subseteq \overline{g}$ such that

- For every $g_1 \in \overline{g}$, $g_1 \in A$ iff $\neg g_1 \notin A$.
- For every $g_1 \vee g_2 \in \overline{g}$, $g_1 \vee g_2 \in A$ iff either $g_1 \in A$ or $g_2 \in A$.
- For every $g_1\mathbf{U}g_2 \in \overline{g}$, $g_1\mathbf{U}g_2 \in A$ iff either $g_2 \in A$ or $g_1, \mathbf{X}(g_1\mathbf{U}g_2) \in A$.

The direct graph representing the tableau for a formula g is then constructed as follows

- The nodes of the graph are the atoms of g.
- There is an edge from the atom A to the atom B iff for every $\mathbf{X}g_1 \in \overline{g}$ we have that $\mathbf{X}g_1 \in A$ iff $g_1 \in B$.

As an example, in Figure 3.8 the tableau for $g = p\mathbf{U}q$ is depicted. To simplify the presentation only positive formulas, i.e., those not starting with $\neg$, are mentioned in the states. The idea underlying such construction is that infinite paths starting from some node s induce, by removing the nonpropositional formulas, models of the formulas contained in s. Unfortunately, the construction achieves only part of this aim. In fact, the graph takes care of local consistency, i.e., propositional and next state consistency, but has no way of controlling the fulfillment of until formulas. This means that it can be possible for some paths to have a node containing a until formula $f_1\mathbf{U}f_2$ without any one of the subsequent nodes contain f_2. We will see later how this can be dealt with. A node is said to be fulfilling with respect to a until formula $f_1\mathbf{U}f_2 \in \overline{g}$ if either $f_1\mathbf{U}f_2$ does not belong to the node or f_2 belongs to it. A path is called *fulfilling* if, for every until formula $f_1\mathbf{U}f_2 \in \overline{g}$, it contains infinitely many nodes fulfilling $f_1\mathbf{U}f_2$. Fulfilling paths do induce models.

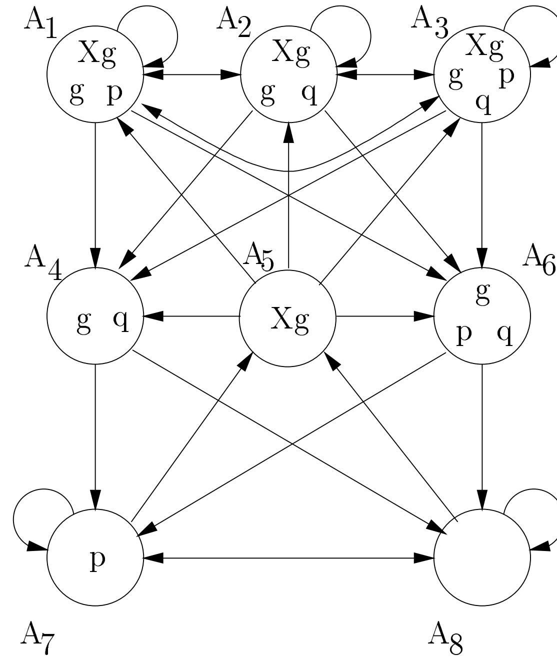

Figure 3.8: Graph-based tableau for $g = p\mathbf{U}q$.

Theorem 3.2.1 *Let $\mathcal{T}_f$ be the tableau for the LTL formula f, and $\pi = s_0 s_1 \ldots$ be a fulfilling path in it. Moreover, let $\pi' = r_0 r_1 \ldots$ be the restriction of π to $\mathcal{P}$, i.e., for every $i \geq 0$, let $r_i = s_i \cap \mathcal{P}$. Then $\pi' \models s_0$.*

Moreover, all the models are encoded through some fulfilling path.

Theorem 3.2.2 *Let $\mathcal{T}_f$ be the tableau for the LTL formula f, and $\xi \models f$. Then, in $\mathcal{T}_f$, there exists a fulfilling path $\pi = s_0 s_1 \ldots$ such that $f \in s_0$ and ξ is the restriction of π to $\mathcal{P}$.*

Through the above construction, one can build the tableau for the negation of the specifications. The next step is producing the *behavior* graph that encodes the "wrong" paths, i.e., paths that are both in the Kripke structure and in the tableau for the negation of the specifications. The behavior graph is built as follows

- The nodes of the behavior graph are the pairs (s, A), where $s \in S$ and A is an atom consistent with s, i.e., such that the $A \cap \mathcal{P} = L(s)$.
- There is a direct edge from (s, A) to (s', A') iff $R(s, s')$ and there is an edge connecting A to A'.

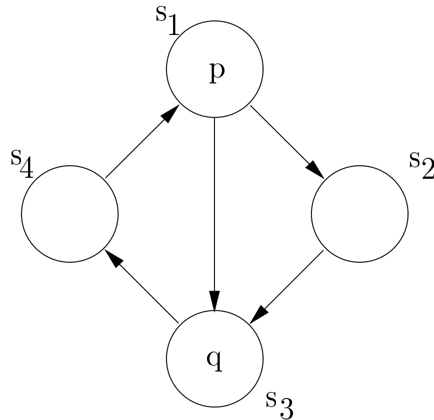

Figure 3.9: A system.

For example, let us consider the system shown in Figure 3.9, where s_1 is the initial state. If one wants to verify this system with respect to the specifications $\neg p\mathbf{V}\neg q$, the behavior graph one obtains is the one shown in Figure 3.10, combining the system with the tableau for $p\mathbf{U}q = \neg(\neg p\mathbf{V}\neg q)$ shown in Figure 3.8

Finally, according to Theorem 3.2.1 and Theorem 3.2.2, the behavior graph has to be searched for infinite paths related to fulfilling paths in the tableau and starting with a node (s, A) such that the negation of the specification $\neg f$ is in A. However, since the fulfillment of the until formulas is not guaranteed by the tableau construction, this step is rather expensive for it involves computing fulfilling maximal nontrivial SCCs. An SCC C of the behavior graph is *fulfilling* if every $f_1\mathbf{U}f_2 \in \overline{\neg f}$ is fulfilled by an atom A such that $(s, A) \in C$ for some state s. Note that a fulfilling SCC, due to its connectivity and being fulfilling, allows for extracting fulfilling paths by allowing for the fulfillment of until formulas whenever this is required. This means that "wrong" paths depart from states s such that there exists a node (s, A) with $\neg f \in A$ that is connected to a fulfilling SCC. With respect to the example shown in Figure 3.10, a fulfilling SCC is $\{\langle s_1, A_7\rangle, \langle s_2, A_5\rangle, \langle s_3, A_4\rangle, \langle s_4, A_8\rangle\}$. Moreover, it is reachable from $\langle s_1, A_1\rangle$ and $g \in A_1$. This means that the system does not satisfy its specifications.

The algorithm we have just depicted has time complexity proportional to $(|S| + |R|) \cdot 2^{\mathcal{O}(|f|)}$, where the exponential factor is introduced by the tableau construction for the specifications. In general, it can be shown that the LTL model checking problem is PSPACE-complete [23]. Finally, it has to be mentioned that the above tableau construction is easy to present but immediately realizes the worst case exponential complexity. However, more efficient incre-

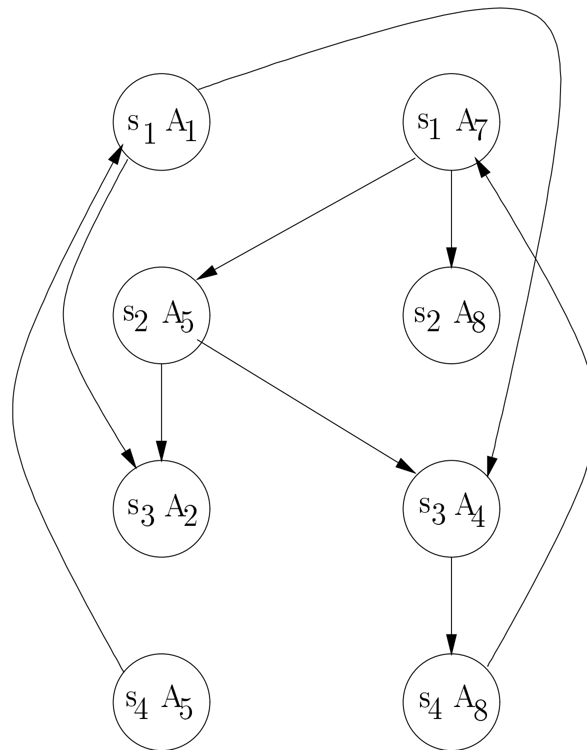

Figure 3.10: Behavior graph.

mental algorithms are available [55, 25].

3.2.2 Automata-based LTL Model Checking

An alternative tableau construction has been given by Vardi and Wolper [84, 85] by exploiting the close relationship existing between linear temporal logic and automata on infinite words [81]. In this approach, both the system and the specification are turned into automata. The system automaton recognizes all the executions of the system, while the specification one recognizes all the models of the *negation* of the specifications. Verification then amounts to check that the two automata do not recognize any common words.

The automata they use for representing the tableau are Büchi automata [8]. A *Büchi automaton* is a tuple $\mathcal{A} = \langle \Sigma, S, S_0, \rho, F \rangle$ where

- Σ is the finite *alphabet*.
- S is a finite set of *states*.
- $S_0 \subseteq S$ is a set of *initial states*.
- $\delta : S \times \Sigma \to 2^S$ is the *transition function*.
- $F \subseteq S$ is a set of *accepting states*.

An *execution* ρ of $\mathcal{A}$ on an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0 s_1 \ldots$ such that

- $s_0 \in S_0$.
- For all $i \geq 0$, we have $s_{i+1} \in \delta(s_i, a_i)$.

An execution ρ is said to be *accepting* if it contains an accepting state infinitely often. We say that a word w is *accepted* by $\mathcal{A}$ if there is an accepting execution of $\mathcal{A}$ over w. The *language* of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of the words accepted by $\mathcal{A}$.

A useful generalization of Büchi automata, which does not increase their expressive power, is given by *generalized Büchi automata*, that is, Büchi automata with a, possibly empty, *set of accepting sets* $F = \{F_1, F_2, \ldots, F_n\}$. In this case, an execution $\rho = s_0 s_1 \ldots$ is accepting if it contains infinitely often a state *from each* accepting set. As opposed to generalized Büchi automata, those with one accepting states are now called *simple* Büchi automata.

Even though more effective constructions to turn LTL formulas into Büchi automata have been given [35, 25], a very simple way to present this translation is to exploit the graph-based tableau construction depicted in the previous section. Indeed, a generalized Büchi automaton $\mathcal{A}_g$ recognizing all the models

of the LTL formula g can be obtained by the graph-based tableau for g by labeling the edges with the propositional information contained in the nodes and by making the accepting states play the role of the fulfilling SCCs, in the sense that fulfilling paths are now seen as accepting executions. Formally, we have that

- The alphabet is $2^{\mathcal{P}}$
- The states of the automaton are the node of the graph.
- The initial states are those containing g.
- The transition function of the automaton corresponds to the edges of the graph labeled with the propositional information contained in the nodes edges depart from, i.e., $\delta(s, s \cap \mathcal{P}) = \{s'$ such that there is an edge connecting s to $s'\}$.
- For each formula of the form $f_1 \mathbf{U} f_2$ occurring in $\overline{g}$, an accepting set $F_{f_1 \mathbf{U} f_2}$ of states containing either $\neg(f_1 \mathbf{U} f_2)$ or f_2 is defined.

As an example, in Figure 3.11 the automaton-based tableau for $p\mathbf{U}q$ correspondent to the graph-based tableau shown in Figure 3.8 is depicted. Initial states are denoted through arrows that do not depart from any state, while accepting states (we have only one accepting set) are denoted as double circles.

Once we build the automaton $\mathcal{A}_{\neg f}$ for the negation of the LTL specification f, we also need to turn the Kripke structure $M = \langle S, R, L\rangle$ into a simple Büchi automaton $\mathcal{A}_M = \langle \Sigma_M, Q_M, Q_{M0}, \delta_M, F_M\rangle$, whose language is the set of the (labels of the) paths of M. This is done as follows:

- The alphabet Σ_M is $2^{\mathcal{P}}$.
- The set Q_M of states is the set S of states of the Kripke structure.
- The initial states Q_{M0} are the whole set of states Q_M.
- The transition function of the automaton is induced by the transition relation of the Kripke structure, i.e., $\delta_M(s, L(s)) = \{s' \in S : R(s, s')\}$.
- There is one accepting set containing all the states, i.e., $F_M = Q_M$.

In Figure 3.12 the automaton corresponding to the Kripke structure depicted in Figure 3.9 is shown.

The next step is to compute the synchronous product $\mathcal{A}_{\neg f} \times \mathcal{A}_M$, that is, the analogous of the behavior graph, in order to accept the intersection of the related languages. Let us start by translating the generalized Büchi automaton $\mathcal{A}_{\neg f} = \langle \Sigma_{\neg f}, Q_{\neg f}, Q_{\neg f_0}, \delta_{\neg f}, F_{\neg f} = \{F_0, \ldots, F_{k-1}\}\rangle$ into an equivalent simple one $\mathcal{A}'_{\neg f} = \langle \Sigma'_{\neg f}, Q'_{\neg f}, Q'_{\neg f_0}, \delta'_{\neg f}, F'_{\neg f}\rangle$. This is done as follows:

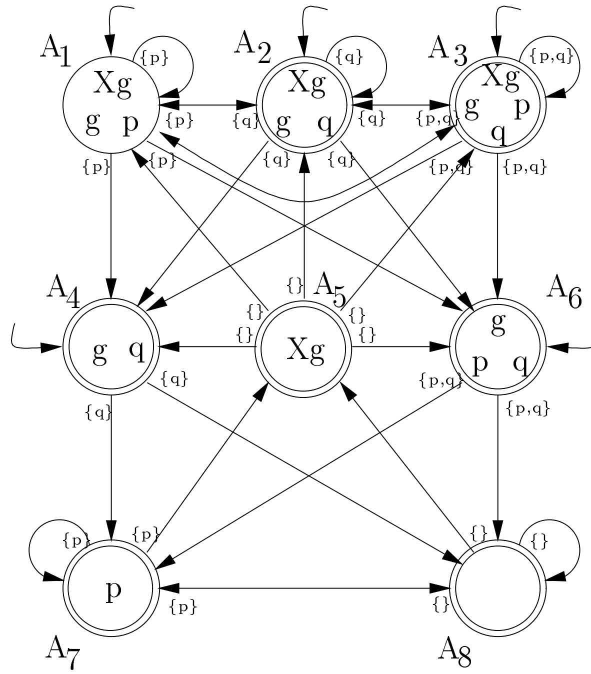

Figure 3.11: Automata-based tableau for $p\mathbf{U}q$.

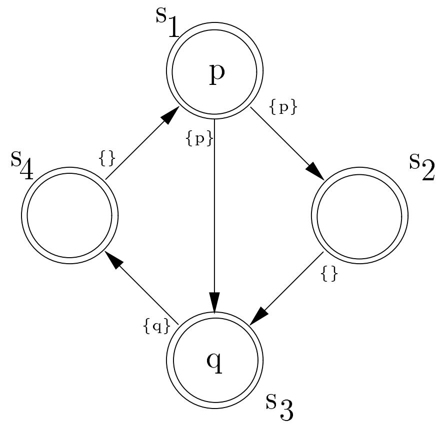

Figure 3.12: A system automaton.

- The alphabet $\Sigma'_{\neg f}$ is $2^{\mathcal{P}}$.
- The set $Q'_{\neg f}$ of states is $Q_{\neg f} \times \{0, \ldots, k-1\}$.
- The set $Q'_{\neg f_0}$ of initial states is $Q_{\neg f_0} \times \{0\}$.
- The transition function, for $i = 0, \ldots, k-1$, is

$$\delta'_{\neg f}((s,i),a) = \begin{cases} \{(s', (i+1)\mathrm{mod}k) : s' \in \delta_{\neg f}(s,a)\} & \text{if } s \in F_i \\ \{(s', i) : s' \in \delta_{\neg f}(s,a)\} & \text{if } s \notin F_i \end{cases}$$

- The set F' of accepting states is $F_0 \times \{0\}$.

The tricky part is to ensure that every F_i is visited infinitely often. In order to achieve this, k copies of the automaton are considered. We move from the copy i to the copy $(i+1)\mathrm{mod}k$ when we hit a state $s \in F_i$. Otherwise, we keep moving in the ith copy. In this way, visiting infinitely often $F_0 \times \{0\}$ amounts to visiting infinitely often every F_i.

The synchronous product between $\mathcal{A}'_{\neg f}$ and $\mathcal{A}_M$ is then defined as follows

- The alphabet is $2^{\mathcal{P}}$.
- The set of states is the product $Q'_{\neg f} \times Q_M$ of the two sets of states.
- The set of initial states is the product $Q'_{\neg f_0} \times Q_{M0}$ of the two sets of initial states.
- The transition function maps each $((s_1, q_1), a) \in (Q_{\neg f} \times Q_M) \times 2^{\mathcal{P}}$ into the set $\{(s_2, q_2) : s_2 \in \delta'_{\neg f}(s_1, a) \text{ and } q_2 \in \delta_M(q_1, a)\}$.
- The set of accepting states is induced by the accepting states of $\mathcal{A}'_{\neg f}$, and is defined as $F'_{\neg f} \times Q_M$.

In Figure 3.13, the automaton corresponding to the composition of the system automaton of Figure 3.12 and the specification automaton of Figure 3.11 is shown.

The last step is to check the emptiness of $\mathcal{L}(\mathcal{A}'_{\neg f} \times \mathcal{A}_M)$: each word belonging to the above language is indeed a witness of the violation of f by M. This check can be carried out in time linear in the size of the product automaton [24] by looking for some accepting state reachable from itself and from some initial state.

Finally, let us note that rather than expressing the specifications as LTL formulas, one can express them directly as Büchi automata. In this way, if from one end one looses the simplicity and conciseness of LTL specifications, from the other end one gains in expressive power. Indeed, for example, LTL

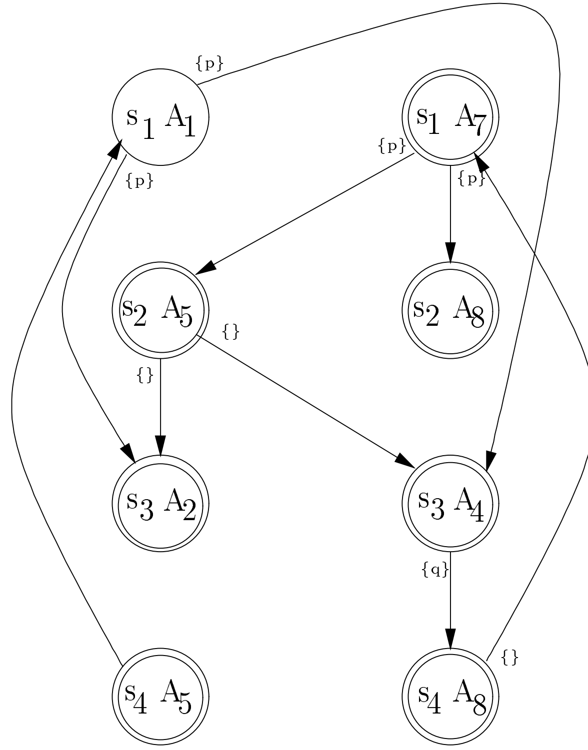

Figure 3.13: Synchronous product.

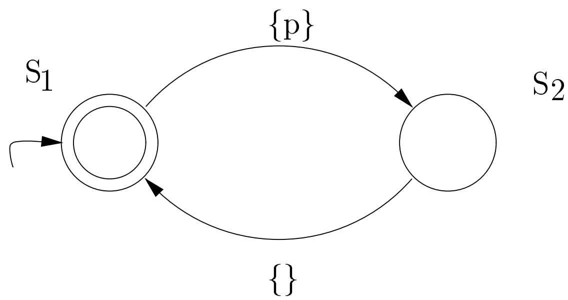

Figure 3.14: Büchi automaton recognizing the set of sequences in which p holds at the even places.

has not counting capabilities: no LTL formula can express the set of sequences in which the proposition p is true in every even state [80], while this set can be simply described by the Büchi automaton shown in Figure 3.14. Note that the upper arrow labeled with p denotes a set of transitions departing from s_1, namely, those related to subsets of $\mathcal{P}$ containing p. In turn, the lower arrow denotes all the transitions departing from s_2.

Chapter 4

Symbolic Model Checking

The algorithmic nature of model checking makes it fully automatic, convenient to use and very attractive to practitioners. On the other hand, model checking is very sensitive to the size of the system. This problem—known as *state-space explosion problem*—is the major limitation of model checking. One of the most important developments in this area is the discovery of *symbolic* model-checking methods [60, 9]. In particular, the use of ordered binary decision diagrams [6] for model representation has yielded model-checking tools that can handle systems with 10^{120} states and beyond [21]. In this chapter we start by introducing the ordered binary decision diagrams and show how to exploit them in order to devise symbolic model checking algorithms for both CTL and LTL.

4.1 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are an economic and efficient way of representing Boolean functions. Thus, through suitable encodings, OBDDs can represent any sets by representing their characteristic functions. Unlike other representations for Boolean functions, OBDDs have the advantage of a canonical form, that is, two equivalent Boolean functions are represented by the same OBDD. This canonical form can be obtained by imposing a total order on the set of Boolean variables. This means that for checking whether two Boolean functions are equivalent, or two sets are equal, we have to compute the related OBDDs and check, in constant time, whether they are the same. According to the same logic, we can check whether a formula is unsatisfiable by computing its OBDD and comparing it with the OBDD for FALSE.

OBDDs are an evolution of *ordered binary decision trees*. An ordered binary decision tree for a set of Boolean variables $\{v_1, \ldots, v_m\}$ is a complete labeled binary tree of height $m + 1$, where the root has height 1. Each nonleaf

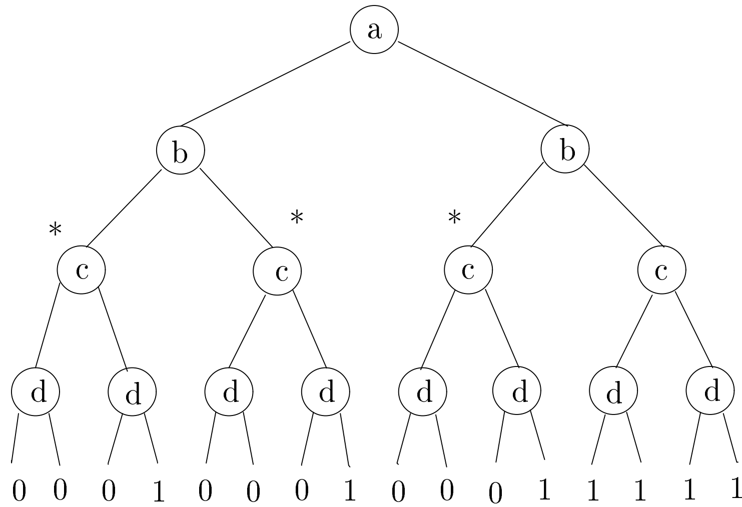

Figure 4.1: A binary decision tree.

node n of height i is labeled with v_i and has two children, $low(n)$ corresponding to the case when v_i is assigned to FALSE (often denoted as 0), and $high(n)$ corresponding to the case when v_i is assigned to TRUE (often denoted as 1). Each leaf node is labeled either with 0 or 1. In Figure 4.1, it is shown the ordered binary decision tree for the function $f(a, b, c, d) = a \wedge b \vee c \wedge d$. The left children are the low ones, while the right children are the high ones. Given a truth assignment for $\{v_1, \ldots, v_n\}$, one can compute the truth value of the function represented by a binary decision tree by traversing it from the root towards the leaves. At each nonleaf node n labeled with v_i, one descends to $high(n)$ if v_i is assigned to TRUE, or to $low(n)$ otherwise.

Ordered binary decision trees are not a very concise representation for Boolean functions. Indeed, they usually contain a lot of redundancy, i.e., distinct but isomorphic subtrees. Two subtrees are isomorphic is there exists a bijective function h mapping nonleaf nodes into nonleaf nodes and leaf nodes into leaf nodes such that for each node n of the tree, the label of n is the same of the label of $h(n)$ and, for nonleaf nodes, $h(low(n)) = low((h(n))$, and $h(high(n)) = high((h(n))$. For example, in Figure 4.1, the nodes marked with an asterisk are isomorphic. Thus, we can obtain a more concise representation by merging together isomorphic subtrees. The result is not a tree anymore, but a *ordered binary decision graph*. More precisely, a ordered binary decision graph is a rooted labeled directed acyclic graph with two types of nodes, terminal nodes and nonterminal nodes. As in the case of the binary decision trees, each nonterminal node n labeled with the variable v_i has two successors,

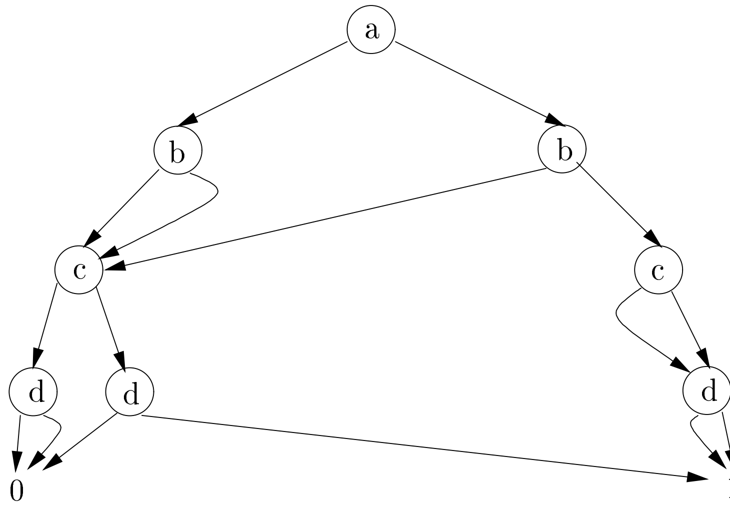

Figure 4.2: A binary decision diagram.

namely *low*(n) and *high*(n), while each terminal node is labeled either with TRUE or FALSE. Each node n in a binary decision diagram defines a Boolean function $f_n(v_1, \ldots, v_n)$ as follows:

- If n is a terminal node:
  - If n is labeled with TRUE, $f_n(v_1, \ldots, v_n) = \text{TRUE}$
  - If n is labeled with FALSE, $f_n(v_1, \ldots, v_n) = \text{FALSE}$
- If n is a nonterminal node labeled with v_i, we have $f_n(v_1, \ldots, v_n) = (\neg v_i \wedge f_{low(n)}(v_1, \ldots, v_n)) \vee (v_i \wedge f_{high(n)}(v_1, \ldots, v_n))$

In Figure 4.2 The binary decision diagram related to the binary decision tree of Figure 4.1 is depicted.

Beside the redundancy introduced by isomorphic subtree, there is redundancy introduced by redundant nodes, that is, those such that their low and high nodes are identical. A redundant node n may be removed, and each arc leading to it can be replaced by one leading to $low(n)(= high(n))$. This last step leads to the OBDD shown in Figure 4.3.

To summarize, the canonical OBDD form is a labeled directed acyclic graph that can be obtained from the ordered binary decision tree by the following two steps:

- Merge isomorphic subtrees into a single subtree.
- Eliminate any nodes whose left and right children are isomorphic.

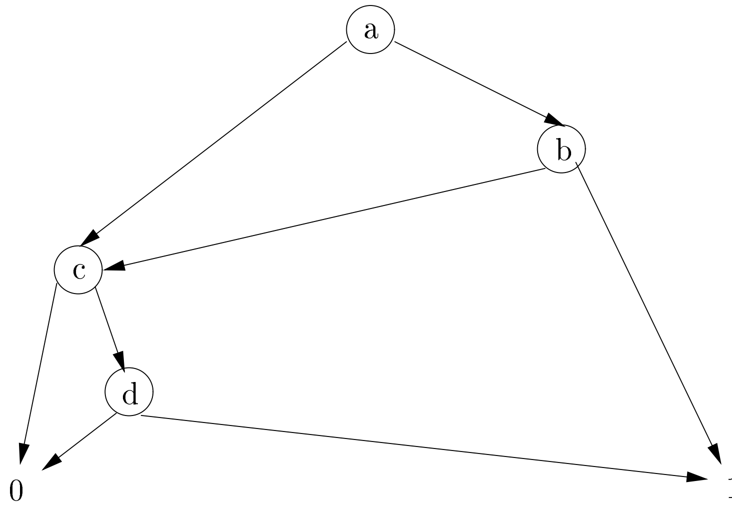

Figure 4.3: An ordered binary decision diagram.

Bryant [6] described an algorithm, called REDUCE, to compute the above steps in a bottom up fashion in time linear with respect to the size of the initial ordered binary decision tree.

Unfortunately, OBDDs are very sensitive to the variable ordering, which is the key point to obtain the canonical form. For example, while the OBDD in Figure 4.3 has 4 nonterminal nodes, by choosing the order a, c, b, d one obtains the 7-nonterminal-node OBDD of Figure 4.4. Moreover, it can be proved that some functions have exponential size OBDDs, no matter what the variable ordering is. One classic example is the function encoding the integer multiplication between two bit vectors [7].

Beside the reduction of an ordered binary decision tree, Bryant also described an algorithm called APPLY that applies an arbitrary Boolean operation $f : \{0,1\}^2 \to \{0,1\}$ to two OBDDs. The operation f can be any Boolean function of two variables.

The algorithm works by recursive descent on the two OBDDs and uses an hash table to store the result returned for each pair of nodes, so that the result for a given pair only has to be computed once. As a result, APPLY has quadratic complexity. Moreover, note that the negation of an OBDD o can be performed in linear complexity, being $\neg o = \text{APPLY}(XOR, o, 1)$.

To see how APPLY works when given a pair of nodes p and q, let us break the problem of computing $f(p, q)$ into cases. First, if both p and q are terminals, then we simply return $f(p, q)$.

If p and q are not terminal and are labeled with the same variable v_i,

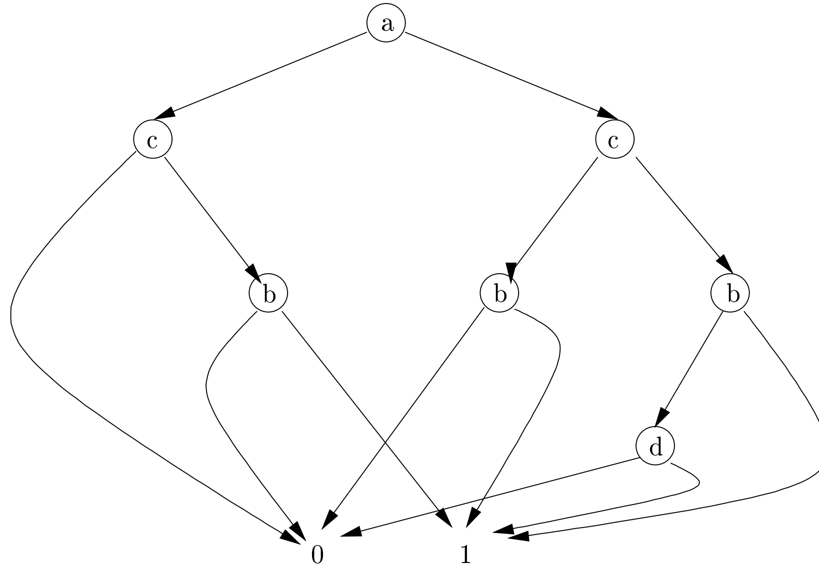

Figure 4.4: Changing the variable ordering.

then we call APPLY recursively to compute $l' = f(low(p), low(q))$ and $h' = f(high(p), high(q))$. If $l' = h'$ then we return l', otherwise we return a node n labeled with v_i such that $low(n) = l'$ and $high(n) = h'$.

If p and q are not terminal and are labeled with different variables v_i and v_j we have two symmetric cases. Let us consider that $v_i < v_j$. Then q does not depend on v_i and it can be shown that $f(p, q)$ correspond to a node n labeled with v_i such that $low(n) = f(low(p), q)$ and $high(n) = f(high(n), q)$.

Bryant also gave the algorithm COMPOSE that can be used for composing two Boolean functions $p : \{0,1\}^n \rightarrow \{0,1\}$ and $q : \{0,1\}^m \rightarrow \{0,1\}^n$. Since $\exists v.p = p(\text{FALSE}) \vee p(\text{TRUE})$, the function COMPOSE could be easily adapted in order to compute $\exists \vec{v}.p$, for a vector $\vec{v}$ of Boolean variables. Therefore, the COMPOSE algorithm could be used to symbolically evaluate $\mathbf{EX}g$ as $\exists \vec{v}.\mathbf{R} \wedge \mathbf{g}$, where $\mathbf{R}$ is the Boolean function representing the transition relation and $\mathbf{g}$ is the function representing the states labeled with g. However, since the performances of the model checker heavily rely on computing $\mathbf{EX}g$, the ad hoc function ANDEXIST has been developed.

The ANDEXISTS function takes in input a vector $\vec{v}$ of Boolean variables and two OBDDs p and q, and computes $\exists \vec{v}.p \wedge q$. It is basically a modification of the APPLY function, where $f(p, q) = p \wedge q$. Indeed, rather that computing the entire OBDD for $p \wedge q$ before applying the existential quantifier, one applies the existential quantifier to the partial results. More in detail, before we return a result, say $r = \langle v, l, h \rangle$, we test the variable v to see if it occurs in the vector

$\vec{v}$. If it does, we call APPLY to compute $l \vee h$, which is exactly $\exists v.f_r$.

4.2 Symbolic Kripke structures

In this section, we show how Kripke structures can be thought of as Boolean functions, that is *symbolically*, and therefore efficiently represented and handled through OBDDS.

The first step is the representation of a finite set D of cardinality $|D| \leq 2^n$. By using a suitable encoding $\psi : \{0,1\}^n \to D$, D can be represented through its characteristic function over the Boolean variables $\{v_1 \ldots, v_n\}$ as

$$f_D(v_1, \ldots, v_n) \text{ iff } \psi(v_1, \ldots, v_n) \in D$$

For example, assuming the encoding $\psi(0,0) = a$, $\psi(0,1) = b$, $\psi(1,0) = c$, $\psi(1,1) = d$, the subset $\{a, b, d\}$ of $\{a, b, c, d\}$ can be represented as the function $\neg v_0 \wedge \neg v_1 \vee \neg v_0 \wedge v_1 \wedge v_0 \wedge v_1$.

A relation R on the domains $D_1, \ldots, D_n$ can be similarly represented as

$$f_R(\vec{v}_1, \ldots, \vec{v}_n) \text{ iff } R(\psi_1(\vec{v}_1), \ldots, \psi_n(\vec{v}_n))$$

where the ψ_i's are the encodings of the domains D_i's. For example, the relation $(a + 1) \equiv_4 b$ between 2-bit numbers, that is, $\{(0,1), (1,2), (2,3), (3,0)\}$, can be encoded as $\{(00, 01), (01, 10), (10, 11), (11, 00)\}$ by exploiting the previous encoding. Therefore, it can be represented through a function of the variables $\{v_0, v_1, v_2, v_3\}$ as $\neg v_0 \wedge \neg v_1 \wedge \neg v_2 \wedge v_3 \vee \neg v_0 \wedge v_1 \wedge v_2 \wedge \neg v_3 \vee v_0 \wedge \neg v_1 \wedge v_2 \wedge v_3 \vee v_0 \wedge v_1 \wedge \neg v_2 \wedge \neg v_3$.

Because of this strong relationship among sets, relations, Boolean functions, and OBDDs, we often use the same name for denoting the different objects belonging to the above classes. That is, as the context requires, the set A becomes an OBDD, a relation, or a Boolean function.

Consider now the fair Kripke structure $M = \langle S, R, L, F = \{F_1, \ldots, F_n\}\rangle$. To represent this structure, we have to describe the set S, the relation R, the labeling L, and the fair sets $F_1, \ldots, F_n$. For the sets S and $F_1, \ldots, F_n$, assuming they contain at most 2^n elements, we consider a Boolean encoding $\psi_S : \{0,1\}^n \to S$ and represent them through their characteristic functions. For the relation R, we use the same encoding twice, once for the current state and one for next state. It is customary to denote the variables encoding the next states as primed version of the ones used for the current state, i.e., representing R as $f_R(\vec{v}, \vec{v}')$, which holds if and only if $R(\psi(\vec{v}), \psi(\vec{v}'))$ does. Finally, the labeling L maps each state into the propositions true that hold in that state. A representation that better fits the approach already described consists in representing L by representing, for each proposition p, the set of states L_p in which p holds. Again, each L_p is represented by using the same encoding used for representing the set of states.

4.3 Fix point characterization of CTL operators

Let $M = \langle S, R, L \rangle$ be a finite Kripke structure and let us consider the complete lattice $(PowerSet(S), \subseteq)$ obtained by partially ordering through the inclusion the set of the subsets of S. The least element of such power set is the empty set, while the greatest element of such power set is S. A *functional* $\mathcal{F}$ in $(PowerSet(S), \subseteq)$ is a function from $PowerSet(S)$ to $PowerSet(S)$. $\mathcal{F}$ is called *monotonic* if it is order preserving, that is, if $A \subseteq B$ implies $\mathcal{F}(A) \subseteq \mathcal{F}(B)$, for each $A, B \subseteq S$. So, for example, for $A \subseteq S$, $\mathcal{F}(X) = X \cap A$ is monotonic, since $X \subseteq X'$ implies $X \cap A \subseteq X' \cap A$. On the other hand, for $\emptyset \neq A \subseteq S$, $\mathcal{F}(X) = A \setminus X$ is not monotonic, since for example, $\emptyset \subset A$ does not imply $\mathcal{F}(\emptyset) = A \subseteq \emptyset = \mathcal{F}(A)$. A functional $\mathcal{F}$ is *union-continuous* when for any non-decreasing infinite sequence of S's subsets $A_1 \subseteq A_2 \subseteq \ldots$, we have that $\cup_i \mathcal{F}(A_i) = \mathcal{F}(\cup_i A_i)$. In turn, a functional $\mathcal{F}$ is *intersection-continuous* when for any non-increasing infinite sequence of S's subsets $A_1 \supseteq A_2 \supseteq \ldots$, we have that $\cap_i \mathcal{F}(A_i) = \mathcal{F}(\cap_i A_i)$. Note that if $\mathcal{F}$ is union or intersection continuous it is also monotonic. A *fixed point* of $\mathcal{F}$ is any $A \subseteq S$ such that $\mathcal{F}(A) = A$. For example, A is a fixed point of $\mathcal{F}(X) = X \cap A$, since $\mathcal{F}(A) = A \cap A = A$. Tarski [79] showed that monotonic functionals on complete lattices have a least fixed point $\mu\mathcal{F}$ defined as

$$\mu\mathcal{F} = \cap_{X : \mathcal{F}(X) = X} X$$

and a greatest fixed point $\nu\mathcal{F}$ defined as

$$\nu\mathcal{F} = \cup_{X : \mathcal{F}(X) = X} X$$

Moreover, if the functional is union-continuous its least fix point can be characterized as

$$\mu\mathcal{F} = \cup_i \mathcal{F}^i(\emptyset)$$

where $\mathcal{F}^i$ means iterating $\mathcal{F}$ i times, that is, $\mathcal{F}^1(X) = \mathcal{F}(X)$ and $\mathcal{F}^{n+1}(X) = \mathcal{F}(\mathcal{F}^n(X))$. On the other hand, if the functional is intersection-continuous its greatest fix point can be characterized as

$$\nu\mathcal{F} = \cap_i \mathcal{F}^i(S)$$

We note that if the set S if finite, then any monotonic functional $\mathcal{F}$ is also union- and intersection-continuous. This is because any infinite sequence of S's subsets $A_1 \subseteq A_2 \subseteq \ldots$ has eventually to stabilize with an element A_m, possibly the whole S, such that $A_m = \cup_i A_i$ while, in turn, any infinite sequence of S's subsets $A_1 \supseteq A_2 \supseteq \ldots$ has eventually to stabilize with an element A_m, possibly the empty set, such that $A_m = \cap_i A_i$.

Now let us identify every CTL formula f with the set $\{s : s \models f\}$ of states in which the formula holds. We note that $\mathbf{EF}p$ is logically equivalent

to $p \vee \mathbf{EXEF}p$. That is, $\mathbf{EF}p$ holds in the current state s when p is true in s or $\mathbf{EF}p$ is true in some successor of s. Two logically equivalent formulas are satisfied by the same set of states. Thus, $\mathbf{EF}p = p \vee \mathbf{EXEF}p$. This makes $\mathbf{EF}p$ a fixed point of the functional $\mathcal{F}(Z) = p \vee \mathbf{EXEF}(Z)$. This functional is monotonic, since if $Z \subseteq Z'$ and there exists a path from a state s to a state satisfying Z then there is also a path form s to a state satisfying Z'. Thus, the functional $\mathcal{F}$ has a least fixed point that it can be shown being exactly $\mathbf{EF}p$. Clarke and Emerson [20] showed that similar fixed point characterizations can be obtained for the other CTL operators:

- $\mathbf{EF}p = \mu(\lambda Z.p \vee \mathbf{EX}(Z))$
- $\mathbf{EG}p = \nu(\lambda Z.p \wedge \mathbf{EX}(Z))$
- $\mathbf{E}(p\mathbf{U}q) = \mu(\lambda Z.q \vee (p \wedge \mathbf{EX}(Z)))$
- $\mathbf{AF}p = \mu(\lambda Z.(p \vee \mathbf{AX}(Z)))$
- $\mathbf{AG}p = \nu(\lambda Z.(p \wedge \mathbf{AX}(Z)))$
- $\mathbf{A}(p\mathbf{U}q) = \mu(\lambda Z.(q \vee (p \wedge \mathbf{AX}(Z))))$

Since we deal with finite Kripke structures, each of the above fixed points can be characterized as the limit of a series obtained by iterating the corresponding functionals, that is

- $\mathbf{EF}p = \cup_i(\lambda Z.(p \vee \mathbf{EX}(Z)))^i(\emptyset)$
- $\mathbf{EG}p = \cap_i(\lambda Z.(p \wedge \mathbf{E}X(Z)))^i(S)$
- $\mathbf{E}(p\mathbf{U}q) = \cup_i(\lambda Z.(q \vee (p \wedge \mathbf{EX}(Z))))^i(\emptyset)$
- $\mathbf{AF}p = \cup_i(\lambda Z.(p \vee \mathbf{AX}(Z)))^i(\emptyset)$
- $\mathbf{AG}p = \cap_i(\lambda Z.(p \wedge \mathbf{AX}(Z)))^i(S)$
- $\mathbf{A}(p\mathbf{U}q) = \cup_i(\lambda Z.(q \vee (p \wedge \mathbf{AX}(Z))))^i(\emptyset)$

Indeed, being S finite, we have that eventually, at most when reaching the whole set S for least fix points or when reaching the empty set for greatest fix points, the above sequences stabilized on the fix points. Thus, the above characterizations give us an effective procedure for computing the basic CTL operators.

4.4 CTL symbolic Model checking

In this section, we show how the fix point characterization of the CTL operators given in Section 4.3 can be exploited in order to define a CTL model checking algorithm implementable through OBDDS. OBDDs will be used to represent the set of states in which a CTL formula holds, while the required computations are carried out through the standard OBDD operations. The function SymbolicCheck, which is depicted in Figure 4.5, when given a CTL formula f returns the set of states where f holds, and is defined on the structure of f.

When f is an atomic proposition, SymbolicCheck(f) simply returns the OBDD for f, while the $\vee$ and $\neg$ connective are implemented through the Apply function.

If f is $\mathbf{EX}f_1$, SymbolicCheck returns the OBDD for $\exists \vec{v}'.(R(\vec{v},\vec{v}') \wedge f_1(\vec{v}'))$ that can be efficiently computed through the function AndExist and Compose. This latter one is used for renaming the variables of f_1 from the current state variable $\vec{v}$ into the next state variable $\vec{v}'$.

Finally, if f is $\mathbf{E}(f_1\mathbf{U}f_2)$ or $\mathbf{EG}f_1$, SymbolicCheck exploits the fix point characterizations of Section 4.3. Indeed, least fix points can be computed through the function shown in Figure 4.6. The function gfp for computing greatest fix points is obtained from the above by replacing the initialization of A with $Z := S$. Such functions can be implemented through OBDDs. Indeed, both the sets Z and Z' can be represented as OBDDs and the transformations induced by the functional $\mathcal{F}$ can be realized through the standard OBDD operations by means of the Apply, AndExist, or Compose functions.

As an example, let us compute $\mathbf{AF}p$ in the Kripke structure shown in Figure 4.7. The labels of a node define the propositions holding in that node. Note that, having the Kripke structure 5 states, the functions for computing least and greatest fix points iterate at most 6 times. The situation after the first iteration, that is, $\mathcal{F}^1(\emptyset) = p \vee \mathbf{AX}\emptyset = p$, is depicted in Figure 4.8. The situation after the second iteration, that is, $\mathcal{F}^2(\emptyset) = p \vee \mathbf{AX}p$, is depicted in Figure 4.9. This is the fix point, since performing another iteration does not result in adding any new states.

As another example, consider the computation of $\mathbf{EG}p$ over the Kripke structure shown in Figure 4.10. The situation after the first iteration, that is, $\mathcal{F}^1(S) = p \wedge \mathbf{EX}(S) = p$, is depicted in Figure 4.11. The situation after the second iteration, that is, $\mathcal{F}^2(S) = p \wedge \mathbf{EX}p$, is depicted in Figure 4.12. The situation after the third iteration, that is, $\mathcal{F}^3(S) = p \wedge \mathbf{EX}(p \wedge \mathbf{EX}p)$, is depicted in Figure 4.13. This is also the fix point, since the next iteration does not remove any further states.

```
function SymbolicCheck(f)
begin
  case f of
    proposition:
      return the OBDD for L_f of the states satisfying f
    ¬f1:
      return ¬(SymbolicCheck(f1))
    f1 ∨ f2:
      return SymbolicCheck(f1) ∨ SymbolicCheck(f2)
    EX f1:
      return SymbolicCheckEX(SymbolicCheck(f1))
    E(f1 U f2):
      return SymbolicCheckEU(SymbolicCheck(f1),
                             SymbolicCheck(f2))
    EG(f1):
      return SymbolicCheckEG(SymbolicCheck(f1))
  end
end

function SymbolicCheckEX(f1)
  begin
    return ∃v⃗'.(R(v⃗, v⃗') ∧ (f1(v⃗')))
  end

function SymbolicCheckEU(f1, f2)
  begin
    return lfp(λZ.f2 ∨ (SymbolicCheckEX(Z) ∧ f1))
  end

function SymbolicCheckEG(f1)
  begin
    return gfp(λZ.f1 ∧ SymbolicCheckEX(Z))
  end
```

Figure 4.5: Function for CTL symbolic model checking.

function LFP($\mathcal{F}$)
  let $Z := \emptyset$
  do
    let $Z' := Z$
    let $Z := \mathcal{F}(Z)$
  until $Z = Z'$
  return Z
end

Figure 4.6: Algorithm for computing the least fix point of a monotonic functional $\mathcal{F}$.

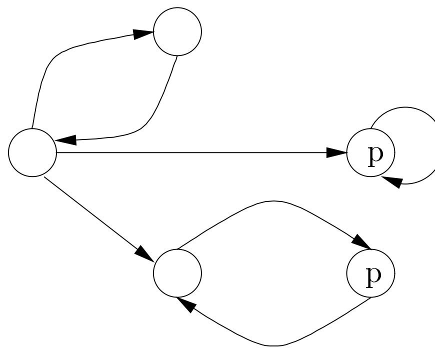

Figure 4.7: A Kripke structure.

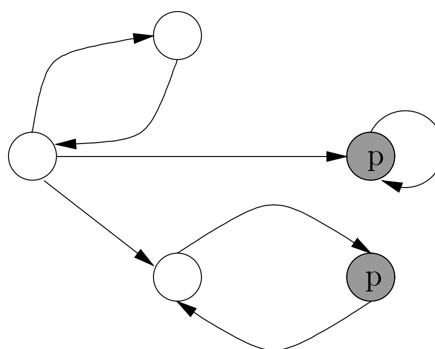

Figure 4.8: Computing $\mathbf{AF}p$, first iteration.

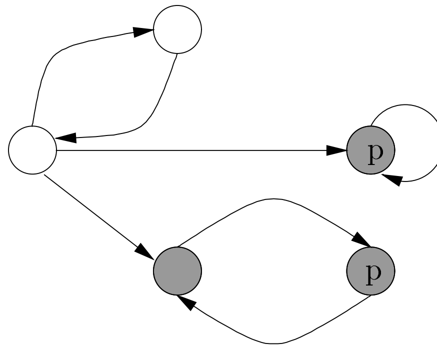

Figure 4.9: Computing $\mathbf{AF}p$, second iteration and fix point.

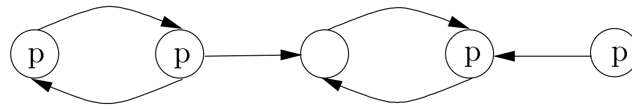

Figure 4.10: A Kripke structure.

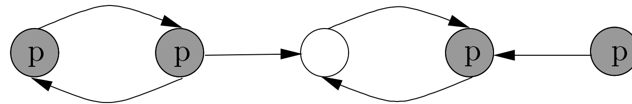

Figure 4.11: Computing $\mathbf{EG}p$, first iteration.

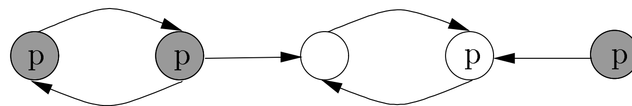

Figure 4.12: Computing $\mathbf{EG}p$, second iteration.

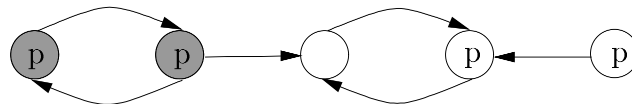

Figure 4.13: Computing $\mathbf{EG}p$, third iteration and fix point.

4.4.1 Symbolic Fair CTL Model Checking

In this section, we show how to extend the algorithm we have given in the previous section, in order to take into account fairness $F = \{F_1, \ldots, F_n\}$. The way we extend the algorithm is the same of the explicit state case, since we start by redefining the computation of $\mathbf{EG}f$ along fair paths and exploit such computation for defining the remaining ones. However, the way in which $\mathbf{EG}f$ is computed is different, in order to make it possible through the standard OBDD operations.

$\mathbf{EG}f$ is fairly satisfied along a path if f invariantly holds in such a path and, for each $F_i \in F$ there exists $s_i \in F_i$ holding infinitely often along the path. It can be shown that the set of the states in which $\mathbf{EG}f$ holds is the largest set Z such that

- Each state in Z satisfies f.
- For each state s in Z and fair set $F_i \in F$, there exists a path in Z of positive length from s to some state in Z satisfying F_i.

Intuitively, from each state contained in such set, it is possible to reach, going through states satisfying f, another state in which F_1 holds. Since this latter state belongs to the set, the reasoning can be iterated reaching all the F_i always satisfying f. Moreover, since the state satisfying F_n is in the set, there is a path satisfying f from this state back to a state satisfying F_1. Therefore, an infinite path always satisfying f and going infinitely often through every F_i can be constructed. Formally, $\mathbf{EG}f$ can expressed as

$$\mathbf{EG}f = \nu(\lambda Z.f \wedge \bigwedge_{F_i \in F} \mathbf{EX}(\mathbf{E}(f\mathbf{U}(Z \wedge F_i))))$$

and, therefore, can be computed through standard symbolic techniques. More in detail, let $fair_i$ be the OBDD for F_i, we have

$$\textsc{SymbolicFairCheck}(\mathbf{EG}f) = \\ \textsc{SymbolicFairCheckEU}(\textsc{SymbolicFairCheck}(f))$$

where

$$\textsc{SymbolicFairCheck}(f_1) = \\ \textsc{gfp}(\lambda Z.f_1 \wedge \\ \bigwedge_{i=1,\ldots,n} \textsc{SymbolicCheckEX}(\textsc{SymbolicCheckEU}(f_1, Z \wedge fair_i))) \quad .$$

Similarly to the explicit state algorithm discussed in Section 3.1.1, the other cases are dealt with by relying on the set *ExistsFairPath* of states from which a fair path departs. This set is computed as *ExistsFairPath* = $\textsc{SymbolicFairCheckEU}(S)$. The function $\textsc{SymbolicFairCheck}$ is depicted in Figure 4.14.

```
function SymbolicFairCheck(f)
begin
  case f of
    proposition:
      return the conjunction of ExistsFairPath with the OBDD for L_f
    ¬f1:
      return ¬(SymbolicFairCheck(f1))
    f1 ∨ f2:
      return SymbolicFairCheck(f1) ∨ SymbolicFairCheck(f2)
    EX f1:
      return SymbolicCheckEX(SymbolicFairCheck(f1)
                                     ∧ ExistsFairPath)
    E(f1 U f2):
      return SymbolicCheckEU(SymbolicFairCheck(f1),
                                     SymbolicFairCheck(f2)
                                     ∧ ExistsFairPath)
    EG(f1):
      return SymbolicFairCheckEG(SymbolicFairCheck(f1))
  end
end

function SymbolicFairCheckEG(f1)
begin
  return gfp(λZ.f1 ∧
⋀_{i=1,...,n} SymbolicCheckEX(SymbolicCheckEU(f1, Z ∧ fair_i)))
end
```

Figure 4.14: Function for fair CTL symbolic model checking.

4.5 LTL symbolic model checking

In this section we show how the fair CTL model checking procedure can be extended in order to deal with LTL specifications. The basic idea is to symbolically encode the graph-based approach to LTL model checking through fair CTL model checking. This is carried out by looking at the behavior graph as a symbolic Kripke structure and by exploiting fairness in order to detect fulfilling paths.

Let us start by symbolically encoding the tableau for the negation $\neg f$ of the specification. To this end, recall that the nodes of the tableau are atoms built on the closure $\overline{\neg f}$, that is, subsets A of $\overline{\neg f}$ such that

- For every $g_1 \in \overline{\neg f}$, $g_1 \in A$ iff $\neg g_1 \notin A$
- For every $g_1 \vee g_2 \in \overline{\neg f}$, $g_1 \vee g_2 \in A$ iff either $g_1 \in A$ or $g_2 \in A$
- For every $g_1 \mathbf{U} g_2 \in \overline{\neg f}$, $g_1 \mathbf{U} g_2 \in A$ iff either $g_2 \in A$ or $g_1, \mathbf{X}(g_1 \mathbf{U} g_2) \in A$

The nodes of the tableau can then be symbolically represented by setting a suitable ordering $f_1, f_2, \ldots, f_n$ among the formulas in $\overline{\neg f}$, by introducing the set $N_T = \{V_g : g \in \overline{\neg f}\}$ of variables, and by defining

$$\begin{aligned}
&\mathit{TableauNode}(V_{f_1}, \ldots, V_{f_n}) = \\
&\quad (\textstyle\bigwedge_{V_{\neg g} \in N_T} V_{\neg g} \leftrightarrow \neg V_g) \\
&\quad \wedge (\textstyle\bigwedge_{V_{g_1 \vee g_2} \in N_T} V_{g_1 \vee g_2} \leftrightarrow V_{g_1} \vee V_{g_2}) \\
&\quad \wedge (\textstyle\bigwedge_{V_{g_1 \mathbf{U} g_2} \in N_T} V_{g_1 \mathbf{U} g_2} \leftrightarrow V_{g_2} \vee (V_{g_1} \wedge \mathbf{X}(V_{g_1 \mathbf{U} g_2})))
\end{aligned}$$

Therefore, the edges of the tableau can be symbolically represented as

$$\begin{aligned}
&\mathit{TableauEdge}(V_{f_1}, \ldots, V_{f_n}, V'_{f_1}, \ldots, V'_{f_n}) = \\
&\quad \mathit{TableauNode}(V_{f_1}, \ldots, V_{f_n}) \\
&\quad \wedge \mathit{TableauNode}(V'_{f_1}, \ldots, V'_{f_n}) \\
&\quad \wedge \textstyle\bigwedge_{V_{\mathbf{X} f_1} \in N_T} V_{\mathbf{X} f_1} \leftrightarrow V'_{f_1}
\end{aligned}$$

Let now $\{w_1, \ldots, w_m\}$ be the set of new variables required to symbolically represent the Kripke structure $M = \langle S, R, L \rangle$, let $\psi : \{w_1, \ldots, w_m\} \to S$ be the Boolean encoding of the states of M, and let *State*, *TransitionRelation*, and *Labeling* be the symbolic representations of the states, of the transition relation, and of the labeling of M, respectively. More in detail, let *Labeling* be a vector of Boolean functions, one for every atomic propositions p, such that $\mathit{Labeling}_p(w_1, \ldots, w_m)$ holds if and only if $p \in L(\psi(w_1, \ldots, w_m))$.

The behavior graph can then be looked at as the fair Kripke structure $B = \langle S_B, R_B, L_B, F_B \rangle$, which can be symbolically represented starting from

the variables $w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}$ as follows. The set S_B of states is symbolically represented as

$$\begin{array}{l} \mathit{BehaviorState}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) = \\ \quad \mathit{State}(w_1, \ldots, w_m) \\ \quad \wedge\ \mathit{TableauNode}(V_{f_1}, \ldots, V_{f_n}) \\ \quad \wedge \bigwedge_{p \in \mathcal{P}} \mathit{Labeling}_p(w_1, \ldots, w_m) \leftrightarrow V_p \end{array}$$

The relation R_B is symbolically represented as

$$\begin{array}{l} \mathit{BehaviorRelation}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}, w'_1, \ldots, w'_m, V'_{f_1}, \ldots, V'_{f_n}) = \\ \quad \mathit{BehaviorState}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) \\ \quad \wedge\ \mathit{BehaviorState}(w'_1, \ldots, w'_m, V'_{f_1}, \ldots, V'_{f_n}) \\ \quad \wedge\ \mathit{TransitionRelation}(w_1, \ldots, w_m, w'_1, \ldots, w'_m) \\ \quad \wedge\ \mathit{TableauEdge}(V_{f_1}, \ldots, V_{f_n}, V'_{f_1}, \ldots, V'_{f_n}) \end{array}$$

The labeling L_B is represented, for each $p \in \mathcal{P}$, as

$$\begin{array}{l} \mathit{BehaviorLabeling}_p(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) = \\ \quad \mathit{BehaviorState}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) \wedge V_p \end{array}$$

Finally, for each $g_1 \mathbf{U} g_2 \in \overline{\neg f}$, a fairness set $F_{g_1 \mathbf{U} g_2}$ is defined and represented as

$$\begin{array}{l} \mathit{BehaviorFairness}_{g_1 \mathbf{U} g_2}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) = \\ \quad \mathit{BehaviorState}(w_1, \ldots, w_m, V_{f_1}, \ldots, V_{f_n}) \wedge (\neg V_{g_1 \mathbf{U} g_2} \vee V_{g_2}) \end{array}$$

A "wrong" path then exists if the following conditions holds:

- The path starts from a state in which $V_{\neg f}$ holds.
- The path goes through each fair set infinitely often.

and, therefore, can be encoded by means of the CTL formula

$$V_{\neg f} \wedge \mathbf{EG}(\textsc{true})$$

that has to be evaluated according to the fair CTL semantics.

Chapter 5

Planning as Model Checking

Planning as model checking [15, 37, 19, 2, 18, 26, 16, 27] is a new planning paradigm that seems to be very promising in order to produce automatic good-performance planners for nonclassical planning. The main idea underlying this paradigm is that, as in model checking, planning problems are faced model-theoretically. That is, planning domains are formalized as semantic models, properties of planning domains are described through temporal formulas, and planning is carried out by verifying whether semantic models satisfy temporal formulas. Looking at planning from this perspective introduces many new important features:

- The approach is *well-founded*: Planning problems are given a clear and intuitive formalization in temporal logic.
- The approach is *general*: The same framework can be used to naturally tackle many different aspects of planning, e.g., many initial states, partial observability, nondeterministic domains, and extended goals, that is, not only goals of attainment.
- The approach is *practical*: By exploiting the large amount of techniques developed in the field of model checking, it is possible to devise efficient algorithms that generate plans automatically and that can deal with large size problems.

In this chapter, we describe two different approaches to planning as model checking, one based on symbolic techniques and another relying on automata on infinite words.

5.1 Symbolic Approach to Planning

Cimatti et al. introduced in [15] the idea to use symbolic model checking techniques for facing planning problems. Such idea has since then been extended

for dealing with nondeterministic domains in several ways. More in detail, [15] proposes an algorithm for generating *weak plans*, that is, plans that may achieve the goal but are not guaranteed to do so; [19] proposes an algorithm to generate *strong plans*, that is, plans that are guaranteed to achieve a desired goal in spite of nondeterminism; and [18, 26] extends [19] to generate *strong cyclic plans*, whose aim is to encode iterative trial-and-error strategies like "pick-up the block until succeed", as described in Section 1.4. Finally, [16] shows how to perform conformant planning symbolically.

In this section, we provide a formal definition of weak, strong, and strong cyclic plans based on the branching time temporal logic CTL and present the related planning algorithms. The formalization is obtained by exploiting the universal and existential path quantifiers of CTL, as well as the "always" and "eventually" temporal connectives. Indeed, the idea is that a weak plan is such that *there exists* an execution that *eventually* achieves the goal, a strong plan is such that *all* the executions *eventually* achieve the goal, and a strong cyclic plan is such that *for each* possible execution, *always* during the execution, there *exists* the possibility of *eventually* achieving the goal.

5.1.1 Planning Problems

In this framework a nondeterministic planning domain can be described in terms of *fluents*, which may assume different values in different *states*, *actions* and a *transition function* describing how (the execution of) an action leads from one state to possibly many different states.

Definition 5.1.1 (Planning Domain) *A planning domain D is a 4-tuple $\langle F, S, A, R\rangle$ where F is the finite set of fluents, $S \subseteq 2^F$ is the set of states, A is the finite set of actions, and $R : S \times A \mapsto 2^S$ is the transition function.*

Fluents belonging (not belonging) to some state s are assigned to TRUE (FALSE) in s. Our definitions deal with Boolean fluents while examples are easier to describe through fluents ranging over generic finite domains. For non-Boolean variables, we use a Boolean encoding similarly to [32]. $R(s, a)$ returns all the states the execution of a from s can lead to. The action a is said to be *executable* in the state s if $R(s, a) \neq \emptyset$.

A nondeterministic planning problem is a planning domain, a set of initial states and a set of goal states.

Definition 5.1.2 (Planning Problem) *A planning problem P is a 3-tuple $\langle D, I, G\rangle$ where D is the planning domain, $I \subseteq S$ is the set of initial states and $G \subseteq S$ is the set of goal states.*

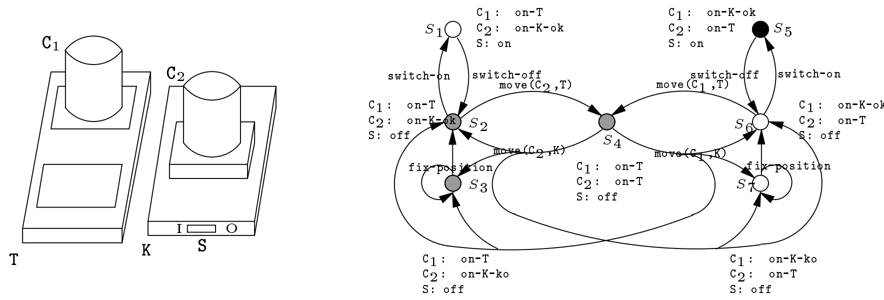

Figure 5.1: An example (left) and its formalization (right).

Both I and G can be represented through two Boolean functions $\mathcal{I}$ and $\mathcal{G}$ over F, which define the sets of states in which they hold. From now on, we switch between the two representations, as sets or functions, as the context requires.

Nondeterminism occurs twice in the above definitions. First, we have a set of initial states, and not a single initial state. Second, the execution of an action from a state is a set of states, and not a single state.

As an explanatory example, let us consider the situation depicted in Figure 5.1 (left). A tray (T) provides two positions in which two containers ($\mathtt{C}_1$ and $\mathtt{C}_2$) for solutions may be hosted. In addition, a kettle (K) may host one container for boiling its solution. The kettle is provided with a switch (S) that can operate only if the container is well positioned on the kettle. This situation can be formalized as shown in Figure 5.1 (right). The set F of (non-Boolean) fluents is $\{\mathtt{C}_1, \mathtt{C}_2, \mathtt{S}\}$. $\mathtt{C}_1$ and $\mathtt{C}_2$ represent the positions of the containers, and can be `on-T` (on tray), `on-K-ok` (on kettle, steady), or `on-K-ko` (on kettle, not steady). `S` represents the status of the kettle's switch (`on` or `off`). The set of states is represented by the nodes of the graph, which define fluents' values. The set of actions is represented by the edges' labels. The actions move($\mathtt{C}_1$,T), move($\mathtt{C}_2$,T), `switch-on`, and `switch-off`, are deterministic; while move($\mathtt{C}_1$,K), move($\mathtt{C}_2$,K), and `fix-position`, are not. Indeed, when moving containers from the tray to the kettle, it can happen the containers are not correctly positioned. Moreover, it can be possible the wrong container is picked up and moved upon the kettle. Thus, $R(S_4, \mathtt{move}(\mathtt{C}_1, \mathtt{K})) = R(S_4, \mathtt{move}(\mathtt{C}_2, \mathtt{K})) = \{S_2, S_3, S_6, S_7\}$. Still, when trying to settle a container, it is possible getting no effect. Thus, $R(S_3, \mathtt{fix\text{-}position}) = \{S_2, S_3\}$ and $R(S_7, \mathtt{fix\text{-}position}) = \{S_6, S_7\}$. The planning problem is to boil the solution contained in $\mathtt{C}_1$ starting from a situation where $\mathtt{C}_1$ is on the tray and the kettle's switch is off, that is, $\mathcal{I}$ is $\mathtt{C}_1 = \mathtt{on\text{-}T} \wedge \mathtt{S} = \mathtt{off}$ (grey nodes, in Figure 5.1), and $\mathcal{G}$ is $\mathtt{C}_1 = \mathtt{on\text{-}K\text{-}ok} \wedge \mathtt{S} = \mathtt{on}$ (black node, in Figure 5.1).

State	Action
S_1	`switch-off`
S_3	`fix-position`
S_2	`move(C`$_2$`,T)`
S_4	`move(C`$_1$`,K)`
S_4	`move(C`$_2$`,K)`
S_7	`fix-position`
S_6	`switch-on`

Figure 5.2: A state-action table.

5.1.2 Plans as State-Action Tables

When dealing with nondeterminism, plans have to be able to represent conditional and iterative behaviors. In this framework plans are defined as *state-action tables* (resembling universal plans [73]) that associate actions to states. The execution of a state-action table can result in conditional and iterative behaviors. Intuitively, a state-action table execution can be explained in terms of a reactive loop that senses the state of the world and chooses one among the corresponding actions, if any, for the execution until the goal is reached. Therefore, a state-action table results in a memoryless plan.

Definition 5.1.3 (State-Action Table) *A state-action table SA for a planning problem P is a set of pairs $\{\langle s,a\rangle : s \in S \setminus G,\ a \in A,$ and a is executable in $s\}$.*

The states of a state-action table may be any state, except for those in the set of goal states. Intuitively, this corresponds to the fact that when the plan achieves the goal no further action needs to be executed. Hereafter, we write $\textsc{States}(SA)$ for denoting the set of states in the state-action table SA, i.e., $\textsc{States}(SA) = \{s : \exists a \in A \text{ such that } \langle s,a\rangle \in SA\}$, and $\textsc{Closure}(SA)$ to denote the states that are not in the SA but that are reachable by executing pairs in SA or are goal states, i.e., $\textsc{Closure}(SA) = \{s \notin \textsc{States}(SA) : \langle s',a'\rangle \in SA, s \in R(s',a')\} \cup G$. The following particular case occurs when $\textsc{Closure}(SA) = G$.

Definition 5.1.4 (Total State-Action Table) *A state-action table SA for a planning problem P is* total *if, for all $\langle s,a\rangle \in SA$, $R(s,a) \subseteq \textsc{States}(SA) \cup G$.*

Intuitively, in a total state-action table, each state that can be reached by executing an action either is a goal state or has a corresponding action in the state-action table. The notion of total state-action table is important in

order to capture strong (cyclic) plans, i.e., plans that must be specified for all possible outcomes of actions. In Figure 5.2, a total state-action table related to our example is shown.

Given a notion of plan as a state-action table, weak, strong, and strong cyclic plans can be formalized in terms of CTL specifications on the possible executions of state-action tables. A preliminary step is to formalize the notion of execution of a state-action table

Definition 5.1.5 (Execution) *Let SA be a state-action table for the planning problem P. An* execution *of SA starting from the state $s_0 \in \textsc{States}(SA) \cup \textsc{Closure}(SA)$ is an infinite sequence $s_0 s_1 \ldots$ of states in S such that, for all $i \geq 0$, either $s_i \in \textsc{Closure}(SA)$ and $s_i = s_{i+1}$, or $s_i \notin \textsc{Closure}(SA)$ and, for some $a \in A$, $\langle s_i, a \rangle \in SA$ and $s_{i+1} \in R(s_i, a)$.*

Executions are infinite sequences of states. Depending on nondeterminism, we may have many possible executions corresponding to a state-action table. Each nongoal state s_i has as successor a state s_{i+1} reachable from s_i by executing an action corresponding to s_i in the state-action table; when the sequence reaches a goal state, the execution is extended with an infinite sequence of the same goal state. Of course, nontotal state-action tables may induce also executions *dangling* at nongoal states, i.e., executions reaching a nongoal state for which no action is provided.

The executions of a state-action table SA for the planning problem P can be encoded as paths of the Kripke structure K_{SA}^P *induced* by SA.

Definition 5.1.6 (Induced Kripke Structure) *Let SA be a state-action table for the planning problem P. The Kripke structure K_{SA}^P induced by SA is defined as*

- $W_{SA}^P = \textsc{States}(SA) \cup \textsc{Closure}(SA)$;
- $T_{SA}^P(s, s')$ *iff* $s \in \textsc{Closure}(SA)$ *and* $s = s'$, *or* $\langle s, a \rangle \in SA$ *and* $s' \in R(s, a)$;
- $L_{SA}^P(s) = s$.

Note that having introduced the closure guarantees the totality of the induced Kripke structure.

5.1.3 Weak Planning

Weak plans are such that, for all the initial states, *there exists* at least one execution that *eventually* achieves the goal. Formally, we have the following

1. **function** WEAKPLAN(P)
2. $WP := \emptyset$; $OldWP := \bot$
3. **while** ($I \not\subseteq$ STATES(WP) $\cup\, G$ **and** $OldWP \neq WP$) **do**
4. $OldWP := WP$
5. $WP := WP \,\cup$ WEAKPREIMAGE(P, WP)
6. **endwhile**
7. **if** ($I \subseteq$ STATES(WP) $\cup\, G$)
8. **then return** WP
9. **else return** *Fail*

Figure 5.3: An algorithm for weak planning.

Definition 5.1.7 (Weak Plan) *A* weak plan *for a planning problem P is a state-action table SA for P such that $\mathcal{I} \subseteq W^P_{SA}$ and, for all $s \in \mathcal{I}$, we have $K^P_{SA}, s \models \mathbf{EF}\mathcal{G}$.*

In Figure 5.3 a simple regressive algorithm for computing weak plans is shown. The idea underlying this and the following algorithms is that *sets* of states (instead of single states) are manipulated during the search. This allows the algorithm to be efficiently implemented through OBDDs and, therefore, to target domains involving large state spaces as shown by the experimental results in [18].

The algorithm starts with the empty state-action table in WP (line 2) and works backward gathering in WP state-action pairs whose execution may lead to G (lines 3–6). This is realized through the function WEAKPREIMAGE (line 5) that, given a planning problem P and a state-action table WP, returns the pairs that are related to states that are neither in WP nor in G and through which states of WP or G can be reached in one step. Formally, WEAKPREIMAGE(P, WP) $= \{\langle s, a\rangle : s \in S, a \in A, s \not\in$ STATES(WP) $\cup\, G$, and $R(s,a) \cap$ (STATES(WP) $\cup\, G$) $\neq \emptyset\}$. The algorithm keeps expanding WP until either it contains $I \setminus G$ or it can not be expanded anymore (line 3). In the former case, the algorithm returns WP (line 8), while in the latter one it returns failure (line 9).

For example, Figure 5.3 is a weak plan for the planning problem shown in Figure 5.1. After one iteration of the **while** statement at line 3, the last row of the table is gathered, while after two iterations the last four rows are collected. After another iteration, the third row is introduced in the table. Note that, due to the definition of WEAKPREIMAGE, in this iteration the pair $\langle s_6, \texttt{move(C}_1\texttt{,T)}\rangle$ is not inserted in the state-action table, since it is related to a state that has already been inserted. Finally, after four iterations, the first two rows are collected. The algorithm then stops since all the initial states

have been gathered.

The above algorithm always terminates, is correct and complete.

Theorem 5.1.1 *Let P be a planning problem, then*

1. WEAKPLAN(P) *terminates.*
2. WEAKPLAN(P) *returns a weak plan for P if and only if one exists.*

Proof: (1) The termination follows from the fact that the **while** statement at line 3 implements a monotonic functional over a finite domain. Indeed, it keeps adding state-action pairs to WP and therefore it eventually terminates, possibly when all the pairs have been introduced.

(2) (only if) By definition of induced Kripke structure, we have that the goal states satisfy $\mathbf{EF}\mathcal{G}$. Thus, let us focus on nongoal states. By induction on the number n of iterations of the **while** statement at line 3, and denoting as WP_n the state-action table WP after the nth iteration, let us prove that $K^P_{WP_n}, s \models \mathbf{EF}(\mathcal{G})$, for all $s \in \text{STATES}(WP_n)$. If $n = 0$, then $\text{STATES}(WP_0) = \emptyset$ and the claim trivially holds. If $n > 0$, either no pairs have been added, and we conclude by inductive hypothesis, or WP_n is obtained from WP_{n-1} by adding some pairs for which at least one of their outcomes is in $\text{STATES}(WP_{n-1})$. We conclude by definition of $K^P_{WP_n}$ and by inductive hypothesis.

(if) Let us suppose that a weak plan $\widehat{WP}$ for P exists. Then, in $K^P_{\widehat{WP}}$, each initial state s is either a goal state or is connected to a goal state through some minimal length path $s_1 \dots s_{n+1}$, where $s = s_1$, $s_1, \dots, s_n \notin G$, and $s_{n+1} \in G$. By definition of $K^P_{\widehat{WP}}$, the path $s_1 \dots s_{n+1}$ corresponds to some state-action pairs $\langle s_1, a_1 \rangle, \dots, \langle s_n, a_n \rangle$ such that, for $i = 1, 2, \dots, n$, we have $s_{i+1} \in R(s_i, a_i)$.

Let us now show that after k iterations of the **while** statement at line 3, $k = 1, \dots, n$, we have that $s_{n-k+1} \in \text{STATES}(WP_k)$ and that all the states in WP_h are connected to G in $K^P_{WP_k}$ through a path of length k. If $k = 1$ then WEAKPREIMAGE introduces the pair $\langle s_n, a_n \rangle$ in WP_1. If $k > 1$, let us note that $s-k+1$ cannot belong to WP_{k-1} since otherwise the inductive hypothesis would contrast the choice of minimal length path. By inductive hypothesis $s_{n-(k-1)+1} \in \text{STATES}(WP_{k-1})$ and therefore the pair $\langle s_{n-k+1}, a_{s-k+1} \rangle$ is inserted in WP_k. Since $WP_{k-1} \subseteq WP_k$, $s_{n-(k-1)+1}$ is connected to G by a path of length $k-1$ in $K^P_{WP_k}$ too. Being s_{n-k+1} connected to $s_{n-(k-1)+1}$, we have that s_{n-k+1} is connected to G by a path of length k in $K^P_{WP_k}$.

Since for each i we have that $WP_i \subseteq WP_{i+1}$, all the states of $\widehat{WP}$ are eventually gathered in some $WP_{\hat{k}}$ and, therefore, the algorithm terminates with success.

```
1. function StrongPlan(P)
2.    SP:= ∅; OldSP:=⊥
3.    while (I ⊈ States(SP) ∪ G and OldSP≠SP) do
4.       OldSP:=SP
5.       SP:=SP ∪ StrongPreImage(P,SP)
6.    endwhile
7.    if (I ⊆ States(SP) ∪ G)
8.       then return SP
9.       else  return Fail
```

Figure 5.4: An algorithm for strong planning.

5.1.4 Strong Planning

Strong plans are such that *all* executions *eventually* achieve the goal. Formally, we have the following

Definition 5.1.8 (Strong Plan) *A* strong plan *for a planning problem P is a total state-action table SA for P such that $\mathcal{I} \subseteq W^P_{SA}$ and, for all $s \in \mathcal{I}$, we have $K^P_{SA}, s \models \mathbf{AF}\mathcal{G}$.*

In Figure 5.4 a simple regressive algorithm for computing strong plans is shown. Again, sets of states (rather than single states) are manipulated during the search, allowing the algorithm to be efficiently implemented through OBDDs.

The algorithm resembles the one given for computing weak plans, but for the function WeakPreImage that is replaced by StrongPreImage. StrongPreImage introduces a state-action pair if it is not related to an already present state or to a goal state, and if all its nondeterministic outcomes are contained in the so far computed state-action table or in the goal. Formally, when given a problem set P and a state-action table SP, StrongPreImage is defined as $\text{StrongPreImage}(P, SP) = \{\langle s, a\rangle : s \in S, a$ is executable in $s, s \notin (\text{States}(SP) \cup G),$ and $R(s,a) \subseteq \text{States}(SP) \cup G\}$.

For example, no strong plan exists for the planning problem shown in Figure 5.1. Indeed, after one iteration, the last row of the state-action table in Figure 5.2 is gathered, but the second iteration results in the fix point since no other state-action pairs can be introduced.

The above algorithm always terminates, is correct and complete.

Theorem 5.1.2 *Let P be a planning problem, then*

1. StrongPlan(P) *terminates.*
2. StrongPlan(P) *returns a strong plan for P if and only if one exists.*

Proof: (1) The termination follows from the fact that the **while** statement at line 3 implements a monotonic functional over a finite domain. Indeed, it keeps adding state-action pairs to SP and therefore it eventually terminates, possibly when all the pairs have been introduced.

(2) (only if) By definition of induced Kripke structure, we have that the goal states satisfy $\mathbf{AF}\mathcal{G}$. Thus, let us focus on nongoal states. By induction on the number n of iterations of the **while** statement at line 3, and denoting as SP_n the state-action table SP after the nth iteration, let us prove that SP_n is total and $K^P_{SP_n}, s \models \mathbf{AF}\mathcal{G}$, for all $s \in \textsc{States}(SP_n)$. If $n = 0$, then $\textsc{States}(SP_0) = \emptyset$ and the claim trivially holds. If $n > 0$, either no pairs have been added, and we conclude by inductive hypothesis, or SP_n is obtained from SP_{n-1} by adding some pairs through $\textsc{StrongPreImage}$. The totality of SP_n follows by inductive hypothesis and by definition of $\textsc{Strong-PreImage}$. Moreover, let $s_0 s_1 \ldots$ be an infinite path in $K^P_{SP_n}$ starting from $s_0 \in \textsc{States}(K^P_{SP_n})$. Since $K^P_{SP_{n-1}}$ is total and, by definition of $\textsc{Strong-PreImage}$, none of the states related to the new pairs is in $\textsc{States}(K^P_{SP_{n-1}})$, either $s_0 s_1 \ldots$ is a path in $K^P_{SP_{n-1}}$ or $s_1 s_2 \ldots$ is path in $K^P_{SP_{n-1}}$ and s_0 is one of the new states. Thus, since by inductive hypothesis $K^P_{SP_{n-1}}, s \models \mathbf{AF}\mathcal{G}$ for all $s \in \textsc{States}(SP_{n-1})$, we have that $K^P_{SP_n}, s \models \mathbf{AF}\mathcal{G}$, for all $s \in \textsc{States}(SP_n)$.

(if) Let us suppose that a strong plan $\widehat{SP}$ for P exist and restrict $\widehat{SP}$ to the states reachable from the initial ones. More in detail, let us define $SP'_0 = \{\langle s,a\rangle \in \widehat{SP} : s \in I\}$ and $SP'_{n+1} = SP'_n \cup \{\langle s,a\rangle \in \widehat{SP} : \langle s',a'\rangle \in SP'_n \text{ and } s \in R(s',a')\}$. Since for all n we have that $SP'_n \subseteq SP'_{n+1}$ and the number of state-action pairs is finite, we can set $SP' = SP'_m$ such that $SP'_{m+1} = SP'_m$. By induction on $k = 1, \ldots, m$, it is then trivial to show that for all $s \in SP'_k$ there exists $s_0 \in I$ such that s_0 is connected to s in $K^P_{SP'}$.

SP' is total and $K^P_{SP'}, s \models \mathbf{AG}\mathcal{G}$ for all $s \in \textsc{States}(SP')$. Indeed, let $\langle s,a\rangle \in SP'$ and let it have been introduced in SP'_l, for some $0 \leq l \leq m$. Since $\widehat{SP}$ is total and by definition of SP'_{l+1}, all the state-action pairs related to states that are outcomes of $\langle s,a\rangle$ are then present in SP'_{l+1}. This, recalling that for all n we have that $SP'_n \subseteq SP'_{n+1}$, means that SP' is total. Let now $r_0 r_1 \ldots$ be a path in $K^P_{SP'}$ starting from the state $r_0 \in \textsc{States}(SP')$. If $r_0 \notin I$, since r_0 is reachable from some initial state s_0, the above path can be extended to a path $s_0 s_1 \ldots r_0 r_1 \ldots$. Let $q_0 q_1 \ldots$ be $r_0 r_1 \ldots$ if $r_0 \in I$ or $s_0 s_1 \ldots r_0 r_1 \ldots$ otherwise. $q_0 q_1 \ldots$ eventually hits a goal state. Indeed, if this was not the case, since SP' is total, we would have that for each $i \geq 0$ there exists $\langle q_i, a_i\rangle \in SP'$ such that $q_{i+1} \in R(q_i, a_i)$. Since $SP' \subseteq \widehat{SP}$, $q_0 q_1 \ldots$ would be a path in $K^P_{\widehat{SP}}$ as well, and this is not possible, since SP' is a strong plan for P. Therefore, $K^P_{SP'}, s \models \mathbf{AF}\mathcal{G}$ for all $s \in SP'$.

As a consequence, there are no cycles in $K^P_{SP'}$ involving states contained

in $\textsc{States}(SP')$. SP' can then be stratified by defining $SP''_0 = \emptyset$ and $SP''_{n+1} = SP''_n \cup \{\langle s, a\rangle \in SP' : s \notin \textsc{States}(SP''_n)$ and $R(s,a) \subseteq \textsc{States}(SP''_n) \cup G\}$, and setting $SP'' = SP''_l$ such that $SP''_l = SP''_{l+1}$. Note that $\textsc{States}(SP'') = \textsc{States}(SP')$. Indeed, if this was not the case, one could consider all the pairs related to the states that are in SP' but not in SP''. Since none of them can be added to SP''_l, by totality of SP', they would induce a cycle in $K^P_{SP'}$. Indeed, let $\langle s_1, a_1\rangle \in SP' \setminus SP''$. Since it can not be added to SP'', it means that it has an outcome towards a state $s_2 \in \textsc{States}(SP') \setminus \textsc{States}(SP'')$, related to the pair $\langle s_2, a_2\rangle$. The claim then follows from the fact that the reasoning can be iterated and the set from which the pairs are taken is finite.

Finally, denoted as SP_k the state-action pairs gathered by the algorithm in SP after the kth iteration of the **while** statement at line 3, we show that the **while** statement terminates because $I \subseteq \textsc{States}(SP_k) \cup G$, for some k. Indeed, if this was not the case, the above **while** statement would terminate because, for some k, we would have that $SP_k = SP_{k+1}$ but $I \setminus G \not\subseteq \textsc{States}(SP_k)$. Therefore, the set $\{\langle s, a, e\rangle : s$ is a nongoal state not belonging to $\textsc{States}(SP_k)$ and $\langle s, a\rangle$ has been inserted in $SP''_e\}$ is not empty. Let us select from the above set one of the tuples related to the minimum e, say $\langle s, a, e\rangle$. Because of this choice, $\langle s, a\rangle$ should be introduced in SP_{k+1} by StrongPreImage, and this is absurd.

5.1.5 Strong Cyclic Planning

Strong cyclic plans are such that for *all* the executions, *always* during the execution, there *exists* the possibility of *eventually* reaching the goal. This means that executions might eventually loop forever on a set of states, but if they terminate they are guaranteed to achieve the goal. Formally, we have the following

Definition 5.1.9 (Strong Cyclic Plan) *A* strong cyclic plan *for a planning problem P is a total state-action table SA for P such that $\mathcal{I} \subseteq W^P_{SA}$ and, for all $s \in \mathcal{I}$, we have $K^P_{SA}, s \models$ $\mathbf{AGEF}\mathcal{G}$.*

The algorithm for computing strong cyclic plans is presented in two steps: first, algorithms computing basic strong cyclic plans are introduced (Figures 5.5 and 5.8), and then an algorithm for improving such basic solutions is given (Figure 5.10).

Given a planning problem P, StrongCyclicPlan(P) (Figure 5.5) generates strong cyclic plans. The algorithm starts with the largest state-action table in SCP (line 2), and repeatedly removes pairs that either spoil SCP totality or are related to states from which the goal cannot be reached (line 5). If the resulting SCP contains all the initial states (line 7), the algorithm returns it (line 8), otherwise *Fail* is returned (line 9).

```
1.  function StrongCyclicPlan(P)
2.     SCP:= {⟨s,a⟩ : s ∈ S \ G and a is executable in s}; OldSCP:=⊥
3.     while (OldSCP≠SCP) do
4.        OldSCP:=SCP
5.        SCP:=PruneUnconnected(P, PruneOutgoing(P, SCP))
6.     endwhile
7.     if (I ⊆ States(SCP) ∪ G)
8.        then return SCP
9.        else  return Fail

14. function PruneOutgoing(P, SA)
15.    Outgoing := ComputeOutgoing(P, SA)
16.    while (Outgoing ≠ ∅) do
17.       SA:=SA\Outgoing
18.       Outgoing := ComputeOutgoing(P, SA)
19.    endwhile
20.    return SA

21. function PruneUnconnected(P, SA)
22.    ConnectedToG := ∅; OldConnectedToG := ⊥
23.    while ConnectedToG ≠ OldConnectedToG do
24.       OldConnectedToG:=ConnectedToG
25.       ConnectedToG:=SA ∩ OneStepBack(P, ConnectedToG)
26.    endwhile
27.    return ConnectedToG
```

Figure 5.5: The algorithm.

Pairs spoiling SCP totality are pruned by function PruneOutgoing (lines 14–20), which iteratively removes state-action pairs that can lead to nongoal states for which no action is considered. Its core is the function ComputeOutgoing that, for a planning problem P and a state-action table SA, is defined as $\{\langle s,a\rangle \in SA : R(s,a) \not\subseteq (\text{States}(SA) \cup G)\}$. With respect to the example shown in Figure 5.6 (left), during the first iteration, PruneOutgoing removes $\langle S_4, e\rangle$ and, during the second one, it removes $\langle S_3, b\rangle$, giving rise to the situation shown in Figure 5.6 (middle).

Having removed the dangling executions results in disconnecting S_2 and S_3 from the goal, and give rise to a cycle in which executions may get stuck with no hope of terminating. This point, however, was not clear in the work presented in [18]. States from which the goal cannot be reached have to be pruned away. This task is accomplished by the function PruneUncon-

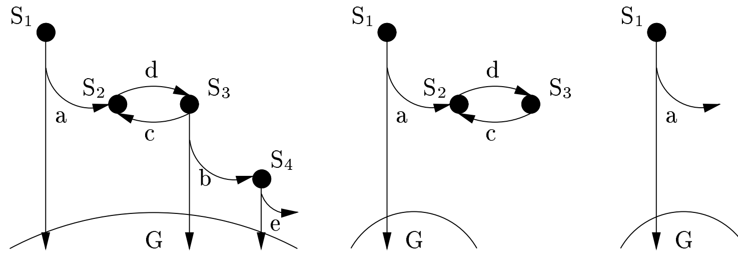

Figure 5.6: Pruning the state-action table.

NECTED (lines 21–27) that, when given with a planning problem P and a state-action table SA, loops backwards inside the state-action table from the goal (line 25) to return the state-action pairs related to states from which the goal is reachable. Looping backward is realized through the function OneStepBack that, when given with a planning problem P and a state-action table SA', returns all the state-action pairs possibly leading to states of SA' or G. Formally, $\text{OneStepBack}(P, SA') = \{\langle s, a\rangle : s \in S \setminus G, a \in A, R(s,a) \cap (\text{States}(SA') \cup G) \neq \emptyset\}$. Note the similarity with WeakPreImage, from which OneStepBack differs because it does not care whether a state have been already inserted. With respect to the example shown in Figure 5.6 (middle), PruneUnconnected removes both $\langle S_2, d\rangle$ and $\langle S_3, c\rangle$, producing the situation shown in Figure 5.6 (right). Having removed the above pairs re-introduces dangling executions and, therefore, requires to apply the pruning phase once more, leading to the empty set. In general, the pruning phase has to be repeated until the putative strong plan SCP is not changed either by PruneOutgoing or by PruneUnconnected (line 3).

As an alternative (see Figure 5.8), rather than starting with the largest state-action table, one could start with an empty state-action table in $AccSA$ (line 2) and incrementally extend it (line 4) until either a strong cyclic plan containing all the initial states is found, or $AccSA$ is not extendible anymore (line 3).

For example, Figure 5.7 is a strong cyclic plan for the planning problem shown in Figure 5.1. Considering the algorithm in Figure 5.8, after one iteration of the **while** statement at line 3, the last row of the table is gathered, while after two iterations the last four rows are collected. After another iteration, the third and the fourth rows are introduced in the table. Note that during this iteration, unlike WeakPreImage, OneStepBack inserts the pair $\langle s_6$,`move(C`$_1$`,T)`$\rangle$ in the state-action table. Finally, after four iterations, the first two rows are collected. The algorithm then stops since all the initial states have been gathered, the state-action table is total and each state is somehow

State	Action
S_1	`switch-off`
S_3	`fix-position`
S_2	`move(C`$_2$`,T)`
S_6	`move(C`$_1$`,T)`
S_4	`move(C`$_1$`,K)`
S_4	`move(C`$_2$`,K)`
S_7	`fix-position`
S_6	`switch-on`

Figure 5.7: A state-action table.

1. **function** StrongCyclicPlan(P)
2. $\quad SCP := \emptyset$; $AccSA := \emptyset$; $OldAccSA := \bot$
3. $\quad$ **while** ($I \not\subseteq$ States(SCP) $\cup\, G$ **and** $AccSA \neq OldAccSA$) **do**
4. $\quad\quad OldAccSA := AccSA$; $AccSA :=$ OneStepBack(P, $AccSA$)
5. $\quad\quad SCP := AccSA$; $OldSCP := \bot$
6. $\quad\quad$ **while** ($OldSCP \neq SCP$) **do**
7. $\quad\quad\quad OldSCP := SCP$
8. $\quad\quad\quad SCP :=$ PruneUnconnected(P, PruneOutgoing(P, SCP))
9. $\quad\quad$ **endwhile**
10. $\quad$ **endwhile**
11. $\quad$ **if** ($I \subseteq$ States(SCP) $\cup\, G$)
12. $\quad\quad$ **then return** SCP
13. $\quad\quad$ **else return** *Fail*

Figure 5.8: The incremental algorithm.

connected to the goal set.

The strong cyclic plans returned by StrongCyclicPlan can be improved in two directions. Consider the example in Figure 5.9, where S_3 is the initial state. The strong cyclic plan returned by StrongCyclicPlan for such example comprises all the possible state-action pairs of the planning problem. Note, however, that the pair $\langle S_1, a\rangle$ is absolutely useless, since it is unreachable from the initial state. Furthermore, the pair $\langle S_4, d\rangle$ is useless as well, because it moves the execution away from the goal. Indeed, when reaching S_4 from S_3, one does not want to go back to S_3 through d. The algorithm for getting rid of the above situations is shown in Figure 5.10.

Function PruneUnreachable loops forward, inside the state-action table returned by the basic algorithm, collecting state-action pairs related to

states that can be reached from the initial ones. Its core is the function ONESTEPFORTH (line 7) that, when given with a planning problem P and a state-action table *Reachable*, returns the set of pairs related to states reachable by executing actions in *Reachable*. Formally, ONESTEPFORTH(P, *Reachable*) $= \{\langle s, a\rangle : s \in S, a \in A, a$ is executable in s and there exists $\langle s', a'\rangle \in$ *Reachable* such that $s \in R(s', a')\}$. *Reachable* is initialized with the pairs related to initial states by GETINIT (line 4), defined as GETINIT$(P, SCP) = \{\langle s, a\rangle \in SCP : s \in I\}$. With respect to Figure 5.9, this first optimization phase chops out the pair $\langle S_1, a\rangle$ while, with respect to the state-action table of Figure 5.2, $\langle S_1,$ `switch-off`$\rangle$ is removed.

Function SHORTESTEXECUTIONS chops out all the pairs $\langle s, a\rangle$ that do not start one of the shortest executions leading from s to the goal. Indeed, executions passing through s can still reach the goal through one of the shortest ones. Shortest executions are gathered in *Shortest* as a set of state-action pairs by looping backward (line 14) inside the (optimized through PRUNEUNREACHABLE) state-action table returned by the basic algorithm, and by introducing new pairs only when related to states that have not been visited yet. Indeed, note that this time the function WEAKPREIMAGE is used in place of ONESTEPBACK. With respect to Figure 5.9, this second optimization phase chops out the pair $\langle S_4, d\rangle$ while, with respect to the state-action table of Figure 5.2, $\langle S_6,$ `move(`C_1`,T)`$\rangle$ is removed.

The algorithms for generating and optimizing strong cyclic plans are guaranteed to terminate, are correct and complete. The results rely on the following Lemmas.

Lemma 5.1.1 *Let $K = \langle S, R, L\rangle$ be a Kripke structure, and $G \subseteq S$. Then, for all $s \in S$ we have that $K, s \models$ **AGEF**G if and only if for all $s \in S$ we have that $K, s \models$ **EF**G.*

Proof: (only if) Each state s can be thought of as the starting point of a path in K. Since $K, s \models$ **AGEF**G, it follows that $K, s \models$ **EF**G.

(if) Let us consider a generic path $s_0 s_1 \ldots$ in K. Since for each $i \geq 0$ we

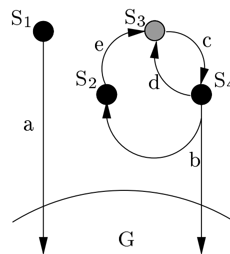

Figure 5.9: Problems of the basic algorithm.

1. **function** OPTIMIZE(P, SCP)
2. **return** SHORTESTEXECUTIONS(P, PRUNEUNREACHABLE(P, SCP))

3. **function** PRUNEUNREACHABLE(P, SCP)
4. $Reachable := $ GETINIT(P, SCP); $OldReachable := \bot$
5. **while** ($Reachable \neq OldReachable$) **do**
6. $OldReachable := Reachable$
7. $Reachable := Reachable \cup SCP \cap$ ONESTEPFORTH(P, $Reachable$)
8. **endwhile**
9. **return** $Reachable$

10. **function** SHORTESTEXECUTIONS(P, SCP)
11. $Shortest := \emptyset$; $OldShortest := \bot$
12. **while** ($Shortest \neq OldShortest$)
13. $OldShortest := Shortest$
14. $Shortest := Shortest \cup SCP \cap$ WEAKPREIMAGE$(P, Shortest)$
15. **endwhile**
16. **return** $Shortest$

Figure 5.10: Optimization.

have that $K, s_i \models \mathbf{EF}G$, we conclude that $K, s_0 \models \mathbf{AGEF}G$.

Lemma 5.1.2 *Let P be a planning problem, SA be a state-action table for P, and $SA' \subseteq SA$ be a total state-action table for P. Then,* PRUNEOUTGOING(P, SA) *returns a total state-action table SA'' such that $SA' \subseteq SA'' \subseteq SA$.*

Proof: First of all, let us note that PRUNEOUTGOING(P, SA) terminates, since the **while** statement at line 16 implements a monotonic functional over a finite domain. Indeed, it keeps removing state-action pairs from SA and, therefore, it eventually terminates, possibly producing the empty set. This also means that the state-action table returned by the function is included in SA.

Let us now denote as SA_k the state-action table SA after the kth iteration of the **while** statement at line 16. Since PRUNEOUTGOING terminates when $SA_m = SA_{m+1}$, for some m, and this happens when no pairs have an outcome that is neither a goal state or a state in SA_m, then SA_m is total.

Finally, by induction on k, let us prove that $SA_k \supseteq SA'$. If $n = 0$ then the claim trivially holds. If $n > 0$ we conclude by inductive hypothesis, by definition of COMPUTEOUTGOING, and because SA' is total.

Lemma 5.1.3 *Let P be a planning problem, SA be a state-action table for P, and $SA' \subseteq SA$ be a total state-action table for P such that $K^P_{SA'}, s \models \mathbf{EF}\mathcal{G}$, for*

all $s \in$ STATES(SA'). *Then,* PRUNEUNCONNECTED(P, SA) *returns a state-action table* SA'' *such that*

1. $SA \supseteq SA'' \supseteq SA'$.

2. $K^P_{SA''}, s \models \mathbf{EF}\mathcal{G}$, *for all* $s \in$ STATES(SA'').

Proof: First of all, let us note that PRUNEUNCONNECTED(P, SA) terminates, since the **while** statement at line 23 implements a monotonic functional over a finite domain. Indeed, it keeps adding pairs to *ConnectedToG* from SA and, therefore, it eventually terminates, possibly when having added the whole SA. This also means that $SA \supseteq SA''$.

Let us now denote as $ConnectedToG_k$ the state-action table *ConnetedToG* after the kth iteration of the **while** statement at line 23. Moreover, let $s \in$ STATES(SA') and $s_1 \dots s_{n+1}$ be a path in $K^P_{SA'}$ connecting $s = s_1$ to $s_{n+1} \in G$, and $s_1, \dots, s_n \in$ STATES(SA'). By definition of $K^P_{SA'}$, that path corresponds to n pairs $\langle s_1, a_1\rangle, \dots, \langle s_n, a_n\rangle \in SA'$ such that for $i = 1, \dots, n$, we have $s_{i+1} \in R(s_i, a_i)$. By induction on $k = 0, \dots, m$, we prove that $K^P_{ConnectedToG_k}, s \models \mathbf{EF}\mathcal{G}$, for all $s \in$ STATES$(ConnectedToG_k)$; that $ConnectedToG_k \subseteq ConnectedToG_{k+1}$; and that $s_{n+1-k} \in$ STATES$(ConnectedToG_k) \cup G$. If $k = 0$ the claim holds because $ConnectedToG_0 = \emptyset$ and $s_{n+1} \in G$. If $k > 0$, $ConnectedToG_k$ is either $ConnectedToG_{k-1}$, and we conclude by inductive hypothesis and by definition of ONESTEPBACK, or has been obtained from $ConnectedToG_{k-1}$ by adding some pairs, each of which has at least one outcome leading to the goal or to $ConnectedToG_{k-1}$. By definition of $K^P_{ConnectedToG_k}$ and by inductive hypothesis, this means that $K^P_{ConnectedToG_k}, s \models \mathbf{EF}\mathcal{G}$, for all $s \in$ STATES$(ConnectedToG_k)$. Moreover, since by inductive hypothesis $ConnectedToG_{k-1} \subseteq ConnectedToG_k$, all the pairs added by ONESTEPBACK to $ConnectedToG_{k-1}$ are added by the function to $ConnectedToG_k$ as well. This means that $ConnectedToG_k \subseteq ConnectedToG_{k+1}$. Finally, by inductive hypothesis and definition of ONESTEPBACK, we have that $s_{n+1-k} \in$ STATES$(ConnectedToG_k)$.

The claim we have already proved immediately implies (2) and allows us to state that STATES$(SA') \subseteq$ STATES(SA''). This means that (1) holds, since SA' is total and by definition of ONESTEPBACK, when a pair $\langle s, a\rangle$ is introduced in $ConnectedToG_k$, we have that all the pairs $\langle s', a'\rangle$ having at least one outcome leading to s or to a goal state are introduced in $ConnectedToG_{k+1}$.

Let us start by proving termination and soundness of the algorithm presented in Figure 5.5.

Theorem 5.1.3 *Let* P *be a planning problem. Then*

1. STRONGCYCLICPLAN(P) *terminates.*

2. StrongCyclicPlan(P) *returns a strong cyclic plan for P if and only if one exists.*

Proof: (1) The termination follows from the fact that, by Lemmas 5.1.2 and 5.1.3, the **while** statement at line 3 implements a monotonic functional over a finite domain. Indeed, it keeps removing state-action pairs from SCP and therefore it eventually terminates, possibly when producing the empty set.

(2) (only if) Since by Lemmas 5.1.3 and 5.1.2 $SCP \subseteq$ PruneUnreachable(P,PruneOutgoing(p, SCP)) $\subseteq$ PruneOutgoing(p, SCP) $\subseteq SCP$, we have that the state-action table SCP returned by the algorithm is a fix point of both PruneOutgoing and PruneUnconnected. Its totality, guaranteed by Lemma 5.1.2, is preserved by PruneUnconnected thank to Lemma 5.1.3. Moreover, Lemmas 5.1.1 and 5.1.3 imply that, for all $s \in I$, $K^P_{SCP}, s \models \mathbf{AGEF}\mathcal{G}$.

(if) Let us suppose that a strong cyclic plan $\widehat{SCP}$ for P exist. Let us denote with SCP_k the state-action table SCP after the kth iteration of the **while** statement at line 3. Then, it sufficies to show that for every k, SCP_k contains $\widehat{SCP}$. By induction on k. If $k = 0$ the claim trivially holds. If $k > 0$ we conclude by inductive hypothesis and Lemmas 5.1.2 and 5.1.3.

Let us now consider the algorithm presented in Figure 5.8.

Theorem 5.1.4 *Let P be a planning problem. Then*

1. StrongCyclicPlan(P) *terminates.*

2. StrongCyclicPlan(P) *returns a strong cyclic plan for P if and only if one exists.*

Proof: (1) The termination follows from Lemmas 5.1.2 and 5.1.3 and from the fact that both the **while** statements at lines 3 and 6 implements monotonic functionals over a finite domain. Indeed, the former one keeps adding state-action pairs to $AccSA$ and therefore eventually terminates, possibly when having added the whole set of pairs. The latter one keeps removing state-action pairs from SCP and therefore eventually terminates, possibly when producing the empty set.

(2) (only if) Since by Lemmas 5.1.3 and 5.1.2 $SCP \subseteq$ PruneUnreachable(P,PruneOutgoing(p, SCP)) $\subseteq$ PruneOutgoing(p, SCP) $\subseteq SCP$, we have that the state-action table SCP returned by the algorithm is a fix point of both PruneOutgoing and PruneUnconnected. Its totality, guaranteed by Lemma 5.1.2, is preserved by PruneUnconnected thank to Lemma 5.1.3. Moreover, Lemmas 5.1.1 and 5.1.3 imply that, for all $s \in I$, $K^P_{SCP}, s \models \mathbf{AGEF}\mathcal{G}$.

(if) Let us suppose that a strong ciclic plan $\widehat{SCP}$ for P exist and, as in Theorem 5.1.2, define SCP' as the restriction of $\widehat{SCP}$ to the states reachable from the initial ones. That is, let us define $SCP'_0 = \{\langle s, a\rangle \in \widehat{SCP} : s \in I\}$, $SCP'_{n+1} = SCP'_n \cup \{\langle s, a\rangle \in \widehat{SCP} : \langle s', a'\rangle \in SCP'_n$ and $s \in R(s', a')\}$, and set $SCP' = SCP'_m$ such that $SCP'_m = SCP'_{m+1}$. Again, SCP' is total. Moreover, $K^P_{SCP'}, s \models \mathbf{AGEF}\mathcal{G}$ for all $s \in \textsc{States}(SCP')$. Indeed, since $\widehat{SCP}$ is strong cyclic plan for P and each $s \in \textsc{States}(SCP')$ is an initial state or is reachable from some initial state in $\widehat{SCP}$, it has to be that s is connected to some goal state in $K^P_{\widehat{SCP}}$ by some path $s_1 \dots s_{n+1}$, where $s = s_1$, and $s_1, \dots, s_n \in \textsc{States}(\widehat{SCP})$, and $s_{n+1} \in G$. By definition of $K^P_{\widehat{SCP}}$, such path corresponds then to n pairs $\langle s_1, a_1\rangle, \dots, \langle s_n, a_n\rangle \in \widehat{SCP}$ such that for $i = 1, \dots, n$, we have that $s_{i+1} \in R(s_i, a_i)$. Supposing that s has been introduced in SCP'_k, by induction on $i = 1, \dots, n$, we can show that $\langle s_i, a_i\rangle \in SCP'_{i-1+k}$. If $i = 1$ the claim follows from the fact that when introducing s all the related pairs are introduced as well. If $i > 1$ the claim holds by inductive hypothesis and by definition of SCP'_{i-1+k}. This means that s is connected to some goal state in $K^P_{SCP'}$ as well.

Now, denoting as $AccSA_k$ the state-action table $AccSA$ after the kth iteration of the **while** statement at line 3, the existence of the path $s_1 \dots s_{n+1}$ connecting in $K^P_{SCP'}$ $s = s_1$ to $s_{n+1} \in G$ ensures that $s \in \textsc{States}(AccSA_n)$. This can be showed by proving by induction on $k = 1, \dots, n$ that $s_{n+1-k} \in \textsc{States}(AccSA_k)$. If $k = 1$ the claim holds because $s_{n+1} \in R(s_n, a_n) \cap G$ and by definition of OneStepBack. If $k > 1$ we conclude by inductive hypothesis and by definition of OneStepBack. This means that $AccSA$ can be expanded so that to include all the states of SCP'.

Further, being SCP' total, this amounts to the possibility of expanding $AccSA$, to obtain say $AccSA_m$, so that to contain SCP'. Indeed, as a state $s \in \textsc{States}(SCP')$ is inserted in $AccSA_k$, we have that all the pairs having s as outcome are inserted in $AccSA_{k+1}$ by OneStepBack. Therefore, either a positive answer is given by the algorithm in $m' < m$ steps, or $AccSA_m$ is accumulated.

Let us denote with SCP_k the state-action table SCP after the kth iteration of the **while** statement at line 6 when dealing with $AccSA_m$. By induction on k we show that SCP_k contains SCP'. If $n = 0$ the claim trivially holds. If $k > 0$ we conclude by inductive hypothesis and Lemmas 5.1.2 and 5.1.3.

Finally, let us show that the optimization step does not spoil the computed strong cyclic plan.

Theorem 5.1.5 *Let* SCP *be a strong cyclic plan for the planning problem P. Then* Optimize(P, SCP) *is a strong cyclic plan for P.*

Proof: PRUNEUNREACHABLE simply implements the reduction of $\widehat{SCP}$ ($\widehat{SP}$) to SCP' (SP') that we have already discussed in Theorem 5.1.4 (Theorem 5.1.2).

Let us focus on SHORTESTEXECUTIONS. Since each state s occurring in $SCP' = \text{PRUNEUNREACHABLE}(P, SCP)$ is connected to some goal state in $K^P_{SCP'}$ there is a minimal length path $s_1 \ldots s_{n+1}$ connecting $s = s_1$ to $s_{n+1} \in G$, and $s_1, \ldots, s_n \in \text{STATES}(SCP')$. Let us denote as $Shortest_k$ the state-action table gathered by the algorithm after the kth iteration of the **while** statement at line 12, and prove by induction on $k = 1, \ldots, n$ that $\{s_{n+1-i} : i = 1, \ldots, k\} \subseteq \text{STATES}(Shortest_k)$ and that each $s \in Shortest_k$ is connect to G through a path of length k in $K^P_{Shortest_k}$. If $k = 1$ the claim holds by definition of WEAKPREIMAGE. If $k > 1$, $s_{n+1-k} \notin \text{STATES}(Shortest_k)$, since otherwise the inductive hypothesis would contradict the choice of a minimal length path. Therefore, we conclude by inductive hypothesis and by definition of WEAKPREIMAGE.

As a consequence of the above claim, we have that $\text{STATES}(SCP') = \text{STATES}(\text{SHORTESTEXECUTIONS}(SCP'))$ and, since SCP' is total, that so is $SCP'' = \text{SHORTESTEXECUTIONS}(SCP')$. Moreover, the above claim and Lemma 5.1.1 also imply that $K^P_{SCP''}, s \models \mathbf{AGEF}\mathcal{G}$ for all $s \in SCP''$.

5.2 Automata-based Approach to Planning

De Giacomo and Vardi [27] have shown how to face planning in deterministic domains through the automata-based approach, focusing on temporally extended goals and partial observability. In this approach, both the planning domain and the goal are looked at as automata on infinite words, and are then suitably combined in order to select the paths in the planning domain that are compatible with the goal.

5.2.1 Planning Problems

A planning problem is formalized as a pair $\langle \mathcal{T}, \mathcal{G} \rangle$ where $\mathcal{T}$ is a deterministic finite *transition system* modeling the planning domain and the initial states, and $\mathcal{G}$ is a Büchi automaton modeling the goal. As a consequence of having described the goal as a Büchi automaton, the interesting executions of $\mathcal{T}$ are the infinite ones.

A finite transition system $\mathcal{T}$ is a tuple $\langle W, W_0, Act, R, Obs, \pi \rangle$ where

- W is the finite set of *states*.
- $W_0 \subseteq W$ is the set of *initial states*.
- Act is the finite set of *actions*.

- $R : W \times Act \to W$ is the total deterministic *transition function*, that is, R is defined for each state and action.
- Obs is the finite set of *observations*, which model the observable part of states.
- $\pi : W \to Obs$ is the *observability* function, which returns the observable part of the states.

An *execution* of the transition system is an infinite sequence of states $w_0 w_1 \ldots \in W^\omega$ such that

- $w_0 \in W_0$
- For all $i \geq 0$ there exists $a_i \in Act$ such that $w_{i+1} = R(w_i, a_i)$.

A *trace* is what we can observe of an execution. For example, $\pi(w_0)\pi(w_1)\ldots$ is the trace corresponding to the execution $w_0 w_1 \ldots$. The *observable behavior* of $\mathcal{T}$ is the set of all possible traces of $\mathcal{T}$.

The goal $\mathcal{G}$ is specified through a Büchi automaton $\langle Obs, S, S_0, \delta, F \rangle$ where:

- Obs plays the role of the alphabet of the automaton.
- S is the finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $\delta : S \times Obs \to 2^S$ is the nondeterministic transition function.
- $F \subseteq S$ is the set of accepting states.

5.2.2 Planning with Complete Information

We start to consider a simplified case, by assuming complete information on the initial states and full observability of states.

In this case, the problem domain and the initial states are modeled through a transition system $\mathcal{T} = \langle W, W_0, Act, R, Obs, \pi \rangle$ where

- $W_0 \subseteq W$ is a singleton set containing the initial state, which is unique since completely specified.
- $Obs = W$ and $\pi : W \to Obs$ is simply the identity function, since we are assuming full observability.

A *plan* p for $\mathcal{T}$ is an infinite sequence of actions $a_0 a_1 \ldots \in Act^\omega$. The *execution* of p from the initial state $w_0 \in W_0$ is an infinite sequence $w_0 w_1 \ldots \in W^\omega$ such that $w_{i+1} = R(w_i, a_i)$, for all $i \geq 0$. The *trace* $tr(p, w_0)$ is the infinite

sequence $\pi(w_0)\pi(w_1)\ldots \in Obs^\omega$. A plan p *realizes* a goal specification $\mathcal{G}$ if and only if $tr(p, w_0) \in L(\mathcal{G})$.

A plan that realizes a goal specification can be synthesized by checking for nonemptiness the Büchi automaton $\mathcal{TG} = \langle Act, S_{\mathcal{TG}}, S_{\mathcal{TG}_0}, \delta_{\mathcal{TG}}, F_{\mathcal{TG}}\rangle$ where

- Act is the alphabet of the automaton.
- $S_{\mathcal{TG}} = S \times W$.
- $S_{\mathcal{TG}_0} = S_0 \times W_0$.
- $(s_j, w_j) \in \delta_{\mathcal{TG}}((s_i, w_i), a)$ iff $w_j = R(w_i, a)$ and $s_j \in \delta(s_i, \pi(w_i))$.
- $F_{\mathcal{TG}} = F \times W$.

Indeed, $\mathcal{TG}$ is the synchronous product of $\mathcal{T}$ and $\mathcal{G}$, which encodes the traces of $\mathcal{T}$ that are compatible with $\mathcal{G}$. The emptiness check consists in looking for an accepting state s_f reachable from itself and from some initial state. From such a path, say $(s_0, w_0)(s_1, w_1)\ldots(s_f, w_f)\ldots(s_f, w_f)\ldots$ we have that the plan can be extracted by chosing a_i such that $(s_{i+1}, w_{i+1}) \in \delta_{\mathcal{TG}}((s_i, w_i), a_i)$.

Since $\mathcal{TG}$ can be built on-the-fly while checking for its nonemptiness, one can look for an accepting state reachable from the initial one and from itself by using a nondeterministic algorithm that only needs $\mathcal{O}(\log(|W|) + \log(|S|))$ bits for storing the accepting, the current, and the next states. Moreover, if we adopt a compact, i.e., logarithmic, representation of the transition system, then planning in the above setting becomes PSPACE. However, it has to be noted that only certain transition systems are compactly representable, since the number of transition functions is $|W|^{|W|}$, while those distinguishable with $\mathcal{O}(\log|W|)$ bits are $2^{\mathcal{O}(\log(|W|))} = |W|^{\mathcal{O}(1)}$. The PSPACE complexity is the complexity of planning in STRIPS [10], which can be seen as a special case of the setting considered here when the goal automaton encodes goals of attainment. Moreover, since STRIPS is PSPACE-hard [10], we can conclude that planning in the setting above is NLOGSPACE-complete, or PSPACE-complete with respect to a compact representation of the transition system.

5.2.3 Conformant Planning with Incomplete Information

We now turn to consider a more general case dealing with partial information about the initial state and partial observability of states. However, we still stick to generate plans as sequences of actions.

In this case, the transition system is $\mathcal{T} = \langle W, W_0, Act, R, Obs, \pi\rangle$ where W, R, Act, and Obs are as before, but $W_0 = \{w_{00}, \ldots, w_{0k-1}\}$, for some $k > 1$, to reflect the uncertainty about the initial state, and π is not the identity function anymore.

A *plan* p for $\mathcal{T}$ is an infinite sequence of actions $a_0 a_1 \ldots \in Act^\omega$. The *execution* of p starting from $w_{0h} \in W_0$ is the infinite sequence of states $w_{0h} w_{1h} \ldots \in W^\omega$ such that $w_{i+1h} = R(w_{ih}, a_i)$. The *trace* $tr(p, w_{0h})$ of p starting from w_{0h} is the infinite sequence $\pi(w_{0h})\pi(w_{1h})\ldots$. A plan p *realizes* the goal $\mathcal{G} = \langle Obs, S, S_0, \delta, F \rangle$ if and only if $tr(p, w_{0h}) \in L(\mathcal{G})$, for $h = 0, \ldots, k-1$.

To synthesize such a plan we work as before by checking for the nonemptiness of the product Büchi automaton $\mathcal{TG} = \langle Act, S_{\mathcal{TG}}, S_{\mathcal{TG}_0}, \delta_{\mathcal{TG}}, F_{\mathcal{TG}} \rangle$, whose construction is slightly more involved, since we have to keep trace of k concurrent executions, and is given as generalized Büchi automaton:

- Act is the alphabet of the automaton.
- $S_{\mathcal{TG}} = S^k \times W^k$.
- $S_{\mathcal{TG}_0} = S_0^k \times \{(w_{00}, \ldots, w_{0k-1})\}$.
- $(\vec{s}_j, \vec{w}_j) \in \delta_{\mathcal{TG}}((\vec{s}_i, \vec{w}_i), a)$ iff, for $h = 0, \ldots, k-1$, we have that $w_{jh} = R(w_{ih}, a)$ and $s_{jh} \in \delta(s_{ih}, \pi(w_{ih}))$.
- $F_{\mathcal{T}} = \{F \times S^{k-1} \times W^k, S \times F \times S^{k-2} \times W^k, \ldots, S^{k-1} \times F \times W^k\}$.

Again, the nonemptiness check can be solved by a nondeterministic algorithm. However, this time, the required space is $\mathcal{O}(k \cdot (\log(|W|) + \log(|S|)))$. Moreover, the algorithm can be shown to be PSPACE-complete or, when assuming a compact representation of the transition system, EXPSPACE-complete [27].

It is interesting to note that the above results also holds when one has full observability of states and the goal automaton encodes goals of attainment. On the other hand, plan existence in STRIPS with incomplete information on the initial situation is PSPACE-complete [3]. This means that, when generalizing the problem domain through a transition system, one does pay a price with respect to more traditional approaches but, for the same price, one has for free temporally generalized goals and partial observability of states.

5.2.4 Conditional Planning with Incomplete Information

In this section we consider two proposals for conditional plans in the general setting discussed above. Let us start with the one given in [27].

A *vector plan* $\vec{p}$ is an infinite sequence of vectors of actions $\vec{a}_0 \vec{a}_1 \ldots \in (Act^k)^\omega$. The *execution* of the hth component of $\vec{p}$ starting from $w_{0h} \in W_0$, denoted by $exe_h(\vec{p}, w_{0h})$, is the infinite sequence of states $w_{0h} w_{1h} \ldots$ such that $w_{i+1h} = R(w_{ih}, a_{ih})$. The *trace* $trace_h(\vec{p}, w_{0h})$ of the hth component of $\vec{p}$ starting from $w_0 \in W_0$ is the infinite sequence $\pi(w_{0h})\pi(w_{1h})\ldots$. The vector plan $\vec{p}$ *realizes* the goal $\mathcal{G}$ if and only if $trace_h(\vec{p}, w_{0h}) \in L(\mathcal{G})$, for $h = 0, \ldots, k-1$.

So far, a vector plan is simply the parallel composition of k sequential plans, each one starting from a different initial state. *Conditional plans* are vector plans whose actions agree on executions with the same observations.

To formally define conditional plans, we introduce the following notion of equivalence on finite executions. Let $w_{0l} \dots w_{nl}$ and $w_{0m} \dots w_{nm}$ two finite executions of the components l and m, respectively. Then

$$w_{0l} \dots w_{nl} \sim w_{0m} \dots w_{nm} \text{ iff } \pi(w_{0l}) \dots \pi(w_{nl}) = \pi(w_{0m}) \dots \pi(w_{nm})$$

A *conditional plan* $\vec{p}$ is a vector plan such that given the finite executions $w_{0l} \dots w_{nl}$ and $w_{0m} \dots w_{nm}$ of a pair of components l and m, we have that $a_{nl} = a_{nm}$ whenever $w_{0l} \dots w_{nl} \sim w_{0m} \dots w_{nm}$.

Again, the synthesis of a conditional plan goes through the construction of a Büchi automaton encoding such plans, and then through the nonemptiness check. Specifically, we build the generalized Büchi automaton $\mathcal{TG} = \langle Act^k, S_{\mathcal{TG}}, S_{\mathcal{TG}_0}, \delta_{\mathcal{TG}}, F_{\mathcal{TG}} \rangle$ where:

- Act^k is the alphabet of the automaton.
- $S_{\mathcal{TG}} = S^k \times W^k \times \mathcal{E}_k$, where $\mathcal{E}_k$ is the set of equivalence relations on the set $\{0, \dots, k-1\}$.
- $S_{\mathcal{TG}_0} = S_0^k \times \{(w_{00}, \dots, w_{0k-1})\} \times \equiv_0$, where $i \equiv_0 j$ iff $w_{0i} = w_{0j}$.
- $(\vec{s}_j, \vec{w}_j, \equiv_j) \in \delta_{\mathcal{TG}}((\vec{s}_i, \vec{w}_i, \equiv_i), \vec{a})$ iff, for $h = 0, \dots, k-1$, we have that
  - $w_{jh} = R(w_{ih}, a_h)$ and $s_{jh} \in \delta(s_{ih}, \pi(w_{ih}))$
  - if $l \equiv_i m$ then $a_l = a_m$
  - $l \equiv_j m$ iff $l \equiv_i m$ and $\pi(w_{jl}) = \pi(w_{jm})$
- $F_{\mathcal{TG}} = \{F \times S^{k-1} \times W^k \times \mathcal{E}_k, \dots, S^{k-1} \times F \times W^k \times \mathcal{E}_k\}$

However, consider the planning domain shown in Figure 5.11, where w_1 and w_2 are the initial states, and suppose that $\pi(w_1) \neq \pi(w_2)$ and $\pi(w_3) = \pi(w_4)$. Hence, since $\pi(w_1) \neq \pi(w_2)$, we have that $w_1 w_3 \not\sim w_2 w_4$ and that different actions can be associated to w_3 and w_4, even though they are not distinguishable.

To solve this problem, we require that actions associated to states that are indistinguishable are the same. Formally, a conditional plan $\vec{p}$ is now a vector plan such that given the finite executions $w_{0l} w_{1l} \dots w_{nl}$ and $w_{0m} w_{1m} \dots w_{nm}$ of a pair of components l and m, we have that $a_{nl} = a_{nm}$ if $\pi(w_{nl}) = \pi(w_{nm})$. This affects the construction of $\mathcal{TG} = \langle Act^k, S_{\mathcal{TG}}, S_{\mathcal{TG}_0}, \delta_{\mathcal{TG}}, F_{\mathcal{TG}} \rangle$ as follows

- Act^k is the alphabet of the automaton.

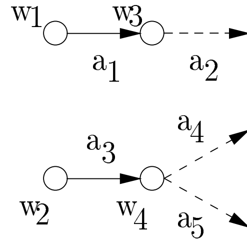

Figure 5.11: A planning domain.

- $S_{\mathcal{TG}} = S^k \times W^k$.
- $S_{\mathcal{TG}_0} = S_0^k \times \{(w_{00}, \ldots, w_{0k-1})\}$.
- $(\vec{s}_j, \vec{w}_j) \in \delta_{\mathcal{TG}}((\vec{s}_i, \vec{w}_i, \vec{a})$ iff, for $h = 0, \ldots, k-1$, we have that
  - $w_{jh} = R(w_{ih}, a_h)$ and $s_{jh} \in \delta(s_{ih}, \pi(w_{ih}))$
  - $a_l = a_m$ if $\pi(w_{il}) = \pi(w_{im})$.
- $F_{\mathcal{TG}} = \{F \times S^{k-1} \times W^k, S \times F \times S^{k-2} \times W^k, \ldots, S^{k-1} \times F \times W^k\}$.

In this latter case, the conditional plan returned by the emptiness check can be put in a more convenient form using **case** statements testing the observable part of states. Finally, observing that for storing an equivalence relation on $\{0, \ldots, k-1\}$ we only need k bits, it can be shown that planning in the above setting is, for both the proposals, PSPACE-complete, or EXPSPACE-complete with respect to a compact representation of $\mathcal{T}$.

5.2.5 Improved Automata Generation for Linear Temporal Logic

Even though the previous section deals with goals expressed in a very general way as Büchi automata, it is often more comfortable to express them as LTL formulas like, for example, $\mathbf{F}g$ for goal of attainment, or $\mathbf{GF}g$ for going infinitely often through g. A translator like the one described in Section 3.2.2 is then used to produce the goal Büchi automaton. However, this translation is critical for two reasons. First, it can result in an exponential number of states and, therefore, several algorithms can yield very different results. Second, since the goal automaton is put in product with the planning domain automaton, each difference in its construction is amplified by the usually huge size of this latter one. For these reasons, it is clearly desirable to keep the goal automaton as small as possible, and to work on-the-fly, that is, to detect that a plan exists by constructing and visiting only some part of the search space containing it.

So far, the state-of-the-art on-the-fly algorithm for turning LTL formulas into automata has been the one presented in [35]. We refer to that algorithm as GPVW. That paper also discusses several possible improvements. We refer to the improved algorithm as GPVW+. In the rest of this section we present, and describe experiments with, a new algorithm for building an automaton from a linear temporal logic formula. Such algorithm, hereafter LTL2AUT, though being based on GPVW+, is geared towards building smaller automata in less time. The improvements are based on simple syntactic techniques, carried out on-the-fly when states are processed, that allow us to eliminate the need of storing some information. Experimental results demonstrate that GPVW+ significantly outperforms GPVW and show that LTL2AUT further outperforms GPVW+, with respect to both the size of the generated automata and computation time. The testing has been performed following a newly developed methodology, which, inspired by the methodologies proposed in [61] and [36] for propositional and modal K logics, is based on randomly generated formulas.

The Core

LTL2AUT, GPVW+, and GPVW can be obtained by suitably instantiating the core we are about to present. The instantiation affects some functions that, in what follows, are highlighted through the SMALL CAPITAL font. The central part of the core is the computation of a *cover* of a set A of formulas, that is, a possibly empty set $C = \{C_i : i \in I\}$ of sets of formulas such that $\bigwedge_{\mu \in A} \mu \leftrightarrow \bigvee_{i \in I} \bigwedge_{\eta_i \in C_i} \eta_i$.

Covers The algorithm for computing covers is defined by extending the propositional tableau in order to allow it to deal with temporal operators. The algorithm works with formulas in *negation normal form*, that is, such that negations only occur in front of propositions. Such formulas are built by combining *literals*, that is, propositions and their negations, through the $\vee$ and $\wedge$ propositional operators, and the **X**, **U** and **V** temporal operators. The formulas that are not decomposed by the tableau construction are called *elementary*, and corresponds to TRUE, FALSE, literals and next-time formulas. A set of formulas is said to be elementary if all its formulas are. Unlike elementary formulas, nonelementary formulas can be decomposed, according to the tableau rules of Figure 5.12, so that $\mu \leftrightarrow \bigwedge_{\beta_1 \in \alpha_1(\mu)} \beta_1 \vee \bigwedge_{\beta_2 \in \alpha_2(\mu)} \beta_2$. Note that the fundamental rules used for decomposing temporal operators are the identity $\mu \mathcal{U} \eta \equiv \eta \vee (\mu \wedge X(\mu \mathcal{U} \eta))$ and its dual $\mu \mathcal{V} \eta \equiv \eta \wedge (\mu \vee X(\mu \mathcal{V} \eta))$. The line numbers in the following description refer to the algorithm appearing in Figure 5.13. The algorithm handles the following data structures:

ToCover The set of formulas to be covered but still not processed.

μ	$\alpha_1(\mu)$	$\alpha_2(\mu)$
$\mu_1 \wedge \mu_2$	$\{\mu_1, \mu_2\}$	$\{\mathrm{F}\}$
$\mu_1 \vee \mu_2$	$\{\mu_1\}$	$\{\mu_2\}$
$\mu_1 \mathcal{U} \mu_2$	$\{\mu_2\}$	$\{\mu_1, X(\mu_1 \mathcal{U} \mu_2)\}$
$\mu_1 \mathcal{V} \mu_2$	$\{\mu_2, \mu_1\}$	$\{\mu_2, X(\mu_1 \mathcal{V} \mu_2)\}$

Figure 5.12: Tableau rules.

```
1  function Cover(A)
2   return cover(A, ∅, ∅, ∅)

3  function cover(ToCover, Current, Covered, Cover)
4   if ToCover = ∅
5     then return Cover ∪ {Current}
6     else  select μ from ToCover
7           remove μ from ToCover and add it to Covered
8           if HAS_TO_BE_STORED(μ)
9             then Current = Current ∪ {μ}
10          if CONTRADICTION(μ, ToCover, Current, Covered)
11            then return Cover
12            else  if REDUNDANT(μ, ToCover, Current, Covered)
13                    then return cover(ToCover, Current, Covered, Cover)
14                    else  if μ is elementary
15                            then return cover(ToCover, Current ∪ {μ},
16                                              CoveredCover)
17                            else  return cover(ToCover ∪ (α1(μ) \ Current),
18                                              Current, Covered,
19                                         cover(ToCover ∪ (α2(μ) \ Current),
20                                              Current, Covered, Cover))
```

Figure 5.13: Cover computation.

Current The element of the cover currently being computed.

Covered The formulas already processed and covered by *Current*.

Cover The cover so far computed.

When computing the current element of the cover, the algorithm first checks whether all the formulas have been covered (line 4). If so, *Current* is ready to be added to *Cover* (line 5). If a formula μ has still to be covered (line 6), the algorithm checks whether μ has to be stored in the current element of the cover (line 8) and, if so, adds it to *Current* (line 9). Processing μ can be avoided in two cases: If there is a contradiction involving it (line 10) or it is redundant (line 12). In the former case, *Current* is discarded (line 11), while in the latter one μ is discarded (line 13). Finally, if μ does need to be covered, it is covered according to its syntactic structure. If μ is elementary, it is covered simply by itself (line 15). Otherwise, μ is covered by covering, according to the tableau rules appearing in Figure 5.12, either $\alpha_1(\mu)$ (line 17) or $\alpha_2(\mu)$ (line 19). This is justified by recalling that $\mu \leftrightarrow \bigwedge_{\beta_1 \in \alpha_1(\mu)} \beta_1 \vee \bigwedge_{\beta_2 \in \alpha_2(\mu)} \beta_2$.

The Automaton Construction Our goal is to build a labeled generalized Büchi automaton recognizing exactly all the models of a linear time temporal logic formula ψ. The presentation and the proof of correctness of the algorithm are simplified if we slightly modify the definition of Büchi automata by moving the labeling from the transition function to states.

More in detail, a generalized Büchi automaton is now a quadruple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F} \rangle$, where

- $\mathcal{Q}$ is a finite set of *states*.
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*.
- $\delta : \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$ is the *transition function*.
- $\mathcal{F} \subseteq 2^{2^{\mathcal{Q}}}$ is a, possibly empty, set of sets of accepting states $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$.

An *execution* of $\mathcal{A}$ is an infinite sequence $\rho = q_0 q_1 q_2 \ldots$ such that

- $q_0 \in \mathcal{I}$.
- For all $i \geq 0$, $q_{i+1} \in \delta(q_i)$.

ρ is *accepting execution* if, for each $F_i \in \mathcal{F}$, there exists $q_i \in F_i$ that appears infinitely often in ρ.

A labeled generalized Büchi automaton is a triple $\langle \mathcal{A}, \mathcal{D}, \mathcal{L} \rangle$, where

- $\mathcal{A}$ is a generalized Büchi automaton.
- $\mathcal{D}$ is some finite domain.
- $\mathcal{L} : \mathcal{Q} \to 2^{\mathcal{D}}$ is the *labeling function.*

A labeled generalized Büchi automaton *accepts* a word $\xi = x_0x_1x_2\ldots$ from $\mathcal{D}^\omega$ iff there exists an accepting execution $\rho = q_0q_1q_2\ldots$ of $\mathcal{A}$ such that $x_i \in \mathcal{L}(q_i)$, for each $i \geq 0$.

A labeled generalized Büchi automaton $\mathcal{A} = \langle\langle\mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}\rangle, \mathcal{D}, \mathcal{L}\rangle$ as defined above can be translated into an equivalent traditionally defined Büchi automaton $\mathcal{A}' = \langle\Sigma, \mathcal{S}, \mathcal{S}_0, \delta', \mathcal{F}'\rangle$ by replacing each transition in $\mathcal{A}$ with $|\mathcal{L}(s)|$ transitions each of which is labeled with an element of $\mathcal{L}(s)$. Formally:

- $\Sigma = \mathcal{D}$
- $\mathcal{S} = \mathcal{Q}$
- $\mathcal{S}_0 = \mathcal{I}$
- $\delta'(s, a) = \{s' : s' \in \delta(s) \text{ and } a \in \mathcal{L}(s)\}$
- $F' = \mathcal{F}$

The algorithm for turning LTL formulas into Büchi automata is presented in two phases. First, we introduce the automaton structure, i.e., its states, which are obtained as covers, initial states, and transition function. The line numbers in the following description refer to this part of the algorithm, which appears in Figure 5.14. Then, we complete such structure by defining labeling and acceptance conditions.

The algorithm starts by computing the initial states as cover of $\{\psi\}$ (line 2). A set U of states whose transition function has still to be defined is kept. All the initial states are clearly added to U (line 2). When defining the transition function for the state s (line 4), we first compute its successors as cover of $\{\mu : X\mu \in s\}$ (line 5). For each computed successor r, the algorithm checks whether r has been previously generated as a state r' (line 6). If so, it suffices to add r' to $\delta(s)$ (line 7). Otherwise, r is added to Q and $\delta(s)$ (lines 8 and 9). Moreover, r is also added to U (line 10), for $\delta(r)$ to be eventually computed.

The domain $\mathcal{D}$ is $2^{\mathcal{P}}$ and the label of a state s consists of all subsets of $2^{\mathcal{P}}$ that are compatible with the propositional information contained in s. More in detail, let $Pos(s)$ be $s \cap \mathcal{P}$ and $Neg(s)$ be $\{p \in \mathcal{P} : \neg p \in s\}$. Then, $\mathcal{L}(s) = \{X : X \subseteq \mathcal{P} \wedge Pos(s) \subseteq X \wedge X \cap Neg(s) = \emptyset\}$. Finally, we have to impose acceptance conditions. Indeed, our construction allows some executions inducing interpretations that are not models of ψ. This happens because it is possible to procrastinate forever the fulfilling of U-formulas, and

```
1  procedure create_automaton_structure(ψ)
2    U = Q = I = Cover({ψ}), δ = ∅
3    while U ≠ ∅
4          remove s from U
5          for r ∈ Cover({μ : Xμ ∈ s})
6              if ∃r' ∈ Q such that r = r'
7              then δ(s) = δ(s) ∪ {r'}
8              else  Q = Q ∪ {r}
9                    δ(s) = δ(s) ∪ {r}
10                   U = U ∪ {r}
```

Figure 5.14: The algorithm.

arises because the formula $\mu\,\mathcal{U}\eta$ can be covered by covering μ and by promising to fulfill it later by covering $X(\mu\,\mathcal{U}\eta)$. The problem is solved by imposing generalized Büchi acceptance conditions. Informally, for each subformula $\mu\,\mathcal{U}\eta$ of ψ, we define a set $F_{\mu\mathcal{U}\eta} \in \mathcal{F}$ containing states s that either do not promise it or immediately fulfill it. In this way, postponing forever fulfilling a promised $\mathcal{U}$-formula gives not rise to accepting executions anymore. Formally, we set $F_{\mu\mathcal{U}\eta} = \{s \in \mathcal{Q} : \textsc{satisfy}(s, \mu\,\mathcal{U}\eta) \rightarrow \textsc{satisfy}(s, \eta)\}$ where, again, SATISFY is a function that will be subject to instantiation.

GPVW, GPVW+, and LTL2AUT

GPVW is obtained by instantiating the Boolean functions parameterizing the previously described core in the following way. HAS_TO_BE_STORED(μ) returns T. CONTRADICTION($\mu, ToCover, Current, Covered$) returns T iff μ is F or μ is a literal such that $\neg\mu \in Current$. REDUNDANT($\mu, ToCover, Current, Covered$) returns F. SATISFY($s, \mu$) returns T iff $\mu \in s$.

For GPVW+ we have the following instantiations. HAS_TO_BE_STORED(μ) returns T iff μ is a $\mathcal{U}$-formula or μ is the righthand argument of a $\mathcal{U}$-formula. CONTRADICTION(μ, $ToCover$, $Current$, $Covered$) returns T iff μ is F or the negation normal form of $\neg\mu$ is in $Covered$. REDUNDANT(μ, $ToCover$, $Current$, $Covered$) returns T iff μ is $\eta\mathcal{U}\nu$ and $\nu \in ToCover \cup Current$, or μ is $\eta\mathcal{V}\nu$ and η, $\nu \in ToCover \cup Current$. SATISFY($s, \mu$) returns T iff $\mu \in s$.

GPVW+ attempts to generate less states than GPVW by reducing the formulas to store in $Current$ and by detecting redundancies and contradictions as soon as possible. Indeed, by reducing the formulas to store in $Current$, GPVW+ increases the possibility of finding matching states, while early detection of contradictions and redundancies avoids producing the part of the automaton for dealing with them. However, GPVW+ still does not solve

some basic problems. First, states obtained by dealing with a $\mathcal{U}$-formula contain either the $\mathcal{U}$-formula or its righthand argument. So, for example, states generated for the righthand argument of $\mu\mathcal{U}\eta$ are equivalent to, but do not match, prior existing states generated for η. Second, redundancy and contradiction checks are performed by explicitly looking for the source of redundancy or contradiction. So, for example, a $\mathcal{U}$-formula whose righthand argument is a conjunction is considered redundant if such conjunction appears among the covered formulas, but it is not if, instead of the conjunction, its conjuncts are present.

LTL2AUT overcomes the above problems in a very simple way: Only the elementary formulas are stored in $Current$, while information about the nonelementary ones is derived from the elementary ones and the ones stored in $ToCover$ using quick syntactic techniques. More in detail, we inductively define the set $\mathcal{SI}(A)$ of the formulas *syntactically implied* by the set of formulas A as follows

- $\mathrm{T} \in \mathcal{SI}(A)$,
- $\mu \in \mathcal{SI}(A)$, if $\mu \in A$,
- $\mu \in \mathcal{SI}(A)$, if μ is non-elementary and either $\alpha_1(\mu) \subseteq \mathcal{SI}(A)$ or $\alpha_2(\mu) \subseteq \mathcal{SI}(A)$.

LTL2AUT requires then the following settings. HAS_TO_BE_STORED(μ) returns F. CONTRADICTION(μ, $ToCover$, $Current$, $Covered$) returns T iff the negation normal form of $\neg\mu$ belongs to $\mathcal{SI}(ToCover \cup Current)$. REDUNDANT($\mu$, $ToCover$, $Current$, $Covered$) returns T iff $\mu \in \mathcal{SI}(ToCover \cup Current)$ and, if μ is $\eta\mathcal{U}\nu$, $\nu \in \mathcal{SI}(ToCover \cup Current)$. SATISFY($s, \mu$) returns T iff $\mu \in \mathcal{SI}(s)$. The special attention to the righthand arguments of $\mathcal{U}$-formulas in the redundancy check is for avoiding discarding information required to define the acceptance conditions.

The Test Method

The method we have adopted is based on two analyses:

Average-behavior analysis: For a fixed number N of propositional variables and for increasing values L of the length of the formulas, a problem set $\mathcal{PS}_{\langle F,N,L\rangle}$ of F random formulas is generated and given in input to the procedures to test. After the computation, a statistical analysis is performed and the results are plotted against L. The process can be repeated for different values of N.

Temporal-behavior analysis: For a fixed number N of propositional variables, a fixed length L of the formulas, and for increasing values P of

the probability of generating the temporal operators $\mathcal{U}$ and $\mathcal{V}$, a problem set $\mathcal{PS}_{\langle F,N,L,P\rangle}$ of F random formulas is generated and given in input to the procedures to test. After the computation, a statistical analysis is performed and the results are plotted against P. The process can be repeated for different values of N and L.

When generating random formulas from a formula space, for example defined by the parameters N, L, and P, our target is to cover such space as uniformly as possible. This requires that, when generating formulas of length L, we produce formulas of length exactly L, and not up to L. Indeed, in the latter way, varying L, we give preference to short formulas. Random formulas parameterized by N, L, and P, are then generated as follows. A unit-length random formula is generated by randomly choosing, according to uniform distribution, one variable. From now on, unless otherwise specified, randomly chosen stands for randomly chosen with uniform distribution. A random formula of length 2 is generated by generating $op(p)$, where op is randomly chosen in $\{\neg, X\}$ and p is a randomly chosen variable. Otherwise, with probability $\frac{P}{2}$ of choosing either $\mathcal{U}$ or $\mathcal{V}$ and probability $\frac{1-P}{4}$ of choosing $\neg$, X, $\wedge$, or $\vee$, the operator op is randomly chosen. If op is unary, the random formula of length L is generated as $op(\mu)$, for some random formula μ of length $L-1$. Otherwise, if op is binary, for some randomly chosen $1 \leq S \leq L-2$, two random formulas μ_1 and μ_2 of length S and $L-S-1$ are produced, and the random formula $op(\mu_1, \mu_2)$ of length L is generated. Since the set of operators we use is $\{\neg, X, \wedge, \vee, \mathcal{U}, \mathcal{V}\}$, random formulas for the average-behavior analysis are generated by setting $P = \frac{1}{3}$. Note that parentheses are not considered. Indeed, our definition generates a syntax tree that makes the priority between the operators clear.

In both the above analyses, the parameters we are interested in are the size of the automata, namely states and transitions, and the time required for their generation. When comparing two procedures Π_1 and Π_2 with respect to some problem set $\mathcal{PS}_{\langle F,N,L,P\rangle}$ and parameter θ, we perform the following statistical analyses.

- $\frac{E(\Pi_1,\theta,\mathcal{PS}_{\langle F,N,L,P\rangle})}{E(\Pi_2,\theta,\mathcal{PS}_{\langle F,N,L,P\rangle})}$: first, we compute the mean value of the outputs of Π_1 and Π_2 separately, and then consider their ratio.

- $E(\frac{\Pi_1}{\Pi_2},\theta,\mathcal{PS}_{\langle F,N,L,P\rangle})$: we first compute the ratio of the outputs of Π_1 and Π_2 separately for each sample of the problem set, and then the mean value of such ratios.

Results

LTL2AUT, GPVW, and GPVW+ have been implemented on the top of the same kernel, and are accessible through command line options. The code

consists of 1400 lines of `C` plus 110 lines for a `lex/yacc` parser. The code has been compiled through `gcc version 2.7.2.3` and executed under the `SunOS 5.5.1` operating system on a `SUNW UltraSPARC-II/296 1G`.

We start by comparing the three algorithms with respect to the first statistical analysis. LTL2AUT and GPVW+ have been compared, according to the test method discussed in Section 5.2.5, on 5700 randomly generated formulas. The results are shown in Figure 5.15. For the average behavior analysis, LTL2AUT and GPVW+ have been compared on 3300 random formulas generated, according to our test method, for $F = 100$, $N = 1, 2, 3$, and $L = 5, 10, \ldots, 55$. Formulas have been collected in 3 groups, for $N = 1, 2, 3$, and inside each group partitioned into 11 problem sets of 100 formulas each, for $L = 5, 10, \ldots, 55$. For each group, $\frac{E(\text{LTL2AUT, states }, \mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, states }, \mathcal{PS}_{\langle 100,N,L\rangle})}$, $\frac{E(\text{LTL2AUT, transitions }, \mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, transitions }, \mathcal{PS}_{\langle 100,N,L\rangle})}$, and $\frac{E(\text{LTL2AUT, time }, \mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, time }, \mathcal{PS}_{\langle 100,N,L\rangle})}$ have been plotted against L. The results show that LTL2AUT clearly outperforms GPVW+, with respect to both the size of automata and computation time. Indeed, just considering formulas of length 30, LTL2AUT produces on the average less than 60% of the states of GPVW+ (for transitions situation is even better) spending on the average less than 30% of the time of GPVW+. Moreover, the initial phase, in which LTL2AUT does have a time overhead with respect to GPVW+, affects formulas, for $L = 5$ and $N = 3$, which are solved by LTL2AUT in at most 0.000555 CPU seconds, as opposed to the most demanding sample for $L = 55$ and $N = 3$, which is solved by LTL2AUT in 6659 CPU seconds. For the temporal-behavior analysis, LTL2AUT and GPVW+ have been compared over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, $L = 20, 30$, and $P = 0.\overline{3}, 0.5, 0.7, 0.95$. Note that $P = 0.\overline{3}$ is the probability we have assumed for the average-behavior analysis. Formulas have been collected in 3 groups, for $N = 1, 2, 3$, and inside each group partitioned into 2 sub-groups, for $L = 20, 30$. Each sub-group has still been partitioned into 4 problem sets, for $P = 0.\overline{3}, 0.5, 0.7, 0.95$. For each sub-group, we have plotted $\frac{E(\text{LTL2AUT, states }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, states }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}$, $\frac{E(\text{LTL2AUT, transitions }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, transitions }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}$, and $\frac{E(\text{LTL2AUT, time }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, time }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}$ against P. Again, the results demonstrate that LTL2AUT clearly outperforms GPVW+.

The comparison between GPVW+ and GPVW, whose results are shown in Figure 5.16, follows the lines of the previous one, by only changing some parameters for allowing GPVW to compute in reasonable time. The average-behavior analysis has been carried out over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, and $L = 5, 10, \ldots, 40$. The temporal-behavior analysis has been performed over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, and $P = 0.\overline{3}, 0.5, 0.7, 0.95$. The results show that

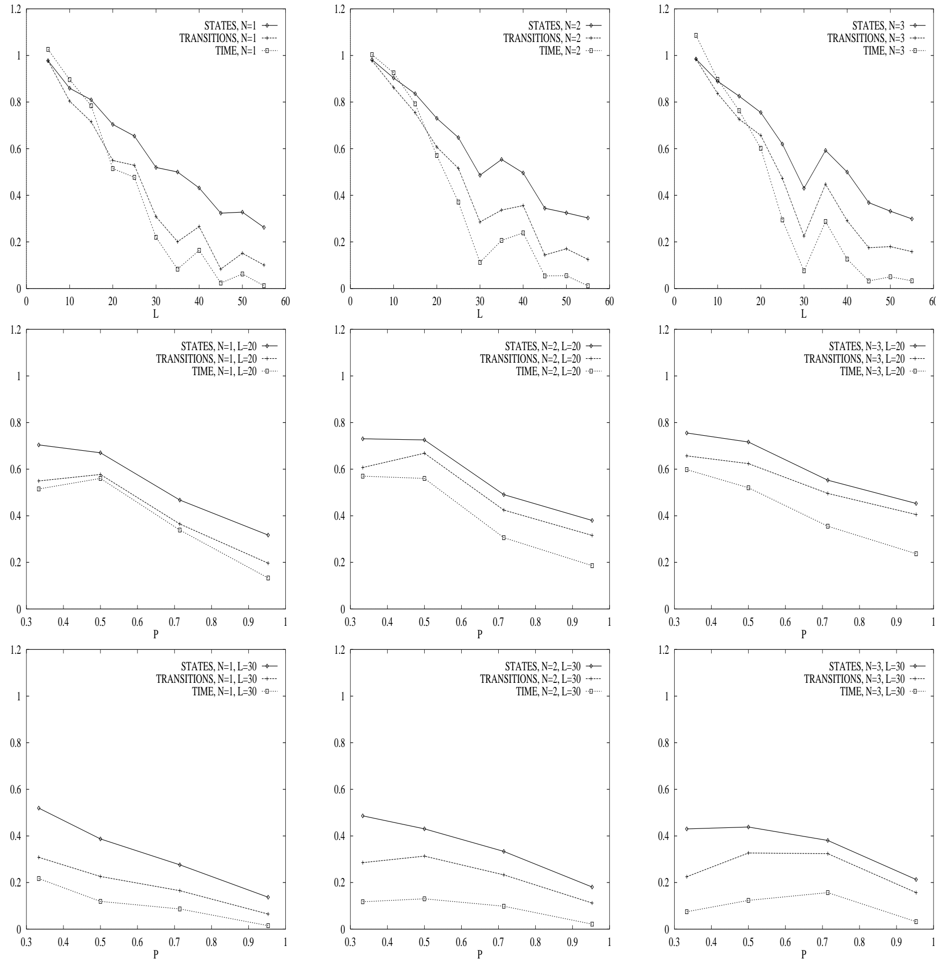

Figure 5.15: LTL2AUT vs. GPVW+. Upper row: Average-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 55$. Middle and lower rows: Temporal-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 20, 30$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

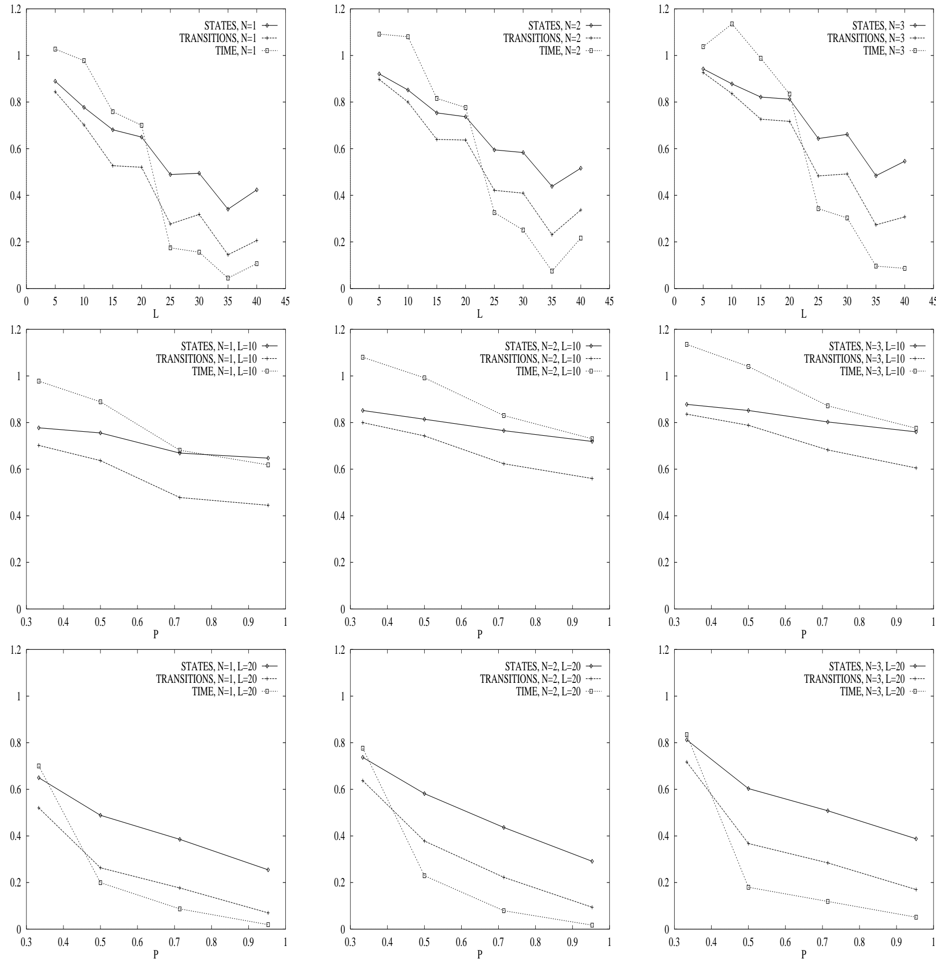

Figure 5.16: GPVW+ vs. GPVW. Upper row: Average-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 40$. Middle and lower rows: Temporal-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

GPVW+ clearly outperforms GPVW both in the size of automata and, after an expected initial phase, also in time. The initial phase interests formulas, for $L = 10$ and $N = 3$, which are solved by GPVW+ in at most 0.004226 CPU seconds, as opposed to the hardest sample for $L = 40$ and $N = 3$, which is solved by GPVW+ in 178 CPU seconds.

The direct comparison between LTL2AUT and GPVW is shown in Figure 5.17. Note that LTL2AUT behaves better than GPVW+ in the initial phase, in which both LTL2AUT and GPVW+ pay a time overhead with respect to GPVW.

Finally, with respect to the second statistical analysis, the results of the comparison are shown in Figures 5.18, 5.19, and 5.20. This analysis gives equal weight to easy and hard instances, while the previous one gives more weight to hard instances. Therefore, since the outputs related to samples belonging to same problem sets turn out to be very heterogeneous (up to 5 orders of magnitude), the comparison of this analysis and the previous one indicates that the gap among the procedures increases for hard formulas.

Proof of Correctness of LTL2AUT

The main theorem is the following:

Theorem 5.2.1 *The automaton $\mathcal{A}(\psi)$ constructed for the LTL formula ψ recognizes exactly all the models of ψ.*

Proof: The two directions are proved in Lemma 5.2.8 and Lemma 5.2.11 below.

Let us first extend some definitions.

Definition 5.2.1 *A* pseudo-execution *ρ is an infinite sequence $s_0 s_1 s_2 \ldots$ of states such that, for each $i \geq 0$, $s_{i+1} \in \delta(s_i)$. ρ is* accepting *if, for each $F_i \in \mathcal{F}$, there exists $q_i \in F_i$ that appears infinitely often in ρ. ρ* accepts *the word ξ if it is accepting and, for $i \geq 0$, $\xi(i) \in \mathcal{L}(s_i)$. A* finite pseudo-execution *ρ is a finite sequence $s_0 s_1 s_2 \ldots s_n$ of states such that, for each $i \in \{0, \ldots, n-1\}$, we have that $s_{i+1} \in \delta(s_i)$.*

Let us note that the cover computation shown in Figure 5.13 when instantiated for LTL2AUT does not uses the argument *Covered*. For this reason, in what follows we will not mention such argument. Moreover, in order to simplify the notation, we write A instead of $\bigwedge_{f \in A} f$, or TRUE in the case of the empty set, for a set A of formulas.

Lemma 5.2.1 *Let ToCover be a set of formulas, Current be elementary set, and $Cover = \{Cover_i : i \in I\}$ be a set of elementary sets. Then the call* **cover**$(ToCover, Current, Cover)$ *returns a set $\{C_j : j \in J\}$ such that, for all $j \in J$*

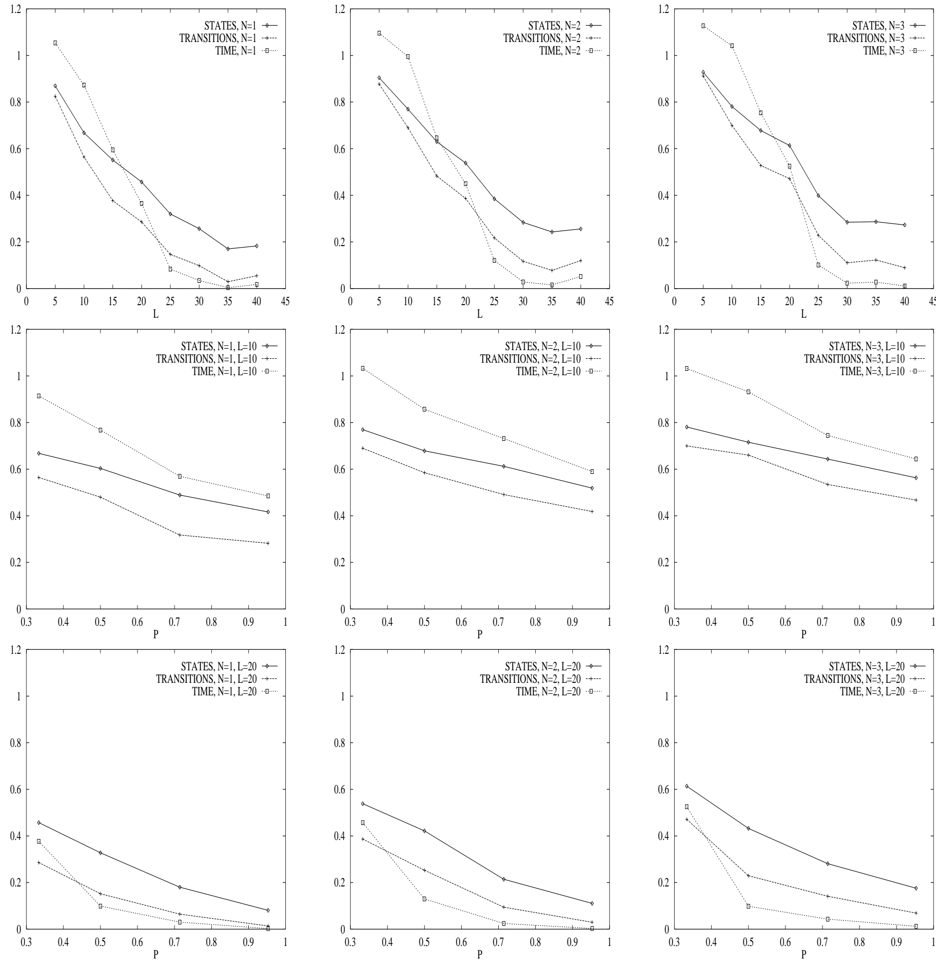

Figure 5.17: LTL2AUT vs. GPVW. Upper row: Average-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 40$. Middle and lower rows: Temporal-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

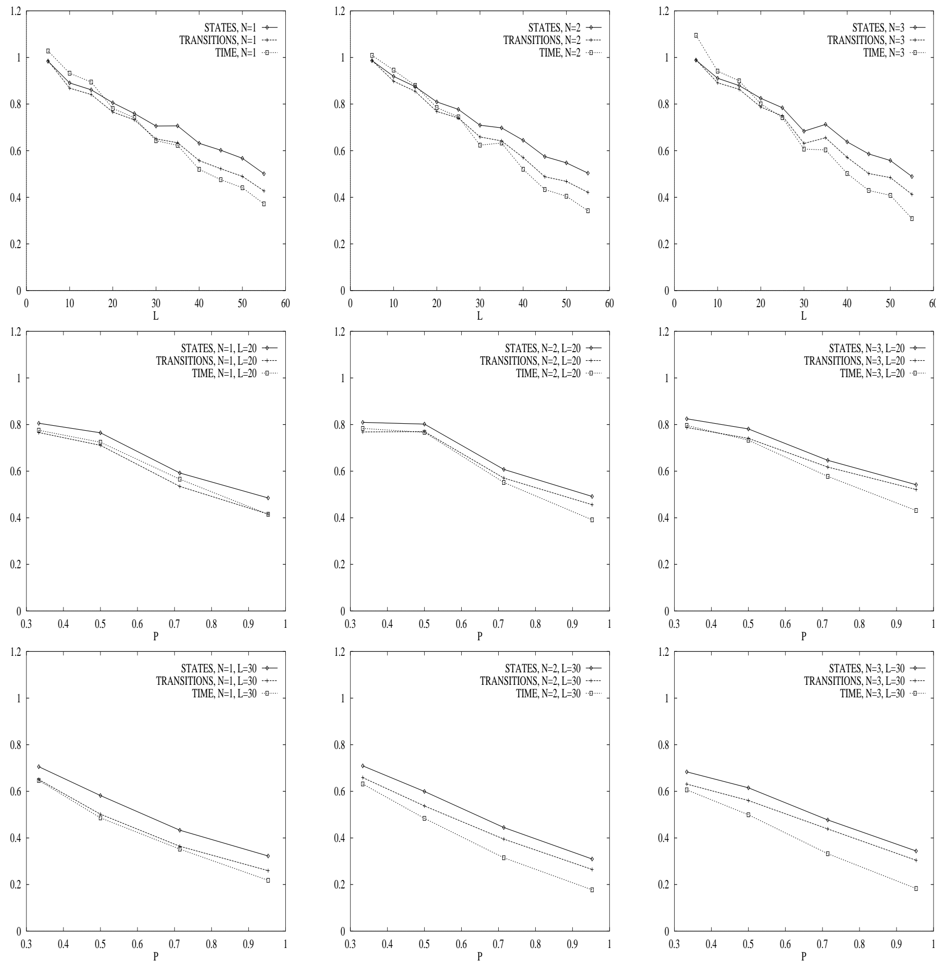

Figure 5.18: LTL2AUT vs. GPVW+. Upper row: Average behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 55$. Middle and lower rows: Temporal behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 20, 30$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

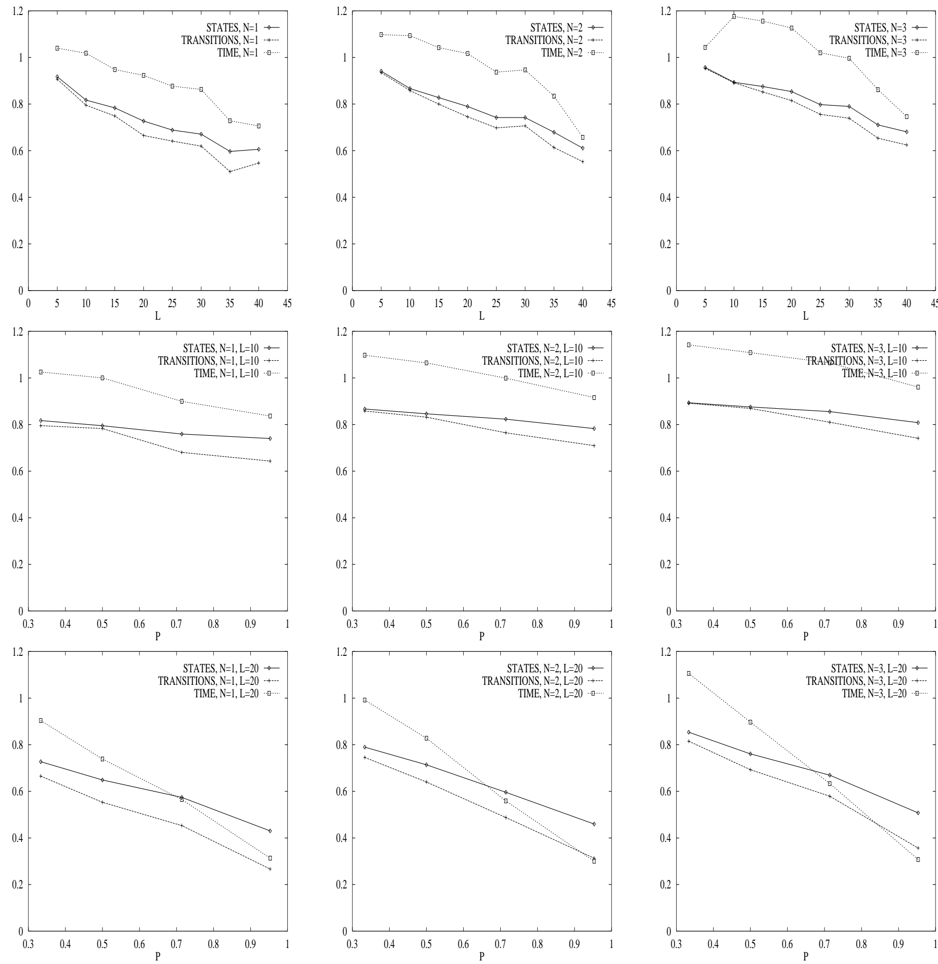

Figure 5.19: GPVW+ vs. GPVW. Upper row: Average behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 40$. Middle and lower rows: Temporal behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

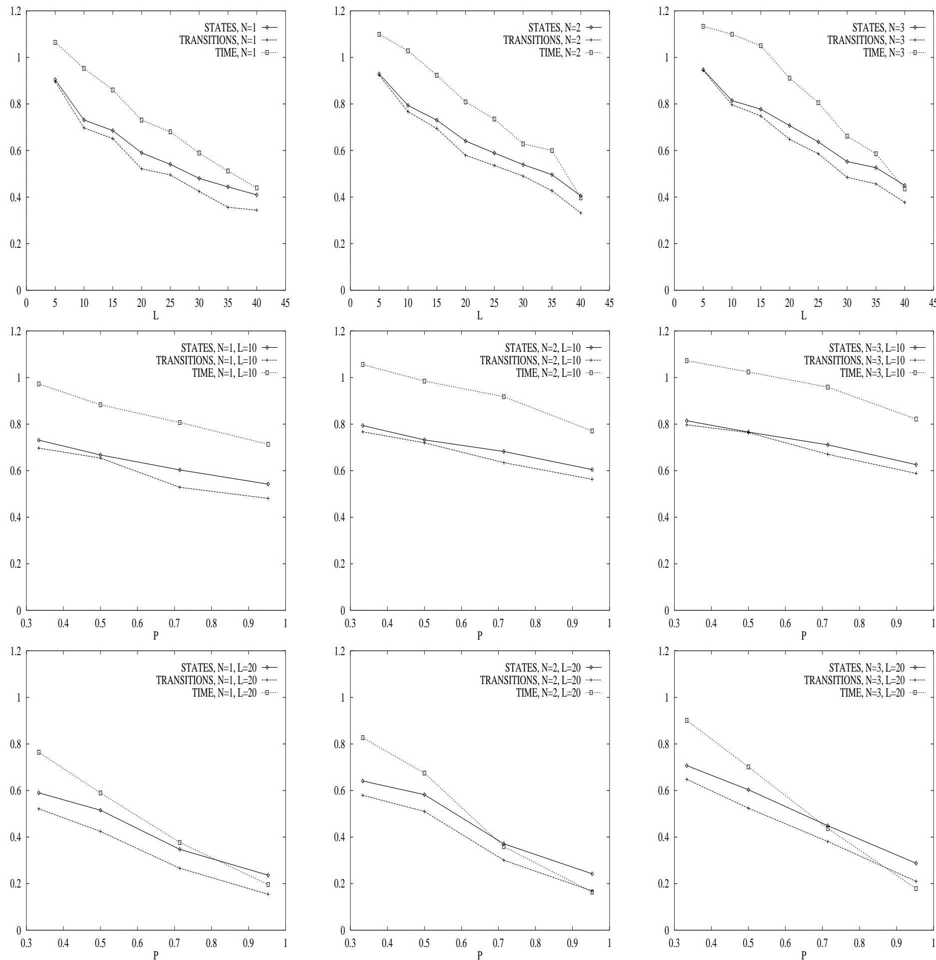

Figure 5.20: LTL2AUT vs. GPVW. Upper row: Average behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 40$. Middle and lower rows: Temporal behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

1. *C_j is elementary*

2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$

3. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$

Proof: By induction over $n = \sum_{f \in ToCover} \mathcal{C}(\mu)$ where

$$\mathcal{C}(\mu) = \begin{cases} 1 & \text{if } \mu \text{ is elementary} \\ \mathcal{C}(\mu_1) + \mathcal{C}(\mu_2) + 1 & \text{if } \mu \text{ is } \mu_1 \vee \mu_2 \text{ or } \mu_1 \wedge \mu_2 \\ \mathcal{C}(\mu_1) + \mathcal{C}(\mu_2) + 2 & \text{if } \mu \text{ is } \mu_1 \mathcal{U} \mu_2 \text{ or } \mu_1 \mathcal{V} \mu_2 \end{cases}$$

If $n = 0$ then $ToCover = \emptyset$ and the function returns $\{Current\} \cup Cover$. Then, for each $j \in J$

1. C_j is elementary, due to the hypothesis.

2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow Current \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$.

3. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i = Current \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$, since C_j is either some $Cover_i$ or $Current$.

If $n > 0$, let $f \in ToCover$ be the chosen formula and $ToCover' = ToCover \setminus \{f\}$. Let us consider the possible cases. If a contradiction in $ToCover \cup Current$ is detected because the negation normal form of $\neg f$ belongs to $\mathcal{SI}(ToCover' \cup Current)$, then the function returns $Cover$ and

1. C_j is elementary, due to the hypothesis.

2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{i \in I} Cover_i = \bigvee_{j \in J} C_j$.

3. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$, since C_j is some $Cover_i$.

If f is redundant in $ToCover' \cup Current$, that is, $f \in \mathcal{SI}(ToCover' \cup Current)$, then the function returns **cover**$(ToCover', Current, Cover)$. By inductive hypothesis, this latter call returns a set $\{C_j : j \in J\}$ such that, for all $j \in J$

1. C_j is elementary.

2. $(ToCover' \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$.

3. $(ToCover' \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$.

Therefore, for all $j \in J$

1. C_j is elementary.

2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$, since $f \in \mathcal{SI}(ToCover' \cup Current)$.

3. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$, since $f \in \mathcal{SI}(ToCover' \cup Current)$.

Let us now consider the cases when f is handled according to its syntactic structure. If f is elementary, then the function returns **cover**$(ToCover', Current \cup \{f\}, Cover)$. By inductive hypothesis, this latter call returns a set $\{C_j : j \in J\}$ such that, for all $j \in J$

1. C_j is elementary.
2. $(ToCover' \cup Current \cup \{f\}) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$.
3. $(ToCover' \cup Current \cup \{f\}) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$.

Therefore

1. C_j is elementary.
2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{j \in J} C_j$.
3. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$.

Finally, if f is nonelementary, the function returns **cover**$(ToCover' \cup (\alpha_1(f) \setminus Current), Current,$ **cover**$(ToCover' \cup (\alpha_2(f) \setminus Current), Current, Cover)$. By inductive hypothesis, the inner call returns $\{D_h : h \in H\}$ such that, for all $h \in H$

1. D_h is elementary.
2. $(ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow \bigvee_{h \in H} D_h$.
3. $(ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(D_h)$.

while the outer call returns $\{C_j : j \in J\}$ such that

1. C_j is elementary.
2. $(ToCover' \cup (\alpha_1(f) \setminus Current) \cup Current) \vee \bigvee_{h \in H} D_h \leftrightarrow \bigvee_{j \in J} C_j$.
3. $(ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current) \vee \bigvee_{h \in H} D_h \in \mathcal{SI}(C_j)$.

Thus, for each $j \in J$

1. C_j is elementary.
2. $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow (ToCover' \cup \{\alpha_1(f) \vee \alpha_2(f)\} \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow (ToCover' \cup \{\alpha_1(f)\} \cup Current) \vee (ToCover' \cup \{\alpha_2(f)\} \cup Current) \vee \bigvee_{i \in I} Cover_i \leftrightarrow (ToCover' \cup \{\alpha_2(f)\} \cup Current) \vee \bigvee_{h \in H} D_h \leftrightarrow \bigvee_{j \in J} C_j$.

3. If $(ToCover' \cup (\alpha_1(f) \setminus Current) \cup Current) \vee \bigvee_{h \in H} D_h \in \mathcal{SI}(C_j)$ due to $(ToCover' \cup (\alpha_1(f) \setminus Current) \cup Current)$, then $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$. Otherwise, for some $h \in H$, $D_h \in \mathcal{SI}(C_j)$ and, therefore, $(ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$, that is, $(ToCover \cup Current) \vee \bigvee_{i \in I} Cover_i \in \mathcal{SI}(C_j)$

Corollary 5.2.1 *Let ToCover be a set of formulas, then the call* **Cover**(A) *returns a set* $\{C_j : j \in J\}$ *such that, for all* $j \in J$

1. C_j *is elementary.*
2. $ToCover \leftrightarrow \bigvee_{j \in J} C_j$.
3. $ToCover \in \mathcal{SI}(C_j)$.

Lemma 5.2.2 *Let* s *be a state of* $\mathcal{A}(\psi)$ *such that* $\mathbf{X}(f) \in \mathcal{SI}(s)$. *Then, for all successors* r *of* s, *we have that* $f \in \mathcal{SI}(r)$.
Proof: If $\mathbf{X}(f) \in \mathcal{SI}(s)$ then $\mathbf{X}(f) \in s$. Since s' successors have been computed as **Cover**$(\{f : \mathbf{X}(f) \in s\})$, we conclude by Corollary 5.2.1 (3).

Lemma 5.2.3 *Let* $s_0 s_1 \ldots s_n$ *be a finite pseudo-execution of* $\mathcal{A}(\psi)$ *such that* $f \mathcal{U} g \in \mathcal{SI}(s_0)$. *Then one of the following holds:*

1. *For all* $i \in \{0, \ldots, n\}$, *we have that* $f, f \mathcal{U} g \in \mathcal{SI}(s_i)$ *and* $g \notin \mathcal{SI}(s_i)$.
2. *There exists* $i \in \{0, \ldots, n\}$ *such that* $g \in \mathcal{SI}(s_i)$ *and, for all* $0 \leq j < i$, $f, f \mathcal{U} g \in \mathcal{SI}(s_j)$.

Proof: By induction on n. If $n = 0$, we conclude by definition of $\mathcal{SI}()$. Otherwise, by definition of $\mathcal{SI}()$, either $g \in \mathcal{SI}(s_0)$, and (2) holds, or $g \notin \mathcal{SI}(s_0)$ but $f, \mathbf{X}(f \mathcal{U} g) \in \mathcal{SI}(s_0)$. In this latter case, by Lemma 5.2.2, $f \mathcal{U} g \in \mathcal{SI}(s_1)$ and we conclude by inductive hypothesis.

Lemma 5.2.4 *Let* $s_0 s_1 \ldots$ *be a pseudo-execution of* $\mathcal{A}(\psi)$ *such that* $f \mathcal{U} g \in \mathcal{SI}(s_0)$. *Then one of the following holds:*

1. *For all* $i \geq 0$, *we have that* $f, f \mathcal{U} g \in \mathcal{SI}(s_i)$ *and* $g \notin \mathcal{SI}(s_i)$.
2. *There exists* $i \geq 0$ *such that* $g \in \mathcal{SI}(s_i)$ *and, for all* $0 \leq j < i$, $f, f \mathcal{U} g \in \mathcal{SI}(s_j)$.

Proof: If (1) does not hold, there exists $\hat{\imath} \geq 0$ such that either $\{f, f \mathcal{U} g\} \not\subseteq \mathcal{SI}(s_{\hat{\imath}})$ or $g \in \mathcal{SI}(s_{\hat{\imath}})$. Consider the finite sequence $s_0 s_1 \ldots s_{\hat{\imath}}$. For Lemma 5.2.3, we conclude that (2) holds.

Lemma 5.2.5 *Let* $s_0 s_1 \ldots s_n$ *be a finite pseudo-execution of* $\mathcal{A}(\psi)$ *such that* $f \mathcal{V} g \in \mathcal{SI}(s_0)$. *Then one of the following holds:*

1. *For all* $i \in \{0, \ldots, n\}$, *we have that* $g, f\mathcal{V}g \in \mathcal{SI}(s_i)$ *and* $f \notin \mathcal{SI}(s_i)$.

2. *There exists* $i \in \{0, \ldots, n\}$ *such that* $f, g \in \mathcal{SI}(s_i)$ *and, for all* $0 \leq j < i$, $g, f\mathcal{V}g \in \mathcal{SI}(s_j)$ *and* $f \notin \mathcal{SI}(s_j)$.

Proof: By induction on n. If $n = 0$, we conclude by definition of $\mathcal{SI}()$. Otherwise, by definition of $\mathcal{SI}()$, $g \in \mathcal{SI}(s_0)$ and either $f \in \mathcal{SI}(s_0)$, and (2) holds, or $f \notin \mathcal{SI}(s_0)$ but $\mathbf{X}(f\mathcal{V}g) \in \mathcal{SI}(s_0)$. In this latter case, by Lemma 5.2.2, $f\mathcal{V}g \in \mathcal{SI}(s_1)$ and we conclude by inductive hypothesis.

Lemma 5.2.6 *Let* $s_0 s_1 \ldots$ *be a pseudo-execution of* $\mathcal{A}(\psi)$ *such that* $f\mathcal{V}g \in \mathcal{SI}(s_0)$. *Then one of the following holds:*

1. *For all* $i \geq 0$, *we have that* $g, f\mathcal{V}g \in \mathcal{SI}(s_i)$ *and* $f \notin \mathcal{SI}(s_i)$.

2. *There exists* $i \geq 0$ *such that* $f, g \in \mathcal{SI}(s_i)$ *and, for all* $0 \leq j < i$, $g, f\mathcal{V}g \in \mathcal{SI}(s_j)$ *and* $f \notin \mathcal{SI}(s_j)$.

Proof: If (1) does not hold, there exists $\hat{\imath} \geq 0$ such that either $\{g, f\mathcal{V}g\} \not\subseteq \mathcal{SI}(s_{\hat{\imath}})$ or $f \in \mathcal{SI}(s_{\hat{\imath}})$. Consider the finite sequence $s_0 s_1 \ldots s_{\hat{\imath}}$. For Lemma 5.2.5, we conclude that (2) holds.

Lemma 5.2.7 *Let* $\rho = s_0 s_1 \ldots$ *be an accepting pseudo-execution of* $\mathcal{A}(\psi)$ *over* ξ. *Then* $\xi \models \mathcal{SI}(s_0)$.

Proof: By induction over the structure of the formulas in $\mathcal{SI}(s_0)$. Note that, due to the contradiction check in the function **cover**, for all the states s of $\mathcal{A}(\psi)$, we have that FALSE $\notin s$. The base case is then for TRUE, for which the thesis trivially follows, and for the propositional literals, for which the thesis holds because of the definition of the labeling.

If $f \vee g \in \mathcal{SI}(s_0)$, it has to be that either $f \in \mathcal{SI}(s_0)$ or $g \in \mathcal{SI}(s_0)$ and we conclude by inductive hypothesis.

In the case $f \wedge g \in \mathcal{SI}(s_0)$, it has to be that both $f \in \mathcal{SI}(s_0)$ and $g \in \mathcal{SI}(s_0)$ and we conclude by inductive hypothesis.

When $\mathbf{X}(f) \in \mathcal{SI}(s_0)$, by Lemma 5.2.2 it has to be that $f \in \mathcal{SI}(s_1)$ and we conclude by inductive hypothesis.

If $f\mathcal{U}g \in \mathcal{SI}(s_0)$, we have two cases. If $f\mathcal{U}g$ is not a subformula of ψ, then it has to be that $g \in \mathcal{SI}(s_0)$, and we conclude by inductive hypothesis. Otherwise, since ρ is accepting, according to Lemma 5.2.4, only the second case is possible. We conclude by inductive hypothesis and by semantic definition of the until operator.

If $f\mathcal{V}g \in \mathcal{SI}(s_0)$, we conclude by Lemma 5.2.6, by inductive hypothesis, and by semantic definition of the release operator.

Lemma 5.2.8 *Let ρ be an accepting execution of $\mathcal{A}(\psi)$ over ξ. Then $\xi \models \psi$.*
Proof: It follows by Lemma 5.2.7, by definition of initial states, and by Lemma 5.2.1 (3).

Lemma 5.2.9 *Let $ToCover$ and $Current$ be set of formulas, and $Cover$ be set of sets. Then $Cover \subseteq$ **cover**$(ToCover, Current, Cover)$.*
Proof: By induction on $n = \sum_{f \in ToCover} \mathcal{C}(f)$. If $n = 0$, then $ToCover = \emptyset$ and $Cover \subseteq Cover \cup \{Current\} =$ **cover**$(ToCover, Current, Cover)$.

If $n > 0$, let f be the selected formula and $ToCover' = ToCover \setminus \{f\}$. Let us consider the several cases. If a contradiction is detected, then $Cover =$ **cover**$(ToCover, Current, Cover)$.

If a redundancy is detected, then we conclude by inductive hypothesis, since **cover**$(ToCover, Current, Cover) =$ **cover**$(ToCover', Current, Cover)$.

Let us now consider the cases when f is handled according to its syntactic structure. If f is elementary, we conclude by inductive hypothesis, since **cover**$(ToCover, Current, Cover) =$ **cover**$(ToCover', Current \cup \{f\}, Cover)$. Finally, if f is nonelementary, we have that **cover**$(ToCover', Current \cup \{f\}, Cover) =$ **cover**$(ToCover' \cup (\alpha_1(f) \setminus Current), Current,$ **cover**$(ToCover' \cup (\alpha_2(f) \setminus Current), Current, Cover))$ and, again, we conclude by inductive hypothesis.

Lemma 5.2.10 *Let $ToCover$ be a set of formulas, $\{f_i \mathcal{U} g_i : i \in I\} \subseteq ToCover$, $Current$ be elementary set, $Cover$ be set of elementary sets, and ξ such that $\xi \models ToCover \cup \{g_i : i \in I\}$. Then, the call* **cover**$(ToCover, Current, Cover)$ *returns the set $\{C_j : j \in J\}$ containing the elementary set $C_{\hat{\jmath}}$ such that*

1. *$\xi \models C_{\hat{\jmath}}$.*

2. *$ToCover \cup Current \cup Cover\{g_i : i \in I\} \in \mathcal{SI}(C_{\hat{\jmath}})$.*

Proof: By induction over $n = \sum_{f \in ToCover} \mathcal{C}(f)$. If $n = 0$ then $ToCover = \emptyset$ and therefore **cover**$(ToCover, Current, Cover) = \{Current\} \cup Cover$. Choosing $C_{\hat{\jmath}} = Current$ we have

1. $\xi \models C_{\hat{\jmath}} = ToCover \cup Current \cup \{g_i : i \in G\}$, by initial hypothesis.

2. $ToCover \cup Current \cup \{g_i : i \in G\} = Current \in \mathcal{SI}(C_{\hat{\jmath}})$.

If $n > 0$, let f be the selected formula and $ToCover' = ToCover \setminus \{f\}$. Because of the initial hypothesis, it cannot be the case of contradiction.

If f is redundant we have two cases. First, let f be $f_{\hat{\imath}} \mathcal{U} g_{\hat{\imath}} \in \{f_i \mathcal{U} g_i : i \in I\}$. In this case, by inductive hypothesis, **cover**$(ToCover, Current, Cover) =$ **cover**$(ToCover', Current, Cover) = \{C_j : i \in J\}$ such that for $\hat{\jmath} \in J$

1. $\xi \models C_{\hat{\jmath}}$

2. $ToCover' \cup Current \cup \{g_i : f_i \mathcal{U} g_i \neq f\} \in \mathcal{SI}(C_{\hat{\jmath}})$

Therefore, choosing the same $C_{\hat{\jmath}}$, we conclude since $g_{\hat{\imath}} \in \mathcal{SI}(ToCover' \cup Current)$. Otherwise, if f is not in $\{f_i \mathcal{U} g_i : i \in I\}$, by inductive hypothesis, $\mathbf{cover}(ToCover, Current, Cover) = \mathbf{cover}(ToCover', Current, Cover) = \{C_j : i \in J\}$ such that for $\hat{\jmath} \in J$

1. $\xi \models C_{\hat{\jmath}}$.

2. $ToCover' \cup Current \cup \{g_i : i \in I\} \in \mathcal{SI}(C\hat{\jmath})$.

Again, choosing the same $C_{\hat{\jmath}}$, we conclude because $f \in \mathcal{SI}(ToCover' \cup Current)$.

Let us now analyze the cases when f is handled according to its syntactic structure. If f is elementary, then $\mathbf{cover}(ToCover, Current, Cover) = \mathbf{cover}(ToCover', Current \cup \{f\}, Cover) = \{C_j : i \in J\}$ such that, by inductive hypothesis, there exists $\hat{\jmath} \in J$

1. $\xi \models C_{\hat{\jmath}}$.

2. $ToCover' \cup Current \cup \{f\} \cup \{g_i : i \in I\} \in \mathcal{SI}(C_{\hat{\jmath}})$.

That is, choosing the same $C_{\hat{\jmath}}$, the thesis.

When f is nonelementary, we have two cases. First, let f be $f_{\hat{\imath}} \mathcal{U} g_{\hat{\imath}} \in \{f_i \mathcal{U} g_i : i \in I\}$. In this case we have that $\mathbf{cover}(ToCover, Current, Cover) = \mathbf{cover}(ToCover' \cup (\{g_{\hat{\imath}}\} \setminus Current), Current, \mathbf{cover}(ToCover' \cup (\{f_{\hat{\imath}}, \mathbf{X}(f_{\hat{\imath}} \mathcal{U} g_{\hat{\imath}})\} \setminus Current), Current, Cover)) = \{C_j : i \in J\}$ such that, by inductive hypothesis, there exists $\hat{\jmath} \in J$

1. $\xi \models C_{\hat{\jmath}}$.

2. $ToCover' \cup (\{g_{\hat{\imath}}\} \cup Current) \cup \{g_i : f_i \mathcal{U} g_i \neq f\} \in \mathcal{SI}(C_{\hat{\jmath}})$.

That is, choosing the same $C_{\hat{\jmath}}$, the thesis. Finally, if $f \notin \{f_i \mathcal{U} g_i : i \in I\}$, we have that either $\xi \models ToCover' \cup (\alpha_1(f) \setminus Current) \cup Current \cup \{g_i : i \in I\}$ or $\xi \models ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current \cup \{g_i : i \in I\}$. In the first case, we have that $\mathbf{cover}(ToCover, Current, Cover) = \mathbf{cover}(ToCover' \cup (\alpha_1(f) \setminus Current), Current, \mathbf{cover}(ToCover' \cup (\alpha_2(f) \setminus Current), Current, Cover)) = \{C_j : i \in J\}$ such that, by inductive hypothesis, there exists $\hat{\jmath} \in J$

1. $\xi \models C_{\hat{\jmath}}$.

2. $ToCover' \cup (\alpha_1(f) \setminus Current) \cup Current \cup \{g_i : i \in I\} \in \mathcal{SI}(C_{\hat{\jmath}})$.

That is, by choosing the same $C_{\hat{\jmath}}$, the thesis. In the second case, we have that $\mathbf{cover}(ToCover' \cup (\alpha_2(f) \setminus Current), Current, Cover) = \{C_j : i \in J\}$ and, by inductive hypothesis, there exists $C_{\hat{\jmath}}$ such that

1. $\xi \models C_{\hat{j}}$.

2. $ToCover' \cup (\alpha_2(f) \setminus Current) \cup Current \cup \{g_i : i \in I\} \in \mathcal{SI}(C_{\hat{j}})$.

Choosing the same $C_{\hat{j}}$, we conclude by Lemma 5.2.9.

Corollary 5.2.2 *Let $ToCover$ be a set of formulas, $\{f_i \mathcal{U} g_i : i \in I\} \subseteq ToCover$, and ξ be an LTL interpretation such that $\xi \models ToCover \cup \{g_i : i \in I\}$. Then, the call* **Cover**$(ToCover)$ *returns the set $\{C_j : j \in J\}$ containing the elementary set $C_{\hat{j}}$ such that*

1. $\xi \models C_{\hat{j}}$.

2. $ToCover \cup \{g_i : i \in I\} \in \mathcal{SI}(C_{\hat{j}})$.

Lemma 5.2.11 *Let $\xi \models \psi$. Then ξ is accepted by $\mathcal{A}(\psi)$.*
Proof: Let us show how an accepting run over ξ can be constructed. Since the initial states are generated as **Cover**$(\{\psi\})$, by Corollary 5.2.1 (2), there is a state $s_0 \in \mathcal{I}$ such that $\xi \models s_0$. In general, having built the fragment of execution $s_0 s_1 \ldots s_n$ such that $\xi_i \models s_i$, let us show how to choose s_{n+1}. Let $U_n = \{\mu_k \mathcal{U} \eta_k : k \in K\} \subseteq \mathcal{SI}(s_n)$, $\{\eta_k : k \in K\} \cap \mathcal{SI}(s_n) = \emptyset$, be the set of the until formulas that are not fulfilled immediately. Since $\{\eta_k : k \in K\} \cap \mathcal{SI}(s_n) = \emptyset$, we have that $\xi_n \models \{X(\mu_k \mathcal{U} \eta_k) : k \in K\}$. Therefore, for each $p > 0$, we can define $U_{n,p} = \{\mu_k \mathcal{U} \eta_k \in U_n : p = \min\{q > 0 : \xi_{n+q} \models \eta_k\}\}$. By induction on p, we now prove that for all $\mu_k \mathcal{U} \nu_k \in U_{n,p}$, there exists $1 \leq l \leq p$ such that s_{n+l} is accepting with respect to $\mu_k \mathcal{U} \nu_k$, that is $\nu_k \in \mathcal{SI}(s_{n+l})$. Let us start to consider the case when $p = 1$. The successors of s_n are computed as **Cover**$(\{f : \mathbf{X}(f) \in s_n\}) \supseteq U_{n,1}$ and $\xi_{n+1} \models \{f : \mathbf{X}(f) \in s_n\} \cup \{g_i : f_i \mathcal{U} g_i \in U_{n,1}\}$. Through Corollary 5.2.2, s_{n+1} can be chosen such that $\xi_{n+1} \models s_{n+1}$ and $\{g_i : f_i \mathcal{U} g_i \in U_{n,1}\} \in \mathcal{SI}(s_{n+1})$. Let us consider now the case $p > 1$, and let $\mu_k \mathcal{U} \nu_k \in U_{n,p}$. Since by Corollary 5.2.1 (3) $\mu_k \mathcal{U} \nu_k \in \mathcal{SI}(s_{n+1})$, either $\nu_k \in \mathcal{SI}(s_{n+1})$ or $\mathbf{X}(\mu_k \mathcal{U} \nu_k) \subseteq \mathcal{SI}(s_{n+1})$. In this latter case, $\mu_k \mathcal{U} \nu_k \in U_{n+1,p-1}$, and we conclude by inductive hypothesis.

Chapter 6

Conclusions and Related Work

In this thesis we have advanced the state-of-the-art in planning as model checking according to two directions.

First, we have presented a formal account for weak, strong, and strong cyclic planning in nondeterministic domains. We have formalized the notion of weak plans, i.e., plans that may achieve the goal but are not guaranteed to; strong plans, i.e., plans that are guaranteed to achieve the goal in spite of nondeterminism; and strong cyclic plans, i.e., plans encoding iterative trial-and-error strategies that always have a possibility of terminating and, when they do, are guaranteed to achieve the goal in spite of nondeterminism. More in detail, weak plans are those whose executions satisfy the CTL formula $\mathbf{EF}\mathcal{G}$, strong plans are those whose executions satisfy $\mathbf{AF}\mathcal{G}$, and strong cyclic plans are plans whose executions satisfy $\mathbf{AGEF}\mathcal{G}$, where $\mathcal{G}$ is a propositional formula representing the set of goal states. We have proven that the algorithms given in [15, 19] compute weak and strong plans respectively, and have defined a new symbolic algorithm for strong cyclic planning that is guaranteed to generate strong cyclic plans and to terminate. Indeed, the algorithm given in [18] did not satisfy the formal specification, since it could generate plans whose executions might get stuck inside cycles with no hope of terminating. All the three algorithms have been implemented in MBP, a planner built on top of the symbolic model checker NuSMV [14], which is currently used in an application for the "Italian Space Agency" (ASI) [12]. A future goal is to extend the planning task from the task of finding a plan which leads to a set of states (the goal) to the task of synthesizing a plan which satisfies some specifications in some temporal logic. This makes the planning task very close to controller synthesis [65, 66, 83, 1, 49], which considers both exogenous events and nondeterministic actions. Due to its generality, however, the work

in synthesis does not always allow for concise solutions as state-action tables, i.e., memoryless plans. Moreover, it is to be investigated how it can express and deal with strong cyclic plans.

As we have already underlined, most of the work in planning is focused on deterministic domains and only recently some works have extended classical planners to "contingent" planners, which generate plans with conditionals, or to "conformant" planners that, unrealistically, try to find strong solutions as sequences of actions. Nevertheless, neither existing contingent nor existing conformant planners are able to generate strong cyclic plans. Due to its generality, deductive planning frameworks can be used to specify desired plans in nondeterministic domains. Nevertheless, the automatic generation of plans in these deductive frameworks is still an open problem. Another very expressive framework has been proposed in [13], and exploits process algebra and mu-calculus for reasoning about nondeterministic and concurrent actions. Unfortunately, this framework does not deal with the problem of plan generation. However, it would be interesting to investigate the possibility of embedding strong cycling planning in both the above frameworks. Some works propose an approach that is similar to the automata-based approach to planning. The TLplan system [2] allows for control strategies expressed in LTL and implements a forward chaining algorithm that has strong similarities with the LTL standard model checking [85]. However, the planner deals only with deterministic domains. Moreover, it is not clear how LTL based frameworks can be extended to express strong or strong cyclic solutions, where both a universal and existential path quantifiers are required, and to generate them. Finally, in planning based on Markov Decision Processes [28, 40, 11], policies (much like state-action tables) are constructed from stochastic automata, where actions induce transitions with an associated probability, and states have an associated reward. The planning task is then reduced to constructing optimal policies with respect to rewards and probability distributions. As a consequence, one has no control on the structure of the generated plan and, therefore, no concept of weak, strong, or strong cyclic planning seems expressible.

With respect to the automata-based approach, we have shown that the algorithm for building an automaton from a linear temporal logic formula can be significantly improved. Since the problem is PSPACE-complete and the automaton obtained from the goal has to be combined with the, usually huge, one representing the planning domain, our result can dramatically affects the performances of the planner. Moreover, we have proposed a test methodology that can be also used for evaluating other LTL deciders, and whose underlying concept, namely targeting a uniform coverage of the formula space, can be exported to other logics. Of course, the notion of uniform coverage can be further refined, and this is part of our future work. In particular, we plan to adapt to LTL the probability distributions proposed in [61] for propositional logic and

adapted in [36] to the modal logic K. These distributions assigns equal probabilities to formulas of the same structure (e.g., 3-CNF in the propositional case). We are also planning to extend the concept of syntactic implication to a semantic one and, finally, to explore automata generation in the symbolic framework.

An alternative automata construction for temporal specifications [45] starts with a two-state automaton that is repeatedly "refined" until all models of the specifications are realized. Due to this refinement process, however, this algorithm can not be used in an on-the-fly fashion. Another approach could be turning the on-the-fly decision procedure presented in [74] into a procedure for automata construction. It is not clear, however, whether and how this modification could be done, for that procedure is geared towards finding and representing one model, but not all models.

Bibliography

[1] Asarin, E., Maler, O., and Pnueli, A. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System II* (1995), vol. 999 of *LNCS*, Springer Verlag.

[2] Bacchus, F., and Kabanza, F. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence* (1998). Submitted for pubblication.

[3] Bäckström, C. Equivalence and tractability results for SAS+ planning. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning* (Cambridge, MA, Oct. 1992), W. Nebel, Bernhard; Rich, Charles; Swartout, Ed., Morgan Kaufmann, pp. 126–140.

[4] Bayardo, R. J., and Schrag, R. Using CSP look-back techniques to solve exceptionally hard SAT instances. *Lecture Notes in Computer Science 1118* (1996), 46–60.

[5] Blum, A., and Furst, M. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)* (Aug. 1995), pp. 1636–1642.

[6] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.

[7] Bryant, R. E. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers 40*, 2 (Feb. 1991), 205–213.

[8] Büchi, J. R. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress for Logic, Methodology, and Philosophy of Science* (1962), Stanford Univ. Press, pp. 1–1.

[9] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98*, 2 (June 1992), 142–170.

[10] Bylander, T. Tractability and artificial intelligence. *JETAI: Journal of Experimental and Theoretical Artificial Intelligence 3* (1991).

[11] Cassandra, A., Kaelbling, L., and Littman, M. Acting optimally in partially observable stochastic domains. In *Proceedings of AAAI-94* (1994), AAAI-Press.

[12] Cesta, A., Riccucci, P., Daniele, M., Traverso, P., Giunchiglia, E., Piaggio, M., and Shaerf, M. Jerry: a system for the automatic generation and execution of plans for robotic devices - the case study of the Spider arm. In *Proceedings of ISAIRAS-99* (1999).

[13] Chen, X., and de Giacomo, G. Reasoning about nondeterministic and concurrent actions: A process algebra approach. *Artificial Intelligence 107*, 1 (1999), 29–62.

[14] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. NuSmv: a reimplementation of smv. Tech. Rep. 9801-06, IRST, Trento, Italy, January 1998.

[15] Cimatti, A., Giunchiglia, E., Giunchiglia, F., and Traverso, P. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *ECP97* (1997), pp. 130–142.

[16] Cimatti, A., and Roveri, M. Conformant planning via model checking. In *Proceedings of the 5th European Conference on Planning (ECP-99)* (1999), LNCS.

[17] Cimatti, A., and Roveri, M. Conformant Planning via Model Checking. In *Proceedings of ECP99* (1999).

[18] Cimatti, A., Roveri, M., and Traverso, P. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceedings of AAAI98* (1998).

[19] Cimatti, A., Roveri, M., and Traverso, P. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceedings of AIPS98* (1998).

[20] Clarke, E., and Emerson, E. Characterizing Properties of Parallel Programs as Fixpoints. In *7th International Colloquium on Automata, Languages and Programming* (1981), vol. 85 of *Lecture Notes in Computer Science*, Springer-Verlag.

[21] Clarke, E., Grumberg, O., and Long, D. Verification Tools for Finite State Concurrent Systems. In *A Decade of Concurrency-Reflections and Perspectives* (Noordwijkerhout, Netherlands, June 1993), J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 803 of *Lecture Notes in Computer Science*, REX School/Symposium, Springer-Verlag, pp. 124–175.

[22] Clarke, E. M., and Emerson, E. A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs* (Yorktown Heights, New York, May 1981), D. Kozen, Ed., vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 52–71.

[23] Clarke, E. M., and Sistla, A. P. The complexity of propositional linear temporal logics. *Journal of the ACM 32* (1985), 733–749.

[24] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. Memory efficient algorithms for the verification of temporal properties. In *Proceedings of Computer-Aided Verification (CAV '90)* (Berlin, Germany, June 1991), E. M. Clarke and R. P. Kurshan, Eds., vol. 531 of *LNCS*, Springer, pp. 233–242.

[25] Daniele, M., Giunchiglia, F., and Vardi, M. Y. Improved automata generation for linear temporal logic. In *Proceedings of Computer-Aided Verification (CAV'99)* (1999), vol. 1633 of *LNCS*, pp. 249–260.

[26] Daniele, M., Traverso, P., and Vardi, M. Y. Strong cyclic planning revisited. In *Proceedings of the 5th European Conference on Planning (ECP-99)* (1999), LNCS.

[27] de Giacomo, G., and Vardi, M. Automata-theoretic approach to planning with temporally extended goals. In *Proceedings of ECP99* (1999).

[28] Dean, T., Kaelbling, L., Kirman, J., and Nicholson, A. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence 76*, 1-2 (1995), 35–74.

[29] Draghicescu, I. A., and Clarke, E. M. Expressibility results for linear-time and branching-time logics. In *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (Berlin, May 30–June 3 1989), J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 354 of *LNCS*, Springer, pp. 428–437.

[30] Emerson, E. A. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier, 1990, ch. 16, pp. 995–1072.

[31] Emerson, E. A., and Halpern, J. Y. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM 33* (1986), 151–178.

[32] Ernst, M., Millstein, T., and Weld, D. Automatic SAT-compilation of planning problems. In *Proceedings of IJCAI-97* (1997).

[33] Ernst, M. D., Millstein, T. D., and Weld, D. S. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)* (San Francisco, Aug. 23–29 1997), Morgan Kaufmann Publishers, pp. 1169–1177.

[34] Fikes, R. E., and Nilsson, N. J. STRIPS: A new approach to the application of Theorem Proving to Problem Solving. *Artificial Intelligence 2*, 3-4 (1971), 189–208.

[35] Gerth, R., Peled, D., Vardi, M., and Wolper, P. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification* (Warsaw, Poland, 1995), Chapman & Hall, pp. 3–18.

[36] Giunchiglia, F., and Sebastiani, R. Building decision procedures for modal logics from propositional decision procedures: the case study of modal K. In *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)* (Berlin, July30 Aug.–3 1996), M. A. McRobbie and J. K. Slaney, Eds., vol. 1104 of *LNAI*, Springer, pp. 583–597.

[37] Giunchiglia, F., and Traverso, P. Planning as Model Checking. In *Proceedings of the 5th European Conference on Planning (ECP-99)* (1999), LNCS.

[38] Green, C. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, D. C., May 1969), D. E. Walker and L. M. Norton, Eds., William Kaufmann, pp. 219–240.

[39] Haas, A. R. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence*, F. M. Brown, Ed. Morgan Kaufmann, 1987.

[40] Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. SPUDD: Stochastic Planning using Decision Diagrams. In *Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI99)* (1989). To appear.

[41] Kauts, H., and Selman, B. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search* (jun 1998), pp. 58–60.

[42] Kautz, H., McAllester, D., and Selman, B. Encoding plans in propositional logic. In *KR'96: Principles of Knowledge Representation and Reasoning*, L. C. Aiello, J. Doyle, and S. Shapiro, Eds. Morgan Kaufmann, San Francisco, California, 1996, pp. 374–384.

[43] Kautz, H., and Selman, B. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence* (Vienna, Austria, Aug. 1992), B. Neumann, Ed., John Wiley & Sons, pp. 359–363.

[44] Kautz, H., and Selman, B. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference* (Menlo Park, Aug. 4–8 1996), AAAI Press / MIT Press, pp. 1194–1201.

[45] Kesten, Y., Manna, Z., McGuire, H., and Pnueli, A. A decision algorithm for full propositional temporal logic. In *Proceedings of Computer-Aided Verification (CAV'93)* (Elounda, Greece, June 1993), C. Courcoubertis, Ed., vol. 697 of *LNCS*, Springer, pp. 97–109.

[46] Koehler, J., Nebel, B., Hoffman, J., and Dimopoulos, Y. Extending planning graphs to an ADL subset. In *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning* (Berlin, Sept. 24– 26 1997), S. Steel and R. Alami, Eds., vol. 1348 of *LNAI*, Springer, pp. 273–285.

[47] Korf, R. E. Search: A survey of recent results. In *Exploring Artificial Intelligence: Survey talks from the national conferences on Artificial Intelligence*. Morgan Kaufman, 1988, pp. 239–237.

[48] Korf, R. E. Linear-space best-first search: Summary of results. In *Proceedings of the 10th National Conference on Artificial Intelligence* (San Jose, CA, July 1992), W. Swartout, Ed., MIT Press, pp. 588–588.

[49] Kupferman, O., and Vardi, M. Synthesis with incomplete information. In *Proceedings of 2nd International Conference on Temporal Logic* (1997), pp. 91–106.

[50] LAMPORT, L. Sometimes is sometimes not never — but not always. In *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages (POPL)* (1980).

[51] LICHTENSTEIN, O., AND PNUELI, A. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 13–16, 1985), ACM SIGACT-SIGPLAN, ACM Press, pp. 97–107.

[52] LIFSCHITZ, V. Formal theories of action (preliminary report). In *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (Milan, Italy, Aug. 1987), J. McDermott, Ed., Morgan Kaufmann, pp. 966–972.

[53] LIFSCHITZ, V. Towards a metatheory of action. In *KR'91: Principles of Knowledge Representation and Reasoning*, J. Allen, R. Fikes, and E. Sandewall, Eds. Morgan Kaufmann, San Mateo, California, 1991, pp. 376–386.

[54] LONG, D., AND FOX, M. Efficient implementation of the plan graph in stan. *Journal of Artificial Intelligence Research 10* (1999), 87–115.

[55] MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[56] MANNA, Z., AND WALDINGER, R. A theory of plans. In *Reasoning about actions and plans: Proceeding of 1986 workshop* (Temberline, Oregon, 1986), M. Georgeff and A. Lansky, Eds., Morgan Kaufmann, pp. 11–46.

[57] MARQUES-SILVA, J. P., AND SAKALLAH, K. A. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools for Artificial Intelligence* (1996).

[58] MCCARTHY, J. Programs with common sense. In *Proceedings of the Symposium on Mechanisation of Thought Processes* (London, 1958), vol. 1, Her Majesty's Stationery Office, pp. 77–84.

[59] MCCARTHY, J., AND HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4* (1969), 463–502.

[60] MCMILLAN, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[61] Mitchell, D., Selman, B., and Levesque, H. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence* (San Jose, CA, July 1992), W. Swartout, Ed., MIT Press, pp. 459–465.

[62] Penberthy, J., and Weld., D. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92* (1992).

[63] Peot, M., and Smith, D. Conditional Nonlinear Planning. In *Proceedings of AIPS-92* (1992), pp. 189–197.

[64] Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)* (Providence, Rhode Island, Oct. 31–Nov. 2 1977), IEEE, IEEE Computer Society Press, pp. 46–57.

[65] Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages* (1989).

[66] Pnueli, A., and Rosner, R. On the synthesis of an asynchronous reactive module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming* (1989), vol. 372 of *LNCS*, pp. 652–671.

[67] Prior, A. N. *Time and Modality.* Oxford University Press, Oxford, U.K., 1957.

[68] Pryor, L., and Collin, G. Planning for Contingency: a Decision Based Approach. *J. of Artificial Intelligence Research 4* (1996), 81–120.

[69] Queille, J. P., and Sifakis, J. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, Lecture Notes in Computer Science, Vol. 137. SV, Berlin/New York, 1982, pp. 337–371.

[70] Reiter, R. A logic for default reasoning. *Artificial Intelligence 13*, 1–2 (Apr. 1980), 81–132.

[71] Russel, S. Efficient memory-bounded search algorithms. In *Proceedings of the 10th European Conference on Artificial Intelligence* (1992).

[72] Sacerdoti, E. D. *A structure for plans and behaviour.* American Elsevier, 1975.

[73] Schoppers, M. J. Universal plans for Reactive Robots in Unpredictable Environments. In *Proceedings of IJCAI87* (1987), pp. 1039–1046.

[74] SCHWENDIMANN, S. A new one-pass tableau calculus for PLTL. *Lecture Notes in Computer Science 1397* (1998), 277–291.

[75] SELMAN, B., LEVESQUE, H. J., AND MITCHELL, D. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (Menlo Park, California, 1992), P. Rosenbloom and P. Szolovits, Eds., American Association for Artificial Intelligence, AAAI Press, pp. 440–446.

[76] SMITH, D., AND WELD, D. Conformant Graphplan. In *AAAI98* (1998), pp. 889–896.

[77] STEEL, S. Action under Uncertainty. *J. of Logic and Computation, Special Issue on Action and Processes 4*, 5 (1994), 777–795.

[78] STEPHAN, W., AND BIUNDO, S. A New Logical Framework for Deductive Planning. In *Proceedings of IJCAI93* (1993), pp. 32–38.

[79] TARSKI, A. A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math. 5* (1955), 285–309.

[80] THAYSE, A., Ed. *From Modal Logic to Deductive Databases.* John Wiley & Sons, Chichester, 1989.

[81] THOMAS, W. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science Publishers B. V., 1990, ch. 4, pp. 133–191.

[82] TOUATI, H. J., BRAYTON, R. K., AND KURSHAN, R. Testing language containment for ω-automata using BDD's. *Information and Computation 118*, 1 (Apr. 1995), 101–109.

[83] VARDI, M. Y. An automata-theoretic approach to fair realizability and synthesis. In *Proceedings of Computer-Aided Verification (CAV'95)* (1995), vol. 939 of *LNCS*, pp. 267–278.

[84] VARDI, M. Y., AND WOLPER, P. An automata-theoretic approach to automatic program verification. In *lics86* (1986), pp. 332–344.

[85] VARDI, M. Y., AND WOLPER, P. Reasoning about infinite computations. *Information and Computation 115*, 1 (15 Nov. 1994), 1–37.

[86] WARREN, D. Generating Conditional Plans and Programs. In *Proceedings of AISB-76* (1976).

[87] WELD, D., ANDERSON, C., AND SMITH, D. Extending Graphplan to Handle Uncertainty and Sensing Actions. In *Proceedings of AAAI98* (1998), pp. 897–904.

[88] ZHANG, H. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated deduction* (Berlin, July 13–17 1997), W. McCune, Ed., vol. 1249 of *LNAI*, Springer, pp. 272–275.