Applications of Description Logic and Causality in Model Checking

by

Shoham Ben-David

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Model checking is an automated technique for the verification of finite-state systems that is widely used in practice. In model checking, a model M is verified against a specification φ, exhaustively checking that the tree of all computations of M satisfies φ. When φ fails to hold in M, the negative result is accompanied by a *counterexample*: a computation in M that demonstrates the failure. State of the art model checkers apply Binary Decision Diagrams (BDDs) as well as satisfiability solvers for this task. However, both methods suffer from the state explosion problem, which restricts the application of model checking to only modestly sized systems. The importance of model checking makes it worthwhile to explore alternative technologies, in the hope of broadening the applicability of the technique to a wider class of systems.

Description Logic (DL) is a family of knowledge representation formalisms based on decidable fragments of first order logic. DL is used mainly for designing ontologies in information systems. In recent years several DL reasoners have been developed, demonstrating an impressive capability to cope with very large ontologies.

This work consists of two parts. In the first we harness the growing ability of DL reasoners to solve model checking problems. We show how DL can serve as a natural setting for representing and solving a model checking problem, and present a variety of encodings that translate such problems into consistency queries in DL. Experimental results, using the Description Logic reasoner FaCT++, demonstrate that for some systems and properties, our method can outperform existing ones.

In the second part we approach a different aspect of model checking. When a specification fails to hold in a model and a counterexample is presented to the user, the counterexample may itself be complex and difficult to understand. We propose an automatic technique to find the computation steps and their associated variable values, that are of particular importance in generating the counterexample. We use the notion of *causality* to formally define a set of causes for the failure of the specification on the given counterexample. We give a linear-time algorithm to detect the causes, and we demonstrate how these causes can be presented to the user as a visual explanation of the failure.

Acknowledgements

I would like to thank my supervisor, Richard Trefler, for his encouragement and support during my years at Waterloo, and for letting me pursue my own research interests. I would like to thank my informal co-supervisor, Grant Weddell, for guiding me through the mysteries of Description Logic, and for his special way of making people feel good about themselves. For me, at least, it worked.

I would like to thank the other members of my examining committee, Mark Aagaard, Amy Felty and John Thistle for reading my work thoroughly, for their helpful suggestions and for making my defense a stimulating experience.

I am grateful to my friends from the IBM Haifa Research Lab, Hana Chockler, Dana Fisman, Avigail Orni and Sitvanit Ruah. Working with you has been so much fun! There is nothing like a Hebrew transatlantic phone conversation to boost up one's motivation. I hope to continue collaborating with you in the future.

I owe a lot to my friends in the Watform Lab, Vlad Ciubotariu, Shahram Esmaeilsabzali, Naghmeh Ghaffari, Alma Juarez-Dominguez, Zarrin Langari and Ali Taleghani, who have made the Watform Lab a pleasant place to work in. Thanks to Shahram for the long conversations about history, religion and life in general. Thanks to Ali for being such a cheerful and supportive office-mate, and special thanks to Zarrin for her continuous friendship, optimism and support.

I would not have survived the years in the University of Waterloo without my supportive family. The love, encouragement and understanding I got from Shai have been invaluable. I would like to thank Shalev, Naama and Sheffi for making my life cheerful and for being such incredible young people.

Finally, I am grateful to my mother, for her love and support in the last ... well, many years, and for never giving up the hope of seeing me getting a PhD.

To the memory of my father

who planted in me the love for Mathematics

Contents

List of Figures

Chapter 1

Introduction

Hardware and software systems have become an integral component of our everyday lives, and we use them for larger and larger parts of our routine activities. While today's world cannot be imagined without these systems, software and hardware programs are full of errors, that often make them unreliable. Errors can be very expensive (e.g., the floating-point division bug of Intel's Pentium processor [Hal95], cost $500,000,000 of damage) and worse – life threatening (e.g., the Therac-25 accidents [LT93] cost the lives of four people). The main reason for the unreliability of today's hardware and software systems is their growing complexity that makes them extremely difficult to verify. In fact, in the hardware industry, verification is recognized as the most resource-consuming component of the design process, taking over 60% of the development time and effort. Finding new verification methods and developing better verification tools can therefore have a significant impact on today's industry.

Verification of software and hardware systems is traditionally done using *testing*: the system is given sequences of legal input behaviors and the outputs are analyzed compared to some expected results. For large systems, both the generation of test cases and the analysis of the results are often automated. However, for any large enough system, running test cases cannot guarantee coverage of all possible behaviors: there are simply too many cases (possibly an infinite number of them) to be covered. Thus, when testing is the only method used for verification, systems are delivered to the market with many possible cases untested. For many systems however, especially safety-critical ones, this is not enough, and a higher degree of coverage is required. In order to meet this requirement, *formal verification* methods have been developed, where mathematical techniques are applied to perform the verification. When properly applied, formal verification methods can guarantee correctness of a system with respect to its specification.

Formal verification is generally divided into two main approaches, the *deductive* approach and the *algorithmic* one. The first is known as *Theorem Proving* [GM93, BKM95, KM97], and involves the development of a mathematical *proof* for the correctness of a given system with respect to its specification. Since developing a proof is a hard task, and in most cases cannot be done automatically, theorem provers are interactive tools that allow the user to specify the main steps of a proof, avoiding, as much as possible, the tedious parts of it.

This work concentrates on the algorithmic approach to formal verification, known as *model checking* ([CE81, QS82], c.f.[CGP00]). Model checking is a fully automated technique for verifying finite-state systems, that is very effective in the verification of hardware and software programs. In model checking, a model M, given as a set of state variables V and their next-state relations, is verified against a specification φ. If the specification holds on the tree of all computations of M we denote it $M \models \varphi$. When φ fails to hold in M, the model checker provides a *counterexample* [CGMZ95]: a computation of M that demonstrates the failure.

Specifications to be checked are given in *temporal logic* – a dialect of modal logic with modalities referring to time. The main temporal logics used in practice are *Linear Temporal Logic* (LTL) [Pnu77] and *Computation Tree Logic* (CTL) [CE81]. Temporal logic specifications, whether given in LTL or in CTL, are divided into two main types [Lam77]: *safety* formulas, stating that "something bad never happens", and *liveness* formulas, asserting that “something good eventually happens”. The violation of a safety formula can be shown by a finite prefix of a computation path, leading to an erroneous state, while the violation of a liveness formula can only be shown by an infinite path, or a loop, in the case of a finite system. Liveness formulas are therefore considered more difficult to verify. In many cases, a liveness formula is accompanied by a *fairness constraint* requiring that the violating loop satisfies some fairness condition.

The main challenge in model checking is called the *state space explosion* problem, where the number of states in the model grows exponentially in the number of variables describing it. Different approaches exist to deal with this problem. They can be roughly divided into *explicit* state methods, that are mostly applied to software systems, and *implicit* state (or *symbolic*) methods that are better applied to hardware models. In this work we concentrate on symbolic methods for hardware model checking.

In symbolic model checking the system under verification is represented as sets of states and transitions, and Boolean functions are used to manipulate those sets. Two main symbolic methods are used to perform model checking. The first, known as *Symbolic Model Verifier* (*SMV*) [McM93] is based on Binary Decision Diagrams (BDDs) [Bry86] for representing the state space as well as for performing the model checking procedure. The second is known as *Bounded Model Checking* (*BMC*) [BCCZ99]. Using this method, the model description is unfolded k times (for a given bound k). The unfolded model as well as the *negation* of the specification are then translated into a propositional formula, and a satisfiability solver is applied to the formula to find a satisfying assignment. Such an assignment, if found, demonstrates an error in the model.

The introduction of the BDD-based model checking method and later on the satisfiability based ones, have significantly improved the performance and applicability of model checking, and have brought the field from a completely theoretical one in the early eighties into a wide-spread practical technique, used in industry [BDEGW03, Ger01, AAH$^+$03]. However, the state explosion problem remains the main problem of this field, restricting the application of model checking to only modestly sized systems. The importance of verification in general and model checking in particular, makes it worthwhile to explore alternative technologies, in the hope of enabling the application of the technique to a wider class of systems.

In the first part of this thesis we explore the possibility of using *Description Logic reasoning* to solve model checking problems. Description Logic (DL) ([BCM$^+$03]) is a family of knowledge representation formalisms, mainly used for specifying ontologies for information systems. The basic elements in description logic are *atomic concepts* (sets of individuals) and *atomic roles* (binary relations between individuals). There are many *dialects* of description logic that differ from each other by the constructs they allow for building new concepts from existing ones. The most commonly used dialect is called *Attributive Language with Complement*, or $\mathcal{ALC}$. Given two concepts $\mathtt{C}_1$ and $\mathtt{C}_2$, and a role $\mathtt{R}$, the DL dialect $\mathcal{ALC}$ allows the construction of the concepts $\neg\mathtt{C}_1$ (all individuals that do not belong to the set represented by $\mathtt{C}_1$), $\mathtt{C}_1 \sqcap \mathtt{C}_2$ (the individuals that belong to both $\mathtt{C}_1$ and $\mathtt{C}_2$), and $\forall\mathtt{R}.\mathtt{C}_1$ (the individuals $\mathtt{a}$, such that for all $\mathtt{b}$ where $\mathtt{R}(\mathtt{a},\mathtt{b})$ holds, $\mathtt{b}$ belongs to $\mathtt{C}_1$). In general, the more expressive a DL dialect is, the more complex it is to reason about.

Description Logic is used for describing ontologies and reasoning about them. An ontology $\mathcal{T}$ is called a *terminology*, and consists of a set of *concept inclusion dependencies*. Each inclusion dependency has the form $\mathtt{C}_1 \sqsubseteq \mathtt{C}_2$, and asserts containment properties of relevant concepts in an underlying domain, e.g., that *cows* are included in *animals*

$$\mathrm{COW} \sqsubseteq \mathrm{ANIMAL},$$

and also in *those things that do not eat animals*

$$\mathrm{COW} \sqsubseteq \forall \mathit{eats}.\neg\mathrm{ANIMAL}.$$

In this latter case, *eats* is an example of a *role*. The main reasoning service provided by a DL system is the *concept consistency* service, that is, for a given terminology $\mathcal{T}$ and concept $\mathtt{C}$, to determine if there is a non-empty interpretation of $\mathtt{C}$, that also satisfies each inclusion dependency in $\mathcal{T}$. We denote concept consistency as $\mathcal{T} \models_{dl} \mathtt{C}$ (note that we use " $\models_{dl}$ " for consistency in DL to differentiate it from "$\models$" that is used in the model checking world). In recent years several DL reasoners have been introduced, such as FaCT++ [TH06], Pellet [SPG$^+$07] and Racer [HM01], demonstrating growing capability to reason about large ontologies.

We show how Description Logic technology can be used for symbolic model checking. We present a variety of encodings of model checking problems as Description Logic terminologies over different dialects. In all cases we provide a linear encoding of a *model description* (or program) and a specification as a DL terminology, and pose a query in such a way that interpretations correspond to errors in the system. We present several methods to support bounded model checking of safety properties, that result in a natural symbolic representation of the sets of states and transitions. Experimental results comparing the different methods are surprising: although the methods are closely related, they perform significantly different.

We then present an encoding for model checking of liveness formulas in DL. Our main contribution for this type of formulas is the introduction of an algorithm to support fairness constraints in DL. This algorithm enhances the tableaux algorithm for DL reasoning, and it is thus of interest to the DL community. On the other hand, it introduces a novel approach to fair path detection, and thus has the potential of improving model checking performance for some cases.

The second part of this work tackles a different aspect of model checking: the analysis of a counterexample. When a formula fails to hold in a model, the first step in debugging the problem is to examine the counterexample in order to understand the error it demonstrates. In many cases, however, the task of understanding the counterexample is non-trivial, and may require a significant manual effort.

An explanation of a counterexample deals with the question: *what values on the com-*

putation trace cause it to falsify the specification? To deal with this problem, we adapt the formal definition of causality of Halpern and Pearl [HP01]. We view a counterexample trace as a matrix $M \times N$ of values, where M is the number of time units (the *length*) of the counterexample, and N the number of variables appearing in the counterexample. An entry (i, j) in the matrix corresponds to the value of variable j at time i. We look for those entries in the matrix that are causes for the first failure of φ on π, according to the definition of [HP01]. We show that the complexity of detecting an exact causal set is NP-complete, based on the complexity result for causality in binary models [EL01]. We then present an over-approximation algorithm whose complexity is linear in the size of the formula and in the length of the counterexample. Our contribution is both theoretical, in defining the set of causes, and practical, in introducing the explanation algorithm that is used in practice.

1.1 Related Work

Model checking using DL

The connection between knowledge-based reasoning and model checking has been explored before. Gottlob et al in [GGV00, GGV02] analyzed the expressive power of *Datalog* statements, and compared them to known temporal logics. Sahasrabudhe in [Sah04] performed model checking of telephony feature interactions by using SQL on an explicit state representation of the model, and compared the results with model checking a similar explicit state representation using the model checker SMV [McM93]. Both Sahasrabudhe and Gottlob et al, however, used an explicit representation of the model, as opposed to the representation of the model *description* that we propose. This difference is crucial, since in many cases the Kripke structure for the model is too big to be built, and symbolic methods must be used.

Our paper [BDTW06] was the first to suggest the use of Description Logic reasoners for model checking. However, the method described in that paper required a synchronization between the progress of different state variables, that resulted in a blow-up in the number of explored states.

Since the DL dialects we use are fragments of first order logic, our method can be viewed as performing model checking using deductive methods. The work of Tuominen [Tuo88, Tuo89] is close to ours in this sense. Tuominen used theorem proving for the verification of Petri-net systems. He used *deterministic propositional dynamic logic* (DPDL) to represent his systems, a logic that is more expressive (and thus more complex to verify) than the DL dialects that we use.

Finally, Dovier and Quintarelli in [DQ01] were interested in the opposite direction: they translated a knowledge-base into a Kripke structure, and a query into a temporal logic formula. They then used a model checker to make inferences about the knowledge-base.

Counterexample explanation

The problem of *understanding* a counterexample has attracted a significant amount of attention in recent years (see for example [CIW$^+$01, JRS02, DRS03, BNR03, Gro04, GK04, CG05, SQL05, WYIG06, GSB07, SB07, SFBD08]). These works, however, concentrated on a different aspect of *understanding* of a counterexample. Mainly, they addressed the question of finding the root cause of the failure in the *model* and proposed automatic ways to extract more information about the model, to ease the debugging procedure. Naturally, the algorithms proposed in the above mentioned works involve implementation in a specific tool. For example, the BDD procedure of [JRS02] would not work for a SAT based model checker like those of [Gro04, BNR03]. In contrast, the method we propose is independent of the tool that generated the counterexample, and can be applied as an external layer to any model checking tool.

There are several works that tie the definition of causality by Halpern and Pearl to formal verification. Most closely related to our work is the paper by Chockler et. al [CHK08], in which causality and its quantitative measure, responsibility, are viewed as a refinement of coverage in model checking. In another work, causality and responsibility are used to improve the refinement techniques of symbolic trajectory evaluation (STE) [CGY08].

1.2 Overview of Thesis

In Chapter 2 we give the needed background for the thesis. In Section 2.1 we discuss topics in model checking: we describe the temporal logics used in the thesis, give the basic definition of a *model* and define an example model and specification that are used in the rest of the document. We briefly discuss the two main symbolic model checking methods: McMillan's symbolic model checking using BDDs [McM93], and bounded model checking [BCCZ99] based on Satisfiability solving. Section 2.2 presents basic facts about Description Logic [BCM$^+$03]. We describe the syntax and semantics of common dialects, explain how ontologies are defined in description logic, and use an example to demonstrate a reasoning service provided by a DL reasoner.

In Chapters 3 and 4 we present our results on model checking using Description Logic. In Chapter 3 we present the symbolic encoding of a model description, and define the different methods for bounded model checking of safety formulas. We prove the correctness of our encodings and discuss experimental results. The work described in this chapter appears in [BDTW07a, BDTW07b, BDTW08, BDTTW08]. In Chapter 4 we show, for the same encoding, how liveness formulas can be described. Since fairness cannot be expressed in the dialects used in this document, we propose a method to implement fairness checking in DL. This work appears in [BDPT$^+$09]. Chapter 5 is devoted to explanation of a counterexample. We define causality in counterexamples, analyze its complexity and propose an approximation algorithm. The work is based on [BBDC$^+$09]. Chapter 6 concludes the document.

Chapter 2

Background

This chapter gives the needed background of model checking and description logic. We start with model checking in the section below, and discusses Description Logic in Section 2.2 .

2.1 Model Checking

Model checking ([CE81, QS82], c.f.[CGP00]) is a technique for the formal verification of hardware and software systems. In model checking, a model M is verified against a specification φ. If the specification holds in the model we denote it $M \models \varphi$. For our discussion, the system under verification, or the *model*, is assumed to have a finite number of Boolean state variables, that may simultaneously change their values as time progresses. The mathematical representation of such a model is called a Kripke structure, and its formal definition is given in Section 2.1.1 below. Synchronous hardware systems are naturally translated into Kripke structures, as these are indeed composed of variables that work in parallel, changing their value at a clock's tick. For asynchronous hardware designs, as well as for software programs, some sort of abstraction may be needed in order to adapt them to the model of a Kripke structure.

In order to verify a given model we need to specify its desired behavior. In model checking, specifications are given as temporal logic formulas [Pnu77] – a language that allows specifying the behavior of the program variables over time. In Section 2.1.2 we present the two main temporal logics that are used in practice, namely, LTL and CTL, and discuss different categories of formulas. In Sections 2.1.3 and 2.1.4 we discuss the two main existing methods for symbolic model checking.

2.1.1 Kripke Structures

A Kripke structure is a labeled directed graph, defined in the context of Modal Logic. We describe here a restricted type of Kripke structure that is used to model reactive systems. We associate a Kripke structure with a finite set V of Boolean variables. Each node in the graph is labeled with a subset of V (the variables that are assigned 1 in the node). Thus each node in the graph represents a *state* of the modeled system. Different nodes in the graph must be labeled with different sets, that is, each state of the system can appear at most once in the Kripke structure. Thus, if V includes n variables, the Kripke structure may have at most 2^n nodes. An edge from one node to another means that a transition is possible in the system, from a given state to the next, in one time unit. From each node there must exist at least one outgoing edge (that is, there are no “dead-ends” in the system). The mathematical definition of a Kripke structure is given below.

Definition 1 (Kripke Structure)**.** Let V be a set of Boolean variables. A *Kripke structure* M over V is a four tuple $M = (S, I, R, L)$ where

1. S is a finite set of states.

2. $I \subseteq S$ is the set of initial states.

3. $R \subseteq S \times S$ is the transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.

4. $L : S \to 2^V$ is a function that labels each state with the set of variables true in that state.

We view each state s as a truth assignment to the variables V. We view a set of states as a Boolean function over V, characterizing the set. For example, the set of initial states I is considered as a Boolean function over V. Thus, if a state s belongs to I, we write $s \models I$. Similarly, if $v_i \in L(s)$ we write $s \models v_i$, and if $v_i \notin L(s)$ we write $s \models \neg v_i$. We say that $w = s_0, s_1, ...$ is a *path* in M if $s_0 \models I$ and $\forall i$, $0 \leq i$, $(s_i, s_{i+1}) \in R$.

Example 2. Figure 2.1 draws the states and transitions of a Kripke structure, called `Simple`. The initial state is colored dark, and the label of each state is the value of the vector (v_1, v_2, v_3).

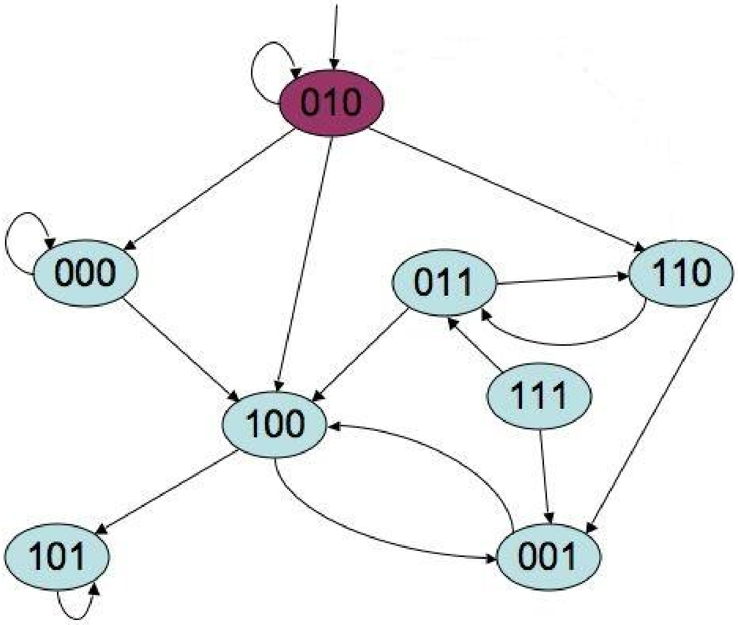

Figure 2.1: The Kripke structure “Simple model”

Kripke structures are used for modeling the behaviors of real hardware and software systems. However, in practice the full Kripke structure of a system is usually too big and too complex to be explicitly described. Rather, a model is given as a set of Boolean variables $V = \{v_1, ..., v_n\}$, their initial values and their next-state assignments. Moreover, for every reasonably-sized system, the Kripke structure is too big to be explicitly built. Rather, systems are described by giving the behavior of every state variable separately. We concentrate on hardware, where systems are naturally described in this way by existing Hardware Description Languages (HDL). In standard HDLs however, the system is deterministic, and multiple behaviors can only be due to the behavior of the inputs. The input language of *SMV* [McM93] allows a more complex non-deterministic behavior. Our notation is an abstraction of the input language of *SMV*.

Definition 3 (Model Description)**.** Let $V = \{v_1, ..., v_n\}$ be a set of Boolean variables. A tuple $MD = (I_{MD}, [\langle c_1, c'_1\rangle, ..., \langle c_n, c'_n\rangle])$ is a *Model Description* over V where I_{MD}, c_i, c'_i are Boolean expressions over V.

The semantics of a model description is a Kripke structure $M_{MD} = (S, I_M, R, L)$, where $S = 2^V$, $L(s) = s$, $I_M = \{s | s \models I_{MD}\}$, and $R = \{(s, s') : \forall 1 \leq i \leq n,\ s \models c_i$ implies $s' \models \neg v_i$ and $s \models c'_i \wedge \neg c_i$ implies $s' \models v_i\}$.

Intuitively, a pair $\langle c_i, c'_i\rangle$ defines the next-state assignment of variable v_i in terms of the current values of $\{v_1, ..., v_n\}$. That is,

$$\text{next}(v_i) = \begin{cases} 0 & \text{if } c_i \\ 1 & \text{if } c'_i \wedge \neg c_i \\ \{0,1\} & \text{otherwise} \end{cases}$$

where the assignment $\{0, 1\}$ indicates that for every possible next-state value of variables $v_1, ...v_{i-1}, v_{i+1}, ..., v_n$ there must exist a next-state with $v_i = 1$, and a next-state with $v_i = 0$.

Example 4. For the model `Simple` given in figure 2.1, the next state assignments are given below.

$$\text{next}(v_1) = \begin{cases} 0 & \text{if } v_1 \wedge v_2 \\ 1 & \text{if } v_3 \wedge \neg(v_1 \wedge v_2) \\ \{0,1\} & \text{otherwise} \end{cases} \quad \text{next}(v_2) = \begin{cases} 0 & \text{if } \neg v_2 \\ \{0,1\} & \text{otherwise} \end{cases} \quad \text{next}(v_3) = \begin{cases} 0 & \text{if } v_1 \\ 1 & \text{otherwise} \end{cases}$$

The full model description is given by

$$\texttt{Simple} = (I, [\langle v_1 \wedge v_2, v_3 \rangle, \langle \neg v_2, v_1 \wedge \neg v_1 \rangle, \langle \neg v_1, v_1 \rangle])$$

over $V = \{v_1, v_2, v_3\}$ with $I = \neg v_1 \wedge v_2 \wedge \neg v_3$.

This example shall be used throughout this document to demonstrate our methods.

2.1.2 Temporal Logic

Temporal Logic is a dialect of Modal Logic. The use of Temporal Logic for the specification of reactive systems was first suggested by Pnueli in [Pnu77] and has since been accepted as the major language for the specification of such systems. Several types of temporal logics exist in the literature, with the most commonly used ones being LTL [Pnu77] and CTL [CE81]. We describe the logics LTL and CTL below, and then discuss different types of temporal logic formulas, known as *safety* and *liveness* formulas.

Linear Temporal Logic

Given a finite set AP of atomic propositions, formulas of LTL are recursively defined as follows:

- Every atomic proposition is an LTL formula.
- If φ and ψ are LTL formulas then so are:
  - $\neg\varphi$ • $\varphi \wedge \psi$ • $X\varphi$ • $[\varphi U \psi]$

Additional operators are defined as syntactic sugaring of those above:

- $true = \neg p \vee p$ • $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$ • $F\varphi = [true\, U \varphi]$
- $\varphi \rightarrow \psi = \neg\varphi \vee \psi$ • $G\varphi = \neg F \neg\varphi$ • $[\varphi W \psi] = [\varphi U \psi] \vee G\varphi$

The formal semantics of LTL formulas is defined with respect to an infinite computation path. A computation path is a sequence of states $w = s_0, s_1, s_2, ...$ where s_i is a truth assignments to the atomic propositions AP. We sometimes use w to denote a finite prefix of a path. The suffix of a computation, $s_j, s_{j+1}, s_{j+2}, ...$ is denoted by w^j. Given a prefix ρ and a path w, we denote $\rho \cdot w$ the concatenation of them. We use $w \models \varphi$ to indicate that the LTL formula φ holds on the computation w. The semantics of $\models$ is inductively defined as follows.

- $w \models p$ IFF $s_0 \models p$
- $w \models \neg\varphi$ IFF $w \not\models \varphi$
- $w \models \varphi \wedge \psi$ IFF $w \models \varphi$ and $w \models \psi$
- $w \models \ X\varphi$ IFF $w^1 \models \varphi$
- $w \models [\varphi U \psi]$ IFF $\exists k \geq 0$ such that $w^k \models \psi$ and for all $0 \leq j < k, w^j \models \varphi$

It is common practice to view computations satisfying an LTL formula as infinite *words* over the alphabet 2^{AP}, where the letters of the alphabet are the states of the computation paths [WVS83, VW86, SVW87]. Under this interpretation, one can talk about the *language* accepted by an LTL formula, referring to the set of words satisfying the formula. Languages that can be accepted by LTL formulas are omega-regular languages. Such languages are accepted by Büchi automata. We give the definition of a Büchi automaton below.

Definition 5 (Büchi automaton)**.** A Büchi automaton is a 4-tuple (S, I, δ, F) where

- S is a finite set of states
- $I \subseteq S$ is a set of initial states
- $\delta \subseteq S \times S$ is a transition relation

- $F \subseteq S$ is a set of accepting states

An infinite sequence of states is accepted by a Büchi automaton if and only if it contains infinitely many accepting states.

In most cases, the verification of an LTL formula φ is done by first building a Büchi automaton A that accepts $\neg\varphi$ [Var96], and then verifying that A accepts no computation of the model M.

Negation Normal Form

For both LTL and CTL (see below) it is possible to transfer formulas into equivalent ones in Negation Normal Form (NNF), where negations are allowed on atomic propositions only. For example, the LTL formula $\neg\mathbf{G}(p \rightarrow \mathbf{X}q)$, where a temporal operator is negated, is equivalent to $\mathbf{F}(p \wedge \mathbf{X}\neg q)$, that is in NNF. The transformation is straightforward using the temporal operations defined above.

Computation Tree Logic

Computation Tree Logic (CTL) is a branching time logic. This means that time is viewed as a tree, where one state may have more than one successive state. To capture this, branching time logics introduce, on top of the temporal operators used for LTL, two *Path Quantifiers*: the A path quantifier stands for "All paths", and the E path quantifier stands for "there exists a path". In CTL, a path quantifier must immediately precede a temporal operator. A formula in NNF form, consisting of the A path quantifier only, is called an ACTL formula If only the E path quantifier exists it is called an ECTL formula. The formal definition of CTL is then given as follows:

Definition 6 (Computation Tree Logic)**.** Given a finite set AP of atomic propositions, formulas of CTL are recursively defined as follows:

- Every atomic proposition is a CTL formula.

- If φ and ψ are CTL formulas then so are:

  - $\neg\varphi$ - $\varphi \wedge \psi$ - $AX\varphi$ - $EX\varphi$ - $A[\varphi U \psi]$ - $E[\varphi U \psi]$

Additional operators are defined as syntactic sugaring of those above:

- $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$
- $AF\varphi = A[\mathit{true}\; U \varphi]$
- $AG\varphi = \neg E[\mathit{true}\; U \neg\varphi]$
- $A[\varphi\, V \psi] = \neg E[\neg\varphi\, U \neg\psi]$
- $A[\varphi\, W \psi] = \neg E[\neg\psi\, U \neg\varphi \wedge \neg\psi]$
- $\varphi \rightarrow \psi = \neg\varphi \vee \psi$
- $EF\varphi = E[\mathit{true}\; U \varphi]$
- $EG\varphi = \neg A[\mathit{true}\; U \neg\varphi]$
- $E[\varphi\, V \psi] = \neg A[\neg\varphi\, U \neg\psi]$
- $E[\varphi\, W \psi] = \neg A[\neg\psi\, U \neg\varphi \wedge \neg\psi]$

The intuitive semantics of CTL operators are given in Figure 2.2. The formal semantics of CTL formulas is defined with respect to a Kripke structure $M = (S, I, R, L)$ over a set of variables $V = \{v_1, ..., v_k\}$. The notation $M, s \models \varphi$, means that the formula φ is true in state s of the model M.

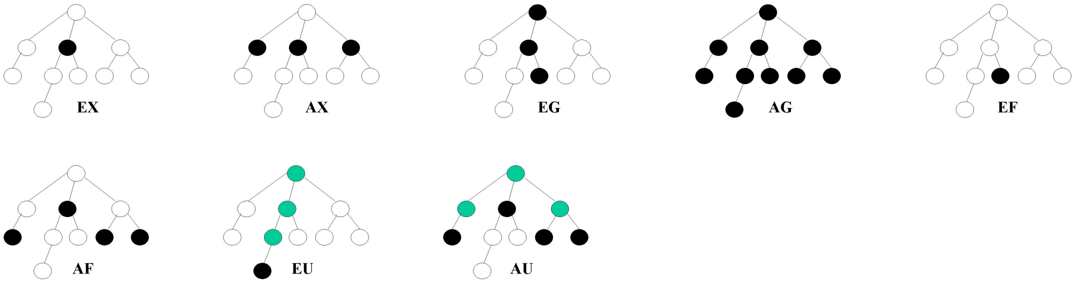

Figure 2.2: CTL Operators

- $M, s \models p$ iff $s \models p$

- $M, s \models \neg\varphi$ iff $M, s \not\models \varphi$

- $M, s \models \varphi \wedge \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$

- $M, s_0 \models\ AXp$ iff for all paths $(s_0, s_1, ...)$, $M, s_1 \models p$
- $M, s_0 \models\ EXp$ iff for some path $(s_0, s_1, ...)$, $M, s_1 \models p$
- $M, s_0 \models A[\varphi U \psi]$ iff for all paths $(s_0, s_1, ...)$, for some i, $M, s_i \models \psi$ and for all $j \leq i$,$M, s_j \models \varphi$
- $M, s_0 \models E[\varphi U \psi]$ iff for some path $(s_0, s_1, ...)$, for some i, $M, s_i \models \psi$ and for all $j \leq i$,$M, s_j \models \varphi$

We say that a Kripke structure $M = (S, I, R, L)$ satisfies a CTL formula φ ($M \models \varphi$) if for all $s_i \in I$, $M, s_i \models \varphi$.

Polarity of Subformulas

Let φ be a temporal logic formula and let ψ be an occurrence of a subformula in φ. We say that ψ has a *positive polarity* in φ, if ψ is under the scope of an even number of negations. Otherwise, we say that ψ has a *negative polarity* in φ. For example, for $\varphi = \neg\mathbf{G}(p \wedge \neg\mathbf{X}q)$, the subformula $\psi = \mathbf{X}q$ has a positive polarity, and $\psi' = p$ has a negative polarity. Note that if φ is given in NNF, only propositions can have a negative polarity in φ.

The Common Fragment of LTL and ACTL

Monika Maidl in [Mai00] has investigated the relationship between CTL and LTL, and characterized the fragment of ACTL that can be expressed in LTL. This fragment is called ACTL^{det}, and its inductive definition is given below, where the operator W stands for "weak until".

- Every atomic proposition is an ACTL^{det} formula.
- If p is a proposition, φ and ψ are ACTL^{det} formulas then so are:

- $\varphi \wedge \psi$
- $(p \wedge \varphi) \vee (\neg p \wedge \psi)$
- $AX\varphi$
- $A[(p \wedge \varphi)U(\neg p \wedge \psi)]$
- $A[(p \wedge \varphi)W(\neg p \wedge \psi)]$

It is interesting to note that most of the CTL formulas written in practical applications belong to the common fragment of ACTL and LTL [BDFR05].

Safety and Liveness Formulas

Temporal logic specifications, whether written in LTL or in CTL, are divided into three basic categories [Lam77, AS85, AS87, MT01]: *liveness* properties, *safety* properties and formulas that are combinations of the two. Informally, a safety formula states that “something *bad* never happens” while a liveness formula asserts that “something *good* will eventually happen”. A somewhat more formal definition was given by Alford et al. in ([AAH$^+$85] cf. [Kin94]), defining as *safety* formulas whose violation can be shown by a finite prefix of a computation path, while the violation of a liveness formula must contain an infinite path (a loop, in case of a finite model). For example, the LTL formula $\mathbf{G}(p \rightarrow \mathbf{X}q)$ is a safety formula, since, in order to show violation, it is enough to present a finite prefix of a computation path that leads to a state where p holds, followed by a state where q does not hold. The formula $\mathbf{G}(req \rightarrow \mathbf{F}(ack))$ on the other hand is a liveness formula, because the violation of it must show a state where req holds followed by an infinite path along which ack never holds.

Fairness

When verifying a liveness formula it is many times the case that the formula should only be verified on computation paths that are *fair* according to some notion. The simplest and most commonly used fairness constraint states that some Boolean condition p must hold on the path infinitely often. (This constraint can be described by the LTL formula GFp, with *p* being a Boolean expression over the set of variables V). When the fairness constraint `fairness p` is given, a legal counterexample for a liveness formula (on a finite model) should therefore include a loop on which the liveness formula fails, but where the expression p holds at least on one state in the loop.

Translating Temporal Logic Formulas into Automata

As mentioned earlier, a popular method for model checking an LTL specification φ, is to first build a Büchi automaton $A_{\neg\varphi}$ for the *negation* of φ with size exponential in the size of the formula [Var96, BFH05]. For formulas in the common fragment of LTL and ACTL, this automaton is of size linear in the size of the formula [Mai00].

Once $A_{\neg\varphi}$ is built, the parallel composition of $A_{\neg\varphi}$ with the model M, denoted $A_{\neg\varphi}||M$, is itself a Büchi automaton, and its language can be checked for emptiness (if the set of computations is not empty, it contains counterexamples for φ). Since the accepting condition of a Büchi automaton requires visiting a set of states infinitely often, the model checking of φ amounts to searching for a fair path in $A_{\neg\varphi}||M$. For safety formulas, the Büchi condition is not needed. Rather, the automaton built is used as a non-deterministic finite automaton (NFA) that has accepting states (accepting error computations) [KV99]. For safety formulas as well, when a formula belongs to the common fragment of LTL and ACTL, the NFA built for it is linear in the size of the formula [BBDL98, Mai00, BDFR05].

Below we sketch the translation of a safety common-fragment LTL formula φ into a non-deterministic finite automaton that accepts erroneous paths. The translation is done in two phases. In the first phase we produce, given φ, a regular expression r_φ that describes an erroneous computation. The alphabet Σ_φ of r_φ contains Boolean expressions over the propositions appearing in φ, and words accepted by r_φ are sequences of states where letters from Σ_φ hold. The full translation is given in [BBDL98]. We give the flavor of the translation using a few examples. We use the letter t to indicate True (the Boolean expression $p \vee \neg p$ for some proposition p). We use $*$ and $\cdot$ in their usual regular-expression meaning.

1. Let $\varphi = G(p)$. We define r_φ to be $(t*) \ \cdot \ \neg p$. Note that this regular expression accepts all computations that include a finite number of states where True holds (any state), followed by a state satisfying $\neg p$. That is, counterexamples for $G(p)$.

2. Let $\varphi = G(p \rightarrow Xq)$. Then $r_\varphi = (t*) \cdot p \cdot \neg q$. A computation path accepted by r_φ demonstrates a finite sequence of states ending with a state satisfying p, followed by a state not satisfying q. This is a counterexample for φ.

3. Let $\varphi = (p \ W \ q)$. In this case we have $r_\varphi = (p \wedge \neg q) * \ \cdot (\neg p \wedge \neg q)$. Paths accepted by r_φ start with a finite number of states where p holds but q does not hold, followed by a state where p stops holding before q arrives. Such a scenario is a counterexample to $(p \ W \ q)$.

The second phase of the translation builds a non-deterministic finite automaton $A_{\neg\varphi}$ accepting the same language as r_φ. There are many known algorithms to achieve this [HU79], where the constructed automaton is of size linear in the size of the regular expression. We give an example of a full translation, from a specification into an automaton, using the automaton building algorithm of [BFR04].

Let us consider a specification stating that one cycle after the signal START appears, the signal BUSY should hold until END arrives. If END never arrives, BUSY should hold forever. We use s to represent START, b for BUSY, and e for END. In LTL, this would be written as follows.

$$\varphi = G(\mathsf{s} \rightarrow X(\mathsf{b} \ W \ \mathsf{e}))$$

Building r_φ as described above, we get

$$r_\varphi = (\mathsf{t}*) \ \cdot \ \mathsf{s} \ \cdot \ (\mathsf{b} \wedge \neg \mathsf{e}) \cdot (\neg \mathsf{b} \wedge \neg \mathsf{e})$$

The automaton $A_{\neg\varphi}$ is given in Figure 2.1.2. The initial state is state 1, and the accepting state is 4. Note that Figure 2.1.2 can also be seen as a state-machine, since from every state and for every input it is possible to progress to another state. Let this state-machine

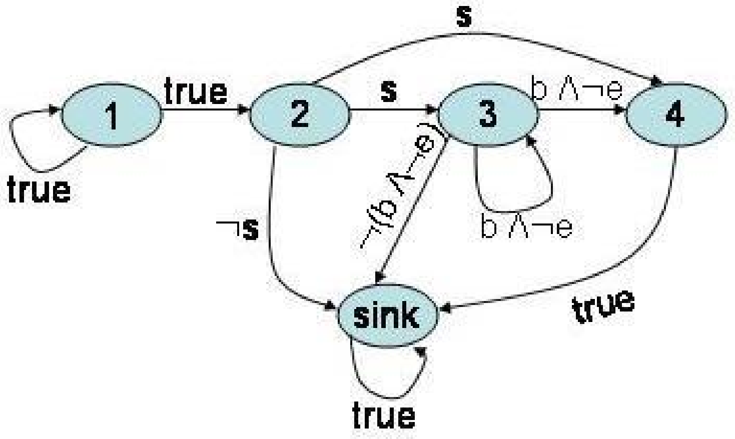

Figure 2.3: An NFA for $G(\mathsf{s} \rightarrow X(\mathsf{b}\ W\ \mathsf{e}))$

be called SM_φ. If φ holds in the model under verification, then it can never be the case that SM_φ reaches state 4 while $(\neg\mathsf{b} \wedge \neg\mathsf{e})$ holds in the state. Model checking of φ can now be carried out by running SM_φ with the model under verification, verifying the formula: $G\neg((SM_\varphi = 4) \wedge (\neg\mathsf{b} \wedge \neg\mathsf{e}))$.

2.1.3 Model Checking

The main challenge in model checking is known as the *state space explosion* problem, where the number of states in the model grows exponentially in the number of variables describing it. In this chapter we briefly describe some of the main methods used in practice to cope with the size problem. We present symbolic model checking of CTL formulas below, and then we sketch the bounded model checking method in Section 2.1.4.

Model Checking of CTL formulas

In [EC80] Emerson and Clarke showed that various branching time properties can be characterized as fixed points of appropriate monotonic functions. Later in [CE81] they introduced the logic CTL, and showed that its operators can be characterized in this way.

To present Clarke and Emerson's theorem, we need to introduce the notion of a *functional*. A functional is denoted $\lambda y.f$ where f is a formula and y is a variable. The variable y acts as a place holder. When applied to a parameter p, the functional $\lambda y.f$ yields f with p substituted for y. For example, if $\gamma = \lambda y.(x \wedge y)$ then $\gamma(true) = x \wedge true = x$. A *fixed point* of a functional γ is any p such that $\gamma(p) = p$. For example if $\gamma = \lambda y.(x \wedge y)$ then $(x \wedge y)$ is a fixed point of γ since $\gamma(x \wedge y) = x \wedge x \wedge y = x \wedge y$.

If γ is a monotonic functional, then it has a *least fixed point* as well as a *greatest fixed point*. The least and greatest fixed points of $\lambda y.f$ are denoted $\mu y.f$ and $\nu y.f$ respectively. A functional γ is union-continuous when for any non-decreasing infinite sequence of sets $p_1 \subseteq p_2 \subseteq ...$, we have $\cup_i \gamma(p_i) = \gamma(\cup_i p_i)$. Similarly, a functional γ is intersection-continuous when for any non-decreasing infinite sequence of sets $p_1 \subseteq p_2 \subseteq ...$, we have $\cap_i \gamma(p_i) = \gamma(\cap_i p_i)$. Tarski [Tar55] showed that if γ is monotonic and union-continuous, then the least fixed point of γ is $\cup_i \gamma^i(false)$ (that is, the union of the sequence obtained by iterating γ with the initial value false). Similarly, if γ is monotonic and intersection-continuous, then the least fixed point of γ is $\cap_i \gamma^i(true)$.

Clarke and Emerson viewed a CTL formula f as a set of states $\{s | s \models f\}$, the states in which the formula is true. Viewing CTL formulas this way, we can observe that the equation $\mathsf{EF}p = p \vee \mathsf{EXEF}p$ holds for all models. Thus $\mathsf{EF}p$ is a fixed point of the functional $\gamma = \lambda y.p \vee \mathsf{EX}y$, and in fact, it is the *least* fixed point of γ. In a similar way, Clarke and Emerson obtained the following characterizations:

1. $\mathsf{EF}p = \mu y.(p \vee \mathsf{EX}y)$

2. $\mathsf{EG}p = \nu y.(p \wedge \mathsf{EX}y)$

3. $\mathsf{E}(qUp) = \mu y.(p \vee (q \wedge \mathsf{EX}y))$

Since the above functionals are monotonic, and the set of states in our models is finite, Tarski's theorems apply, and we get an effective procedure for calculating the fixed points.

For $\mathsf{EF}p$ for example, we get:

$$\mathsf{EF}p = \cup_i(\lambda y.(p \vee \mathsf{EX}y))^i(false)$$

and thus it is enough to iterate EX until a fixed point is found, starting with false. Given a set of states S, calculating $\mathsf{EX}(S)$ is done by going one step backwards from S, to get all states that can reach S in one step through the transition relation.

For this, one needs an efficient way to represent and manipulate sets of states and relations. McMillan in [McM93] showed how this can be done using Binary Decision Diagrams (BDDs) [Bry86], that can be seen as a data structure that is especially efficient for the representation of Boolean functions. McMillan also wrote the first symbolic model checker called SMV [McM93].

2.1.4 Bounded Model Checking

Given a Kripke structure M, a formula φ, and a bound k, bounded model checking (BMC) tries to refute $M \models \varphi$ by proving the existence of a witness to the *negation* of φ, of length k or less. We use the notation M^k to denote the model M bounded by k. The idea of bounded model checking was first proposed in 1999 by Biere, Cimatti, Clarke and Zhu [BCCZ99]. They suggested to unfold a given model and specification k times, using auxiliary variables, making them into a propositional formula, and then use a satisfiability solver to find a satisfying assignment. Such an assignment, if found, serves as a counterexample to φ. The vast development of SAT solvers in recent years (See zChaff [MMZ$^+$01] and MiniSAT [ES04] for example), has made this method into the leading one in the world of hardware model checking.

We describe the translation of a BMC problem into a propositional formula in the next section.

Translating a BMC problem into a propositional formula

The BMC method of [BCCZ99] generates a propositional formula that is satisfiable if and only if $M^k \not\models \varphi$. We describe this method for invariant formulas, e.g. $\varphi = AG(p)$. For such formulas, we have that $M^k \not\models \varphi$ if and only if there exists a path $w = s_0, ..., s_j$ in M, such that $j \leq k$ and $s_j \models \neg p$.

We use the definition of a *model description* (Definition 3), given in Section 2.1.1. Let $MD = (I_{MD}, [\langle c_1, c_1' \rangle, ..., \langle c_n, c_n' \rangle])$ be a model description over a set of variables $V = \{v_1, ..., v_n\}$, and let $\varphi = AG(p)$ be the formula to be verified, with p being a Boolean expression over V. In order to unfold MD until a given bound k, we introduce k sets of new propositional variables $V^1 = \{v_1^1, ..., v_n^1\}, ..., V^k = \{v_1^k, ..., v_n^k\}$. For readability, the original set of variables V, will now be called V^0. According to Definition 3, a pair $\langle c_i, c_i' \rangle$ states that if the condition c_i holds in the current state (in terms of the variables V^0), then the value of v_i in any next state must be 0, while if c_i does not hold but c_i' does hold, then the value of v_i in the next state must be 1. We introduce k conditions $c_i^1, ..., c_i^k$ and k conditions $c_i^{1\prime}, ..., c_i^{k\prime}$ for each pair c_i, c_i', where c_i^j is the condition c_i written in terms of the variables V^j.

For each pair $\langle c_i, c_i' \rangle$ we introduce k propositional formulas:

$$T_i^j = (c_i^j \rightarrow \neg v_i^{j+1}) \wedge (\neg c_i^j \wedge c_i^{j\prime} \rightarrow v_i^{j+1})$$

for $0 \leq j < k$. The propositional formula that represents the unfolded model is composed of three parts:

- The initial condition I, written in terms of the variables V^0.
- The transition formulas T_i^j for $0 \leq j < k$.
- The negation of the specification: $P = \neg p^0 \vee \neg p^1 \vee \neg p^2 \vee ... \vee \neg p^k$, where p^i is the Boolean formula p written in terms of the set of variables V^i.

The model and specification are therefore represented by the propositional formula:

$$I \wedge P \wedge (\bigwedge_{0 \leq j < k, 0 < i \leq n} T_i^j).$$

We demonstrate the translation with the example below.

Example 7. We consider again the model given in Figure 2.1.

$$\texttt{Simple model} = (I, [\langle v_1 \wedge v_2, v_3 \rangle, \langle \neg v_2, v_1 \wedge \neg v_1 \rangle, \langle \neg v_1, v_1 \rangle])$$

over $V = \{v_1, v_2, v_3\}$ with $I = \neg v_1 \wedge v_2 \wedge \neg v_3$, and let $\varphi = AG(v_1 \vee v_2)$. We choose $k = 4$. To translate the model into a propositional formula for bound 4, we introduce 4 copies of the variables, $V^1 = \{v_1^1, v_2^1, v_3^1\}, ..., V^4 = \{v_1^4, v_2^4, v_3^4\}$. We first have to write the initial condition I in terms of the variables V^0:

$$I = \neg v_1^0 \wedge v_2^0 \wedge \neg v_3^0$$

Second, we build the propositional formula P corresponding to the specification:

$$P = ((\neg v_1^0 \wedge \neg v_2^0) \vee (\neg v_1^1 \wedge \neg v_2^1) \vee (\neg v_1^2 \wedge \neg v_2^2) \vee (\neg v_1^3 \wedge \neg v_2^3) \vee (\neg v_1^4 \wedge \neg v_2^4))$$

We now build the formulas T_i^j. For the next-state value of v_1 in time step 0, we have:

$$T_1^0 = ((v_1^0 \wedge v_2^0) \rightarrow \neg v_1^1) \wedge ((\neg(v_1^0 \wedge v_2^0) \wedge \neg v_3^0) \rightarrow v_1^1)$$

T_1^1, T_1^2 and T_1^3 will be the same as T_1^0, with all the top indexes shifted.

For the next-state value of v_2 we get:

$$T_2^0 = (\neg v_2^0 \rightarrow \neg v_2^1) \wedge (v_2^0 \wedge v_1^0 \wedge \neg v_1^0 \rightarrow v_2^1)$$

Note that the right hand side of T_2^0 is equivalent to **True**, thus we get $T_2^0 = (\neg v_2^0 \rightarrow \neg v_2^1)$. Similarly, we have $T_2^1 = (\neg v_2^1 \rightarrow \neg v_2^2)$, $T_2^2 = (\neg v_2^2 \rightarrow \neg v_2^3)$ and $T_2^3 = (\neg v_2^3 \rightarrow \neg v_2^4)$. For the last variable v_3 we have:

$$T_3^0 = (\neg v_1^0 \rightarrow \neg v_3^1) \wedge (v_1^0 \rightarrow v_3^1)$$

and T_3^1, T_3^2, T_3^3 are defined in the same way, with the top indexes shifted as above. The propositional formula for the model, unfolded to depth 4 is then:

$$T^4_{\texttt{Simple}} = I \wedge P \wedge T_1^0 \wedge T_1^1 \wedge T_1^2 \wedge T_1^3 \wedge T_2^0 \wedge T_2^1 \wedge T_2^2 \wedge T_2^3 \wedge T_3^0 \wedge T_3^1 \wedge T_3^2 \wedge T_3^3$$

In order to find a counterexample of length k or less, we need to find a satisfying assignment for $T^4_{\texttt{Simple}}$. Since $T^4_{\texttt{Simple}}$ is a propositional formula, a satisfiability solver can now be applied to it. If no satisfying assignment exists, it means that no bug can be found until cycle 4.

2.2 Description Logic

Description Logic [BCM$^+$03] is a family of knowledge representation formalisms. It has evolved from earlier knowledge representations, such as *network semantics* [Qui67, CQ69], and *frames* [Min81] in an attempt to overcome ambiguities in the semantics of those formalisms.

In description logic, the basic elements are *atomic concepts* and *atomic roles*. Atomic concepts are unary predicate symbols, denoting sets of individuals; atomic roles are binary predicate symbols, used to express relationships between individuals. Complex descriptions of concepts and roles can be built from simpler ones by using *constructors*. Different dialects of Description Logic are distinguished by the constructors they allow. An important feature of description logic is the ability to *infer* about the described knowledge-base: to find implicit facts from the explicit information given.

We present the formal syntax and semantics of different description logic dialects in Section 2.2.1. Section 2.2.2 discusses how knowledge-bases are represented using terminologies (Tboxes) and world descriptions (Aboxes). Section 2.2.3 then presents the basic reasoning algorithm using tableau construction.

2.2.1 Syntax and semantics

The basic description logic dialect that we use in this work is known as $\mathcal{ALC}$ (for *attribute language* with *complement*). In $\mathcal{ALC}$, complex concepts are formed from simpler ones using the following constructors, where A is an atomic concept, C and D are concepts and R is a role:

$$\begin{array}{rl} A & (atomic\ concept) \\ \top & (universal\ concept) \\ \bot & (bottom\ concept) \\ \neg C & (negation) \\ C \sqcap D & (intersection) \\ \forall R.C & (value\ restriction) \end{array}$$

The union operator is defined as $C \sqcup D = \neg(\neg C \sqcap \neg D)$, and existential quantification is defined as $\exists R.C = \neg\forall R.\neg C$. The *semantics* of $\mathcal{ALC}$ is defined with respect to a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set, and $\cdot^{\mathcal{I}}$ is a function mapping every atomic concept to a subset of $\Delta^{\mathcal{I}}$ and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Figure 2.4 presents the semantics of $\mathcal{ALC}$ constructors.

A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
$\top$	$\Delta^{\mathcal{I}}$
$\bot$	$\emptyset$
$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$

Figure 2.4: The semantics of $\mathcal{ALC}$

We review terminologies and reasoning procedures for $\mathcal{ALC}$ in the next sections. In Section 2.2.4 we define other DL dialects that are used in this document.

2.2.2 Terminologies and world descriptions

Complex descriptions of concepts and roles are used to describe the classes of objects of a given domain. The knowledge-base itself is composed of two components: a *terminology* (or *Tbox*) and a *world description* (or *Abox*). The terminology gives a list of *axioms*, that describe relations between concepts in the domain. In the most general case, terminological axioms have the form of a concept inclusion

$$C \sqsubseteq D,$$

where C and D are concepts written in terms of a given dialect. The semantics of a concept inclusion is as expected: an interpretation $\mathcal{I}$ satisfies the concept inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Figure 2.5 presents a terminology with concepts about animals and their eating habits. The second component of a knowledge-base, the Abox, lists assertions about individual names in a specific domain. These will be of the form

$$\boxed{\begin{array}{rcl} \text{Herbivore} & \sqsubseteq & \forall eats.\neg\text{Animal} \\ \text{Omnivore} & \sqsubseteq & \exists eats.\text{Animal} \sqcap \exists eats.\neg\text{Animal} \\ \text{Cow} & \sqsubseteq & \text{Animal} \sqcap \text{Herbivore} \\ \text{Human} & \sqsubseteq & \text{Animal} \sqcap (\text{Herbivore} \sqcup \text{Omnivore}) \end{array}}$$

Figure 2.5: A Tbox with facts about eating habits

$$C(a)\ , \qquad R(b, c)$$

where a, b, c are individual names in the domain, C a concept and R a role. The above assertions state that a is a member of C and b, c are related by R. When an Abox is present, an interpretation $\mathcal{I}$ should also map the individual names: each individual a will be mapped to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. Figure 2.6 gives an example of an Abox.

Cow(DINA)
$eats$(DINA,GRASS)

Figure 2.6: An Abox

2.2.3 Reasoning

Different inference tasks come to mind when dealing with Description Logic. For example, we can consider:

- Subsumption - is one concept more general than another: $C \sqsubseteq D$?
- Consistency - does a given concept C have an interpretation?
- Membership - is the individual i a member of a concept C in all interpretations?

It turns out that the different inference tasks can all be reduced to the question of *consistency*. We use $\models_{dl} C$ to indicate consistency in DL, to differentiate it from satisfaction in the model checking world. Thus, the general consistency problem, with respect to a Tbox $\mathcal{T}$, asks if $\mathcal{T} \models_{dl} C$ holds; that is, if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}}$ is non-empty and such that $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ holds for every $C_1 \sqsubseteq C_2$ in $\mathcal{T}$.

Tableaux algorithms for consistency checking of C with respect to a terminology $\mathcal{T}$, try to prove the consistency by demonstrating the existence of an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}}$ is not empty and all the concept inclusions in $\mathcal{T}$ hold. This is done by syntactically decomposing C, to derive constraints on the structure of such an interpretation. The construction fails if the constraints include a *clash*; that is, if some individual x must be an element of both D and $\neg D$ for some concept D. The algorithm is designed in such a way that it is guaranteed to terminate, and guaranteed to construct an interpretation if one exists.

In practice, the algorithm works on a labeled tree, called a *completion tree*[1], that has a close correspondence to an interpretation for $\mathcal{T}$ and C. We assume that concepts are given in *negation normal form* (NNF), where negations are allowed only on atomic concepts. A concept can be transformed into an equivalent one in NNF by pushing negation inwards, making use of de Morgan's laws and the duality between existential and universal restrictions [HST00]. For a concept C, we write $\mathsf{nnf}(C)$ to denote the NNF of C, write $\dot{\neg}C$ to denote the NNF of $\neg C$, and $\mathsf{sub}(C)$ to denote the set of all sub-concepts of C (including C) and their negations. For a TBox $\mathcal{T}$ we define $\mathsf{sub}(\mathcal{T}) = \bigcup_{(C \sqsubseteq D) \in \mathcal{T}} \mathsf{sub}(C) \cup \mathsf{sub}(D)$.

Definition 8. Let $\mathcal{T}$ be an $\mathcal{ALC}$ TBox and C a concept in NNF. A *completion tree* for C with respect to $\mathcal{T}$ is a directed graph $\mathbf{G} = (V, E, \mathcal{L})$ where each node $x \in V$ is labelled with a set $\mathcal{L}(x) \subseteq \mathsf{sub}(\mathcal{T}) \cup \mathsf{sub}(C)$ and each edge $\langle x, y \rangle \in E$ is labelled with a role name $\mathcal{L}(\langle x, y \rangle) \in R_N$.

If $\langle x, y \rangle \in E$, then y is called a *successor* of x and x is called a *predecessor* of y. If, in addition, $R = \mathcal{L}(\langle x, y \rangle)$, then y is called an R*-successor* of x and x is called an R*-predecessor* of y. *Ancestor* is the transitive closure of predecessor, and *descendant* is the transitive closure of successor.

$\mathbf{G}$ is said to contain a *clash* if for some $A \in \mathsf{NC}$ and node x of $\mathbf{G}$, $\{A, \neg A\} \subseteq \mathcal{L}(x)$.

The tableaux algorithm for checking concept consistency of C with respect to $\mathcal{T}$ starts with the completion tree $\mathbf{G} = (\{r_0\}, \emptyset, \mathcal{L})$ where $\mathcal{L}(r_0) = \{\mathsf{nnf}(C)\}$. $\mathbf{G}$ is then expanded by repeatedly applying the expansion rules given in Figure 2.7, stopping if a clash occurs.

In order to ensure termination we need to restrict the creation of new nodes in the completion tree. The notion of *blocking* is used for this purpose.

Definition 9. A node x is *label blocked* if it has an ancestor y such that $\mathcal{L}(x) \subseteq \mathcal{L}(y)$. In this case, we say that y *blocks* x. A node is *blocked* if either it is label blocked or its predecessor is blocked.

[1] We note that for more expressive DL dialects a completion *graph* may be needed.

$\sqsubseteq$-rule: if 1. $C_1 \sqsubseteq C_2 \in \mathcal{T}$, and
2. $\{\dot{\neg} C_1, \mathsf{nnf}(C_2)\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{\dot{\neg} C_1, \mathsf{nnf}(C_2)\}$

$\sqcap$-rule: if 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, and
2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$

$\sqcup$-rule: if 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, and
2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$

$\exists$-rule: if 1. $\exists R.C \in \mathcal{L}(x)$, x is not blocked, and
2. x has no R-successor y with $C \in \mathcal{L}(y)$,
then create a new node y with $\mathcal{L}(\langle x, y \rangle) = R$
and $\mathcal{L}(y) = \{C\}$

$\forall$-rule: if 1. $\forall R.C \in \mathcal{L}(x)$, and
2. there is an R-successor y of x such that $C \notin \mathcal{L}(y)$
then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$

Figure 2.7: Tableaux expansion rules for $\mathcal{ALC}$

When nodes in a branch of the completion tree resemble ancestor nodes, a block is established to ensure that further applications of $\exists$-rule are not applied to the blocked nodes (and therefore ensure termination).

Definition 10. A completion tree $\mathbf{G}$ is called *complete* if no expansion rule can be applied. $\mathbf{G}$ is *clash-free* if no node contains a clash.

A *tableaux algorithm* for checking concept consistency of an $\mathcal{ALC}$ concept C with respect to a TBox $\mathcal{T}$ builds a completion tree for C. If a complete and clash-free tree can be obtained, the algorithm returns "consistent"; otherwise, if it was unable to build such a tree, it returns "unsatisfiable".

Theorem 11. *(decision procedure, [SS91]) The tableaux algorithm always terminates for a given $\mathcal{ALC}$ concept C and TBox $\mathcal{T}$, and returns "consistent" iff C is satisfiable with respect to a TBox $\mathcal{T}$.*

2.2.4 Other Dialects

We present additional DL dialects that are needed for our results in Chapter 3.

- Inverse roles (indicated by the letter $\mathcal{I}$).

  If R is a role, this construct allows us to define the concept $\forall R^{-}.C$, for any concept C. Given a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, the semantics is defined as $(\forall R^{-}.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} : \forall y.(y, x) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$.

- Nominals (indicated by $\mathcal{O}$).

  This constructor allows the definition of a concept as a set of individuals: $\{s_1, ..., s_k\}$. the semantics, given a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, is as expected: $(\{s_1, ..., s_k\})^{\mathcal{I}} = \{(s_1)^{\mathcal{I}}, ..., (s_k)^{\mathcal{I}}\}$.

- Functional roles (indicated by $\mathcal{F}$).

  Allows defining some or all of the roles as *functionals*. If R is functional and

$\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is given, then for any $x, y_1, y_2 \in \Delta^{\mathcal{I}}$, $\{(x, y_1), (x, y_2)\} \subseteq R^{\mathcal{I}}$ implies $y_1 = y_2$.

Chapter 3

Symbolic Model Checking using Description Logic

We present a variety of encodings for symbolic model checking using Description Logic. For all of those encodings the ontology constructed describes an error in the system, and interpretations if found, provide legal paths from the initial state of the model to a buggy state. Interpretations can thus serve as counterexamples.

In this chapter we give formulations of bounded model checking of *invariance* properties, of the type $\mathsf{G}(p)$, and in Chapter 4 we discuss unbounded model checking of *inevitability* properties ($\mathsf{F}(p)$), where p is a Boolean expression over the set of state variables V. As explained in Section 2.1.2, all LTL properties can be translated into these types of formula via the construction of a Büchi automaton. Note that the CTL formulas $\mathsf{AG}(p)$ and $\mathsf{AF}(p)$ are equivalent to the LTL ones $\mathsf{G}(p)$ and $\mathsf{F}(p)$ respectively. We sometimes use the CTL notation, since the description of an erroneous situation ($\mathsf{EF}(\neg p)$ or $\mathsf{EG}(\neg p)$) is easier in CTL.

The rest of this chapter is organized as follows. In Section 3.1 we show how a model description MD can be represented as a TBox $\mathcal{T}_{MD}$ over the Description Logic dialect $\mathcal{ALC}$. We then prove several lemmas in Section 3.1.1, correlating interpretations satisfying $\mathcal{T}_{MD}$ with sub-models of the Kripke structure M_{MD} described by MD. These lemmas will be used later in this chapter to prove the correctness of our encodings. In Section 3.2 we present various ways of phrasing a bounded model checking problem as a consistency problem in DL. The methods differ from each other in the DL dialects they use as well as in the encodings themselves. We prove the correctness of our encodings in Section 3.3, and in Section 3.4 we sketch an alternative symbolic representation of a model description, and review the changes needed in all of the previously presented encodings. Section 3.5 gives experimental results, and Section 3.6 concludes this chapter with a discussion.

3.1 Modeling A Kripke Structure as a TBox

We start by presenting a natural encoding of a model description MD as a terminology over $\mathcal{ALC}$. Let $MD = (I, [\langle c_1, c_1' \rangle, ..., \langle c_n, c_n' \rangle])$ be a model description for the model $M_{MD} = (S, I, R, L)$, over $V = \{v_1, ..., v_n\}$, as defined in Section 2.1.1.

For each variable $v_i \in V$ we introduce one primitive concept V_i, where V_i denotes $v_i = 1$ and $\neg\mathsf{V}_i$ denotes $v_i = 0$. We introduce one primitive role R corresponding to the transition relation of the model. Given a Boolean expression p over the state variables $v_1, ..., v_n$, we denote $\mathcal{D}(p)$ the concept P derived from p by replacing each v_i in p with V_i, and $\vee, \wedge, \neg$ with $\sqcap, \sqcup, \neg$ respectively. For example, if p = $(\neg v_1 \wedge v_2)$, then $\mathcal{D}(p) = (\neg\mathsf{V}_1 \sqcap \mathsf{V}_2)$.

We define the concept S_0 to represent the set of initial states: $\mathsf{S}_0 = \mathcal{D}(I)$. We define $\mathsf{C}_i = \mathcal{D}(c_i)$, $\mathsf{C}_i' = \mathcal{D}(c_i')$, for all $1 \leq i \leq n$. The TBox $\mathcal{T}_{MD}$ would consist of the following concept inclusions, describing the model: for each pair $\langle c_i, c_i' \rangle$ we introduce the inclusions

$$\begin{array}{rcl} \mathsf{C}_i & \sqsubseteq & \forall\mathsf{R}.\neg\mathsf{V}_i \\ (\neg\mathsf{C}_i \sqcap \mathsf{C}_i') & \sqsubseteq & \forall\mathsf{R}.\mathsf{V}_i \end{array}$$

Interpretations for $\mathcal{T}_{MD}$ will consist of individuals that correspond to states in the system MD. Note that in our DL translation, if an individual σ belongs to a concept V_i it means that the variable v_i has the value 1 in the corresponding state s. The first inclusion ensures that in any interpretation, an individual that belongs to C_i can be related by R only to individuals that do not belong to V_i. Since individuals correspond to states in the model M_{MD}, this means that when c_i holds in a state s, all neighbor states of s must have $v_i = 0$. The above inclusions thus restrict the role R to agree with the definition of R in the model description. Note that for a model description MD over n variables, $\mathcal{T}_{MD}$ will consists of only $2n$ concept inclusions.

As an example, consider the Kripke structure `Simple` presented in Figure 2.1. Its MD is given as $\mathtt{Simple} = (I, [\langle v_1 \wedge v_2, v_3 \rangle, \langle \neg v_2, v_1 \wedge \neg v_1 \rangle, \langle \neg v_1, v_1 \rangle])$ over $V = \{v_1, v_2, v_3\}$ with $I = \neg v_1 \wedge v_2 \wedge \neg v_3$. We build a TBox $\mathcal{T}_{\mathtt{Simple}}$ for it. We introduce three primitive concepts $\mathsf{V}_1, \mathsf{V}_2, \mathsf{V}_3$ and one primitive role R. Figure 3.1 below gives the full TBox.

Note that for simplicity, we omitted the inclusion $(\neg\neg\mathsf{V}_2 \sqcap \mathsf{V}_1 \sqcap \neg\mathsf{V}_1) \sqsubseteq \forall \mathsf{R}.\mathsf{V}_2$ (corresponding to $\neg\mathsf{C}_i \sqcap \mathsf{C}'_i \sqsubseteq \forall\mathsf{R}.\mathsf{V}_i$ for $i = 2$), since the prefix $\neg\neg\mathsf{V}_2 \sqcap \mathsf{V}_1 \sqcap \neg\mathsf{V}_1$ is equivalent to $\bot$. Similarly, the concept $\neg\neg\mathsf{V}_1 \sqcap \mathsf{V}_1$ (corresponding to $\neg\mathsf{C}_3 \sqcap \mathsf{C}'_3$) was replaced by the equivalent V_1.

$$\boxed{\begin{array}{rcl}
\mathsf{S}_0 & \sqsubseteq & \neg\mathsf{V}_1 \sqcap \mathsf{V}_2 \sqcap \neg\mathsf{V}_3 \\
(\mathsf{V}_1 \sqcap \mathsf{V}_2) & \sqsubseteq & \forall\mathsf{R}.\neg\mathsf{V}_1 \\
(\neg(\mathsf{V}_1 \sqcap \mathsf{V}_2) \sqcap \mathsf{V}_3) & \sqsubseteq & \forall\mathsf{R}.\mathsf{V}_1 \\
\neg\mathsf{V}_2 & \sqsubseteq & \forall\mathsf{R}.\neg\mathsf{V}_2 \\
\neg\mathsf{V}_1 & \sqsubseteq & \forall\mathsf{R}.\neg\mathsf{V}_3 \\
\mathsf{V}_1 & \sqsubseteq & \forall\mathsf{R}.\mathsf{V}_3
\end{array}}$$

Figure 3.1: The TBox $\mathcal{T}_{Simple}$

In the subsection below we prove that interpretations of $\mathcal{T}_{MD}$ must correspond to sub-models of the Kripke structure M_{MD}. The propositions presented here will be used later in this chapter to prove the correctness of our encodings.

3.1.1 TBox Interpretations as sub-models of M

Let $MD = (I, [\langle c_1, c_1' \rangle, ..., \langle c_n, c_n' \rangle])$ be a model description for the model $M_{MD} = (S, I, R, L)$, over $V = \{v_1, ..., v_n\}$, and let $\mathcal{T}_{MD}$ be the TBox built for it as described above. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation for $\mathcal{T}_{MD}$. We define a mapping $\mathcal{M} : \Delta^{\mathcal{I}} \rightarrow S$ that relates individuals from $\Delta^{\mathcal{I}}$ to states from S.

Definition 12. $\mathcal{M}(\sigma) = s$ if $\forall i$, $1 \leq i \leq n$, $\sigma \in \mathsf{V}_i^{\mathcal{I}}$ if and only if $s \models v_i$.

Note that $\mathcal{M}$ is a function, since a state s is determined by the value of the variables $v_1, ..., v_n$. The following lemma shows that σ and $\mathcal{M}(\sigma)$ agree also on Boolean expressions over $v_1, ..., v_n$.

Lemma 13. Let b be a Boolean expression over $v_1, ..., v_n$, and $\mathsf{B} = \mathcal{D}(b)$ its corresponding concept. Let $\sigma \in \Delta^{\mathcal{I}}$ be an element in the interpretation $\mathcal{I}$, and let $s = \mathcal{M}(\sigma)$. Then $\sigma \in \mathsf{B}^{\mathcal{I}}$ if and only if $s \models b$.

Proof. By induction on the structure of the Boolean expression b. □

Corollary 14. Let $\sigma_1, \sigma_2 \in \Delta^{\mathcal{I}}$, $\mathcal{M}(\sigma_1) = s_1$, $\mathcal{M}(\sigma_2) = s_2$. If $(\sigma_1, \sigma_2) \in \mathsf{R}^{\mathcal{I}}$ then $(s_1, s_2) \in R$.

Before we prove Corollary 14, we note that the other direction does not hold: if $(s_1, s_2) \in R$ in the system MD it does not necessarily imply that $(\sigma_1, \sigma_2) \in \mathsf{R}^{\mathcal{I}}$. To see this, note that the concept inclusions in $\mathcal{T}_{MD}$ do not enforce any ‘edge’ to exist in an interpretation; they only assert conditions on edges, if they do exist. Thus, an interpretation that has no edges at all, would satisfy all concept inclusions of $\mathcal{T}_{MD}$. Note also that the direction stated in the corollary is the only one needed for our proofs.

Proof of Corollary 14. If $(\sigma_1, \sigma_2) \in \mathsf{R}^{\mathcal{I}}$, then since $\mathcal{I}$ is an interpretation for $\mathcal{T}_{MD}$ we know that $\forall i$, $1 \leq i \leq n$, ($\sigma_1 \in \mathsf{C}_i^{\mathcal{I}}$ implies $\sigma_2 \in \neg\mathsf{V}_i^{\mathcal{I}}$ and $\sigma_1 \in \mathsf{C}_i'^{\mathcal{I}} \cap \neg\mathsf{C}_i^{\mathcal{I}}$ implies $\sigma_2 \in \mathsf{V}_i$). But since $\mathsf{C}_i = \mathcal{D}(c_i)$ and $\mathsf{C}_i' = \mathcal{D}(c_i')$, we get by Lemma 13, that $\forall i$, $1 \leq i \leq n$, ($s_1 \models c_i$ implies $s_2 \models \neg v_i$ and $s_1 \models c_i' \wedge \neg c_i$ implies $s_2 \models v_i$). By Definition 3 we get that $(s_1, s_2) \in R$.

□

Corollary 15. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation for $\mathcal{T}_{MD}$, and let $\sigma_0, \sigma_1, ..., \sigma_m$ be individuals in $\Delta^{\mathcal{I}}$, such that $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$. We define a sequence of states $w_{\mathcal{I}} = s_0, s_1, ..., s_m$ such that $\mathcal{M}(\sigma_i) = s_i$. Then $w_{\mathcal{I}}$ is a path in M_{MD} .

Proof. Follows directly from Corollary 14. □

Corollary 16. Let $w = s_0, s_1, ..., s_m$ be a path in M_{MD}. Let $\mathcal{I}_w = (\Delta^{\mathcal{I}_w}, \cdot^{\mathcal{I}_w})$ be a structure derived from w: $\Delta^{\mathcal{I}_w} = \{\sigma_0, \sigma_1, ..., \sigma_m\}$, and $\cdot^{\mathcal{I}_w}$ maps the individuals in such a way that $\mathcal{M}(\sigma_i) = s_i$ and $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}_w}$. Then $\mathcal{I}_w$ is an interpretation for $\mathcal{T}_{MD}$.

Proof. All the inclusions in $\mathcal{T}_{MD}$ have the form $\mathsf{C}_i \sqsubseteq \forall\mathsf{R}.\neg\mathsf{V}_i$ or $(\neg\mathsf{C}_i \sqcap \mathsf{C}_i') \sqsubseteq \forall\mathsf{R}.\mathsf{V}_i$. We know that for all i, $1 \leq i < n$, $(s_i, s_{i+1}) \in R$. Thus, by the definition, for all j, $1 \leq j \leq n$, ($s_i \models c_j$ implies $s_{i+1} \models \neg v_j$ and $s_i \models c_j' \wedge \neg c_j$ implies $s_{i+1} \models v_j$). Since by construction, $\mathcal{M}(\sigma_i) = s_i$, we get by Lemma 13 that for all j, $1 \leq j \leq n$, ($\sigma_i \in \mathsf{C}_j^{\mathcal{I}}$ implies $\sigma_{i+1} \in \neg\mathsf{V}_j^{\mathcal{I}}$ and $\sigma_i \in \mathsf{C}_j'^{\mathcal{I}} \cap \neg\mathsf{C}_j^{\mathcal{I}}$ implies $\sigma_{i+1} \in \mathsf{V}_j$). Thus the pairs (σ_i, σ_{i+1}) obey the concept inclusions of $\mathcal{T}_{MD}$. Since these pairs are the only ones in $\mathsf{R}^{\mathcal{I}_w}$, the inclusions hold under the interpretation $\mathcal{I}_w$. □

Note that the TBox built so far describes only the model and does not consider the specification to be verified. Legal interpretations include for example the empty interpretation, and are not necessarily useful for verification. In order to use DL reasoning for model checking we need to add restrictions to the terminology to stand for the specification. In all methods described in the sequel, we add concept inclusions or assertions that describe a *bug* in the model. Interpretations will therefore be legal sub-models that demonstrate an erroneous behavior. In the section below we discuss safety properties. Liveness properties are treated in Chapter 4.

3.2 The Different BMC methods

Let $\varphi = AG(p)$ be the formula to be verified, with p being a Boolean expression over the state variables $v_1, ..., v_n$. Recall that in bounded model checking of bound k, one tries to refute the satisfaction of $AG(p)$ in the given model by presenting a path of length k or less, that leads to a state where $\neg p$ holds. In order to encode this as a DL query, we use the TBox $\mathcal{T}_{MD}$ described in Section 3.1, and add two components to it, one describing a bounded path and the other describing a buggy state. Let $\mathcal{T}^k_{MD}$ be the TBox representing both the model and the bounded path, and let C_φ be a concept representing a bug. Model checking is then carried out by asking the DL reasoner to determine whether $\mathcal{T}^k_{MD} \models_{dl} C_\varphi$. If the answer is positive, it means that an interpretation is found for $\mathcal{T}^k_{MD}$ such that C_φ is not empty. Such an interpretation represents a counterexample.

Below we present four encodings of a bounded model checking problem as a consistency query in DL. The methods we describe differ from each other by the way a bounded path of length k is defined, and by the way the formula is represented. We demonstrate each method on the example `Simple` presented in Figure 2.1.

1. **Using $\mathcal{ALC}$**

   For this method we use the terminology $\mathcal{T}_{MD}$ built in section 3.1, and add nothing to encode a bounded path. Rather, we encode the possible existence of a bug at distance k or less, as one concept inclusion. Let $\varphi = AG(p)$, and $\mathsf{P} = \mathcal{D}(p)$ the concept representing p. We define the concept C^1_φ as follows.

$$\mathsf{C}^1_\varphi \sqsubseteq \mathsf{S}_0 \sqcap (\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup ...\exists\mathsf{R}.\neg\mathsf{P})...)))$$

with k nested $\exists$Rs.

As an example, consider the model `Simple`, the bound $k = 4$ and the specification $\varphi = AG(\neg v_2 \vee \neg v_3)$. We build $\mathcal{T}_{\mathtt{Simple}}$ as shown in Section 3.1, and the concept C^1_φ as given below.

$$\mathsf{C}^1_\varphi \sqsubseteq \mathsf{S}_0 \sqcap ((\mathsf{V}_2 \sqcap \mathsf{V}_3) \sqcup \exists\mathsf{R}.((\mathsf{V}_2 \sqcap \mathsf{V}_3) \sqcup \exists\mathsf{R}.((\mathsf{V}_2 \sqcap \mathsf{V}_3) \sqcup \exists\mathsf{R}.((\mathsf{V}_2 \sqcap \mathsf{V}_3) \sqcup \exists\mathsf{R}.(\mathsf{V}_2 \sqcap \mathsf{V}_3)))))$$

Verification will now take place by asking whether $\mathcal{T}_{\mathtt{Simple}} \models_{dl} \mathsf{C}^1_\varphi$.

2. **Using $\mathcal{ALCI}$.**

Recall that the concept S_0 represents the set of initial states of M. If S_1 represents states that can be reached in one step from S_0, then the concept inclusion $\mathsf{S}_1 \sqsubseteq \exists\mathsf{R}^-.\mathsf{S}_0$ must hold (that is, the set S_1 is a subset of all the states that can reach S_0 by going one step backwards using the relation R). Similarly, we denote by S_i subsets of the states reachable in i steps from the set of initial states, and introduce the inclusions

$$\mathsf{S}_i \sqsubseteq \exists\mathsf{R}^-.\mathsf{S}_{i-1}$$

for $0 < i \leq k$. We call this set of concept inclusions $\mathcal{T}_k$.

For $\varphi = AG(p)$, let $\mathsf{P} = \mathcal{D}(p)$ be the concept representing p. We define the concept $\mathsf{C}^2_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup ... \sqcup \mathsf{S}_k)$. If C_φ is consistent with respect to the terminology $\mathcal{T}^k_{MD} = \mathcal{T}_k \cup \mathcal{T}_{MD}$, it means that $\neg p$ holds in a state with distance k or less from the initial state. Model checking is thus reduced to the query: $\mathcal{T}^k_{MD} \models_{dl} \mathsf{C}^2_\varphi$. A positive answer from the DL reasoner indicates an error in M_{MD}.

For the model `Simple`, the bound $k = 4$ and the specification $\varphi = AG(\neg v_2 \vee \neg v_3)$. Figure 3.2 describes $\mathcal{T}^4_{\mathtt{Simple}}$, the TBox representing both the model and bound.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(\mathsf{V}_1 \sqcap \mathsf{V}_2) & \sqsubseteq & \forall \mathsf{R}.\neg\mathsf{V}_1 & \mathsf{S}_0 & \sqsubseteq & (\neg\mathsf{V}_1 \sqcap \mathsf{V}_2 \sqcap \neg\mathsf{V}_3) \\
(\neg(\mathsf{V}_1 \sqcap \mathsf{V}_2) \sqcap \mathsf{V}_3) & \sqsubseteq & \forall \mathsf{R}.\mathsf{V}_1 & \mathsf{S}_1 & \sqsubseteq & \exists \mathsf{R}^{-}.\mathsf{S}_0 \\
\neg\mathsf{V}_2 & \sqsubseteq & \forall \mathsf{R}.\neg\mathsf{V}_2 & \mathsf{S}_2 & \sqsubseteq & \exists \mathsf{R}^{-}.\mathsf{S}_1 \\
\neg\mathsf{V}_1 & \sqsubseteq & \forall \mathsf{R}.\neg\mathsf{V}_3 & \mathsf{S}_3 & \sqsubseteq & \exists \mathsf{R}^{-}.\mathsf{S}_2 \\
\mathsf{V}_1 & \sqsubseteq & \forall \mathsf{R}.\mathsf{V}_3 & \mathsf{S}_4 & \sqsubseteq & \exists \mathsf{R}^{-}.\mathsf{S}_3
\end{array}$$

Figure 3.2: The TBox $\mathcal{T}^4_{Simple}$

For the specification $\varphi = AG(\neg v_2 \vee \neg v_3)$ we get $\mathsf{P} \equiv \neg\mathsf{V}_2 \sqcup \neg\mathsf{V}_3$, and $\mathsf{C}^2_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup \mathsf{S}_2 \sqcup \mathsf{S}_3 \sqcup \mathsf{S}_4)$. Verification is then carried out by asking the query: Is the concept C^2_φ consistent with respect to $\mathcal{T}^4_{\mathtt{Simple}}$?

3. **Using $\mathcal{ALCO}$ and Aboxes.**

   The method described in item (2) above encodes a bounded path with a set of concept inclusions, and thus uses *inverse roles*. We show how a bounded path, as well as the formula to be checked can be encoded as a set of ABox assertions. For a bound k, we introduce $k+1$ individuals, $\mathsf{s}_0, \mathsf{s}_1, ..., \mathsf{s}_k$. The assertion $\mathsf{S}_0(\mathsf{s}_0)$ makes s_0 an initial state, and for $0 \leq i < k$, the assertions $\mathsf{R}(\mathsf{s}_i,\mathsf{s}_{i+1})$ make s_i a state of distance i from the initial state. We call this set of assertions $\mathcal{A}_k$. In order to verify the specification φ we use *nominals*. For $\varphi = AG(p)$ we define the concept $\mathsf{P} = \mathcal{D}(p)$ as before, and define the concept $\mathsf{C}^3_\varphi \sqsubseteq \neg\mathsf{P} \sqcap \{\mathsf{s}_0, ..., \mathsf{s}_k\}$. Verification for this method is done by asking the query: $(\mathcal{T}_{MD}, \mathcal{A}_k) \models_{dl} \mathsf{C}^3_\varphi$.

   For the example `Simple`, with bound $k = 4$ and $\varphi = AG(\neg v_2 \vee \neg v_3)$, we build the ABox

   $$\mathcal{A}_4 = \{\mathsf{S}_0(\mathsf{s}_0), \mathsf{R}(\mathsf{s}_0, \mathsf{s}_1), \mathsf{R}(\mathsf{s}_1, \mathsf{s}_2), \mathsf{R}(\mathsf{s}_2, \mathsf{s}_3), \mathsf{R}(\mathsf{s}_3, \mathsf{s}_4)\}$$

   and the concept

   $$\mathsf{C}^3_\varphi \sqsubseteq \mathsf{V}_2 \sqcap \mathsf{V}_3 \sqcap \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3, \mathsf{s}_4\}$$

Verification is done by asking the query $(\mathcal{T}_{\texttt{Simple}}, \mathcal{A}_k) \models_{dl} \mathtt{C}^3_\varphi$.

Note that the assertions in $\mathcal{A}_k$ form a symbolic path of length $k + 1$ through the model, starting from an initial states. Moreover, this syntactic definition of a path does not depend on the model described in the Tbox $\mathcal{T}_{MD}$.

4. **Using $\mathcal{ALCF}$ and Aboxes.**

This method encodes a bounded path as an ABox as described in item (3) above. However, we use a special encoding for the formula, that involves enhancing both the TBox and the ABox. It is based on two known facts. First, that if $AG(p)$ does not hold in a model then $EF(\neg p)$ does. Second, that $EF(p)$ has a fixpoint representation (Clarke and Emerson's [CE81]):

$$EF(p) \equiv p \vee EX(EF(p))$$

That is, in order for $EF(p)$ to hold in a state, either p should hold in the current state, or there should exist a next state where $EF(p)$ holds. In order to encode this in DL, we need to enhance both the TBox built in section 3.1 and the ABox described in item (3) above.

We first define R to be a *functional role*, to ensure that each individual in the interpretation has at most one outgoing edge through R. We then add an assertion to $\mathcal{A}_k$:

$$\neg\exists \mathsf{R}.\top(\mathsf{s}_k)$$

forcing s_k to be the last state in the interpretation (that is, s_k has no outgoing edges). We then build the Tbox $\mathcal{T}'_{MD}$ by adding one concept inclusion to $\mathcal{T}_{MD}$. We introduce a new concept $\mathtt{EFnotP}$, and define it as follows:

$$\mathtt{EFnotP} \sqsubseteq \neg\mathtt{P} \sqcup \exists\mathtt{R.EFnotP}$$

This inclusion imitates exactly the fixpoint representation of Clarke and Emerson: we first check whether $\neg\mathsf{P}$ holds in the current state; if it does, then a bug was found and we are done. If not, we try to perform the same check on the following states, that are accessible via the role $\mathtt{R}$. Since $\mathtt{R}$ is a functional, we have that $\exists\mathtt{R.EFnotP}$ is the same as $\forall\mathtt{R.EFnotP}$, and it is propagated to the next state. If $\neg\mathtt{P}$ does not hold in the last state, $\exists\mathtt{R.EFnotP}$ is not applicable anymore, and the search stops after k steps.

Finally, we add another assertion to $\mathcal{A}_k$, stating that $\mathtt{s}_0$, the initial state, belongs also to the new concept $\mathtt{EFnotP}$:

$$\mathtt{EFnotP}(\mathtt{s}_0).$$

Let $\mathcal{A}'_k = \mathcal{A}_k \cup \{\mathtt{EFnotP}(\mathtt{s}_0), \neg\exists\mathtt{R}.\top(\mathtt{s}_k)\}$, and $\mathcal{T}'_{MD}$ as defined above. If $(\mathcal{T}'_{MD}, \mathcal{A}'_k)$ is consistent, it means that $\neg p$ holds on one of the states of distance k or less from the initial state.

For the example $\mathtt{Simple}$, $\varphi = AG(\neg v_2 \vee \neg v_3)$ and $k = 4$, we define $\mathtt{R}$ to be a functional role, and add the following inclusion to create $\mathcal{T}'_{\mathtt{Simple}}$:

$$\mathtt{EFnotP} \sqsubseteq (\mathsf{V}_2 \sqcap \mathsf{V}_3) \sqcup \exists\mathtt{R.EFnotP}$$

We then add two assertions to $\mathcal{A}_4$:

$$\mathcal{A}'_4 = \mathcal{A}_4 \cup \{\mathtt{EFnotP}(\mathtt{s}_0), \neg\exists\mathtt{R}.\top(\mathtt{s}_4)\}$$

Verification is now carried out by asking whether $(\mathcal{T}'_{\mathtt{Simple}}, \mathcal{A}'_4)$ is consistent. Note that as in the examples above, we expect the DL reasoner to give an "unsatisfiable" result ("inconsistent" for the other cases), since the formula φ actually holds in $\mathtt{Simple}$.

3.3 Correctness

We relate the satisfaction of φ in the model M_{MD} to the consistency problems stated in the previous section. Let $MD = (I, [\langle c_1, c'_1\rangle, ..., \langle c_n, c'_n\rangle])$ denote a model description for a model $M_{MD} = (S, I, R, L)$, let M^k_{MD} be the restriction of M_{MD} to distance k from the initial states, and let $\varphi = AG(p)$ be a safety formula. Let $\mathcal{T}_{MD}$, $\mathcal{T}^k_{MD}$, $(\mathcal{T}_{MD}, \mathcal{A}_k)$, $(\mathcal{T}'_{MD}, \mathcal{A}'_k)$ be the ontologies built for MD as defined in items (1),(2),(3),(4) respectively in Section 3.2, and let C^1_φ , C^2_φ, C^3_φ be the concepts representing φ, as described in items (1),(2),(3) of Section 3.2 (note that for method (4) no C_φ is defined). Theorems 17 and 18 state that our methods are correct.

Theorem 17. *If $M^k_{MD} \not\models \varphi$ then all the following hold:*
1. $\mathcal{T}_{MD} \models_{dl} \mathsf{C}^1_\varphi$. 2. $\mathcal{T}^k_{MD} \models_{dl} \mathsf{C}^2_\varphi$. 3.$(\mathcal{T}_{MD}, \mathcal{A}_k) \models_{dl} \mathsf{C}^3_\varphi$ 4. $(\mathcal{T}'_{MD}, \mathcal{A}'_k)$ is consistent.

Theorem 18. *If one of the following holds:*
1. $\mathcal{T}_{MD} \models_{dl} \mathsf{C}^1_\varphi$ 2. $\mathcal{T}^k_{MD} \models_{dl} \mathsf{C}^2_\varphi$ 3.$(\mathcal{T}_{MD}, \mathcal{A}_k) \models_{dl} \mathsf{C}^3_\varphi$ or 4. $(\mathcal{T}'_{MD}, \mathcal{A}'_k)$ is consistent
then $M^k_{MD} \not\models \varphi$.

Proof of Theorem 17. Assume that $M^k_{MD} \not\models \varphi$. Then there exists a path in M^k_{MD}, $w = s_0, ..., s_j$, where $j \leq k$, such that $s_0 \models I, \forall l, 0 < l \leq j, (s_{l-1}, s_l) \in R$, and $s_j \not\models p$. We build a finite interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ for $\mathcal{T}_{MD}$, based on w. The set $\Delta^\mathcal{I}$ includes $j + 1$ elements $\sigma_0, ..., \sigma_j$. Each of the primitive concepts V_i is interpreted as a set $\mathsf{V}^\mathcal{I}_i$, such that $\forall l, 0 \leq l \leq j$, $\sigma_l \in \mathsf{V}^\mathcal{I}_i$ if and only if $s_l \models v_i$. Note that for this interpretation, $\mathcal{M}(\sigma_l) = s_l$ (where $\mathcal{M}$ is as given in definition 12). The interpretation $\mathsf{R}^\mathcal{I}$ of the role R is a set of pairs (σ_l, σ_{l+1}), $0 \leq l < j$. By Corollary 16, we know that all concept inclusions of $\mathcal{T}_{MD}$ hold under this interpretation. Note further that since $s_j \not\models p$ we get by Lemma 13 that $\sigma_j \in \Delta^\mathcal{I} \setminus \mathsf{P}^\mathcal{I}$.

We consider each of the four methods separately.

1. We assign $\sigma_0 \in \mathsf{C}^1_\varphi$ (where σ_0 is the individual corresponding to s_0). We need to show that the inclusion

$$\mathsf{C}^1_\varphi \sqsubseteq \mathsf{S}_0 \sqcap (\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup ...\exists\mathsf{R}.\neg\mathsf{P})...)))$$

holds under the interpretation $\mathcal{I}$. This is easy to see: first, $\sigma_0 \in \mathsf{S}_0$, by the mapping $\mathcal{M}$ and Lemma 13. Second, since $\sigma_j \in \Delta^{\mathcal{I}} \setminus \mathsf{P}^{\mathcal{I}}$ and (σ_{j-1}, σ_j) belong to $\mathsf{R}^{\mathcal{I}}$, we have that $\sigma_{j-1} \in (\exists\mathsf{R}.\neg\mathsf{P})^{\mathcal{I}}$. For similar considerations, since $(\sigma_0, \sigma_1), (\sigma_1, \sigma_2), ..., (\sigma_{j-2}, \sigma_{j-1})$ all belong to $\mathsf{R}^{\mathcal{I}}$, we have that
$\sigma_0 \in (\underbrace{\exists\mathsf{R}.\exists\mathsf{R}....\exists\mathsf{R}}_{j}.\neg\mathsf{P})^{\mathcal{I}}$. We have shown an interpretation where all inclusions of $\mathcal{T}_{MD}$ hold, and C^1_φ is not empty, thus $\mathcal{T}_{MD} \not\models_{dl} \mathsf{C}^1_\varphi$.

2. We interpret each primitive concept S_l as $\{\sigma_l\}$ for $0 \leq l \leq j$. The primitive concepts $\mathsf{S}_{j+1}, ..., \mathsf{S}_k$ are interpreted as $\emptyset$. We assign $\sigma_j \in \mathsf{C}^{2^{\mathcal{I}}}_\varphi$. Since $\mathcal{T}^k_{MD} = \mathcal{T}_{MD} \cup \mathcal{T}_k$, it remains to show that all concept inclusions of $\mathcal{T}_k$ hold under this interpretation, and that $\mathsf{C}^2_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup ... \sqcup \mathsf{S}_k)$.

   - Inclusions from $\mathcal{T}_k$: For $l > j$, $\mathsf{S}^{\mathcal{I}}_l = \emptyset$, and are thus included in any other set. In order for $\mathsf{S}_l \sqsubseteq \exists\mathsf{R}^-.\mathsf{S}_{l-1}$ to hold, for $l \leq j$, we need to show that $\mathsf{S}^{\mathcal{I}}_l \subseteq \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} \text{ s.t. } (y,x) \in \mathsf{R}^{\mathcal{I}} \wedge y \in \mathsf{S}^{\mathcal{I}}_{l-1}\}$. Indeed, $\mathsf{S}^{\mathcal{I}}_l = \{\sigma_l\}, \mathsf{S}^{\mathcal{I}}_{l-1} = \{\sigma_{l-1}\}, (\sigma_{l-1}, \sigma_l) \in \mathsf{R}^{\mathcal{I}}$, and (σ_{l-1}, σ_l) is the only pair $(x,y) \in \mathsf{R}^{\mathcal{I}}$ such that $x \in \mathsf{S}^{\mathcal{I}}_{l-1}$. Thus the inclusion holds.

   - We need to show that $\mathsf{C}^2_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup ... \sqcup \mathsf{S}_k)$ holds under the interpretation $\mathcal{I}$. Since $\mathsf{S}_j = \{\sigma_j\}$, we get that $\sigma_j \in (\mathsf{S}^{\mathcal{I}}_0 \cup \mathsf{S}^{\mathcal{I}}_1 \cup ... \cup \mathsf{S}^{\mathcal{I}}_k)$. Since $s_j \models \neg p$, we get by Lemma 13 that $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$. Thus $\sigma_j \in (\neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup ... \sqcup \mathsf{S}_k))^{\mathcal{I}}$ as needed.

3. We interpret the individuals $\mathsf{s}_0, ..., \mathsf{s}_j$ of $\mathcal{A}_k$ as $\sigma_0, ..., \sigma_j$, that already exist in $\Delta^{\mathcal{I}}$. We assign $\sigma_j \in \mathsf{C}^{3^{\mathcal{I}}}_\varphi$, and σ_j is the only individual in $\mathsf{C}^{3^{\mathcal{I}}}_\varphi$. For $\mathsf{s}_{j+1}, ..., \mathsf{s}_k$, we introduce new individuals, $\sigma_{j+1}, ..., \sigma_k$. Since $s_j \not\models p$, we get by Lemma 13 that $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$.

   By the construction of $\mathcal{I}$, it satisfies both $\mathcal{T}_{MD}$ and $\mathcal{A}_k$. It remains to be shown that $\mathsf{C}^3_\varphi \sqsubseteq \neg\mathsf{P} \sqcap \{\mathsf{s}_0, ..., \mathsf{s}_k\}$ under the interpretation $\mathcal{I}$. Since s_j is interpreted as σ_j and

$\sigma_j \notin \mathsf{P}^\mathcal{I}$, we get that $\sigma_j \in (\Delta^\mathcal{I} \setminus \mathsf{P}^\mathcal{I}) \cap \{\mathtt{s}_0^\mathcal{I}, ..., \mathtt{s}_k^\mathcal{I}\}$ as needed.

4. We interpret the individuals $\mathtt{s}_0, ..., \mathtt{s}_j$ of $\mathcal{A}_k$ as $\sigma_0, ..., \sigma_j$, as above. We map all σ_i, $0 \leq i \leq j$, to belong to $\mathtt{EFnotP}$. The assertions in $\mathcal{A}'_k$ therefore hold: $(\mathtt{s}_i^\mathcal{I}, \mathtt{s}_{i+1}^\mathcal{I}) \in \mathsf{R}^\mathcal{I}$, $\mathtt{s}_0^\mathcal{I} \in \mathtt{EFnotP}^\mathcal{I}$ and $\mathtt{s}_k^\mathcal{I} \in (\Delta^\mathcal{I} \setminus \{e \in \Delta^\mathcal{I} : \exists (e, e') \in \mathsf{R}^\mathcal{I}\})$ since there is no outgoing edge from $\mathtt{s}_k^\mathcal{I}$. It remains to be shown that the inclusion $\mathtt{EFnotP} \sqsubseteq \neg\mathsf{P} \sqcup \exists\mathsf{R}.\mathtt{EFnotP}$ holds under the interpretation $\mathcal{I}$. We know that $\sigma_0, \sigma_1, ..., \sigma_j \in \mathtt{EFnotP}^\mathcal{I}$, and only them. We have to show that these individuals belong also to the right hand side of the inclusion. $\sigma_0, \sigma_1, ..., \sigma_{j-1}$ belong there since $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^\mathcal{I}$, $\sigma_{i+1} \in \mathtt{EFnotP}^\mathcal{I}$ for $0 \leq i < j$. Since $\sigma_j \notin \mathsf{P}^\mathcal{I}$ it also belongs there.

This concludes the proof of Theorem 17. □

For the proof of Theorem 18, we need to show the opposite direction, that is, that if an interpretation can be found for one of the ontologies, it means that $M_{MD}^k \not\models \varphi$. The following lemma, derived trivially from Lemma 13 and Corollary 15, shows that it is enough to show, given an interpretation for $\mathcal{T}_{MD}$, that it includes a “bad” sequence of individuals.

Lemma 19. Let $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ be an interpretation for $\mathcal{T}_{MD}$. Let $\varphi = AG(p)$ be a formula and $\mathsf{P} = \mathcal{D}(p)$. If there exist individuals $\sigma_0, \sigma_1, ..., \sigma_j$ in $\Delta^\mathcal{I}$, such that $\sigma_0 \in \mathsf{S}_0^\mathcal{I}$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^\mathcal{I}$ for $0 \leq i < j$, and $\sigma_j \notin \mathsf{P}^\mathcal{I}$ then $M_{MD}^k \not\models \varphi$.

Proof. Let $s_0 = \mathcal{M}(\sigma_0)$, $s_1 = \mathcal{M}(\sigma_1)$,..., $s_j = \mathcal{M}(\sigma_j)$. Since $\mathcal{I}$ is an interpretation of $\mathcal{T}_{MD}$, we know by Corollary 15, that $s_0, s_1, ..., s_j$ is a path in M_{MD}^k. Since $\sigma_j \notin \mathsf{P}^\mathcal{I}$ we have that $s_j \not\models p$ by Lemma 13. Thus $M_{MD}^k \not\models \varphi$. □

Proof of Theorem 18. In all the cases, we assume that an interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ exists, such that the given ontology holds. We then show, for each case that the axioms and assertions imply the existence of a series of individuals, $\sigma_0, \sigma_1, ..., \sigma_j$ such that $\sigma_0 \in \mathsf{S}_0$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^\mathcal{I}$ for $0 \leq i < j$, and $\sigma_j \notin \mathsf{P}^\mathcal{I}$. This is enough by Lemma 19.

1. We know that the inclusion

   $$\mathsf{C}^1_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \underbrace{(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup ...\exists\mathsf{R}.\neg\mathsf{P})...)))}_{k}$$

   holds under the interpretation $\mathcal{I}$, and that C^1_φ is not empty. Let σ_0 be an individual in ${\mathsf{C}^1_\varphi}^{\mathcal{I}}$. Then $\sigma_0 \in \mathsf{S}_0^{\mathcal{I}}$ and also $\sigma_0 \in \underbrace{(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup ...\exists\mathsf{R}.\neg\mathsf{P})...)))}_{k}^{\mathcal{I}}$. If $\sigma_0 \in \mathsf{P}^{\mathcal{I}}$ then there must exist an individual σ_1 such that $(\sigma_0, \sigma_1) \in \mathsf{R}^{\mathcal{I}}$ and $\sigma_1 \in \underbrace{(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup \exists\mathsf{R}.(\neg\mathsf{P} \sqcup ...\exists\mathsf{R}.\neg\mathsf{P})...)))}_{k-1}^{\mathcal{I}}$. For similar considerations, there must exist a series of individuals, $\sigma_2, \sigma_3, ..., \sigma_j$ such that $j \leq k$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$ for $1 \leq i < j$, and $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$. We have found a sequence $\sigma_0, \sigma_1, ..., \sigma_j$ such that $j \leq k$ $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$, and by Lemma 19, $M^k_{MD} \not\models \varphi$.

2. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation showing that $\mathcal{T}^k_{MD} \models_{dl} \mathsf{C}_\varphi$ is consistent. Since $\mathsf{C}^{\mathcal{I}}_\varphi = (\Delta^{\mathcal{I}} \setminus \mathsf{B}^{\mathcal{I}}) \cap (\mathsf{S}^{\mathcal{I}}_0 \cup \mathsf{S}^{\mathcal{I}}_1 \cup ... \cup \mathsf{S}^{\mathcal{I}}_k)$ is not empty in $\mathcal{I}$, it must be the case that for some j, $0 \leq j \leq k$, $(\Delta^{\mathcal{I}} \setminus \mathsf{P}^{\mathcal{I}}) \cap \mathsf{S}^{\mathcal{I}}_j$ is not empty. Let σ_j be an element in $(\Delta^{\mathcal{I}} \setminus \mathsf{P}^{\mathcal{I}}) \cap \mathsf{S}^{\mathcal{I}}_j$. Then $\sigma_j \in (\Delta^{\mathcal{I}} \setminus \mathsf{B}^{\mathcal{I}})$ and also $\sigma_j \in \mathsf{S}^{\mathcal{I}}_j$.

   Since $\mathcal{T}^k_{MD}$ includes the concept inclusion $\mathsf{S}_j \sqsubseteq \exists\mathsf{R}^-.\mathsf{S}_{j-1}$, and $\mathsf{S}^{\mathcal{I}}_j$ is not empty, we deduce that $\mathsf{S}^{\mathcal{I}}_{j-1}$ is not empty, and that $\exists\sigma_{j-1} \in \mathsf{S}^{\mathcal{I}}_{j-1}$, such that $(\sigma_{j-1}, \sigma_j) \in \mathsf{R}^{\mathcal{I}}$. By similar considerations, there must exist a sequence of elements $\sigma_0, ..., \sigma_j \in \Delta^{\mathcal{I}}$, such that for $0 \leq l < j$, $(\sigma_l, \sigma_{l+1}) \in \mathsf{R}^{\mathcal{I}}$, $\sigma_0 \in \mathsf{S}^{\mathcal{I}}_0$ and $\sigma_0 \notin \mathsf{P}^{\mathcal{I}}$. Thus $M^k_{MD} \not\models \varphi$.

3. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation showing that $(\mathcal{T}_{MD}, \mathcal{A}_k) \models_{dl} \mathsf{C}_\varphi$ is consistent. Since $\mathsf{C}_\varphi \sqsubseteq \neg\mathsf{P} \sqcap \{\mathsf{s}_0, ..., \mathsf{s}_k\}$ is not empty, we know that $\mathsf{s}^{\mathcal{I}}_j \notin \mathsf{P}^{\mathcal{I}}$ for some $0 \leq j \leq k$. Let $\sigma_0, \sigma_1, ..., \sigma_j$ be the interpretation of $\{\mathsf{s}_0, ..., \mathsf{s}_j\}$. By the assertions in $\mathcal{A}_k$ we know that $\sigma_0 \in \mathsf{S}^{\mathcal{I}}_0$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$ for $1 \leq i < j$ and $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$ as needed.

4. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation showing that $(\mathcal{T}'_{MD}, \mathcal{A}'_k)$ is consistent. Let $\sigma_0, \sigma_1, ..., \sigma_k$ be the interpretation of $\{\mathsf{s}_0, ..., \mathsf{s}_k\}$. By the assertions in $\mathcal{A}'_k$ we have

that $\sigma_0 \in \mathsf{S}_0^{\mathcal{I}}$ and also $\sigma_0 \in \mathtt{EFnotP}^{\mathcal{I}}$. Since $\mathtt{EFnotP}^{\mathcal{I}} \subseteq \neg\mathsf{P}^{\mathcal{I}} \cup \exists\mathsf{R}^{\mathcal{I}}.\mathtt{EFnotP}^{\mathcal{I}}$ we know that either $\sigma_0 \notin \mathsf{P}^{\mathcal{I}}$ or there exists an element $\gamma \in \Delta^{\mathcal{I}}$ such that $(\sigma_0, \gamma) \in \mathsf{R}^{\mathcal{I}}$, and $\gamma \in \mathtt{EFnotP}^{\mathcal{I}}$. Since R is a functional role, $\gamma = \sigma_1$. For similar considerations, either $\sigma_1 \notin \mathsf{P}^{\mathcal{I}}$ or $\sigma_2 \in \mathtt{EFnotP}^{\mathcal{I}}$, and the same applies to σ_i, $0 \leq i < k$. Since σ_k has no outgoing edge, we get that one of $\sigma_0, , \sigma_1, ..., \sigma_k$ must belong to $\Delta^{\mathcal{I}} \setminus \mathsf{P}^{\mathcal{I}}$. Let this element be σ_j.

We have found a sequence $\sigma_0, , \sigma_1, ..., \sigma_j$ such that $\sigma_0 \in \mathsf{S}_0^{\mathcal{I}}$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$ for $1 \leq i < j$ and $\sigma_j \notin \mathsf{P}^{\mathcal{I}}$ as before, thus $M_{MD}^k \not\models \varphi$.

This concludes the proof of Theorem 18. □

We now turn to investigate a different encoding to a model description as a TBox, that gives rise to new BMC encodings.

3.4 Alternative Encodings

The methods described in the previous section all used the same encoding for the model, which we denoted $\mathcal{T}_{MD}$. In this section we present an alternative encoding of a model, based on the Ramsey-rule [Ram31]. Translated into DL notation, this rule states the following equivalence:

$$\mathsf{C} \sqsubseteq \forall\mathsf{R}.\mathsf{D} \text{ if and only if } \exists\mathsf{R}^{-}.\mathsf{C} \sqsubseteq \mathsf{D}$$

Note that the role R used in $\mathcal{T}_{MD}$ is actually defined by the restrictions imposed by the concept inclusions. We can therefore replace R by a role $\hat{\mathsf{R}}$, equivalent to the inverse of R. The model description MD will be defined by the following inclusions, denoted $\hat{\mathcal{T}}_{MD}$:

$$\begin{array}{rcl} \exists\hat{\mathsf{R}}.\mathsf{C}_i & \sqsubseteq & \neg\mathsf{V}_i \\ \exists\hat{\mathsf{R}}.(\neg\mathsf{C}_i \sqcap \mathsf{C}_i') & \sqsubseteq & \mathsf{V}_i \end{array}$$

Note that $\hat{\mathcal{T}}_{MD}$ is defined over $\mathcal{ALC}$. Let $\varphi = AG(p)$, and $\mathsf{P} = \mathcal{D}(p)$. The four methods of section 3.2 can now be adapted to use the TBox $\hat{\mathcal{T}}_{MD}$.

1. We define the concept $\hat{\mathsf{C}}^1_\varphi$ as follows.

   $$\hat{\mathsf{C}}^1_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \exists\hat{\mathsf{R}}.(\mathsf{S}_0 \sqcup \exists\hat{\mathsf{R}}.(\mathsf{S}_0 \sqcup ...\exists\hat{\mathsf{R}}.\neg\mathsf{S}_0)...)))$$

   with k nested $\exists\hat{\mathsf{R}}$s. Checking whether $\hat{\mathcal{T}}_{MD} \models_{dl} \hat{\mathsf{C}}^1_\varphi$ performs a search backwards, starting from a buggy state (that belongs to $\neg\mathsf{P}$), and trying to reach an initial state in k or less steps.

2. Recall that for the second method we used concept inclusions to encode a bounded path, that needed the inverse roles, thus using the dialect $\mathcal{ALCI}$. Using $\hat{\mathsf{R}}$, we built $\hat{\mathcal{T}}_k$ where only $\mathcal{ALC}$ is needed:

   $$\mathsf{S}_i \ \sqsubseteq \exists\hat{\mathsf{R}}.\mathsf{S}_{i-1}$$

   for $0 < i \leq k$. The encoding of the formula now stays the same:

   $$\hat{\mathsf{C}}^2_\varphi \sqsubseteq \neg\mathsf{P} \sqcap (\mathsf{S}_0 \sqcup \mathsf{S}_1 \sqcup ... \sqcup \mathsf{S}_k)$$

   As we see, for this method the alternative encoding allows us to avoid the use of inverse roles.

3. As in the original encoding, for a bound k we introduce $k+1$ individuals, s_0,s_1 ,...,s_k. However, we encode the path and formula differently. We assert $\neg\mathsf{P}(\mathsf{s}_k)$ to say that s_k is a buggy state. For $0 < i \leq k$, the assertions $\hat{\mathsf{R}}(\mathsf{s}_i,\mathsf{s}_{i-1})$ make s_i a state of distance $k-i$ from the buggy state. We call this set of assertions $\hat{\mathcal{A}}_k$. We now want to use nominals to say that the initial state is reachable in k steps, going backwards from the buggy state s_k: $\hat{\mathsf{C}}^3_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \{\mathsf{s}_0, ..., \mathsf{s}_k\}$. Verification for this method is done by asking the query: $(\hat{\mathcal{T}}_{MD}, \hat{\mathcal{A}}_k) \models_{dl} \hat{\mathsf{C}}^3_\varphi$.

4. As in section 3.2, for the fourth method we define $\hat{\mathsf{R}}$ to be a *functional role*, ensuring that each individual in the interpretation has at most one outgoing edge through $\hat{\mathsf{R}}$.

We use the same TBox and ABox as in item 3 above, but add to them. We add an assertion to $\hat{\mathcal{A}}_k$:

$$\neg\exists\hat{\mathsf{R}}.\top(\mathsf{s}_0)$$

forcing s_0 to be the last state in the interpretation (that is, s_0 has no outgoing edges). We then build the Tbox $\hat{\mathcal{T}}'_{MD}$ by adding one concept inclusion to $\hat{\mathcal{T}}_{MD}$. We introduce a new concept EFs_0, and define it as follows:

$$\mathsf{EFs}_0 \sqsubseteq \mathsf{S}_0 \sqcup \exists\hat{\mathsf{R}}.\mathsf{EFs}_0$$

We now add another assertion to $\hat{\mathcal{A}}_k$, stating that that s_k, the buggy state, belongs also to the new concept EFs_0:

$$\mathsf{EFs}_0(\mathsf{s}_k).$$

Like before, we first check whether S_0 holds in the current state - that is, if the buggy state from which we start is already an initial state; if it is, then a bug was found in an initial state (S_0) and we are done. If not, we try to perform the same check on the following states, that are accessible via the role $\hat{\mathsf{R}}$. Since $\hat{\mathsf{R}}$ is a functional, we have that $\exists\hat{\mathsf{R}}.\mathsf{EFs}_0$ is the same as $\forall\hat{\mathsf{R}}.\mathsf{EFs}_0$, and it is propagated to the next state. If S_0 does not hold in the last state, $\exists\hat{\mathsf{R}}.\mathsf{EFs}_0$ is not applicable anymore, and the search stops after k steps.

Let $\hat{\mathcal{A}}'_k = \hat{\mathcal{A}}_k \cup \{\mathsf{EFs}_0(\mathsf{s}_k), \neg\exists\hat{\mathsf{R}}.\top(\mathsf{s}_0)\}$, and $\hat{\mathcal{T}}'_{MD}$ as defined above. If $(\hat{\mathcal{T}}'_{MD}, \hat{\mathcal{A}}'_k)$ is consistent, it means that I can be reached in k or less steps from a buggy state.

3.5 Experimental Results

We conducted our experiments using the FaCT++ description logic reasoner [TH06]. While other DL reasoners exist, such as Pellet [SPG$^+$07] and Racer [HM01], we found FaCT++ to be more accessible, being a free, open-source and well documented tool. A benchmark comparison reported in [GHT06] suggests that FaCT++ is also one of the leading tools in performance. The tool accepts three input languages. The DIG interface language [BMC03] (defined by the Description Logic Implementation Group), the Owl-DL language and a simple lisp-like input. We worked with the lisp-like input language. We have experimented with the eight methods described in the previous sections, to compare their performances. We used a model derived from the NuSMV example "dme1-16", taken from [NuS], parameterized to enable different model sizes, and ran our experiments on an Intel XEON CPU of 1.8GHz, with a 4GB RAM and Cache size of 512 KB. In table 3.3 below we present run-time results for a model consisting of 85 state variables.

Size	Bound	SAT	1	1*	2	2*	3	3*	4	4*
85	5	0.02	0.02	1.13	1.77	5.3	0.05	125	0.18	–
85	6	0.03	0.03	1.38	287	48	0.08	–	0.25	–
85	7	0.04	0.03	4.3	–	104	0.18	–	0.34	–
85	8	0.05	0.04	59.31	–	604	0.6	–	0.44	–
85	9	0.05	0.05	–	–	–	2.61	–	0.55	–
85	10	0.05	0.05	–	–	–	34	–	0.68	–
85	15	0.08	0.11	–	–	–	–	–	5.85	–
85	17	0.09	0.12	–	–	–	–	–	10	–
85	20	0.12	59.72	–	–	–	–	–	–	–
85	30	0.22	–	–	–	–	–	–	–	–
85	40	0.30	–	–	–	–	–	–	–	–

Figure 3.3: Run times for BMC, small model

Time is given in seconds, and a result of '–' indicates that the run did not terminate within 1200 seconds. Column 2 gives the *bound* of the BMC run. Column 3 presents the results of the same model and formula running using a SAT solver. For this, we used the BMC mode of Cadence-SMV [McM], that invoked zChaff [MMZ$^+$01] as a SAT solver. The columns titled with a number, each refer to the method with a similar number, described in Section 3.2. The columns titled with a starred number (1*, 2* etc), refer to the methods described in Section 3.4 (note that the starred methods are the Ramsey-rule versions of the non-starred ones). Table 3.4 presents results for a larger model, consisting of 425 state variables. The verified formula was an invariant formula that failed on cycle 42. As evident from the table, none of our methods, for the large model as well as for the small one, were capable of searching more than 20 cycles. Thus in all cases the result from the DL reasoner was “unsatisfiable” (meaning that no error was found up to the given bound).

Size	Bound	SAT	1	1*	2	2*	3	3*	4	4*
425	5	0.71	0.17	34	32	58	29.5	–	32	–
425	6	0.72	0.19	42	–	134	39.8	–	43.9	–
425	7	0.78	0.21	87	–	279	51.5	–	55.9	–
425	8	0.80	0.24	600	–	1275	63.8	–	69.4	–
425	9	0.88	0.28	–	–	–	95.9	–	83.8	–
425	10	0.93	0.34	–	–	–	703.6	–	99	–
425	15	1.04	0.79	–	–	–	–	–	423	–
425	17	1.50	1.15	–	–	–	–	–	630	–
425	20	2.34	314	–	–	–	–	–	–	–
425	30	2.76	–	–	–	–	–	–	–	–
425	40	4.56	–	–	–	–	–	–	–	–

Figure 3.4: Run times for BMC, large model

The satisfiability solver we used, zChaff, outperformed all of our encodings as can be seen in the tables. While some of these methods, especially number 1, performed well for lower bounds, they all seem to be very sensitive to the depth of the search, and explode once the bound passes 20.

In all methods, except for number 2, the Ramsey-rule version performed much worse than the original one. In the context of model checking this is not surprising, since *backwards* traversal of the transition relation is known to be more difficult than forward traversal. The exception of Method 2 can be explained by the fact that the original method in this case required backward traversal as well.

It is interesting to note the significant differences between the various DL encodings themselves. While the gap between forward and backward encodings is expected due to the nature of the model checking problem, the difference in performance between forward encodings seems to be related to internal DL algorithms. In Chapter 6 we suggest possible future directions to investigate this phenomena.

3.6 Discussion

It is interesting to compare a typical DL application to the model checking application presented above. The GALEN ontology [RN94] for example, contains close to 25,000 concepts and around 500 concept inclusions, yet queries are resolved in a matter of minutes. In contrast, the examples we use contain only a few hundred concepts and a similar number of concept inclusions, but for big enough k the run does not terminate. The difference, it seems, stems from the different "shape" of the problems. A typical DL application is usually "shallow" in the sense that relations through roles are applied only once, while the model checking application involves concepts that are defined using repeated relations through one role.

The complexity of consistency checking with respect to a general terminology is known to be EXP-time complete in all dialects used in this chapter [Sch91, DM00, Tob01]. The complexity of model checking is known to be PSPACE-complete [SC85, CES86]. At first sight then, it may look as if we try to solve a simple task with a complex algorithm. This is not the case however, for two main reasons.

First, we note that the complexity result for model checking is measured with respect to the full Kripke structure (the *model*) rather than the *model description*, while consistency checking complexity is given in terms of the terminology size. For our encoding, the size of the terminology is linear in the size of the model description. The Kripke structure is in many cases exponentially larger than the description of it, and the main idea of symbolic model checking is to avoid, when possible, the need to build the full Kripke structure.

Second, it is important to note that while consistency checking is EXP-time complete in general, the complexity is in NP for all the bounded model checking methods presented above. To see this, let $n = |MD|$ and k the bound. The size of the model description is $O(n + k)$. If an interpretation is given, (a ‘witness’ for the consistency query), it is of size $O(n \times k)$ (k nodes, each of them of size $O(n)$, assigning values to the n primitive concepts). Verifying that the given interpretation indeed satisfies all concept inclusions will again amount to $O(n \times k)$ calculations. We conclude that, as known, complexity is not a good measurement to assess model checking methods.

Chapter 4

Liveness and Fairness Modeling using Description Logic

In the previous chapter we explored several methods to encode a bounded model checking problem of invariance formulas in DL. We now turn to consider *liveness* formulas, given as $\mathsf{AF}(p)$ with p being a Boolean expression. Such formulas state that p must hold at least once on every path. For a model description MD, we use the same encoding as described in Section 3.1, and here as well, we encode in our terminology a description of a *buggy* path, and use the DL reasoner to find a counterexample for us. In the liveness case, a buggy path would be one on which p never holds. We thus look for a representation of the formula $\mathsf{EG}(\neg p)$. As discussed in Section 2.1.2, liveness formulas are rarely verified without some fairness constraints. In fact, for the main model checking method of LTL, a liveness formula is translated into a Büchi automaton [Var96], and model checking is reduced to finding a "fair" loop.

In Section 4.1 below we give an encoding for $\mathsf{AF}(p)$ formulas over $\mathcal{ALC}$, and prove its correctness. Section 4.2 deals with fairness encoding. We show that fairness cannot be expressed in $\mathcal{ALC}$ or other dialects discussed in this document, and demonstrate that for our needs, fairness can be implemented on top of a tableau reasoning algorithm. In

Section 4.3 we present experimental results, and Section 4.4 concludes this chapter with a discussion.

4.1 The Encoding

As before, we use the fix-point representation of CTL formulas, as defined by Clarke and Emerson in [CE81] (see Section 2.1.3). Thus, $\mathsf{EG}(\neg p)$ is represented as follows:

$$\mathsf{EG}(\neg p) = \neg p \wedge \mathsf{EX}(\mathsf{EG}(\neg p)) \tag{4.1}$$

We use this equation for our translation into DL. Let $MD = (I, [\langle c_1, c'_1\rangle, ..., \langle c_n, c'_n\rangle])$ be a model description for the model $M_{MD} = (S, I, R, L)$ over $V = \{v_1, ..., v_n\}$, and let $\mathcal{T}_{MD}$ be the terminology built for it as described in Section 3.1. Let $\varphi = \mathsf{AF}(p)$ be the formula to be verified, with p being a Boolean expression over the variables $v_1, ..., v_n$, and let $\mathsf{P} = \mathcal{D}(p)$.

We introduce a new concept called $\mathtt{EGnotP}$, and add the following concept inclusion to $\mathcal{T}_{MD}$:

$$\mathtt{EGnotP} \sqsubseteq \neg\mathsf{P} \sqcap \exists \mathsf{R}.\mathtt{EGnotP} \tag{4.2}$$

Note that the expression $\exists\mathsf{R}.\mathsf{C}$ can be seen as taking one step through R, and thus corresponds, in a sense, to the CTL expression $\mathsf{EX}(\mathsf{C})$.

Let $\mathcal{T}^F_{MD}$ be the terminology we get by adding Equation (4.2) to $\mathcal{T}_{MD}$. We define the concept $\mathsf{C}_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \mathtt{EGnotP}$. In order to verify φ, we now check whether C_φ is consistent with respect to our terminology: $\mathcal{T}^F_{MD} \models_{dl} \mathsf{C}_\varphi$?
A positive answer from the DL reasoning tool will be accompanied by an interpretation for $\mathcal{T}^F_{MD}$ in which C_φ is not empty. This interpretation can serve an a witness to $\mathsf{EG}(\neg p)$, or as a counterexample to $\mathsf{AF}(p)$. The following proposition states our result formally.

Proposition 20. *$M_{MD} \not\models \varphi$ if and only if $\mathcal{T}^F_{MD} \models_{dl} \mathsf{C}_\varphi$.*

Proof. ($\Longrightarrow$). Assume that $M_{MD} \not\models \varphi$. Since M_{MD} is a finite kripke structure, this means there exists a loop, that is, a sequence of states s_0, s_1, ... ,s_m such that $s_0 \models I$, $s_i \not\models p$ for $0 \leq i \leq m$, $(s_i, s_{i+1}) \in R$ for $0 \leq i < m$, and $s_m = s_j$ for some $0 \leq j < m$. We use this sequence to build an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for $\mathcal{T}^F_{MD}$. We define m individuals $\sigma_0, \sigma_1, ..., \sigma_{m-1}$ in $\Delta^{\mathcal{I}}$, that correspond to s_0, s_1, ... ,s_{m-1}. We map $\sigma_i \in \mathtt{EGnotP}^{\mathcal{I}}$ for $0 \leq i < m$. We then map each σ_i to the primitive concepts $\mathsf{V}_k^{\mathcal{I}}$ according to s_i as expected: $\sigma_i \in \mathsf{V}_k^{\mathcal{I}}$ if and only if $s_i \models v_k$. Note that since $s_0 \models I$ we get by Lemma 13 that $\sigma_0 \in \mathsf{S}_0^{\mathcal{I}}$, and also $\sigma_i \notin \mathsf{P}^{\mathcal{I}}$ for $0 \leq i < m$, since $s_i \not\models p$. We define $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$ for $0 \leq i < m-1$, and also $(\sigma_{m-1}, \sigma_j) \in \mathsf{R}^{\mathcal{I}}$. Finally, we map $\sigma_0 \in \mathtt{EGnotP}^{\mathcal{I}}$. We need to show that all inclusion in $\mathcal{T}^F_{MD}$ hold under this interpretation. By Corollary 16, we know that all inclusions from $\mathcal{T}_{MD}$ hold.

- For the inclusion $\mathtt{EGnotP} \sqsubseteq \neg\mathsf{P} \sqcap \exists\mathsf{R}.\mathtt{EGnotP}$, note that by the construction of $\mathcal{I}$, all individuals σ_i belong to $\mathtt{EGnotP}^{\mathcal{I}}$. We know also that $\sigma_i \notin \mathsf{P}^{\mathcal{I}}$. Since each individual has an outgoing edge that is also in $\mathtt{EGnotP}^{\mathcal{I}}$ the inclusion holds.

- The inclusion $\mathsf{C}_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \mathtt{EGnotP}$ holds, since σ_0, the only individual in C_φ, belongs also to $\mathsf{S}_0^{\mathcal{I}} \cap \mathtt{EGnotP}^{\mathcal{I}}$.

($\Longleftarrow$). Assume that $\mathcal{T}^F_{MD} \models_{dl} \mathsf{C}_\varphi$. Then there exists an interpretation for $\mathcal{T}^F_{MD}$, such that $\mathsf{C}_\varphi^{\mathcal{I}}$ is not empty. Since $\mathcal{ALC}$ enjoys the *finite model property* [BCM$^+$03], there must exist a finite interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for $\mathcal{T}^F_{MD}$ such that $\mathsf{C}_\varphi^{\mathcal{I}}$ is not empty. Thus there exists an individual $\sigma_0 \in \mathsf{S}_0^{\mathcal{I}} \cap \mathtt{EGnotP}^{\mathcal{I}}$.

Since $\sigma_0 \in \mathtt{EGnotP}^{\mathcal{I}}$ and $\mathtt{EGnotP}^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}} \setminus \mathsf{P}^{\mathcal{I}}) \cap \{e \in \Delta^{\mathcal{I}} : \exists(e, e') \in \mathsf{R}^{\mathcal{I}}$ s.t. $e' \in \mathtt{EGnotP}^{\mathcal{I}}\}$ we know that $\sigma_0 \notin \mathsf{P}^{\mathcal{I}}$, and there must exist $\sigma_1 \in \mathtt{EGnotP}^{\mathcal{I}}$ such that $(\sigma_0, \sigma_1) \in \mathsf{R}^{\mathcal{I}}$. For similar considerations, there exists a sequence of individuals, $\sigma_0, \sigma_1, \sigma_2, ...$, such that $\sigma_i \in \mathtt{EGnotP}^{\mathcal{I}}$, $\sigma_i \notin \mathsf{P}^{\mathcal{I}}$, and $(\sigma_i, \sigma_{i+1}) \in \mathsf{R}^{\mathcal{I}}$ for all i. Since $\mathcal{I}$ is finite there must exist m, j such that $\sigma_m = \sigma_j$. We show that $M_{MD} \not\models \varphi$ by presenting an infinite sequence of states (a loop) in M_{MD} that do not satisfy p. We map each σ_i to a state s_i as usual: $\sigma_i \in \mathsf{V}_k^{\mathcal{I}}$ if and only if $s_i \models v_k$. By Lemma 13, $s_i \not\models p$, since $\sigma_i \notin \mathsf{P}^{\mathcal{I}}$. Also, by Corollary 14, $\forall 0 \leq i \leq m$, $(s_i, s_{i+1}) \in R$ and $(s_m, s_j) \in R$.

□

4.1.1 Other Attempts

We consider two related attempts that can be tempting to be tried.

- It is interesting to try the Ramsey-rule method for encoding a model description (Section 3.4). Recall that using this method, we encode a role $\hat{\mathsf{R}}$ that goes backwards, equivalent to the inverse of the role R. For the BMC methods of Section 3.4, we used this method to go backwards from a buggy state, trying to reach an initial state within the given bound. In our case however, there is no buggy state, as a failure can be demonstrated only by a buggy loop. Suppose that equation 4.2 is changed a bit, to use $\hat{\mathsf{R}}$:

$$\widehat{\mathtt{EGnotP}} \sqsubseteq \neg\mathsf{P} \sqcap \exists\hat{\mathsf{R}}.\widehat{\mathtt{EGnotP}}$$

  Figure 4.1 demonstrates the difference between the concepts $\mathtt{EGnotP}$ and $\widehat{\mathtt{EGnotP}}$. The left hand side of the figure describe $\mathtt{EGnotP}$: individuals in which $\neg p$ holds, and in all the other individuals reachable through R , $\neg p$ holds also. The right hand side describes $\widehat{\mathtt{EGnotP}}$: individuals that when going backwards through R (that is, forward through $\hat{\mathsf{R}}$) can visit only individuals with $\neg p$. For model checking $\mathsf{AF}(p)$,

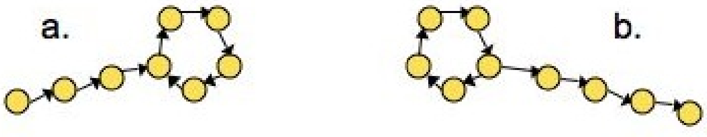

Figure 4.1: Forward vs. backward role modeling

we check the consistency of $\mathsf{S}_0 \sqcap \mathtt{EGnotP}$, that is check whether one of the initial states belong to $\mathtt{EGnotP}$. If that is the case it means that a loop with $\neg p$ is reachable from the initial state. We note that $\mathsf{S}_0 \sqcap \widehat{\mathtt{EGnotP}}$ would not, in general, give us the correct answer. Only if the individual in the intersection is part of the loop. Otherwise the answer would be wrong. Thus we can get a "false negative" if $\mathsf{S}_0 \sqcap \widehat{\mathtt{EGnotP}}$ is found to be inconsistent, although a state might exist in the left "tail", that is not included in $\widehat{\mathtt{EGnotP}}$. We can also get a "false positive", if the intersection is not empty because it includes an individual from the right tail, which actually does not lead to a legal loop.

- It is tempting to try and use the same reasoning to verify a formula $\psi = \mathsf{AG}(p)$: instead of the concept inclusion in (4.2), add the concept $\mathtt{AGp}$ and the following concept inclusion:

  $$\mathsf{AGp} \sqsubseteq \mathsf{P} \sqcap \forall \mathsf{R}.\mathsf{AGp}. \tag{4.3}$$

  Define $\mathsf{C}_\psi \sqsubseteq \mathsf{S}_0 \sqcap \mathtt{AGp}$. Let $\mathcal{T}_{MD}^{F\prime}$ be the terminology we get by replacing Equation (4.2) with Equation (4.3) in $\mathcal{T}_{MD}^{F}$. Note that checking $\mathcal{T}_{MD}^{F\prime} \models_{dl} \mathsf{C}_\psi$ does not give us what we want. To see this, recall that $\mathcal{T}_{MD}^{F\prime} \models_{dl} \mathsf{C}_\psi$ asks whether *there exists* an interpretation $\mathcal{I}$, that satisfies all concept inclusions in $\mathcal{T}_{MD}^{F\prime}$, and for which C_ψ is not empty. Such interpretation does not necessarily include all possible transitions in the given model M_{MD}. In fact, an interpretation that satisfies inclusion (4.2) would be enough for inclusion (4.3) as well. Thus $\mathcal{T}_{MD}^{F\prime} \models_{dl} \mathsf{C}_\psi$ verifies $\mathsf{EG}(p)$ and not $\mathsf{AG}(p)$.

The encoding we have presented so far does not account for fairness constraints. This is the topic of the next section.

4.2 Modeling Fairness

In Section 2.1.2 we explained that fairness constraints are an important component in model checking of liveness formulas. The constraint `fairness p` asserts that the liveness formula should be verified only on infinite paths on which `p` holds infinitely often. In order to encode fairness, we need to express *eventuality* - that some event can be reached within a finite number of steps. Providing a general encoding for this (as done for a bounded path in Section 3.2), is not possible in the DL dialects that were considered so far (subsets of $\mathcal{ALCIOF}$). To see this, recall that $\mathcal{ALCIOF}$ corresponds to a fragment of first order logic. Eventuality, needed for fairness, is equivalent to *reachability*, which cannot be expressed in first order logic (c.f. [MR04]).

More expressive dialects have been defined in the DL literature that can deal with our problem. In [GL97], De Giacomo and Lenzerini propose the embedding of a μ calculus operator to DL, introducing the dialect $\mu\mathcal{ALCQ}$. In a joint work with Calvanese [CGL99] they later expand this dialect to support inverse roles as well as roles with arbitrary arity, introducing the dialect $\mathcal{DLR}_\mu$. While they provide an algorithm to decide consistency problems written in these dialects (using tree automata), those algorithms were never implemented in any existing DL reasoning tool [TW08].

We observe that while fairness cannot be expressed in $\mathcal{ALCIOF}$, it can be easily implemented. In order to find an interpretation for our encoding of liveness properties over $\mathcal{ALC}$ (Section 4.1), the mechanism of *blocking* [HS99] comes into play (see Section 2.2.3): an interpretation is found when a new node y in the expansion is a subset of a previous node x (in this case we say that y is blocked by x)[1]. Such an interpretation demonstrates a loop, or, translated into the model checking world, an infinite sequence of states. In order to support, for example, the fairness constraint `fairness p` we need to make sure that at least one of the nodes in the loop has (or can possibly have) p in it. That is, we allow x to block y only if p appears in some node on the path from x to y .

[1] Other blocking conditions may apply for more expressive DL dialects.

Below we show how fairness can be achieved in tableau based DL reasoning. We assume that the formula to be verified is of the form $\mathsf{AF}(\mathbf{false}\ )$ (or $\mathsf{EG}(\mathbf{true}\ )$), meaning that model checking is reduced to finding a fair cycle.

4.2.1 Realizing Fairness in Tableaux Reasoning

We propose a modification to the tableaux procedure to support fairness. Our procedure is both terminating and sound: if a fair cycle is found, it is a correct one. However, the procedure is not complete, that is, there are cases were a fair cycle exits but our procedure fails to find it. We show that by an iterative application of the algorithm, completeness can also be achieved. In the remainder of this section we discuss the theoretical and implementation considerations for realizing fairness in DL reasoning.

As discussed before, fairness constraints in model checking are Boolean expressions that should be satisfied at least once in a given loop in order for it to be a legal counterexample. The algorithm we present deals with one fairness constraint; if more than one constraint should be considered, a repeated application of the algorithm would be required.

In tableaux reasoning, an interpretation is represented by a completion tree (see Section 2.2.3), and cycles are represented by blocked nodes. If a node y is blocked by a node x_0 then there exists a path of nodes $x_0, x_1 \ldots, x_n, y$ in the completion tree, such that each edge $\langle x_0, x_1 \rangle, \ldots, \langle x_n, y \rangle$ is labeled with R. (note that in the terminologies that we deal with there exists only one role R). Such a blocking path represents a loop.

In order to implement reasoning with fairness, we need to reject those completion trees that correspond to unfair computations. Let FC be a fairness constraint. A completion tree $\mathbf{G}$ is *unfair* with respect to FC if there exists a loop $x_0, \ldots, x_n, y$ such that $FC \notin \mathcal{L}(x_i)$ for all $0 \leq i \leq n$. Otherwise, we say that $\mathbf{G}$ is *fair* with respect to FC.

Modifying Tableaux to Support Fairness

Our approach to implementing fairness is to build a complete and clash-free completion tree and, if it is unfair, to attempt to make it fair by adding the fairness constraint to the label of some node involved in a cycle. To accomplish this, the tableaux algorithm is extended with the new rule illustrated in Figure 4.2. We set the new rule to have a lower priority than all existing rules.

$fairness$-rule: if 1. y is a node blocked by x_0 (let $(x_0, \ldots, x_n, y)$ be the cycle)
2. FC is a fairness constraint such that for every $i, 0 \leq i \leq n, FC \notin \mathcal{L}(x_i)$
then set $\mathcal{L}(x_i) = \mathcal{L}(x_i) \cup \{FC\}$ for some i : $0 \leq i \leq n$

Figure 4.2: Expansion rule for fairness

The tableaux algorithm is enhanced in such a way that a node is not considered *blocked* until a fairness constraint appears in the label of one of its nodes. Note that after applying the $fairness$-rule the completion tree must be updated: a clash may now exist that did not exist before, and labels of nodes may need to change.

Theorem 21. *The tableaux algorithm with $fairness$-rule terminates and is sound (if a complete clash-free fair completion tree for C is found then C is consistent).*

Proof. The algorithm is clearly sound: if a cycle is found where FC holds on one of the nodes then a fair cycle exists. To prove termination, we assume, without loss of generality, that the completion tree is a single path. After a first application of $fairness$-rule to a given blocking loop, there are three cases to consider:

1. It is possible to compute a complete clash-free fair completion tree without a need for a second application of $fairness$-rule.
2. A clash occurs before a second application of the $fairness$-rule, or

3. A subsequent application of $fairness$-rule is required.

Both cases (1) and (2) lead to termination. Case (3) implies that the addition of FC to a label inside the cycle breaks the blocking condition and leads to a new cycle. The algorithm therefore proceeds by adding FC inside the next loop. Again, there are three possible outcomes, with two resulting in termination. In the worst case, there is a sequence of case (3) for which adding FC forces unblocking the last node and moving the blocking loop forward. However, after a finite number of occurrences of case (3), there must eventually be two nodes labeled by FC for which the labels are the same (since the TBox is finite). One of these nodes will then block the other, and the fair loop must then be established. □

Note that while our algorithm is sound, completeness is not guaranteed. That is, there can be cases where a concept is satisfiable with respect to a fairness constraint FC, but the tableaux procedure fails to find an interpretation. To see how this happens, let us consider the example shown in Figure 4.3, that presents two completion trees for the concept C

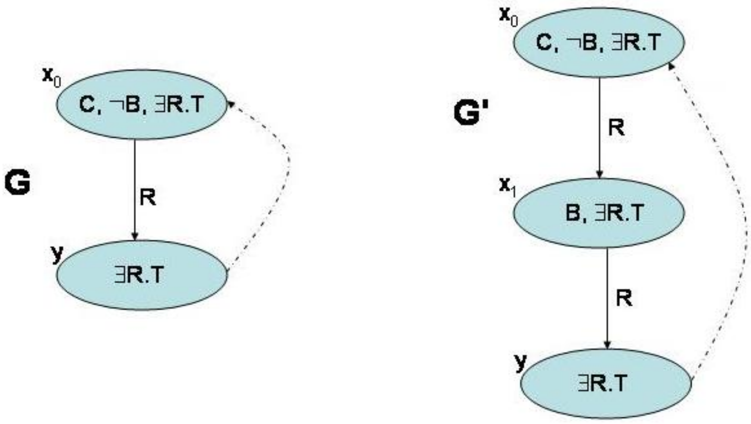

Figure 4.3: Two completion trees for $\mathcal{T}_1$

with respect to the TBox $\mathcal{T}_1 = \{\mathsf{C} \sqsubseteq \neg\mathsf{B}, \top \sqsubseteq \exists\mathsf{R}.\top\}$.

Obviously $\mathcal{T}_1 \models_{dl} \mathsf{C}$, since there exists a complete and clash-free completion tree $\mathbf{G}$ for it as illustrated in the left-hand side of Figure 4.3. On $\mathbf{G}$, the node y is blocked by x_0. If we add the fairness constraint $FC = \mathsf{B}$, the $fairness$-rule will try to add B to the only possible node, x_0, resulting in a clash. Our tableau algorithm will therefore return an “unsatisfiable” result.

A clash-free fair completion tree for C does exist, however, as shown by $\mathbf{G}'$ in Figure 4.3. In order to find it though, we need to allow for a longer blocking cycle. In the definition below, we introduce the notion of n-blocking.

Definition 22. Let n be a non-negative integer. A node y is *n-blocked* by the node x_0 with blocking loop $x_0, \ldots, x_m, y$ if y is blocked by x_0 and $n \leq m$, that is, there are at least n nodes in the blocking loop.

Figure 4.3 gives examples for 0-blocking according to Definition 22 (the completion tree $\mathbf{G}$), as well as 1-blocking ($\mathbf{G}'$).

Note that replacing the original tableaux blocking with n-blocking in the (fair) tableaux algorithm would clearly preserve both termination and soundness. Based on n-blocking, we can now propose a tableaux algorithm that would guarantee completeness, for loop lengths less than or equal to n.

Algorithm 23. Given a concept C, a TBox $\mathcal{T}$, a fairness condition FC and a non-negative integer n, check the unfair consistency of C with respect to $\mathcal{T}$ using the regular tableaux procedure. If it is unsatisfiable, return “unsatisfiable”. Then, for $0 \leq k \leq n$, run the fair tableaux algorithm with k-blocking. Return “satisfiable” if a fair loop is found for some k; otherwise return “unsatisfiable”. □

Theorem 24. *Algorithm 23 is a sound and complete decision procedure for the fair satisfaction of C with respect to $\mathcal{T}$ and FC, with loops up to length n.*

Proof. Termination and soundness are a simple consequence of Theorem 21. Completeness follows from the fact that no fair blocking loops for any possible length not exceeding

n were found. $\square$

Note that for a Kripke structure M and a liveness specification φ there exists n such that if $M \not\models \varphi$, then M contains a fair cycle with length not exceeding n (since M is finite). Thus, it is possible to build the TBox $\mathcal{T}$ using the technique from Section 4.1 and run the procedure suggested in Algorithm 23 to determine if a fair cycle exists.

4.3 Experimental Evaluation

The modified tableaux reasoning procedure described above was implemented by Dmitry Tsarkov on top of FaCT++ [TH06], a state-of-the-art description logic reasoner. In order to run real examples, we wrote a translator from the AIGER [Bie07] format, that builds a terminology as described in Section 4.1. Liveness formulas were translated in the AIGER models into Büchi automata (see section 2.1.2), and the fairness constraints were passed to FaCT++ using a new construct in the interface language.

The models we acquired were originally written in the VIS [BHSV$^+$96] input language, and were translated into AIGER using different tools. We present results running three sets of benchmarks with fairness constraints. The "amba" benchmark encodes an Advanced High Performance Bus. The "vsa" benchmarks encode a simple architecture for a microprocessor. In each of the vsa benchmarks, the number indicates the datawidth of the microprocessor. The "Vending" example is part of the VIS distribution.

Figure 4.4 summarizes our results. Times reported are in seconds and a time of '–' indicates that the run did not finish in the allotted time of 1 hour.

It is evident from Figure 4.4 that our approach is efficacious in certain scenarios. For the "amba" benchmark, our system could not finish in the given time, while VIS was easily able to handle it in a fraction of a second. However, the "vsaR" benchmarks proved simple for our reasoner while VIS was unable to finish in the given time.

Benchmark	Result	Size (vars)	FaCT++	VIS
vsaR - 6	Fail	170	10.8s	–
vsaR - 8	Fail	204	14.4s	–
vending	Pass	64	–	1.1s
amba2 - G3	Pass	63	–	0.7s
amba3 - G3	Pass	77	–	17.7s

Figure 4.4: Run times for the fairness verification tasks

4.4 Discussion

Our method can be seen as “bounded” fair cycle detection, where at each iteration we look for loops of length not shorter than a given bound n. In this aspect it resembles bounded model checking of liveness properties using satisfiability solving. The search algorithm is different however. While our method dynamically searched for a fair loop, in the SAT case a CNF formula is statically created prior to the SAT run, encoding all possible loops up to a given length.

Our method works better when a fair cycle does exist in the model. Note that when a fair cycle does not exist, the bound n on the length of the longest loop is an inadequate over-approximation of the real limit. Our algorithm would thus continue iterating longer than needed before it would reach the conclusion that a fair cycle does not exist. This can be also observed in the results presented in the previous section: when a formula is satisfied (no fair cycle exists), our method performs much worse than VIS, but becomes useful when a bad cycle does exist in the model.

Chapter 5

Counterexample Explanation

An important feature of model checking tools is their ability to provide, when the specification does not hold in a model, a *counterexample* [CGMZ95]: a trace that demonstrates the failure of the specification in the model. This allows the user to analyze the failure, understand its source(s), and fix the specification or model accordingly. In many cases, however, the task of understanding the counterexample is challenging, and may require a significant manual effort.

An explanation of a counterexample deals with the question: *what values on the trace cause it to falsify the specification?* Thus, we face the problem of *causality*. The philosophy literature, going back to Hume [Hum39], has long been struggling with the problem of what it means for one event to cause another. We relate the formal definition of causality of Halpern and Pearl [HP01] to explanations of counterexamples. The definition of causality used in [HP01], like other definitions of causality in the philosophy literature, is based on *counterfactual dependence*. Event A is said to be a *cause* of event B if, had A not happened (this is the counterfactual condition, since A did in fact happen) then B would not have happened. Unfortunately, this definition does not capture all the subtleties involved with causality. The following story, presented by Hall in [Hal02], demonstrates some of the difficulties in this definition. Suppose that Suzy and Billy both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been preempted by Suzy's throw. Thus, according to the counterfactual condition, Suzy's throw is not a cause for shattering the bottle (because if Suzy wouldn't have thrown her rock, the bottle would have been shattered by Billy's throw). Halpern and Pearl deal with this subtlety by, roughly speaking, taking A to be a cause of B if B counterfactually depends on A under some contingency. For example, Suzy's throw is a cause of the bottle shattering because the bottle shattering counterfactually depends on Suzy's throw, under the contingency that Billy doesn't throw.

We adapt the causality definition of Halpern and Pearl from [HP01] to the analysis of a counterexample trace π with respect to a temporal logic formula φ. We view a trace as a matrix of values, where an entry (j, i) corresponds to the value of variable i at time j. We look for those entries in the matrix that are causes for the first failure of φ on π, according to the definition in [HP01]. To demonstrate our approach, let us consider the following example.

Example: A transaction begins when `START` is asserted, and ends when `END` is asserted. Some unbounded number of time units later, the signal `STATUS_VALID` is asserted. Our specification requires that a new transaction must not begin before the `STATUS_VALID` of the previous transaction has arrived. This specification can be written in LTL as

$$\mathbf{G}(\texttt{START} \rightarrow (\neg\texttt{END}\ \mathbf{U}\ (\texttt{END} \wedge \mathbf{X}[\neg\texttt{START}\ \mathbf{U}\ \texttt{STATUS\_VALID}]))).$$

A counterexample for this specification may look like the computation path π shown in Fig. 5.1.

In this example, the failure of the specification on the trace is not trivially evident. Our explanations, displayed as *dots* attract the user's attention to the relevant places, to help in identifying the failure. Note that each dot r is a *cause* of the failure of φ on the trace: switching the value of r would, under some contingency on the other values, change the value of φ on π. For example, if we switch the value of `START` in state 15 from 1 to 0,

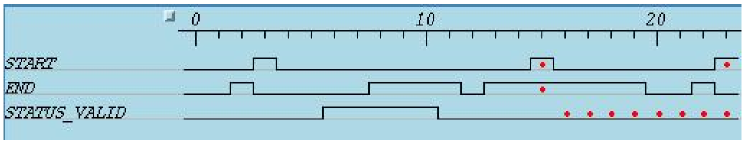

Figure 5.1: A counterexample with explanations

φ would not fail on the given trace anymore (in this case, no contingency on the other values is needed). Thus the matrix entry of the variable START at time 15 is indicated as a cause.

We show that the complexity of detecting an exact causal set is NP-complete, based on the complexity result for causality in binary models ([EL01]). We then present an over-approximation algorithm whose complexity is linear in the size of the formula and in the length of the trace.

5.1 Defining Causality in Counterexamples

A *counterexample* to an LTL formula φ in a Kripke structure K is a computation path $\pi = s_0, s_1, \ldots$ such that $\pi \not\models \varphi$. For a state s_i and a variable v, the labeling function L of K maps the pair $\langle s_i, v \rangle$ to $\{0, 1\}$ in a natural way: $L(\langle s_i, v \rangle) = 1$ if $s_i \models v$, and 0 otherwise. For a pair $\langle s, v \rangle$ in π, we denote by $\langle \hat{s}, v \rangle$ the pair that is derived from $\langle s, v \rangle$ by switching the labeling of v in s. Let π be a path, s a state in π and v a variable in the labeling function. We denote $\pi^{\langle \hat{s}, v \rangle}$ the path derived from π by switching the labeling of v in s on π. This definition can be extended for a set of pairs A: we denote $\hat{A}$ the set $\{\langle \hat{s}, v \rangle | \langle s, v \rangle \in A\}$. The path $\pi^{\hat{A}}$ is then derived from π by switching the value of v in s for all pairs $\langle s, v \rangle \in A$.

One of the ways to define causality is to use the definition of *criticality*: event A is *critical* for event B if, had A not occurred, B would not occur. Event C is then defined to be a *cause* of event B if C can be made *critical* for B by, possibly, changing some conditions. Adapting this definition to causality in counterexamples, we want to say that the value of variable v in state s is critical for the failure of φ on π if, after switching this value, φ does not fail on π any longer.

Before we formally define causes in counterexamples we need to deal with one subtlety: the value of φ on finite paths. While computation paths are infinite, it is often possible to determine that $\pi \not\models \varphi$ after a *finite* prefix of the path. Thus, a counterexample produced by a model checker may demonstrate a finite execution path. We use the notation $\pi[0..k] \models_f \varphi$ to denote "finitely models", and $\pi[0..k] \not\models_f \varphi$ for "finitely falsifies". These are defined as follows.

Definition 25 (Evaluation on finite paths)**.** Let $\pi[0..k]$ be a finite path and φ an LTL formula. We say that:

1. The value of φ is **true** in $\pi[0..k]$ ($\pi[0..k] \models_f \varphi$) iff for all infinite computations ρ, we have $\pi[0..k] \cdot \rho \models \varphi$
2. The value of φ is **false** in $\pi[0..k]$ ($\pi[0..k] \not\models_f \varphi$) iff for all infinite computations ρ, we have $\pi[0..k] \cdot \rho \not\models \varphi$;
3. The value of φ in π is **unknown** ($\pi[0..k]$? φ) iff there exist two infinite computations ρ_1 and ρ_2 such that $\pi[0..k] \cdot \rho_1 \models \varphi$ and $\pi[0..k] \cdot \rho_2 \not\models \varphi$.

Before we define criticality and causality in counterexamples, we note that only part of the values in a counterexample can be relevant for the explanation. We thus need the definition below.

Definition 26 (Bottom value)**.** For a Kripke structure $K = (S, I, R, L)$, a path π in K, and a formula φ, a pair $\langle s, v\rangle$ is said to have a *bottom value* for φ in π, if $L(\langle s, v\rangle) = 0$ and v has a *positive* polarity in φ, or $L(\langle s, v\rangle) = 1$ and v has a *negative* polarity in φ.

Note that a variable v may appear in different polarities in a formula φ. In such a case, we say that $\langle s, v\rangle$ has a bottom value for every state s.

Let φ be an LTL formula that fails on an infinite path $\pi = s_0, s_1, \ldots$, and let k be the smallest index such that $\pi[0..k] \not\models_f \varphi$. If φ does not fail on any finite prefix of π, we take $k = \infty$ (then $\pi[0..\infty]$ naturally stands for π, and we have $\pi \not\models \varphi$).
In the definitions of criticality and causality given below, we assume that k is the smallest index such that $\pi[0..k] \not\models_f \varphi$.

Definition 27 (Criticality in counterexample traces)**.** A pair $\langle s, v\rangle$ is *critical* for the failure of φ on $\pi[0..k]$ if $\pi[0..k] \not\models_f \varphi$, but either $\pi^{\langle \hat{s}, v\rangle}[0..k] \models_f \varphi$ or $\pi^{\langle \hat{s}, v\rangle}[0..k] \; ? \; \varphi$.

That is, switching the value of v in s changes the value of φ on $\pi[0..k]$ (to either **true** or **unknown**). As a simple example, consider the formula $\varphi = \mathbf{G}p$, on $\pi = s_0, s_1, s_2$, labeled $p \cdot p \cdot \neg p$. Then, $\pi[0..2] \not\models_f \varphi$, and $\langle s_2, p\rangle$ is critical for this failure, since switching the value of p in state s_2 changes the value of φ to **unknown**.

Definition 28 (Causality in counterexample traces)**.** A pair $\langle s, v\rangle$ is a *cause* of the failure of φ on $\pi[0..k]$ if there exists a set A of *bottom-valued* pairs, such that the following hold:

- $\langle s, v\rangle \notin A$,
- $\pi^{\hat{A}}[0..k] \not\models_f \varphi$, k is the smallest such index, and
- $\langle s, v\rangle$ is *critical* for the failure of φ on $\pi^{\hat{A}}[0..k]$.

A pair $\langle s, v\rangle$ is defined to be a *cause* for the failure of φ on π, if it can be made *critical* for this failure by switching the values of some bottom-valued pairs. Note that according to this definition, only bottom-valued pairs can be causes. The restriction of allowed changes to bottom-valued pairs is important, since other changes of values may introduce new failures that did not exist on the original counterexample, and thus can lead to “spurious causes” - pairs that are not causes of the original failure, but can be made critical if new failures are introduced. Consider, for example, the formula $\psi_1 = \mathbf{G}(req \rightarrow \mathbf{X}ack)$ and the trace ρ_1 pictured in Figure 5.2. It is clear that the value of req in s_0 is not a cause for the failure of ψ_1, since this request is acknowledged. If we allow changes of any pairs, it is easy to see that $\langle s_0, req\rangle$ is a cause for the failure of ψ_1 because changing the value of req in s_2 and of ack in s_1 makes it critical.

Note that a trace π may demonstrate more than one failure of φ, as we demonstrate in the examples below. We believe that the first failure is the most interesting one for the user. Also, focusing on one failure naturally reduces the set of causes, and thus makes it easier for the user to understand the explanation.

Examples:

1. Consider $\varphi_1 = \mathsf{G}\, p$ and a path $\pi_1 = s_0, s_1, s_2, s_3, (s_4)^\omega$ labeled as $(p) \cdot (p) \cdot (\neg p) \cdot (\neg p) \cdot (p)^\omega$. The shortest prefix of π on which φ_1 fails is $\pi_1[0..2]$. $\langle s_2, p\rangle$ is critical for the failure of φ on $\pi[0..2]$, because changing its value from 0 to 1 changes the value of φ on $\pi[0..2]$ from **false** to **unknown**. Also, there are no bottom-valued pairs in $\pi[0..2]$, thus there are no other causes, which indeed meets our intuition.

2. Consider $\varphi_2 = \mathsf{F}\, p$ and a path $\pi_2 = (s_0)^\omega = (\neg p)^\omega$. The formula φ_2 fails in π_2, yet it does not fail on any finite prefix of π_2. Note that changing the value of any $\langle s_i, p\rangle$ for $i \geq 0$ results in the satisfaction of φ on π, thus all pairs $\{\langle s_i, p\rangle : i \in \mathbb{N}\}$ are critical and hence are causes for the failure of φ_2 on π_2.

3. The following example demonstrates the difference between criticality and causality. Consider $\varphi_= \mathbf{G}(a \wedge b \wedge c)$ and a trace $\pi_3 = s_0, s_1, s_2, \ldots$ labeled as $(\emptyset)^\omega$ (see Figure 5.2). The formula φ_3 fails on s_0, however, changing the value of any signal in one state does not change the value of φ_3. There exists, however, a set A of bottom-valued pairs whose change makes the value of a in s_0 critical: $A = \{\langle s_0, b\rangle, \langle s_0, c\rangle\}$. Similarly, $\langle s_0, b\rangle$ and $\langle s_0, c\rangle$ are also causes.

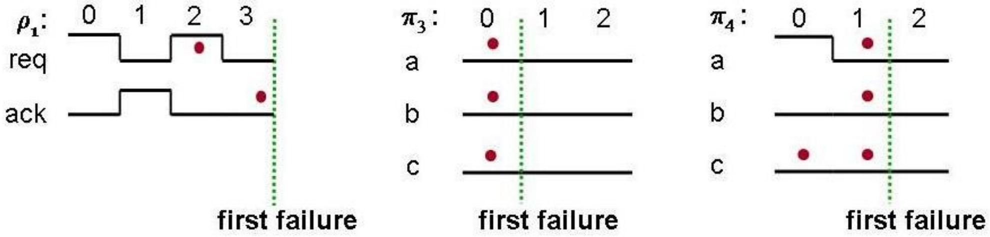

Figure 5.2: Counterexample traces.

5.2 Complexity of computing causality in counterexamples

The complexity of computing causes for counterexamples follows from the complexity of computing causality in binary causal models defined in [HP01].

Lemma 29. Computing the set of causes for falsification of a linear-time temporal specification on a single trace is NP-complete.

Proof. The proof of NP-hardness is based the reduction from computing causality in binary causal models to computing causality in counterexamples. The problem of computing causality in binary causal models is NP-complete [EL01]. The reduction from binary causal models to Boolean circuits and from Boolean circuits to model-checking, shown in [CHK08], is based on the automata-theoretic approach to branching-time model checking ([KVW00]), and proves that computing causality in model checking of branching time specifications is NP-complete. On a single trace linear-time and branching temporal logics coincide, and computing the causes for satisfaction is easily reducible to computing the causes for falsification.

The proof of membership in NP is straightforward: given a path π and a formula φ that is falsified on π, the number of pairs $\langle s, v\rangle$ is $|\varphi| \cdot |\pi|$; for a pair $\langle s, v\rangle$, we can non-deterministically choose a set A of bottom-valued pairs; checking whether changing

L on S makes $\langle s, v\rangle$ critical for the falsification of φ requires model-checking φ on the modified π twice, and thus can be done in linear time. $\square$

5.3 An over-approximation algorithm

The counterexamples we work with have a finite number of states. When representing an infinite path, the counterexample will contain a loop indication, i.e., an indication that the last state in the counterexample is equal to one of the earlier states.

Let φ be a formula, given in negation normal form, and let $\pi[0..k] = s_0, s_1, ..., s_k$ be a non-empty counterexample for it, consisting of a finite number of states and a possible loop indication. We assume that the counterexample contains a loop only if it is necessary for demonstrating the failure. In other words, if $\pi[0..k] \not\models_f \varphi$ then we assume that $\pi[0..k]$ has no loop indication.

We denote by $\pi[i..k]$ the suffix of $\pi[0..k]$ that starts at s_i. The procedure C below produces $C(\pi[i..k], \psi)$, the approximation of the set of causes for the failure of a sub-formula ψ on the suffix of $\pi[0..k]$ that starts with s_i. We invoke the procedure C with the arguments $(\pi[0..k], \varphi)$ to produce the set of causes for the failure of φ on $\pi[0..k]$.

During the computation of $C(\pi[i..k], \varphi)$, we use the auxiliary function val, that evaluates sub-formulas of φ on the given path. It returns 0 if the sub-formula fails on the path and 1 otherwise. The computation of val is done in parallel with the computation of the causality set, and relies on recursively computed causality sets for sub-formulas of φ. The value of val is computed as follows:

- $val(\pi[i..k], true) = 1$
- $val(\pi[i..k], false) = 0$
- For any formula $\varphi \notin \{true, false\}$, $val(\pi[i..k], \varphi) = 1$ iff $C(\pi[i..k], \varphi) = \emptyset$

Algorithm 30 (Causality Set)**.** An approximated causality set C for $\pi[i..k]$ and ψ is computed as follows

- $C(\pi[i..k], true) = C(\pi[i..k], false) = \emptyset$
- $C(\pi[i..k], p) = \begin{cases} \{\langle s_i, p\rangle\} & \text{if } p \notin s_i \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \neg p) = \begin{cases} \{\langle s_i, p\rangle\} & \text{if } p \in L(s_i) \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \mathbf{X}\varphi) = \begin{cases} C(\pi[i+1..k], \varphi) & \text{if } i < k \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \varphi \wedge \psi) = C(\pi[i..k], \varphi) \cup C(\pi[i..k], \psi)$
- $C(\pi[i..k], \varphi \vee \psi) =$
$\begin{cases} C(\pi[i..k], \varphi) \cup C(\pi[i..k], \psi) & \text{if } val(\pi[i..k], \varphi) = 0 \text{ and } val(\pi[i..k], \psi) = 0 \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \mathbf{G}\varphi) =$
$\begin{cases} C(\pi[i..k], \varphi) & \text{if } val(\pi[i..k], \varphi) = 0 \\ C(\pi[i+1..k], \mathbf{G}\varphi) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } i < k \text{ and } val(\pi[i..k], \mathbf{XG}\varphi) = 0 \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], [\varphi\, \mathbf{U}\, \psi]) =$
$\begin{cases} C(\pi[i..k], \psi) \cup C(\pi[i..k], \varphi) & \text{if } val(\pi[i..k], \varphi) = 0 \text{ and } val(\pi[i..k], \psi) = 0 \\ C(\pi[i..k], \psi) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } val(\pi[i..k], \psi) = 0 \text{ and } i = k \\ C(\pi[i..k], \psi) \cup C(\pi[i+1..k], [\varphi\, \mathbf{U}\, \psi]) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } val(\pi[i..k], \psi) = 0 \\ & \quad \text{and } i < k \text{ and } val(\pi[i..k], \mathbf{X}[\varphi\, \mathbf{U}\, \psi]) = 0 \\ \emptyset & \text{otherwise} \end{cases}$

The procedure above recursively computes a set of causes for the given formula φ on the suffix of a counterexample $\pi[i..k]$. At the proposition level, p is considered a cause in the current state if and only if it has a bottom-value in the state. At every level of the recursion, a sub-formula is considered relevant (that is, its exploration can produce causes for falsification of the whole specification) if it has a value of **false** at the current state.

We explain in detail the recursive computation for the *Until* operator, since it is the most difficult to follow.

When examining the computation path $\pi[i..k]$, the subformula $\eta = [\varphi \, \mathbf{U} \, \psi]$ will be explored only in the case that η has a value of **false** on $\pi[i..k]$. There could be one of the three reasons for this:

1. The value of φ on $\pi[i..k]$ is **false** , and the value of ψ on $\pi[i..k]$ is **false**, in which case the causality set will be the union of the causality sets for φ and for ψ.

2. The value of φ on $\pi[i..k]$ is **true** or **unknown** , and the value of ψ on $\pi[i..k]$ is **false**, and $i = k$. In this case we know that the Until formula does not fail on any finite prefix of the counterexample (and therefore it must have a loop). The causality set is the set for ψ, since the reason for the failure is the fact that ψ never holds.

3. The value of φ on $\pi[i..k]$ is **true** or **unknown**, the value of ψ on $\pi[i..k]$ is **false**, and the value of $\mathbf{X}[\varphi \, \mathbf{U} \, \psi]$ on $\pi[i..k]$ is **false**. Here the Until formula has not failed yet, but we know that it will. We thus take as causality sets the set for ψ, to show that it has continuously failed to hold so far, and the set for $\mathbf{X}[\varphi \, \mathbf{U} \, \psi]$.

Lemma 31. The complexity of Algorithm 30 is linear in k and in $|\varphi|$.

Proof. The complexity follows from the fact that each subformula ψ of φ is evaluated at most once at each state s_i of the counterexample π. □

Theorem 32. *The set of pairs produced by Algorithm 30 for a formula φ on a path π is an over-approximation of the set of causes for φ on π according to Definition 28.*

For the proof of Theorem 32, we consider an evaluation graph for φ on a given path π. An LTL formula φ, given in Negation Normal Form, can be decomposed according to the following two rules (that appear, for example, in [Wol85, MP85]):

- $\psi_1 \, \mathbf{U} \, \psi_2 \equiv (\psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \, \mathbf{U} \, \psi_2)))$

- $\mathbf{G}\psi \equiv \psi \wedge \mathbf{X}(\mathbf{G}\psi)$

Given a finite prefix of an execution path $\pi[0..k]$, we can build a labeled AND-OR evaluation graph for φ on $\pi[0..k]$. Each node will be labeled with a state and a formula that should be evaluated in the state. Internal nodes are labeled also by an operator, AND or OR, that indicates how the evaluations of the children nodes are combined. The root of the graph will be labeled with (s_0, φ). A leaf node n labeled (s_i, φ) is expanded according to its label:

- If $\varphi = \psi_1 \wedge \psi_2$, we construct two new nodes, and label them with (s_i, ψ_1) and (s_i, ψ_2). The node n is then labeled also with AND.

- For $\varphi = \psi_1 \vee \psi_2$, the same as item (1) with the label of n being OR.

- If $\varphi = \mathbf{X}\psi$, we add a child node and label it (s_{i+1}, ψ).

- Finally, if $\varphi = \mathbf{G}\psi$ or $\varphi = \psi_1 \, \mathbf{U} \, \psi_2$ we expand the formula according to the expansion rules given above.

For a path π of length k, the graph is expanded according to the above given rules, until all leaf nodes are labeled with one of the following:

- (s_{k+1}, ρ) with ρ being any formula, or

- (s_i, l) where $i \leq k$ and l is a literal (a variable or its negation).

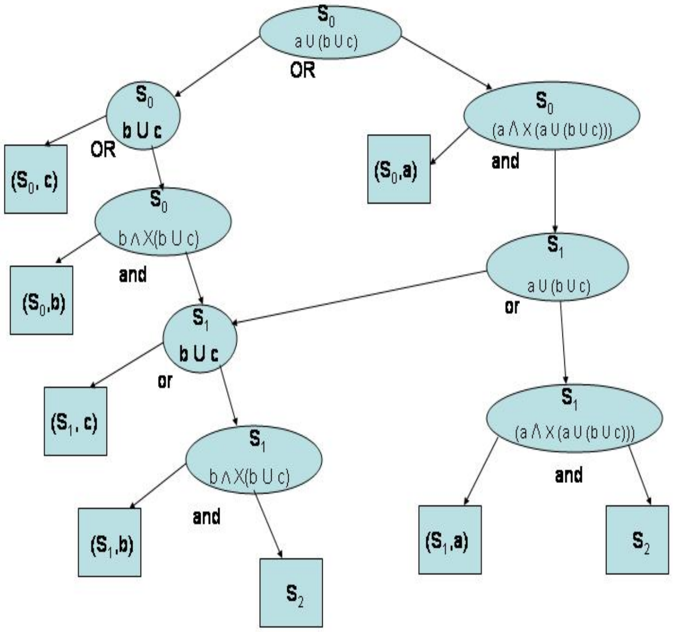

Figure 5.3: An evaluation graph for $a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$

Figure 5.3 demonstrates the AND-OR evaluation graph for the formula $a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$, on a two-state path. To evaluate the formula, we now consider the given path $\pi[0..k]$: every proposition gets a value (**true** or **false**) as indicated in π, and the evaluation is performed bottom-up, starting from the leaves. Note that in general a leaf's value may be unknown, if it depends on values from the state s_{k+1}, that are not given in $\pi[0..k]$. However, in our circumstances this would not affect the evaluation of φ on π since $\pi[0..k]$ is a counterexample, and therefore φ evaluates to **false** on it. Figure 5.4 presents the evaluation graph of $a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$, with the values added for $\pi[0..1] = a \cdot \emptyset$. We use the evaluation graph to prove Theorem 32.

Proof of Theorem 32. For a formula φ and a path $\pi[0..k]$, we examine the evaluation graph as described above. Since φ fails on π, the value of the root is *false*. We look at evaluation paths ε in the graph, that start from a leaf and go backward all the way to

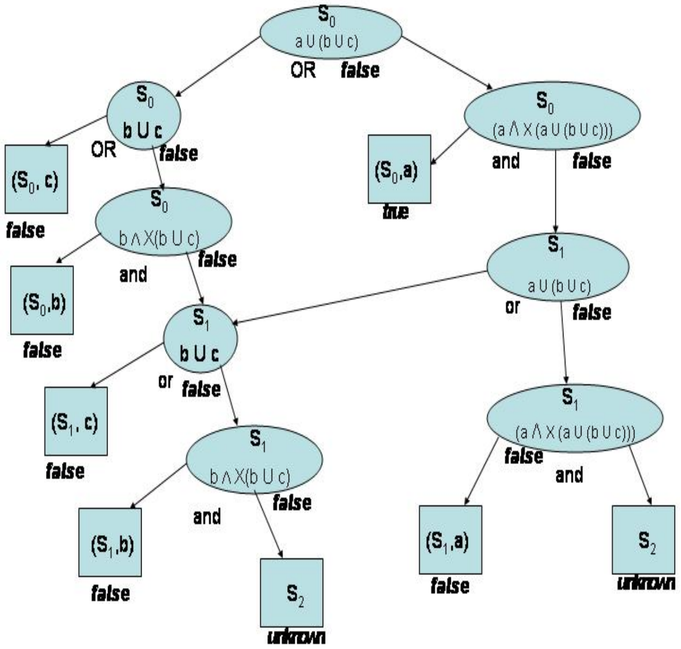

Figure 5.4: Evaluation of $a \, \mathbf{U} \, (b \, \mathbf{U} \, c)$ on $\pi[0..1] = a \cdot \emptyset$

the root. If ε visits only *false* labeled nodes, we call it a **failure path**. The claim below is needed for the rest of the proof.

Claim 33. Let l be a literal in φ and s_i a state in $\pi[0..k]$. We denote v_l the variable corresponding to l (that is, $l = \neg v_l$ or $l = v_l$). Then the following holds.

1. Let ε be a **failure path** in the evaluation graph of φ on $\pi[0..k]$, such that its leaf is labeled with (s_i, l). If l evaluates to **false** in s_i, then the pair $\langle s_i, v_l \rangle$ is a cause according to Algorithm 30.

2. If a pair $\langle s_i, v_l \rangle$ is a cause according to Definition 28, then there must exist a **failure path** ε on which (s_i, l) appears in the leaf label, and l evaluates to **false** in s_i.

By item (2) of Claim 33, if $\langle s_i, v_l \rangle$ is a cause according to Definition 28, then v gets a bottom-value on a leaf of a failure path. But by item (1) of the claim, in this case

$\langle s_i, v_l \rangle$ is also a cause according to Algorithm 30. Thus the algorithm produces an over-approximation of the causal set according to Definition 28. □

Proof of Claim 33.

1. A close examination of Algorithm 30 shows that a pair $\langle s_i, v_l \rangle$ gets into the causality set if v_l assumes a bottom-value in s_i. It will be passed on to the next level of the recursion as long as the sub-formula that v_l belongs to keeps getting *false*. This is the same as visiting only **false** labeled nodes on the way to the root.

2. Suppose the pair $\langle s_i, v_l \rangle$ can be made *critical*. That means that after switching some bottom values, φ still fails on $\pi[0..k]$, but switching $\langle s_i, v_l \rangle$ now changes the value of φ on $\pi[0..k]$. Note that (s_i, l) must be located on a failure path; otherwise switching its value cannot change the value of φ.

□

We note that not all bottom-valued leaves that have a failure path to the root are causes (otherwise Algorithm 30 would always give accurate results). In our experience though, Algorithm 30 gives accurate results for the majority of real-life examples. As an example of a formula on which Algorithm 30 does not give an accurate result, consider $\varphi = a \, \mathbf{U} \, (b \, \mathbf{U} \, c)$ and a trace $\pi = s_0, s_1, s_2, \ldots$ labeled as $a \cdot (\emptyset)^\omega$ (see Figure 5.2). The formula φ fails on π, and $\pi[0..1]$ is the shortest prefix on which it fails. What is the set of causes for failure of φ on $\pi[0..1]$? The pair $\langle s_0, a \rangle$ is not a cause, since it is not bottom-valued. Checking all possible changes of sets of bottom-valued pairs shows that $\langle s_0, b \rangle$ is not a cause. On the other hand, $\langle s_1, a \rangle$ and $\langle s_1, b \rangle$ are causes because changing the value of a in s_1 from 0 to 1 makes φ **unknown** on $\pi[0..1]$, and similarly for $\langle s_1, b \rangle$. The pairs $\langle s_0, c \rangle$ and $\langle s_1, c \rangle$ are causes because changing the value of c in either s_0 or s_1 from 0 to 1 changes the value of φ to **true** on $\pi[0..1]$. The values of signals in s_2 are not causes because the first failure of φ happens in s_1. The causes are represented graphically as red dots in Figure 5.2. By examining the algorithm, we can see that on φ and π it outputs the set of pairs that contains, in addition to the exact set of causes, the pair $\langle s_0, b\rangle$.

5.4 Discussion

The definition of an explanation for a counterexample reflects two major decisions. First, we chose to detect a set of causes for the temporally *first* failure only (that is, the smallest k such that $\pi[0..k] \not\models_f \varphi$). We believe that this is the most beneficial for the user; in many cases, other failures demonstrated by the computation path are a consequence of the first one. For example, in the design of a hardware model, it is common to have an *error* signal, that is never supposed to rise, however, once rises, it stays in this position forever. For the formula $\mathbf{G}\neg$*error*, only the first state were *error*=1 is interesting, since the rest are a consequence of the first.

Another decision made when choosing the definition is that for the first failure, we try to find all values that have any influence on the failure. That is where the definition of *causality*, rather than *criticality*, comes into play. We believe that our explanations, at this stage, should furnish the user with all she needs to debug the error. Choosing to provide a *critical* set as an explanation, would make this set minimal, such that switching any of the values would make the formula pass in the computation path.

Such a set can be detected by translating the counterexample path and the formula into a CNF formula via a BMC translation. This CNF formula would be unsatisfiable, and the unsat core [LS04, MLA$^+$05] provided by the satisfiability solver would be a minimal set that demonstrate the failure. We think, however, that this is not good enough for the user. For example, let our formula be $\mathbf{G}(p \wedge q)$, on a single-state trace $\emptyset$. Then p and q are each an unsat core, and therefore only one of them will be provided. If the user is to debug the problem, however, she must be aware of the failure of both p and q.

Chapter 6

Conclusion and Future Directions

We have approached two different aspects of model checking. In Chapters 3 and 4 we examined different ways to use Description Logic reasoning for symbolic model checking. In Chapter 5 we proposed a method to analyze a counterexample. Below we summarize each of the chapters and discuss future research directions.

Bounded model checking of safety formulas using DL

We have presented several methods to perform bounded model checking of safety properties using Description Logic reasoning. All of these methods have the nice property that the encoding of the problem as a DL ontology is of constant size in terms of the original problem, and once set, the model checking task is performed by the DL reasoning tool, with no intervention. This is in contrast to BDD-based model checking tools that need to custom-build the model checking algorithm using BDDs. For bounded model checking, a given model description MD and a bound k are represented in DL with an ontology of size $|MD| + k$, as opposed to $|MD| \times k$ when translating MD to a propositional formula in order to use a satisfiability solver. Our method can thus be viewed as a natural setting for a symbolic representation of bounded model checking problems, avoiding the need to unfold the model as done for SAT based BMC.

The methods described in Chapter 3 used the DL reasoner as a black box, and no attempt has been made to tune the DL algorithms to better work for the task of model checking. In the future it would be interesting to examine the internal DL algorithms in light of our application. First, the results of Section 3.5 demonstrate that different encodings of the same problem vary dramatically in performance. If this is found to be inherent in the DL algorithms, it would be natural to search for easily detectable conditions, where one encoding can be automatically translated into another, that is easier to reason about. This could improve the reasoning performance for some applications, as demonstrated by our results. Second, at least one of our encodings (the one using $\mathcal{ALC}$), seems to perform very well as long as the bound is small enough. It is interesting to understand what causes the blow-up for larger bounds. If this problem can be overcome, it would enable the use of DL for model checking, and potentially improve the performance of DL reasoning in general.

Finally, when a concept is found to be consistent with respect to a given terminology, the DL reasoner is capable of providing a satisfying interpretation. Since the existence of an interpretation, in our setting, indicates a bug in the model, the interpretation should be translated into a readable counterexample. Note that the interpretation can possibly be only partial, since the tableaux reasoning algorithm may not depend on all concepts of the terminology. For such cases, some mechanism should be developed, to derive the lacking information.

Liveness and fairness using DL

We have approached model checking of liveness formulas using DL reasoning, and showed that a formula of the type AFp can be easily encoded over $\mathcal{ALC}$ when no fairness constraints are involved. When fairness constraints are required, encoding in common dialects of DL is not possible. We showed however, that the tableaux reasoning procedure can be modified to support fairness in bounded model checking. In order to achieve unbounded model checking, the algorithm should be iterated with increasing bounds. This makes our method less efficient when no fair cycle exists (the formula holds in the model) or when the fair loop is long. The experiments we have presented, comparing our method to the model checker VIS, support the observation that our approach would be more beneficial when a fair loop does exist in the model (i.e. a bug is found). More experiments should be performed, however, to understand the extent to which our method can be beneficial.

Model checking of liveness properties is considered more difficult than model checking of safety ones, and special attention has been devoted to this type of formula in the literature, both using BDD-based methods [BGS00, RBS00, BGS06], and using SAT-based techniques [AS04, GGA05]. It has been recognized, however, that no single method can outperform others on all models [BDEGW03, Nev08]. State of the art model checkers invoke multiple algorithms for each model checking problem, presenting the user with the result of the first method to terminate. Our method, if found beneficial for a significant range of models, could fit nicely in such a platform, speeding up verification time for part of the models.

Counterexample explanation

We have shown how the causality definition of Halpern and Pearl [HP01] can be adapted to the task of explaining a counterexample. Since the causality algorithm is applied to a single counterexample, ignoring the model from which it was extracted, no size issues are involved, and the execution time is negligible. An important advantage of our method is the fact that it is independent of the tool that produced the counterexample. When more than one model checking "engine" is invoked to verify a formula, as described in [BDEGW03, Nev08], the independence of the causality algorithm is especially important. We note that our approach, though demonstrated here for LTL specifications, can work in the same manner for ACTL formulas, since on a single computation path, LTL and ACTL formulas coincide.

In the future, it would be interesting to extend our method to other linear temporal logics used in practice, such as PSL [Acc03]. While our definition should hold for this logic as well, the approximation algorithm should be extended without increasing its complexity. Since the algorithm we provided produces an over-approximation of the causality set, it is interesting to see if the language for which it provides the exact causality set could be characterized. Finally, the approach we have presented defines and (approximately) detects a set of causes for the *first* failure of a formula on a trace. While we believe that this information is the most beneficial for the user, other definitions can also be considered.

Bibliography

[AAH^{+}85] M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider. *Distributed systems: methods and tools for specification. An advanced course*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[AAH^{+}03] M. S. Abadir, K. Albin, J. Havlicek, N. Krishnamurthy, and A. K. Martin. Formal verification successes at motorola. *Formal Methods in System Design*, 22(2):117–123, 2003.

[Acc03] Accellera. Accellera property language reference manual. In *http://www.eda.org/ vfv/docs/psl_lrm-1.0.pdf*, pages 101–112, January 2003. Appendix B.

[AS85] B. Alpern and F. B. Schnieder. Defining liveness. In *Information Processing Letters*, pages 21:181–185, October 1985.

[AS87] B. Alpern and F. B. Schnieder. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AS04] Mohammad Awedh and Fabio Somenzi. Proving more properties with bounded model checking. In *CAV*, pages 96–108, 2004.

[BBDC$^+$09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining a counterexample using causality. In *CAV*, 2009.

[BBDL98] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In *Proc.* 10^{th} *International Conference on Computer Aided Verification (CAV'98)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *In Proc. 5th international Tools and Algorithms for the Construction and Analyses of Systems (TACAS'99)*, 1999.

[BCM$^+$03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BDEGW03] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003.

[BDFR05] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. The safety simple subset. In *First International Haifa Verification Conference*, volume 3875 of *Lecture Notes in Computer Science*, pages 14–29. Springer, November 2005.

[BDPT$^+$09] Shoham Ben-David, Jeffrey Pound, Richard Trefler, Dmitry Tsarkov, and Grant Weddell. Fair cycle detection using description logic reasoning. In *Proc. 22nd International Workshop on Description Logics*, 2009.

[BDTTW08] Shoham Ben-David, Richard Trefler, Dmitry Tsarkov, and Grant Weddell. Checking inevitability and invariance using description logic technology, 2008. Technical report CS-2008-28, University of Waterloo.

[BDTW06] Shoham Ben-David, Richard Trefler, and Grant Weddell. Model checking the basic modalities of CTL with description logic. In *Proc. 19th International Workshop on Description Logics*, pages 223–230, 2006.

[BDTW07a] Shoham Ben-David, Richard Trefler, and Grant Weddell. Bounded model checking with description logic reasoning. In *Proc. 16th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, pages 60–72, 2007.

[BDTW07b] Shoham Ben-David, Richard Trefler, and Grant Weddell. Modal vs. propositional reasoning for model checking with description logic. In *Proc. 20th International Workshop on Description Logics*, pages 179–186, 2007.

[BDTW08] Shoham Ben-David, Richard Trefler, and Grant Weddell. Model checking using description logic. *Journal of Logic and Computations*, 2008. doi: 10.1093/logcom/exn062.

[BFH05] Doron Bustan, Dana Fisman, and John Havlicek. Automata construction for PSL, 2005. Technical Report MCS05- 04, Computer Science and Applied Mathematics, The Weizmann Institute of Science.

[BFR04] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. Embedding finite automata within regular expressions. In 1st *International Symposium on Leveraging Applications of Formal Methods (IsoLa'04)*, Paphos, Cyprus, November 2004.

[BGS00] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, 2000.

[BGS06] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in log symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.

[BHSV^{+}96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *CAV*, 1996.

[Bie07] Armin Biere. The AIGER And-Inverter Graph (AIG) Format, 2007. http://fmv.jku.at/aiger/.

[BKM95] R. S. Boyer, M. Kaufmann, and J. Strother Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[BMC03] S. Bechhofer, R. Mller, and P. Crowther. The dig description logic interface. In *Proc. of the 16th Description Logic Workshop*, 2003.

[BNR03] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, 2003.

[Bry86] Randy Bryant. Graph-based algorithms for boolean function manipulation. In *In IEEE Transactions on Computers*, volume C-35(8), pages 677–691, 1986.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CG05] Marsha Chechik and Arie Gurfinkel. A framework for counterexample generation and exploration. In *Proceedings of FASE'05*, pages 217–233, 2005.

[CGL99] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *IJCAI*, pages 84–89, 1999.

[CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 427–432, 1995.

[CGP00] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[CGY08] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *TACAS*, pages 233–248, 2008.

[CHK08] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? *ACM Trans. Comput. Log.*, 9(3), 2008.

[CIW$^+$01] F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *In Proceedings of CHARME'01*, pages 275–292, 2001.

[CQ69] A.M. Collins and M.R. Quillian. Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247, 1969.

[DM00] Francesco M. Donini and Fabio Massacci. Exptime tableaux for $\mathcal{ALC}$. *Artificial Intelligence*, 124(1):87–138, 2000.

[DQ01] A. Dovier and E. Quintarelli. Model checking based data retrieval. In *Revised Papers from the 8th International Workshop on Database Programming Languages, LNCS Vol. 2397*, pages 62–77, 2001.

[DRS03] Y. Dong, C. R. Ramakrishnan, and S. A. Smolka. Model checking and evidence exploration. In *IEEE Conference and Workshops on Engineering Computer Based Systems*, pages 214–223, Huntsville, Alabama, 2003. IEEE.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[EL01] T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. In *Proc. 7th International Joint Conference on Artificial Intelligence*, pages 35–40, 2001.

[ES04] Niklas Een and Niklas Srensson. An extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[Ger01] R. Gerth. Model checking if your life depends on it a view from intel's trenches. In *SPIN*, page 15, 2001.

[GGA05] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Beyond safety: customized sat-based model checking. In *DAC*, pages 738–743, 2005.

[GGV00] George Gottlob, Erich Grädel, and Helmut Veith. Linear Time Datalog for Branching Time Logic. In J. Minker, editor, *Logic-Based Artificial Intelligence*, chapter 19. Kluwer, 2000.

[GGV02] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a deductive query language with linear time model checking. *Computational Logic*, 3(1):42–79, 2002.

[GHT06] Tom Gardiner, Ian Horrocks, and Dmitry Tsarkov. Automated benchmarking of description logic reasoners. In *Description Logics*, 2006.

[GK04] A. Groce and D. Kroening. Making the most of bmc counterexamples. In *SGSH*, July 2004.

[GL97] G. De Giacomo and M. Lenzerini. A uniform framework for concept definitions in description logics. *J. Artif. Intell. Res. (JAIR)*, 6:87–110, 1997.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, USA, 1993.

[Gro04] A. Groce. Error explanation with distance metrics. In *TACAS*, 2004.

[GSB07] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for c programs. *Electr. Notes Theor. Comput. Sci.*, 174(4):95–111, 2007.

[Hal95] Tom R. Halfhill. An error in a lookup table created the infamous bug in intel’s latest processor, 1995. http://www.byte.com/art/9503/sec13/art1.htm.

[Hal02] N. Hall. Two concepts of causation. In J. Collins, N. Hall, and L. A. Paul, editors, *Causation and Counterfactuals*. MIT Press, Cambridge, Mass., 2002.

[HM01] Volker Haarslev and Ralf Moller. Racer system description. In *International Joint Conference on Automated Reasoning (IJCAR'2001)*, volume 2083, pages 701–706, 2001.

[HP01] J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach — part 1: Causes. In *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 194–202, San Francisco, CA, 2001. Morgan Kaufmann Publishers.

[HS99] Ian Horrocks and Ulrike Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999.

[HST00] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. Lect. Notes in Comp. Sci., pages 482–496, 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1979.

[Hum39] D. Hume. *A treatise of human nature*. John Noon, London, 1939.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS02*, pages 445–458, 2002.

[Kin94] E. Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53, June 1994.

[KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[KV99] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In *Proc.* 11^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1633, pages 172–183. Springer-Verlag, 1999.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE–3(2):125–143, 1977.

[LS04] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[Mai00] Monika Maidl. The common fragment of CTL and LTL. In *IEEE Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[McM] K. McMillan. Cadence-smv. http://www.kenmcmil.com/smv.html.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Min81] M. Minsky. A framework for representing knowledge. In *Mind Design*, MIT Press, pages 95–128, 1981.

[MLA$^+$05] Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, and Karem A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *SAT*, pages 467–474, 2005.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *38th Design Automation Conference*, pages 530–535, 2001.

[MP85] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer (Published: 8/1995), New York, NY, USA, 1985.

[MR04] M.Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, Canbridge UK, 2004.

[MT01] Panagiotis Manolios and Richard J. Trefler. Safety and liveness in branching time. In *Logic in Computer Science*, pages 366–377, 2001.

[Nev08] Ziv Nevo. User-friendly model checking: Automatically configuring algorithms with rulebase/pe. In *4th Haifa Verification Conference*, October 2008.

[NuS] NuSMV examples collection. http://nusmv.irst.itc.it/examples/examples.html.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[QS82] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, 1982.

[Qui67] M.R. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12:410–430, 1967.

[Ram31] F. Plank Ramsey. Truth and Probability. *Foundations of Mathematics and Other Logical Essays*, 1931.

[RBS00] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD '00:*

Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, pages 143–160. Springer-Verlag, 2000.

[RN94] A. L. Rector and W. A. Nowlan. The galen project. *Computer Methods and Programs Biomedicine*, 45(1-2):75–78, 1994.

[Sah04] Maneesha Sahasrabudhe. SQL-based CTL model checking for telephony feature interactions. In *A Master Thesis, Univesity of Waterloo, Ontario Canada*, 2004.

[SB07] S. Staber and R. Bloem. Fault localization and correction with qbf. In *SAT*, pages 355–368, 2007.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Sch91] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *Proc. of the 12th International Joint Conference on Artificial Intelligence*, pages 466–471, 1991.

[SFBD08] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *ACM Great Lakes Symposium on VLSI*, pages 77–82, 2008.

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. 5(2), 2007.

[SQL05] S. Shen, Y. Qin, and S. Li. A faster counterexample minization algorithm based on refutation analysis. In *DATE05*, pages 672–677, 2005.

[SS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. 48(1):1–26, 1991.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of mathematics*, 5(2):285–309, 1955.

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[Tob01] Stephan Tobies. Complexity results and practical algorithms for logics in knowledge representation, 2001. PhD Thesis, LuFG Theoretical Computer Science, RWTH-Aachen, Germany.

[Tuo88] H. Tuominen. Elementary net systems and dynamic logic. In *European Workshop on Applications and Theory in Petri Nets*, number 424 in LNCS, pages 453–466, 1988.

[Tuo89] H. Tuominen. Proving properties of elementary net systems with a special-purpose theorem prover. In *Proc. International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 97–104, 1989.

[TW08] D. Toman and G. Weddell, 2008. Personal Communication.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[Wol85] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28:119–135, 1985.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths (extended abstract). In *FOCS*, pages 185–194, 1983.

[WYIG06] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *ATVA*, pages 82–95, 2006.