INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

LTL Software Model Checking

in CPAchecker

Thomas Bunk

Master Thesis

Supervisor	Prof. Dr. Dirk Beyer
Advisor	Dr. Philipp Wendler
Submission Date	April 8th 2019

Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, April 8th 2019 Thomas Bunk

Abstract

Model checking is used to automatically verify a model against a specification. In terms of software model checking, this ensures that the program behaves correctly and does what it is supposed to do. The standard approach is to statically analyze a program, construct an abstract model thereof, and finally perform an exhaustive search of the state space in order to determine whether the specification holds. The latter is mostly given in the form of a temporal logic, because this allows to easily express desirable properties of a system, such as e.g. functional correctness, reachability, safety, or liveness.

In this thesis a specific specification logic, linear temporal logic (LTL), is implemented into the CPAchecker framework. The main objective lies on analyzing programs written in C to be verified for LTL properties, and in particular, liveness-properties thereof. Generally, this can be done by converting a negated LTL formula into an automaton on infinite words (more specifically, a Büchi automaton), and combine this afterwards with a model of the software program. In CPAchecker, this results in an abstract reachability graph (ARG), that has a finite set of states which can be reached from an initial starting state. The correctness requirement is eventually proven by verifying that there is no set of words left which the ARG accepts, i.e., that the language of the ARG is empty.

Acknowledgements

This Master's thesis was written at the "Institut für Informatik" of the Ludwig-Maximilians-Universität München at the chair of Software and Computational Systems Lab.

My special thanks go to my advisers Prof. Dr. Dirk Beyer and Dr. Philipp Wendler, who both provided great assistance by helping me with constructive advise and dedicated support. I especially thank Martin Spiessl for many valuable and wide-ranging discussions that we have had, particularly during the last months of my work. I would also like to thank Karlheinz Friedberger, Dr. Marie-Christine Jakobs, and all the other colleagues from the chair who assisted me by giving instructions on how to utilize the CPAchecker project, and hence contributed to the success of this Master's thesis. Beyond that, many thanks go to Katharina Winter for her careful reading of a draft of this thesis.

Contents

1 Introduction 1

2 Related Work 4

3 LTL Software Model Checking in CPAchecker 6

3.1 The Methodology 6

3.2 Motivating Example 8

4 Preliminaries 13

4.1 Linear Temporal Logic 14

4.1.1 Syntax 15

4.1.2 Semantics 17

4.1.3 Equivalences 20

4.2 Safety and Liveness 22

4.3 Büchi Automaton 23

4.4 CPAchecker 25

4.4.1 Control-flow Automaton 26

4.4.2 CPA Algorithm 28

4.4.3 Configurable Program Analysis 30

4.4.4 Counterexample-guided Abstraction Refinement 31

5 LTL Software Model Checking 33

6 Implementation 38

6.1 Parsing LTL Formulas in CPACHECKER 39

6.2 Trace Abstraction 44

6.2.1 Example 45

7 Conclusion and Future Work 49

Bibliography 51

List of Figures

3.1 Feasibility of sequences 7
3.2 C-Program together with a corresponding CFA 8
3.3 Büchi Automaton in comparison with the negated LTL property . . . 9
3.4 Example of a Büchi program 10

4.1 Temporal modalities in LTL 18
4.2 Transition system with two states 21
4.3 Example of a Büchi automaton 24
4.4 Architecture of CPAchecker 25
4.5 Example of a CFA 27
4.6 Procedural method of CEGAR. 32

5.1 Implementation of LTL model checking in CPAchecker 34

6.1 Example of an ANTLR4 syntax tree 41
6.2 Example C-Program for the trace abstraction 46
6.3 Interpolation automaton and the resulting automaton from the first refinement 47
6.4 Interpolation automaton and the resulting automaton from the second refinement 48

CHAPTER 1

Introduction

Nowadays computing devices are so well established that they accompany us in almost every aspect of our daily life. From computers and smartphones through to Smart TVs or IoT Devices, it is astonishing how ubiquitous they have become by up to this point and how much they have contributed towards making life much more convenient for the endconsumer. Even in mission-critical systems, such as airplanes, automobiles, or entertainment, software is no longer indispensable. In the modern car for example, the software ranges over control functions like breaking, cruise control, or parking assistants, and more ambitious systems are currently in research and development, such as connected cars equipped with internet access, the reduction of energy consumption through advanced software algorithms, or the autonomously driving car. These are all perfect examples for how increasingly complex the software systems are becoming. Not only are they required to function with a good performance in terms of response times and processing capacities, but they are also expected to deliver an error-free experience. Taking high-speed trains for example, it is frustrating for travelers when a train is canceled due to technical problems. This is all the worse, if the problem is caused by a preventable bug in the software. Therefore, it is crucial that formalisms, techniques and tools are provided, which guarantee the correctness and well-functioning of such systems with mathematical rigor.

This masters thesis introduces such a technique, namely *LTL software model checking*, and describes how it is implemented in the java-written framework CPAchecker, an open-source project that allows software verification for programs written in C.

Verification is a form of software quality control, in which the software system is checked for satisfaction of the requirements that have been identified. To put it in other words, verification checks that "we are building the product right", i.e., that the program achieves its goal without the occurrence of any bugs.

Model Checking thereby allows to formally verify software programs in an automated fashion. Both the idea and the term were introduced independently in the early eighties by Clarke and Emerson in [18], and Queille and Sifakis in [40]. Essentially, the problem it describes boils down to: Given a model of a system, check algorithmically whether this model fulfills a given specification. The model is usually represented as a *transition system* (TS), i.e. a directed graph that consists of states and transitions, because it allows the formal description of the behavior of a system in an unambiguous and mathematically precise way. The model is thereby automatically generated from an appropriate dialect or extension of programming languages like Java or C. Having such a TS produced, the model checking technique then explores all possible states of the system in a systematic way. This makes it possible for a model checker to give a final verdict on whether the system model truly satisfies the desired property. Should it be possible to encounter a state in which the property does not hold, a counterexample can be created, which acts as a witness for how the undesired state could be reached by the model. The counterexample is an execution path in the TS that starts in an initial state and contains all the states that are necessary to finally reach the violated state. In software model checking, the violated state usually denotes a bug in the program, in which the counterexample can then be used to understand and fix the underlying problem.

In order to guarantee a rigorous verification, the specification is likewise required to be formulated in an unambiguous and precise formal language. The established way to express such properties are temporal logics. They state explicitly what the system should and should not do. The most common checked properties are:

- Functional correctness: Does the model work the way it is supposed to?
- Reachability: Is there an undesirable state reachable from an initial state?
- Safety: "Something bad will never happen".
- Liveness: "Something good will happen eventually".
- Fairness: Under special circumstances, does an event take place repeatedly?

The temporal logic used in this work for specifying properties is *linear temporal logic* (LTL), according to the name of the verification technique used: *LTL software*

model checking. More specifically, the focus lies on verifying C-programs for general liveness properties.
The goal of this thesis is to implement the above described concept in CPAchecker, a tool for verifying software programs written in C. It is based on the *configurable program analysis*-concept (CPA), that allows to express different verification techniques in a single formalism. Many analyses of well-known concepts of verification methods are implemented using this framework. The major success of CPAchecker has now been proven for several years in the software verification competition SV-COMP, where it has continuously won in various categories, amongst others the category "Overall" [1] for five times.

The thesis is structured as follows: Chapter 2 describes some of the related work that has been made in the area of LTL software model checking so far. Chapter 3 then shows by means of a concrete example how this is approached in theory with respect to CPAchecker. In Chapter 4, a background about chosen topics is provided which might be helpful in further understanding the various components described thereafter in this work. Chapter 5 presents the core algorithm that was developed during this thesis, and describes in detail the elementary components that are required in order to perform LTL software model checking in CPAchecker. Chapter 6 presents some of the technical concepts that are of particular interest, as they are – in this form – completely new in the CPAchecker project. This refers most notably to the process of transforming LTL formulas into automata from its framework, as well as the implementation of so-called *interpolation* automata that allow the execution of *trace abstraction refinements*. Chapter 7 finally draws a conclusion and provides an outlook about possible optimizations for LTL software model checking in CPAchecker.

[1] https://cpachecker.sosy-lab.org/achieve.php (last accessed on March 08, 2019)

CHAPTER 2

Related Work

The techniques used in this work to perform LTL software model checking are based on the methods in "*Fairness Modulo Theory: A New Approach to LTL Software Model Checking*" [23]. This paper was published at the University of Freiburg by Dietsch, Heizmann, Langenfeld, and Podelski, and introduces the method of checking finite prefixes before considering the full infinite path in order to perform LTL model checking. Only if no such infeasibility argument exists, they proceed by searching for a termination argument. The reason behind this approach is the fact that constructing a proof for unsatisfiability is much cheaper than the proof of termination via the construction of a ranking function.

In case the infinite path is not executable for either of the two reasons mentioned above, a trace abstraction is subsequently performed, in which the reason for infeasibility is generalized. This allows to (possibly) exclude plenty of traces in the next refinement iteration, namely all of which are not executable for the same reason of infeasibility. The process of performing a trace abstraction is however quite involved, which is therefore not part of the above mentioned paper. Instead, the reader is referred to [29] and [30], respectively, in which the methods are elaborated extensively.

The whole concept was implemented in the tool ULTIMATE LTL AUTOMIZER[1], which demonstrates the success of this approach.

[1]`https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=ltl_automizer` (last accessed on March 28, 2019)

In TERMINATOR [21], they state to have developed the first known fully automatic verification tool (i.e., first known to the best of their knowledge), that is able to check infinite-state programs for liveness properties. The approach there boils down to reducing the problem of liveness checking into fair termination checking. This is similar to the methods used in this work and ULTIMATE LTL AUTOMIZER, except that for each infinite path a (costly) termination check is performed. This is what we want to avoid by means of doing a check for infeasible finite prefixes beforehand. In case of success, this allows us to do a general refinement in which all fair paths are excluded that are unsatisfiable for the same reason of infeasibility.

Simple Promela Interpreter (SPIN) [32] is possibly the best known tool among the publicly available ones, which has had its development started back in the early eighties, making it one of the first model checking tools available. Its main target is the verification of program models that are written in the *Promela* modelling language. Promela is thereby short for *Process Meta Language*. The property to be checked can be specified as LTL formula, which will be afterwards compiled into a Büchi automaton. SPIN has its own format for expressing such Büchi automata introduced, which is nowadays commonly known as Promela *never claim*. However, despite the popularity of SPIN, it is still rather time-consuming and cumbersome to remodel given code-bases into the Promela modeling language.

NUSMV is a reimplementation and extension of SMV (*Symbolic Model Verifier*) [35], and uses either SAT-based bounded model checking or BDD-based symbolic model checking in order to verify temporal logic properties. While SMV could originally only handle CTL properties on a symbolic model, NUSMV is additionally able to verify program-models against properties given as LTL formulas. However, the verification process is effectively done by reducing LTL model checking to CTL model checking.

CHAPTER 3

LTL Software Model Checking in CPAchecker

3.1 The Methodology

The common approach for LTL software model checking is to build transition systems that represent both the model and the specification. In CPACHECKER, the model is represented by a so-called *control-flow automaton* (CFA). This is a labeled directed graph, where the nodes represent reachable states from a C-program, and the edges are accordingly the transitions in between these states. The specification is provided as LTL-formula, and hence needs to be translated first into a finite state machine (or more precisely, a *Büchi automaton*, in which the final states are visited infinitely often). Both transition systems are then combined afterwards, such that a new kind of transition system is created as a result. In the remainder of this work, this resulting TS will be called *Büchi program*. Depending on the kind of specification, there are different approaches in order to show that the model satisfies the specification. For safety properties, it must be shown that there exists no execution path to an error state along a feasible path. A path is feasible if there exists an executable sequence of states that begins in an initial state and ends up in the final target state. A sequence of states is thereby executable if the corresponding logical formula is computable by an solver for *satisfiable modulo theory*. This concept is called *Reachability Modulo Theory*, and was introduced by Lal and Qadeer 2013 in [34].

For liveness properties, the approach slightly differs. It must be shown that in fact no infinite path exists in the Büchi program, in which the specified state is never reached. In its simplest form, a counterexample is a loop that represents such a path and that does not contain the desired state. However, this is rather difficult to prove, because this path might under given circumstances be very well just a finite prefix that can be extended to another infinite path, which then satisfies the specification once again.

Therefore, to tackle this issue, the Büchi automaton is not created from the original LTL-formula, but instead from its negated form. The resulting Büchi program, i.e. the product of the Büchi automaton and the input program, consists of a distinguished set of nodes. These are used to define infinite paths that visit the set of nodes infinitely many times. Such paths are also denoted as *fair*. In order to show that the original LTL-property holds, the approach is now to not prove the existence of feasible fair paths, but instead to prove the opposite, namely their absence. In other words, if the Büchi program does have a path that is both fair and feasible, this is a violation of the LTL property, and a concrete counterexample for the given program. Otherwise, if the absence of such paths can be shown for the whole Büchi program, the specification then holds and it is proven that the C-program is free of bugs.

To be able to show the absence of infinite paths, it is necessary to prove that each of them eventually terminates. Thus, it is required to compute a *ranking function*. Fig. 3.1 depicts this exemplarily: For the two paths τ_1 and τ_2, a possible ranking function r is $r(x,y) = x - y$.

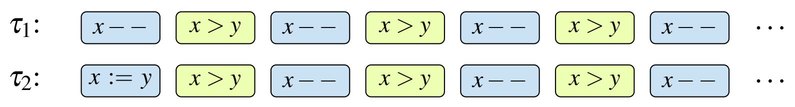


Figure 3.1: Two sequences of statements with a common possible ranking function.

The sequence of states in τ_1 is executable, for as long as x is greater than y. In comparison , in path τ_2 are the first two statements $\boxed{x := y}\boxed{x > y}$ already not feasible, despite its ranking function being fully valid.

There are open-source tools available that compute ranking functions for infinite paths (e.g. [6, 13, 39]), just like in the example above, i.e. such as the infinite input sequences $(\boxed{x := y}\boxed{x > y})^\omega$ and $\boxed{x := y}\boxed{x > y}(\boxed{x := y}\boxed{x > y})^\omega$. However, according to [23], it is always more costly to create a ranking function than to prove the unsatisfiability of a logical formula corresponding to finite prefixes. This is the case in τ_2. Its infinite sequence of statements contains a finite prefix in which the first two statements already contradict each other. This makes the path unexecutable and is sufficient for a *proof of unsatisfiability*. It is hence rendered unnecessary to continue creating a ranking function.

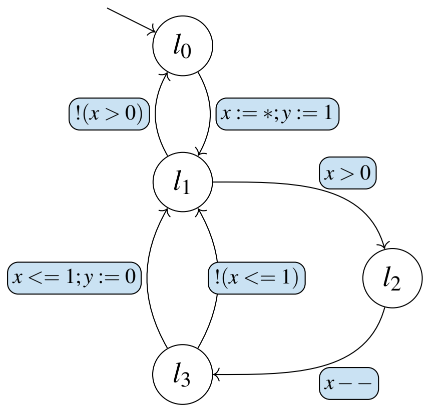
This fact will be exploited in this work. Instead of directly creating a ranking function for a full infinite path, it is always checked first if there exist finite prefixes on the path which are already contradicting each other. Modern SMT-solver are not only able to show the infeasibility of logical formulas, they also provide *interpolants* (e.g. [15, 17]), which can then be used to further generalize the proof of unsatisfiability.

3.2 Motivating Example

In this section it is illustrated by means of an example how LTL software model checking is performed in CPACHECKER. Fig. 3.2a shows the pseudo-code for a program written in C on the left. The program essentially contains two signed integer variables x and y, in which the former gets a non-deterministic value assigned (i.e., a random value from the whole range of positive and negative integer numbers), while the latter is set to 1 for the moment. All the program then does is to continuously decrement the value x for as long as it is strictly positive. Only afterwards, when the point is eventually reached where the value of x is less or equal to 1, the variable y is set to the value 0. On the right in Fig. 3.2b, a CFA is depicted that represents the C-program. The graph is a finite state machine in which the transitions are labels from the program statements, and where the states consist only of non-accepting states (in this example).

```
int x, y;
while (1) {
  x := *;
  y := 1;
  while (x > 0) {
    x--;
    if (x <= 1) {
      y := 0;
    }
  }
}
```

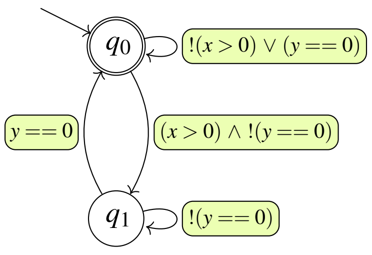
(a) Pseudo-code of a C-program.



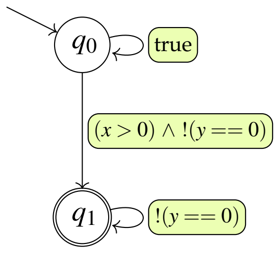
(b) A control-flow automaton.

Figure 3.2: Example of a C-program P which is shown in (a) on the left, while (b) on the right depicts the corresponding representation as CFA.

The specification will be given as LTL property $\varphi = \Box(x > 0 \Rightarrow \Diamond(y = 0))$. In essence, the program shall satisfy the condition, that for all times while x is greater than 0, it follows that eventually the variable y is set to 0. This is expressed in the Büchi-automaton in Fig. 3.3a, which is equivalent to the LTL property. However, as the property has the form of a liveness-property, a Büchi-automata of the negated LTL-formula is instead required to be able to check whether the specification holds. The automata on the left is therefore depicted for the sake of completeness, while the one on the right is the actual one that is required in order to proceed.



(a) Büchi automaton $\mathcal{A}_\varphi$.



(b) Büchi automaton $\mathcal{A}_{\neg\varphi}$.

Figure 3.3: Fig. (a) on the left shows a Büchi automaton $\mathcal{A}_1$ for the LTL-property $\varphi = \Box(x > 0 \Rightarrow \Diamond(y = 0))$. In this graph, the state q_0 is both the initial- and accepting state. In comparison, Fig. (b) on the right shows a Büchi automaton $\mathcal{A}_2$ for the negated LTL property $\neg\varphi$. The accepting state is q_1, and is simultaneously a sink state.

The Büchi automaton in Fig. 3.3b was thus built using the negated LTL formula. The syntax and semantics of LTL are defined in Sect. 4.1. Formally, the LTL-formula is negated as follows:

$$\begin{aligned}
\neg\varphi &= \neg\Box(x>0 \;\Rightarrow\; \Diamond(y==0)) \\
&= \Diamond\neg(x>0 \;\Rightarrow\; \Diamond(y==0)) && //\text{with } \neg\Box\varphi \equiv \Diamond\neg\varphi \\
&= \Diamond\neg(\neg(x>0) \;\lor\; \Diamond(y==0)) \\
&= \Diamond(\neg\neg(x>0) \;\land\; \neg\Diamond(y==0)) \\
&= \Diamond(x>0 \;\land\; \Box\neg(y==0)) && //\text{with } \neg\Diamond\varphi \equiv \Box\neg\varphi
\end{aligned}$$

In the first step the input program and the LTL property both need to be translated into finite automata, as seen above. The next step is then to create the cross-product of these, which results in a Büchi program $\mathcal{B}$. This is depicted in Fig. 3.4. The primary goal in this Büchi program $\mathcal{B}$ is to show that there exists no path that is both fair and feasible. Or to put it in other words, that no path exists that is executable and at the same time able to reach the target locations infinitely often. A target location is an accepting state which is marked in the graph with double circles. If the absence of such paths can indeed be proven, this corresponds to no feasible and fair path in the input program $\mathcal{P}$. Hence it can be concluded that the specification φ holds and our program returns SAFE.

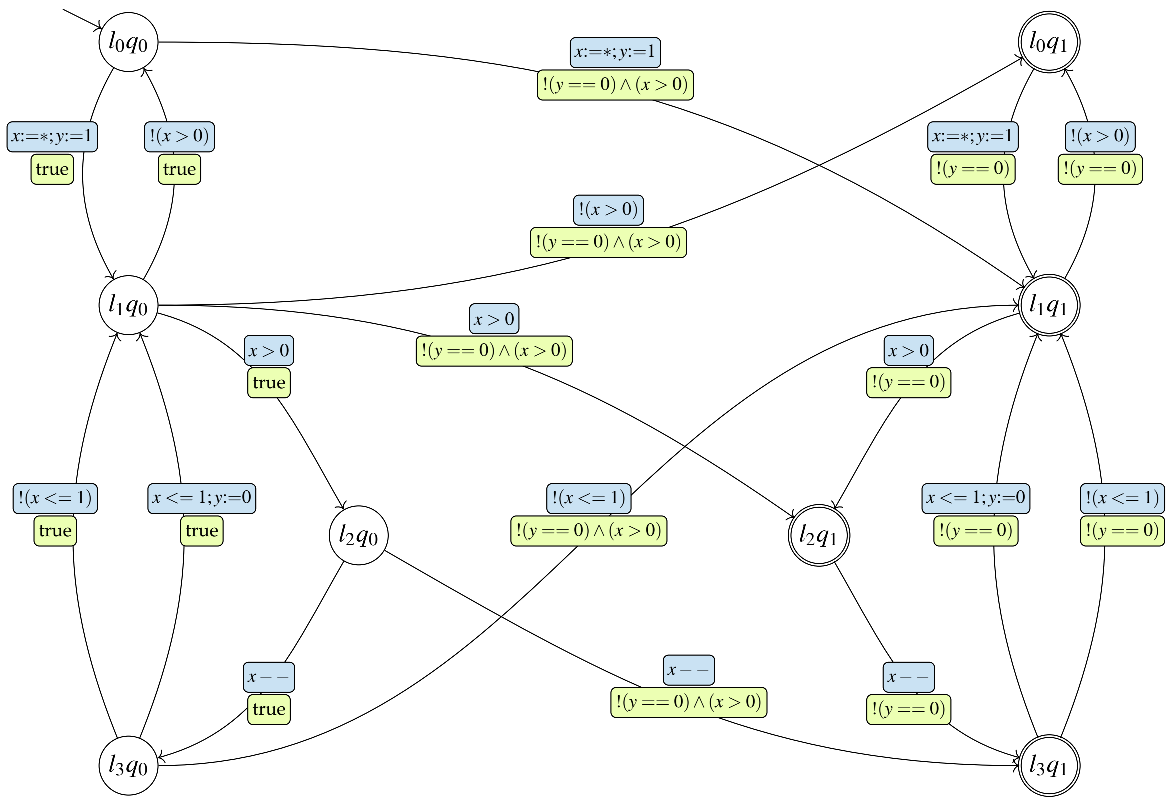


Figure 3.4: The Büchi Program $\mathcal{B}$ is built from the product of the CFA that represents the input program $\mathcal{P}$ (cf. Fig. 3.2b) and the Büchi automaton $\mathcal{B}$ with the property $\neg\varphi$ (cf. Fig. 3.3b).

The Büchi program is built as follows: The locations are pairs (l_iq_i) that result from the cross-product of the CFA locations l_i in $\mathcal{P}$ and the locations of the Büchi automaton q_i in $\mathcal{A}_{\neg\varphi}$. Likewise, the transition labels do also come as pairs $\boxed{s_1}$ $\boxed{s_2}$. The first element $\boxed{s_1}$ stands for a statement of the C-program, and is continuously colored blue in the example graphs (i.e., the CFA and the Büchi program). The second element $\boxed{s_2}$ is an element from the Büchi-automaton, which comes always as an assumption. For a better distinctiveness, these are colored green in the graphs.

In paper [23], a special notion for an infinite sequence of statements is introduced, which is called a *trace* and described as a key concept. In comparison, the term *path* is used to describe a (possibly infinite) sequence of nodes. Analogous to infinite paths is a trace fair, if the labeling corresponds to a set of nodes that are visited infinitely often. Moreover, a trace is called feasible, if the associated path is executable by the original input program $\mathcal{P}$.

A trace τ always comes in the form $\tau_1\tau_2^\omega$, in which τ_1 is a finite prefix and τ_2 an infinite sequence of statements. An example of a fair trace τ in the Büchi program B is as follows:

$$\begin{aligned}\tau_1: &\quad \boxed{x{:=}{*};y{:=}1}\ \boxed{!(y==0)\wedge(x>0)}\ \boxed{!(x>0)}\ \boxed{!(y==0)}\\ \tau_2: &\quad \boxed{x{:=}{*};y{:=}1}\ \boxed{!(y==0)}\ \boxed{!(x>0)}\ \boxed{!(y==0)}\end{aligned}$$

The trace is fair, because the locations (l_0q_1) and (l_1q_1) are both accepting and at the same time traversed infinitely many times. Yet it is not feasible, because in τ_1 the second statement $\boxed{!(y==0)\wedge(x>0)}$ and the third statement $\boxed{!(x>0)}$ contradict each other. Thus every trace in the Büchi program $\mathcal{B}$, that is the labeling of the finite prefix τ_1 is infeasible.

In general, three possible types of infeasibilities are possible. In paper [23], they are called 1.) "*Local infeasibility*", 2.) "*Infeasibility of a finite prefix*", and 3.) "*ω-Infeasibility*".

Local infeasibility: The name refers to the fact that the infeasibility affects only single edges in a Büchi program. More precisely, the labels of the edges consist of pairs, in which the first element is the program statement and the second the assume-edge from the Büchi automaton, and these two elements contradict each other. An example in the Büchi program $\mathcal{B}$ is the following edge:

$$l_1q_0 \quad \boxed{!(x>0)}\ \boxed{!(y==0)\wedge(x>0)} \quad l_0q_1$$

Each trace that traverses along a path with this label is infeasible, because $\boxed{!(x>0)}$ is contradictory to $\boxed{x>0}$. In the Büchi program $\mathcal{B}$, there exists a second such case where the concept of local infeasibility applies, namely the transition from (l_3q_1) to (l_1q_1). The label contains the two statements $\boxed{y{:=}0}$ and $\boxed{!(y==0)}$, which are not executable either.

Infeasibility of a finite prefix: A trace with finite prefixes that eventually traverses along the following sequence of statements is infeasible:

l_0q_0	$\boxed{x{:=}*; y{:=}1}$	$\boxed{\text{true}}$	l_1q_0
l_1q_0	$\boxed{x > 0}$	$\boxed{\text{true}}$	l_2q_0
l_2q_0	$\boxed{x--}$	$\boxed{!(y == 0) \wedge (x > 0)}$	l_3q_1
l_3q_1	$\boxed{!(x <= 1)}$	$\boxed{!(y == 0)}$	l_1q_1
l_1q_1	$\boxed{!(x > 0)}$	$\boxed{!(y == 0)}$	l_0q_1

This is due to the last two statements $\boxed{!(x <= 1)}$ and $\boxed{!(x > 0)}$ from the input program contradicting each other. The trace $\tau_1\tau_2^\omega$ that was presented before is another example for an infeasibility of a finite prefix.

ω-Infeasibility: Every trace that eventually ends up in the following loop is infeasible as well:

l_1q_1	$\boxed{(x > 0)}$	$\boxed{!(y == 0)}$	l_2q_1
l_2q_1	$\boxed{x--}$	$\boxed{!(y == 0)}$	l_3q_1
l_3q_1	$\boxed{!(x <= 1)}$	$\boxed{!(y == 0)}$	l_0q_1

The loop can only be traversed for as long as the value of x is greater than 1. However, the loop is bounded to terminate, because $\boxed{x--}$ will decrease the value of x until it eventually contradicts the statement $\boxed{!(x <= 1)}$. The synthesized ranking function is $f(x) = x$, which is the formal termination argument.

For the Büchi program $\mathcal{B}$ depicted in Fig. 3.4, it can be proven that each of the fair traces is not feasible for one of the described reasons above. This relates to our input program $\mathcal{P}$ not being able to traverse such a path either, thus showing that there is no possibility to somehow violate the LTL property. Thus, we have proven that the specification is indeed satisfied.

CHAPTER 4

Preliminaries

The intuitive meaning of software model checking was already described in the introduction. It answers the question, whether a model of a program $\mathcal{P}$ satisfies a specification φ. This is formally written as $\mathcal{P} \vDash \varphi$ and pronounced "$\mathcal{P}$ is a model for φ".

In comparison to other verification techniques like automated theorem proving or proof checking, model checking comes with a number of distinct advantages:

- The process is fully automatic, and requires minimal human intervention. In particular, no interaction is required during the execution, which also means that this verification technique is well suited for less experienced users.
- Compared to other methods, Model checking is relatively fast in practice. Consider e.g. a proof-checker, which involves the manual construction of proofs. This requires interactive work with the user, and is thus both highly time-consuming, while at the same time extremely error prone.
- In case of failure, a counterexample is provided, whose execution trace is invaluable in order to track down the reason as to why the specification did not hold.

Model checking is beyond that able to cover all possible behaviors of the system. This is in contrast to techniques such as testing or simulation, where only a single behavior is considered at a time. The reason is that in the approach of model checking, an exhaustive exploration of the state-space is performed. However, this leads to the most serious drawback of model checking, which is known as the *state explosion problem*. It usually occurs in large systems with of a huge number of states, because the size of the state-space can grow (at least) exponentially in the number of its processes and variables. This results in the model being too large to fit in the available amount of computer memory. A solution to this problem is an ongoing scientific process, and one of the main driving forces behind the continuous efforts in the research of model checkers and the used techniques.

The process of model checking is done in the following different phases:

1. *Modeling*: In the first step, both the specification and the program must be converted into a formalism that is accepted by a model checker. In CPAchecker, for the former this is done using the property specification language LTL, which is afterwards converted into a Büchi automaton that accepts the same language of words. LTL and Büchi automata are defined in Sect. 4.1 and Sect. 4.3, respectively, and is mainly based on [5]. The program is formalized in form of an control-flow automaton (CFA), which is described in Sect. 4.4.1.

2. *Running phase*: In the next phase, the model checking algorithm is performed to check the validity of the property in all states of the program model. The process itself is executed fully automatic. This is elaborated in **??**.

3. *Analysis*: Depending on the outcome of the last phase, the user may need to manually analyze the result. In case the property is valid, it can be concluded that the model is indeed satisfied. If however the result is that the property does not hold, the counterexample needs to be analyzed in order to track down the source of the bug.

4.1 Linear Temporal Logic

In order to express a specification, there are different temporal logics in use. This section introduces linear temporal logic (LTL), and defines its syntax and semantics. LTL was introduced by Amir Pnueli in [27, 38] and is a propositional logic that is extended by modalities referring to time. However, there is no explicit notion of time, i.e. an exact timing of events is not supported. Instead, these modalities rather allow to specify the relative order, in which these events occur. An example would be that a condition either holds at all times, or that it becomes true at least once eventually. The temporal modalities provide operators to describe these events along a single path, with the following two operators being the most common ones:

$\Diamond$	*"Finally"* – a property is satisfied at some point in the future
$\Box$	*"Globally"* – a property is satisfied now and forever in the future

In literature, there are partly different expressions used for the above modalities. The modal operator $\Diamond$ is often also referred to as *eventually*, whereas the operator $\Box$ is commonly also denoted with *always* or *henceforth*. Most of the temporal modalities have additionally both a textual and a symbolical form. For example, "$\Diamond\varphi$" is a symbolical expression and has the same meaning as the textual "$F\varphi$", and the same is true for "$\Box\varphi$" and "$G\varphi$"

LTL is a *linear-time* logic. Another common kind of temporal logic is the *branching-time* logic. Two popular such logics in computer science are *computational tree logic* (CTL), which was introduced by Clarke and Emerson in [18], and *CTL** from Emerson and Halpern, which was introduced in [25]. Therein, the model can be regarded as a tree-like structure, however, the future is not determined. That is, there are different paths, and it is unknown from the starting state which one of them will finally be taken. In LTL, in comparison, the formulas contain a single universal quantifier, meaning that a path formula f holds for every such path. All formulas in LTL are thus implicitly universally quantified.

4.1.1 Syntax

LTL formulas are constructed from a finite set of propositional variables AP, the logical operators like conjunction $\wedge$ and negation $\neg$, and the basic temporal modal operators X (pronounced *next*) and U (pronounced *until*). The atomic proposition $p \in AP$ stands for a state label in the set APs. In the context of this work, the atomic proposition is thus an assertion about values from variables of a C-program, such as e.g. "$x > 0$" or "$y! = 0$. The X-modality is an unary prefix operator, which takes one LTL formula as argument. Formally, $X\varphi$ holds in the current state, if φ holds in the next state. The U-operator is a binary infix operator, and requires two LTL formulas as arguments. The modality $\varphi_1 U \varphi_2$ holds for two LTL formulas φ_1, φ_2, if φ_1 holds from the current moment for at least so long, until φ_2 is applied.
LTL formulas over the set AP of atomic propositions can be defined inductively as follows:

- if $p \in AP$, then p is an LTL formula
- if φ_1 and φ_2 are LTL formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $X\varphi_1$, and $\varphi_1 U \varphi_2$ are LTL formulas.

In general, the temporal modalities are either in unary prefix form or in binary infix form. An example for the former is $X\varphi$, where the operator X stands in front of the LTL-formula φ, whereas for the latter, the operator is enclosed by LTL-formulas (e.g. $\varphi_1 U \varphi_2$ for the operator U). As for the precedence order, unary operators bind stronger than binary ones. However, while $\neg$ and X bind equally strong, there are differences within the binary operators. Here, the temporal operators bind stronger than the propositional operators, i.e., U takes precedence over $\wedge$, $\vee$, and $\Rightarrow$. Parentheses are left out wherever appropriate, e.g. instead of $(\varphi_1)U(\varphi_2)$, this is henceforth simply written as $\varphi_1 U \varphi_2$. Finally, the binding of modal operators is right-associative, e.g. $\varphi_1 U \varphi_2 U \varphi_3$ has the same meaning as $\varphi_1 U(\varphi_2 U \varphi_3)$.

With the use of Boolean algebra, all of the other logical and temporal operators can be derived successively. Given two LTL formulas φ_1, φ_2, the truth-values $true$ and $false$, as well as the propositional logical connectives $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalency), and $\oplus$ (exclusive or) can be obtained as follows:

$$\begin{aligned} false &\equiv \varphi_1 \wedge \neg\varphi_1 \\ true &\equiv \neg false \\ \varphi_1 \vee \varphi_2 &\equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \Rightarrow \varphi_2 &\equiv \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \Leftrightarrow \varphi_2 &\equiv (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \\ \varphi_1 \oplus \varphi_2 &\equiv (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1) \end{aligned}$$

The two at the beginning of this section mentioned modal operators $\Diamond$ and $\Box$ can be formally derived using the until-operator U:

$$\Diamond = true\ U\ \varphi \tag{4.1}$$

$$\Box = \neg\Diamond\neg\varphi \tag{4.2}$$

From the two equations above, the finally-operator $\Diamond$ has the following intuitive meaning: $\Diamond\varphi$ is satisfied, if at some point in the future φ holds. $\Box\varphi$, on the other hand, holds, if there is no single moment in time, in which $\neg\varphi$ does not hold. This is equivalent to φ holding from now on forever.

The temporal modalities can be arbitrarily combined. This leads to nested modalities with a new meaning. The most common composite operators are the following two:

$$\Diamond\Box\varphi \qquad \text{“eventually forever } \varphi\text{” (stability, or also progress)} \tag{4.3}$$

$$\Box\Diamond\varphi \qquad \text{“infinitely often } \varphi\text{” (recurrence)} \tag{4.4}$$

Again, if only $\Box\varphi$ is considered, this means that at all points in time beginning from now on, φ must hold. However, if $\Diamond$ is added to the front, the proposition is then synonymous to $\Diamond(\Box\varphi)$, and means that $\Box\varphi$ must hold forever, but only from some time onward in the future. This is also called *stability* (also known as *non-progress*), because once $\Box\varphi$ holds, the sequence of states in which φ does not hold can never be left afterwards. $\Box\Diamond\varphi$ on the other hand states that φ will occur at least once, however, this applies at all points in time. This is the same as saying that φ holds infinitely often, and hence guarantees that there is some progress, because φ will at some point in time recur either way.

Further common composite modalities are:

$\varphi_1 \Rightarrow \Diamond\varphi_2$	“$\varphi_1$implies eventually φ_2” (response)
$\varphi_1 \Rightarrow \varphi_2 U \varphi_3$	“φ_1 implies φ_2 until φ_3” (precedence)
$\Diamond\varphi_1 \Rightarrow \Diamond\varphi_2$	“eventually φ_1 implies eventually φ_2” (correlation)

4.1.2 Semantics

An atomic proposition is a set of program states. Let $\sum$ be an alphabet over a finite set of propositional variables, in which the propositions can either evaluate to true or false. This has the same meaning as 2^{AP}, which is in the following used as expression for the alphabet. A letter A is an element of the alphabet. A word over 2^{AP} is a sequence of states $A_0A_1A_2...$ such that $\forall\, i \geq 0 :\ A_i \in 2^{AP}$. A prefix w' of the word $w = A_1A_2...$ is a finite word $B_1B_2...B_n$ with $B_i = A_i$ for all $0 \leq i \leq n$. $(2^{AP})^*$ denotes the set of finite words over 2^{AP}, while $(2^{AP})^\omega$ indicates the set of all infinite sequences over the alphabet 2^{AP}.

The semantics of LTL in the following are defined by an interpretation over words. Let φ be an LTL formula over AP. The language $Words(\varphi)$ contains all infinite words over the alphabet 2^{AP} that satisfy φ. The words induced by φ is

$$Words(\varphi) = \{\sigma \in (2^{AP})^\omega \ \mid\ \sigma \models \varphi\} \tag{4.5}$$

with the satisfaction relation $\models\ \subseteq\ (2^{AP})^\omega\ \times\ LTL$.

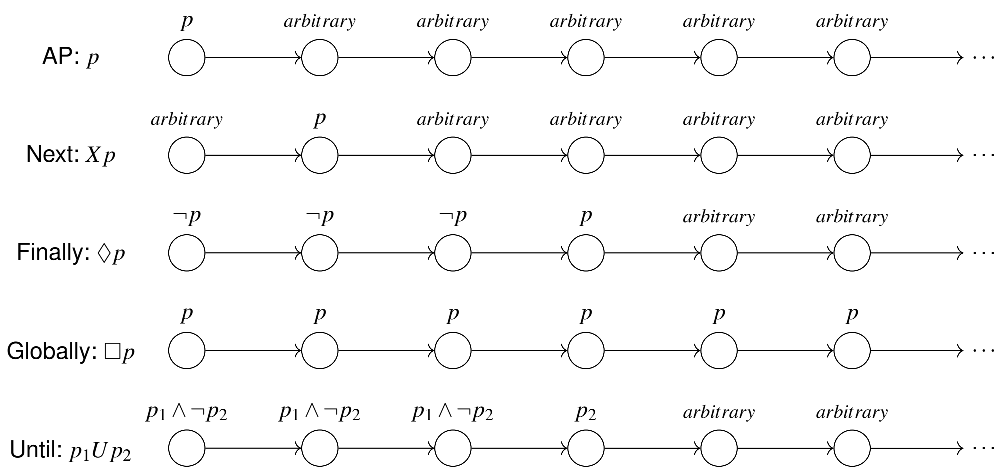


Figure 4.1: Evaluation of temporal modalities over a sequence of states. The left-hand side states the LTL formulas, while the right-hand side depicts the corresponding paths.

A linear temporal property either holds on an infinite sequence of statements, or on a single position of a word. This is illustrated in Fig. 4.1 – for an atomic proposition p, the LTL formula is fulfilled, when the property holds on the first state of the sequence, i.e. at position 0 in this case. An LT property is defined as a subset of all infinite words in the set of atomic propositions AP. A word either satisfies an LTL property or not. In correspondence with the definition given in Eq. (4.5), the semantics for each LTL property is defined subsequently in detail. In general, φ denotes an LTL formula, and σ is an infinite word from the set of APs, such that $\sigma = A_0 A_1 A_2 ... \in (2^{AP})^{\omega}$.

Boolean constant: If an LTL formula is given as truth-value *true*, a word σ is trivially always satisfied:

$$\sigma \models true \tag{4.6}$$

Atomic proposition: An atomic proposition p is satisfied, when it holds in the first statement A_0 of word $\sigma = A_0 A_1 A_2 ...$:

$$\sigma \models p \qquad \text{iff } p \in A_0 \tag{4.7}$$

Negation: The negation of an atomic proposition φ is satisfied, if for a word σ the proposition does not hold in the first statement of that word:

$$\sigma \models \neg\varphi \qquad \text{iff } \sigma \not\models \varphi \tag{4.8}$$

Conjunction: The conjunction of two LTL formulas φ_1 and φ_2 is satisfied for a word σ, if σ satisfies both φ_1 and φ_2. Formally this is defined as follows:

$$\sigma \models \varphi_1 \wedge \varphi_2 \qquad \text{iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \tag{4.9}$$

Next: The next-operator is used to specify that a word is satisfied in an immediate next state of a word. I.e., given a word $\sigma = A_0 A_0 A_2 ... \in (2^{AP})^\omega$ and an LTL specification $X\varphi$, then φ is satisfied if and only if it holds onward from the direct next statement, which is the letter A_1:

$$\sigma \models X\varphi \qquad \text{iff } \sigma[1...] = A_1 A_2 ... \models \varphi \tag{4.10}$$

Until: For two LTL formulas φ_1 and φ_2, the until-operator U states that φ_1 is satisfied for at least so long, until φ_2 is satisfied once. This is reflected in the definition, which consists therefore of two parts:

- φ_2 must hold in the word at some point in time. This is enforced by φ_1.
- Until then, φ_1 is required to hold. Note however that if φ_2 is true at all points in word σ, then φ_1 does not necessarily need to be true either for the whole until-operator $\varphi_1 U \varphi_2$ to be satisfied.

Formally, the definition of the operator U is as follows:

$$\sigma \models \varphi_1 U \varphi_2 \qquad \text{iff } \exists j \geq 0 \text{ such that } \sigma_j \models \varphi_2 \text{ and for all } 0 \leq i \leq j,\ \sigma_i \models \varphi_1 \tag{4.11}$$

The intuitive meaning of the finally- and globally-operators have already been described in equation (4.1) and (4.2). These are subsequently formally defined:

Finally: The LTL formula $\Diamond\varphi$ (also denoted $F\varphi$) is satisfied, if φ holds in a word σ at any point in its sequence of statements. The semantics of the finally-operator is derived from the until-operator, which is stated in the following as $true\ U\ \varphi$. The definition is thus immediate from Eq. (4.1) and the semantics from the until-operator:

$$\sigma \models \Diamond\varphi \qquad \text{iff } \exists i \geq 0.\ \sigma[i...] \models \varphi \tag{4.12}$$

Globally: The globally operator $\Box\varphi$ (also written as $G\varphi$) states that in a word σ, the LTL formula φ has to hold in all of its statements. Formally, the operator is defined as a derivation from the finally-operator, i.e. $\neg\Diamond\neg\varphi$. This expands to the following statement:

$$\sigma \models \Box\varphi \qquad \text{iff } \forall i \geq 0.\ \sigma[i...] \models \varphi \tag{4.13}$$

Finally-Globally and **Globally-Finally:** The intuitive meaning of the nested operators $\Diamond\Box\varphi$ and $\Box\Diamond\varphi$ are described in (4.3) and (4.4), respectively. The semantics follow directly by applying the equations (4.12) and (4.13) for the finally- and globally-operator from above:

$$\sigma \models \Diamond\Box\varphi \quad \text{iff } \exists i \geq 0.\ \forall j \geq i.\ \sigma[j...] \models \varphi \tag{4.14}$$

$$\sigma \models \Box\Diamond\varphi \quad \text{iff } \forall i \geq 0.\ \exists j \geq i.\ \sigma[j...] \models \varphi \tag{4.15}$$

4.1.3 Equivalences

As LTL extends propositional logic, all laws for logical equivalences are likewise valid, such as the commutative properties (e.g. $p \vee q \equiv q \vee p$), the associative properties (e.g. $(p \vee q) \vee r \equiv p \vee (q \vee r)$), or the laws of De Morgan (e.g. $\neg(p \wedge q) \equiv \neg p \vee \neg q$). Generally speaking, two logical formulas are equivalent, whenever their truth-values are the same in all models. In the exact same manner do these equivalence rules also exist for temporal modalities. Two LTL formulas φ, ψ are said to be equivalent, written $\varphi \equiv \psi$, if for all words induced by φ and all words induced by ψ the following holds: $Words(\varphi) = Words(\psi)$. Below are some of the most common equivalence rules stated with respect to the temporal modalities:

Negation propagation:

$$\begin{aligned} \neg X\varphi &\equiv X\neg\varphi \\ \neg\Diamond\varphi &\equiv \Box\neg\varphi \\ \neg\Box\varphi &\equiv \Diamond\neg\varphi \end{aligned}$$

Idempotency:

$$\begin{aligned} \Diamond\Diamond\varphi &\equiv \Diamond\varphi \\ \Box\Box\varphi &\equiv \Box\varphi \\ \varphi U(\varphi U\psi) &\equiv \varphi U\psi \\ (\varphi U\psi)U\psi &\equiv \varphi U\psi \end{aligned}$$

Absorption:

$$\begin{aligned} \Diamond\Box\Diamond\varphi &\equiv \Box\Diamond\varphi \\ \Box\Diamond\Box\varphi &\equiv \Diamond\Box\varphi \end{aligned}$$

Expansion:

$$\begin{aligned} \varphi U \psi &\equiv \psi \vee (\varphi \wedge X(\varphi U \psi)) \\ \Diamond\varphi &\equiv \varphi \vee X\Diamond\varphi \\ \Box\varphi &\equiv \varphi \wedge X\Box\varphi \end{aligned}$$

Distributivity:

$$\begin{aligned} X(\varphi \vee \psi) &\equiv (X\varphi) \vee (X\psi) \\ X(\varphi \wedge \psi) &\equiv (X\varphi) \wedge (X\psi) \\ X(\varphi U \psi) &\equiv (X\varphi) U (X\psi) \\ \Diamond(\varphi \vee \psi) &\equiv \Diamond\varphi \vee \Diamond\psi \\ \Box(\varphi \wedge \psi) &\equiv \Box\varphi \wedge \Box\psi \end{aligned}$$

All of the equivalences can be formally derived. As an example, the equivalence between $\neg\Diamond\varphi \equiv \Box\neg\varphi$ for an LTL formula φ and word σ is shown below:

$$\begin{aligned} & \sigma \models \neg\Diamond\varphi & \\ \text{iff} \quad & \neg\exists i \geq 0.\ \sigma[i...] \models \varphi & \text{(Def. of } \Diamond\text{, c.f. Eq. (4.12))} \\ \text{iff} \quad & \forall i \geq 0.\ \sigma[i...] \not\models \varphi & \text{(Def. of negation of } \exists\text{-quantor)} \\ \text{iff} \quad & \forall i \geq 0.\ \sigma[i...] \models \neg\varphi & \text{(Def. of } \neg\text{, c.f. Eq. (4.8))} \\ \text{iff} \quad & \sigma \models \Box\neg\varphi & \text{(Def. of } \Box\text{, c.f. Eq. (4.13))} \end{aligned}$$

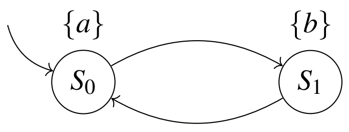


Figure 4.2: State-based transition system TS with two nodes S_0 and S_1. For $a, b \in AP$, it holds that $TS \models \Diamond a \wedge \Diamond b$, and $TS \not\models \Diamond(a \wedge b)$.

An peculiarity for the distribution laws is the duality between $\Diamond$ and disjunction, and $\Box$ and conjunction, respectively. This is insofar important, as when interchanging the logical operators $\wedge$ and $\vee$ for the respective equations, the equations are no longer equivalencies:

$$\Diamond(\varphi \wedge \psi) \not\equiv \Diamond\varphi \wedge \Diamond\psi \quad \text{and} \quad \Box(\varphi \vee \psi) \not\equiv \Box\varphi \vee \Box\psi$$

This is demonstrated in Fig. 4.2. The transition system TS has two states S_0 and S_1, in which the former contains the letter a and the latter the atomic proposition b. It is apparent that the transition system satisfies $\Diamond a \wedge \Diamond b$, since both of the states S_0 and S_1 will eventually be reached. However, because a and b will never hold concurrently within one state, the formula $\Diamond(a \wedge b)$ is hence also never fulfilled, thus showing that the two properties are not equivalent. For the same reason, this also applies to $\Box(a \vee b)$ and $\Box a \vee \Box b$.

4.2 Safety and Liveness

In linear temporal logic, there are two main classes of properties that can be expressed using this specification language. These are called *safety*-properties and *liveness*-properties. Alpern and Schneider proved in [2] that every temporal formula can be written as the conjunction of a safety and a liveness property, thus making these two types of properties so fundamental. They are subsequently elaborated in more detail.

Safety properties: These express that "nothing bad" will happen, ever, during the execution of a program. In general, a safety property evaluates to true for an infinite behavior, if (and only if) it is true for every finite prefix of that behavior. There are two possible ways to formulate them, which is either by

1. stating the illegal executions, i.e. "what may not happen", or alternatively by
2. stating the legal executions, i.e. "what may happen" (though, this is not necessarily required to happen)

The complement of the legal executions are the illegal executions. It is more often both easier and more reliably to state the legal runs, because if something bad were to happen, the violation of the property would have to occur within a finite number of states. A formal definition for safety properties was provided by Alpern and Schneider in [3] as follows:

$$\forall \sigma \in S^\omega : \ \sigma \models \varphi \qquad \text{iff } \forall i \geq 0 : \ \exists w \in S^\omega : \ \sigma[0..i]w \models \varphi \tag{4.16}$$

where φ is the property, S the set of program states (which is 2^{AP} in our case), S^* the set of finite sequences of states, and S^ω the set of infinite sequences of states. The above definition states that the safety property φ is satisfied by an infinite word $\sigma \in S^\omega$, if and only if this is true for each finite prefix $\sigma[0..i]$. Otherwise, if a violation occurred, there exists a finite prefix σ such that the end of the prefix at position i, denoted $\sigma[i]$, would then mark the point in time where the violation first happens. This is a concrete counterexample in which the violation is witnessed, and no matter how often this finite prefix is extended to another infinite path, it still witnesses the violation of the desired property.

Liveness properties: These kind of properties express that "something good" will happen, eventually, during the execution of a program. However, due to their nature, they can never be refuted by observing only a finite sequence of statements. More precisely, the occurrence of "something good" does not even have to be observable in a fixed interval of time. Instead, it is fully possible that this first happens at a later stage. Every finite prefix of an infinite sequence of statements can therefore be extended to an infinite trace, in which the statements then satisfy the property again. A formal definition is as follows (cf. Alpern and Schneider in [3]):

$$\forall \sigma \in S^* : \ \sigma \vDash \varphi \qquad \text{iff } \exists \beta \in S^\omega : \ \sigma\beta \vDash \varphi \tag{4.17}$$

where again φ is the property, S the set of program states, S^* the set of finite sequences of states, and S^ω the set of infinite sequences of states. The equation states that the property φ is a liveness property, if and only if each finite sequence σ of states can be extended to an infinite sequence of statements such, that the composition $\sigma\beta$ fulfills the specification. In contrast to safety properties, liveness does not stipulate that "something good" always happens, only that it does so eventually. The end of σ, i.e. the last statement of the finite prefix does thereby mark the point in time, where something good happens such that the specification φ is satisfied.

4.3 Büchi Automaton

Finite automata allow to define languages over finite words. However, as seen in the previous section, the properties for our cause require automata for languages over infinite words. This can be achieved via *Büchi automatons*, which were introduced by J. R. Büchi, a Swiss logician, in [14]. These automata are finite automata over infinite words, and allow to specify LTL properties whose permitted executions represent accepting words. A Büchi automaton is defined as a five-tuple $\mathcal{A} = (S, S_0, \sum, \rightarrow, F)$, in which the contained parts have the following meaning:

- S is a finite set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $\sum$ is the finite alphabet of the automaton,
- $\rightarrow \subseteq (S \times \sum \times S)$ is non-deterministic transition relation, and
- $F \subseteq S$ is the (Büchi-) *acceptance condition*. The elements in F are called final or accepting states.

An infinite word σ over the alphabet $\sum$ is an infinite sequence $A_0A_1A_2...$ of symbols $A_i \in \sum$ for all $i \geq 0$. $\sum^{\omega}$ denotes the set of all infinite words over $\sum$. A language of infinite words, also called ω-language, is any subset of $\sum^{\omega}$. We write $s \xrightarrow{l} s'$ to denote that $(s,l,s') \in \rightarrow$.

A *run* r on a word $\sigma = A_0A_1A_2... \in \sum^{\omega}$ denotes an infinite alternating sequence of states $s_0s_1s_2... \in S$, such that $s_0 \in S_0$ and $\forall i \geq 0 : s_i \xrightarrow{l_i} s_{i+1}$. A run is thus an allowed sequence of states that an automaton may pass through while it reads the input. A Büchi automaton *accepts* a run $r = s_0s_1s_2... \in S$ if it contains infinitely many accepting states, i.e., if and only if $s_i \in F$ for infinitely many indices $i \in \mathbb{N}$. This is equivalent to $inf(r) \cap F \neq \emptyset$. A word $\sigma \in \sum^{\omega}$ is accepted by $\mathcal{A}$ if there is an accepting run on σ. The *language* $\mathcal{L}(\mathcal{A})$ of a Büchi automaton $\mathcal{A}$ is the set of all words from $\sum^{\omega}$ that are accepted by $\mathcal{A}$.

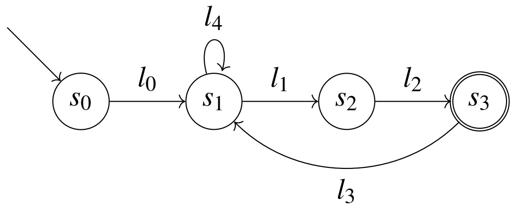


Figure 4.3: A Büchi automaton $\mathcal{A}$. An infinite accepting run is $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \xrightarrow{l_3} s_1 \xrightarrow{l_1} ...$, that ends up traversing the looping states s_1, s_2, and s_3. The set of accepting runs defines the language $\mathcal{L}(\mathcal{A})$ of the Büchi automaton $\mathcal{A}$.

According to this definition, the Büchi automaton accepts an input if there is a run along the statements of a word φ in which the accepting states F are visited infinitely often. Since the set of F is finite, there must be at least one state $s \in S$ that is infinitely often visited along the word σ. This is demonstrated in Fig. 4.3. The automaton consists of the alphabet $\sum = \{l_0, l_1, l_2, l_3\}$. It can be seen there how a possible run traces an infinite path which starts at the initial state $s_0 \in S_0$, eventually reaches the final state $s_4 \in F$, and thereafter keeps looping back to s_4 infinitely often. This run is written $s_0s_1s_2s_3s_1s_2s_3...$, or in short, $s_0(s_1s_2s_3)^{\omega}$. It is accepting, because it visits the accepting state s_3 infinitely often. The word on which the run traces along is accordingly $l_0(l_1l_2l_3)^{\omega}$.
Another example of a run is $s_0s_1^{\omega}$ for the word $l_0l_4^{\omega}$. This run is however not accepting, because the accepting state s_3 is never visited in this example. The same is true for runs of the form $s_0(s_1s_2s_3)^*s_4^{\omega}$. Here, the accepting state is visited, but only for finitely

many times. The language accepted by this (deterministic) Büchi automaton is given by the ω-regular expression

$$l_0 l_4^* l_1 l_2 (l_3 l_4^* l_1 l_2)^\omega$$

In general, an LTL property is usually denoted as a formula φ, though, it is possible to translate it into an equivalent nondeterministic Büchi automaton (NBA) [5] such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$. In the remainder of this work it is thus assumed that for each LTL property φ we have a Büchi Automaton $\mathcal{A}$ available.

4.4 CPAchecker

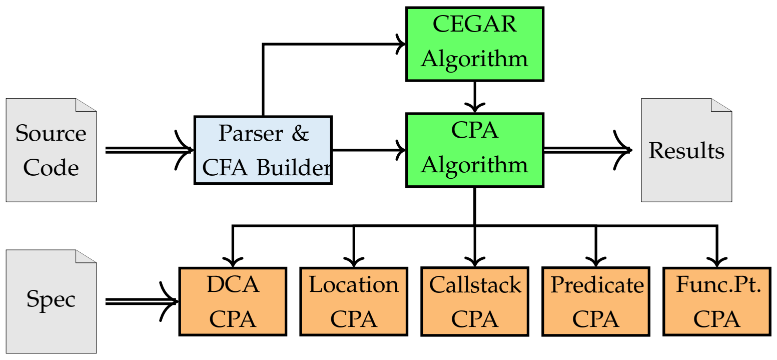


Figure 4.4: Simplified architecture of CPAchecker.

CPAchecker [10] is an open-source framework written in Java, and has the greater goal of verifying programs that are written in C. For this reason, the framework provides different approaches of program analysis techniques, such as e.g. predicate analysis [11], explicit-state model checking [12], or k-induction [7].
Fig. 4.4 gives a rough overview of the architecture of CPAchecker, with respect to the components that are relevant most for this work. For the input, a specification together with a source code written in C is required. In a first step, the source code is parsed and then transformed into an intermediate representation, which is in the case of CPAchecker a *control-flow automaton* (CFA). The concept thereof is explained in Sect. 4.4.1 in more detail. Afterwards a CEGAR algorithm will be started. CEGAR is short for *counterexample-guided abstraction refinement*, and is an abstraction technique that iteratively refines an abstract model in an automated fashion, until either a real counterexample is found, or the overall absence of a violation can be concluded, which then proves our program to be safe. For each iteration in the CEGAR algorithm, a CPA-algorithm is started that computes an *abstract reachability graph* (ARG). The ARG consists of reachable abstract states, on

which graph algorithms are later executed on, in order to check whether reachable cycles with target states exist. If such traces with cycles can be found, a check for feasibility is performed immediately after. Each of the executable traces exemplifies a concrete counterexample that witnesses a violation in our program. Otherwise, if these traces are not executable, the CEGAR algorithm will start its next iteration, in which the CPA algorithm is started from anew. The idea is to recreate the ARG, this time however united with a new finite automaton that recognizes the set of all infeasible traces from the last iteration. This allows to exclude all such violating traces in the newly computed ARG, since the automaton makes sure that the states within the cycles are no longer accepting. The whole procedure of refining the ARG is afterwards continued for as long as executable traces with loops can be found, in which the cycles within still contain accepting states.

The CPA algorithm is elaborated on in Sect. 4.4.2. In order to run successfully, it is dependent on several other *configurable program analyses* (CPAs), all of which are necessary to build the ARG: The *DCA* CPA consists of automata, including the Büchi automaton that represents the specification. Any other automaton results from an iteration step of CEGAR. These so-called *Interpolation automata* can be seen as a form of precision that only affect local states of the ARG. As for the further CPAs, the *location* CPA is used to track the program locations, the *callstack* CPA to track the function callstack, and the *function pointer* CPA is used to track function pointers in the program. The concept of CPAs is described in Sect. 4.4.3, while the functionality of CEGAR is explained in the following section (i.e., Sect. 4.4.4) in more detail.

4.4.1 Control-flow Automaton

In order to verify a program, it is usually first modeled into an intermediate representation. In CPAchecker, a control-flow automaton (CFA) is used to represent such a program, as it allows for a good abstraction of the software-based model. A CFA is a directed graph that represents program states and a finite set of program variables in blocks. This allows the traversal of all possible paths during the execution of a program.

A CFA $\mathcal{A}$ over a given set of program operations Ops is defined as $\mathcal{A} = (Loc, l_0, G)$, where Loc is the finite set of nodes called the *locations* of the program, $l_0 \in Loc$ is the initial starting location, and G is the set of control-flow edges labeled with program statements, with $G \subseteq Loc \times Ops \times Loc$. An edge (l, op, l') can be denoted $l \xrightarrow{op} l'$, and describes the control flow from a predecessor location l to a successor location l' by means of a program operation $op \in Ops$. CPAchecker supports several types of program operations Ops:

The first one is the assignment operation $x := expr$ for a finite set of program variables Var, with $x \in Var$, and $expr$ being an expression over Var. The next operation is the assume operation $[x]$, where x is a Boolean operation over Var. Further operations are a noop operation *noop*, in which the transitions are traversed without any further effect, and a function call operation together with a return operation to accordingly represent function calls. A CFA has one distinguished initial state. As for the exit states, they are in this work not required in the classical sense. To elaborate, an exit state usually means some kind of bottom state that does not have any successor state. The focus in this work lies however on infinite sequences of statements, thus only program models are considered where each location has at least one outgoing edge, such that $\forall l \in Loc, \exists op \in Ops, \exists l' \in Loc: (l, op, l') \in G$. For each location where no such outgoing transition exists, a selfloop is added using the *noop* operation.

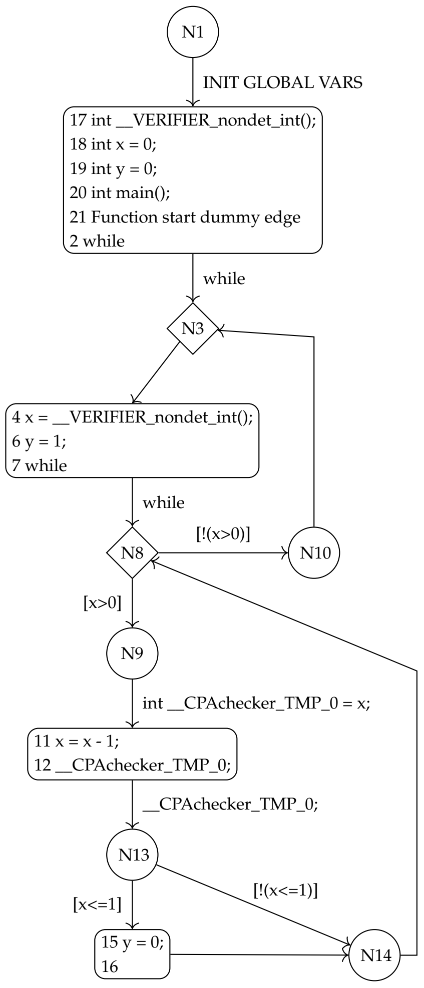


Figure 4.5: Concrete example for a CFA in CPACHECKER. It represents the program depicted in Fig. 3.2a.

Fig. 4.5 depicts the interprocedural CFA which represents the program from the example chapter Sect. 3.2 (c.f. Fig. 3.2a). The starting state l_0 in this graph is the location node N1, and every subsequent location is afterwards reachable by an edge either labeled with a statement from the program, or by a *noop* operation in which the edge label is omitted. There are no function calls in the example program, therefore neither do call and return operations occur in this CFA.

A path is a (possibly infinite) sequence of locations $l_0 l_1 l_2 ...$. The starting location l_0 of a path is always required to be the initial starting location of the CFA. A *trace* τ of a program $\mathcal{P}$ is an infinite sequence of operations $\tau = op_0 op_1 op_2 ...$ such that τ is the edge labeling of an infinite program path. The set of all traces in a program are formally defined as follows:

$$T(\mathcal{P}) = \{op_0 op_1 ... \in Ops^{\omega} \mid \exists l_0 l_1 ... : (l_i, op_i, l_{i+1}) \in \ G \text{ for all } i \geq 0\}$$

A trace τ is feasible, if for the corresponding sequence of program states $l_0 l_1 l_2 ... \in Loc$ a transition relation for each of these states in G exist. This is also called a *program execution*, i.e. we say that a trace τ is feasible if it has a program execution, otherwise we say that it is infeasible. Informally speaking, a trace is executable if it is not only syntactically, but also semantically correct.

4.4.2 CPA Algorithm

A possible concept for program analysis is the *configurable program analysis* (CPA) as introduced by Beyer, Henzinger, and Théoduloz in [8]. This was later extended to *configurable program analysis with precision adjustment* (CPA+) [9], together with an algorithm such that reachability analyses can be performed. However, as the precision used there has no practical use in this work, we will keep using the descriptions given in [8] in the remainder of this thesis.

The concept of configurable program analysis allows to automatically verify a program, in which the two major approaches of *program analysis* [1] and *model checking* [20] are combined. In [8] it is demonstrated how they are both subcases of each other, thus making this approach possible. The difference is that program analysis makes efficient yet inaccurate analyses, which means that there is a risk of producing an overwhelming number of false alarms. In contrast, model checking performs expressive analyses, such that the produced results are sound. That is, if a program is declared to be save, it is indeed safe. The downside of this approach is however the expensiveness that comes with it, i.e. it does not scale to large programs. The challenge lies thus in finding a trade-off between a good precision that keeps false positives from occurring, while at the same time aiming for a best possible efficiency such that a satisfying performance can be achieved.

Alg. 1 depicts the CPA algorithm. In general, it is a waitlist-based reachability algorithm in which a state-space exploration is performed. The main objective lies in computing a set of abstract states whom are reachable from an initial state. Using this approach, the algorithm can be used either for e.g. data-flow analysis, model checking, or anything in-between the spectrum of these two approaches.

Algorithm 1 CPA Algorithm (taken from [8], Algorithm 1)

```
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
       an initial abstract state $e_0 \in E$, where $E$ denotes the set of elements of
       the lattice of $D$
Output: a set of reachable abstract states
Variables: a set $\mathsf{reached} \subseteq E$, a set $\mathsf{waitlist} \subseteq E$
 1: $\mathsf{waitlist} := \{e_0\}$
 2: $\mathsf{reached} := \{e_0\}$
 3: while $\mathsf{waitlist} \neq \emptyset$ do
 4:   choose $e$ from $\mathsf{waitlist}$
 5:   $\mathsf{waitlist} := \mathsf{waitlist} \setminus \{e\}$
 6:   for all $e'$ with $e \rightsquigarrow e'$ do
 7:     for all $e'' \in \mathsf{reached}$ do
 8:       // combine with existing abstract state
 9:       $e_{new} := \mathsf{merge}(e', e'')$
10:       if $e_{new} \neq e''$ then
11:         $\mathsf{waitlist} := (\mathsf{waitlist} \cup \{e_{new}\}) \setminus \{e''\}$
12:         $\mathsf{reached} := (\mathsf{reached} \cup \{e_{new}\}) \setminus \{e''\}$
13:     if $\neg\mathsf{stop}(e', \mathsf{reached})$ then
14:       $\mathsf{waitlist} := \mathsf{waitlist} \cup \{e'\}$
15:       $\mathsf{reached} := \mathsf{reached} \cup \{e'\}$
16: return $\mathsf{reached}$
```

The algorithm takes as input a CPA $\mathbb{D}$ and an initial abstract state e_0. It operates on two sets *reached* and *waitlist*, that both consist of abstract states. The former contains all states that have already been visited by the algorithm, and are hence declared as "reached". The waitlist contains all states whose successor states are yet to be explored. In the beginning, only the state e_0 is included in both of the sets. For as long as the waitlist is not empty, an abstract state e is taken and removed from it. An own strategy determines which state is chosen next. The algorithm then proceeds by computing all abstract successor states e' of e, based on the transfer relation $\rightsquigarrow$ of the CPA $\mathbb{D}$. Two computations are now subsequently made: First, all abstract successors e' are *merged* with each of the already computed, reachable states $e'' \in \mathsf{reached}$ by using the merge operator of $\mathbb{D}$. As a result, a new abstract state e_{new} is created that may or may not be equal to e''. In case of the former, e'' is removed from both the reached set and the waitlist, and instead e_{new} is added to both of the sets. At this point it does not matter whether the state e'' has already been visited by the algorithm so far, i.e. this is done completely independent from e'' being in either the waitlist or the reached set. Afterwards, a second computation is carried out, which is the so-called *stop*-check. Using the stop-operator of $\mathbb{D}$, the algorithm checks now whether the current abstract state e' is already covered in the reached set. In case it is not, e' is added to both the reached set and the waitlist, such that it can be explored in a later iteration. In case there are still abstract states left to be explored in the waitlist, the algorithm then continues by picking one and analyzing it in the next iteration. Otherwise, if all abstract states have been explored, the CPA algorithm terminates and returns the reached set, which contains all of the analyzed states. In CPAchecker, an ARG is used afterwards that allows to represent these in an abstract model.

4.4.3 Configurable Program Analysis

CPAchecker is based on configurable program analyses (CPA), which provide the means for expressing different verification techniques in a single formal setting. The definitions are taken again from [8]. Formally, a CPA is defined as a tuple $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, where D is the abstract domain, $\rightsquigarrow$ the transfer relation, merge the merge operator, and stop the stop operator. These are explained in the following in more detail:

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by a set C of concrete states, a semi-lattice $\mathcal{E}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ consists of a (possibly infinite) set E of lattice elements and a partial order $\sqsubseteq \subseteq E \times E$ over the elements of E. The elements E of $\mathcal{E}$ are thereby called the *abstract states* of the analysis. The join operator $\sqcup : E \times E \to E$ of $\mathcal{E}$ is defined through $\sqsubseteq$ and denotes the least upper bound for its two parameters. The top element $\top \in E$ is the least upper bound of E. The concretization function $[\![\cdot]\!] : E \to 2^C$ allows to assign to each abstract state the set of concrete states that they represent.

2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ assigns to each abstract state $e \in E$ all possible abstract successor states $e' \in E$ with respect to control-flow edges $g \in G$ from the CFA. We write $e \overset{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if there exists a $g \in G$ with $e \overset{g}{\rightsquigarrow} e'$.

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ combines the information of two abstract states. Depending on the first element e, this may result in a new abstraction state that can be anything between the second element e' and the top element $\top$. The resulting state can hence only be more abstract than the second parameter e'. This guarantees that our analysis is sound, and is formally denoted as $e' \sqsubseteq \mathsf{merge}(e, e')$. The two merge operators used most are merge^{sep} and merge^{join}:

$$\mathsf{merge}^{join}(e, e') = e \sqcup e'$$
$$\mathsf{merge}^{sep}(e, e') = e'$$

The operator merge^{join} *weakens* the second parameter depending on the first argument. This is also called *widening*. Using this operator, the performance of the analysis can be increased at the cost of precision, since only the successor states of the newly created abstract state need to be computed subsequently (as opposed to computing the successors of both e and e'). The increase in performance is due to the resulting state (possibly) representing more concrete states. $\mathsf{merge}^{sep}(e,e')$ on the other hand does not weaken any abstract state. This merge operator always returns the second parameter e'.

4. The *stop operator* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ is used for coverage checks and is also called *termination check*. It determines whether an abstract state $e \in E$ is covered by a set of states $R \in 2^E$ that is given as second parameter. If e is covered by R, then $\mathsf{stop}(e,R)$ returns *true* and the CPA algorithm skips analyzing the successor states of e. The two stop operators used most are thereby stop^{sep} and stop^{join}:

$$\mathsf{stop}^{sep}(e,R) = \exists e' \in R : e \sqsubseteq e'$$
$$\mathsf{stop}^{join}(e,R) = e \sqsubset \bigsqcup_{e_i \in R} e_i$$

The operator stop^{sep} checks every abstract state in R separately, i.e. whether a state $e' \in R$ exists that represents at least all concrete states of e. The termination check stop^{join} first joins all states in R and then checks if this subsumes the first state e. The soundness of a termination check is achieved if the stop operator fulfills that

$$\mathsf{stop}(e,R) = \mathit{true} \text{ implies } [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$$

4.4.4 Counterexample-guided Abstraction Refinement

Model checking relies on an exhaustive exploration of the state space in order to find counterexamples. The state space can grow thereby exponentially, thus making the search both costly in memory and time. For a large program with complex structures, this typically ends up in a huge state space. In general, this issue can be tackled by computing an over-approximating abstraction of the program, such that a combinatorial blow-up (i.e. a state-space explosion) can be avoided. For this reason, the abstraction should be as coarse as possible in order to keep the state space small. However, if the abstraction is kept too coarse, this is also by no means a satisfying solution, as this might leads to false alarms. A good abstraction needs therefore to also be precise enough that such erroneous reports do not appear. With *counterexample-guided abstraction refinement* (CEGAR) [19], an abstraction technique is used that allows to iteratively refine an abstract model making use of automatically derived counterexamples.

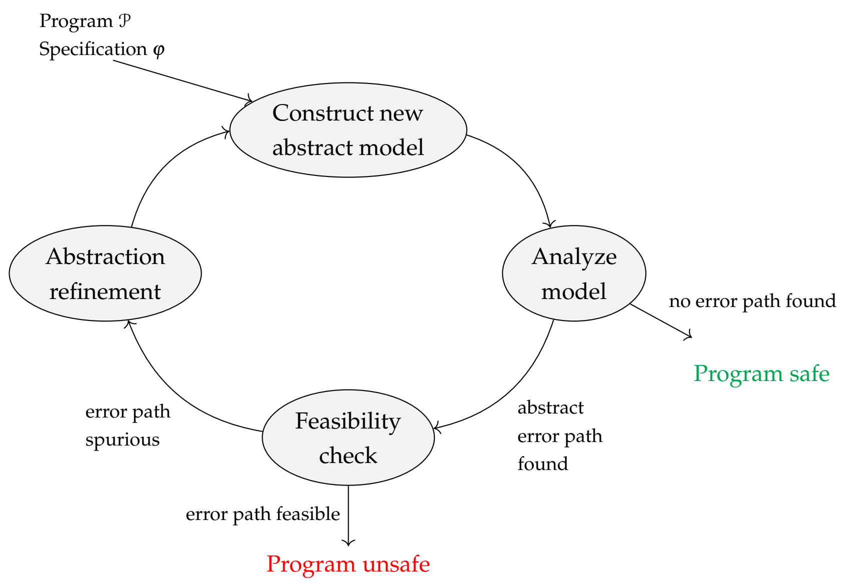


Figure 4.6: Procedural method of CEGAR.

A counterexample is an interpretation in which the violation for a given property is witnessed. In software model checking, the counterexample is thereby an error path through the program, in which the reasoning does not hold, i.e. where the premises of such a path are true while the conclusion is not. Fig. 4.6 gives an overview of the approach. Given a program $\mathcal{P}$ and a property φ, an abstract model is constructed, which is afterwards analyzed for the existence of an error path. If no such path can be found, this relates to our input program $\mathcal{P}$ having no possibility to traverse a path either in which the property does not hold. It can thus be concluded that the program $\mathcal{P}$ is safe, and the analysis therefore terminates. Otherwise, should an abstract error path be found during the analysis, it needs to be checked for feasibility next. The reasoning behind this is that the reported counterexample might be spurious, because it only exists within the abstract model of the program, not however in the program itself. The feasibility check is performed by checking whether the path is executable with respect to the concrete program semantics. Should the error path be truly feasible, the analysis terminates and reports our program to be unsafe, together with a counterexample that witnesses the violation of the specification. Otherwise, the error path is spurious, which means that the abstract model was too course. The infeasible error path is afterwards used for an abstraction refinement, that is performed on the current model. Finally, CEGAR then proceeds with the next iteration of this newly computed abstraction.

CHAPTER 5

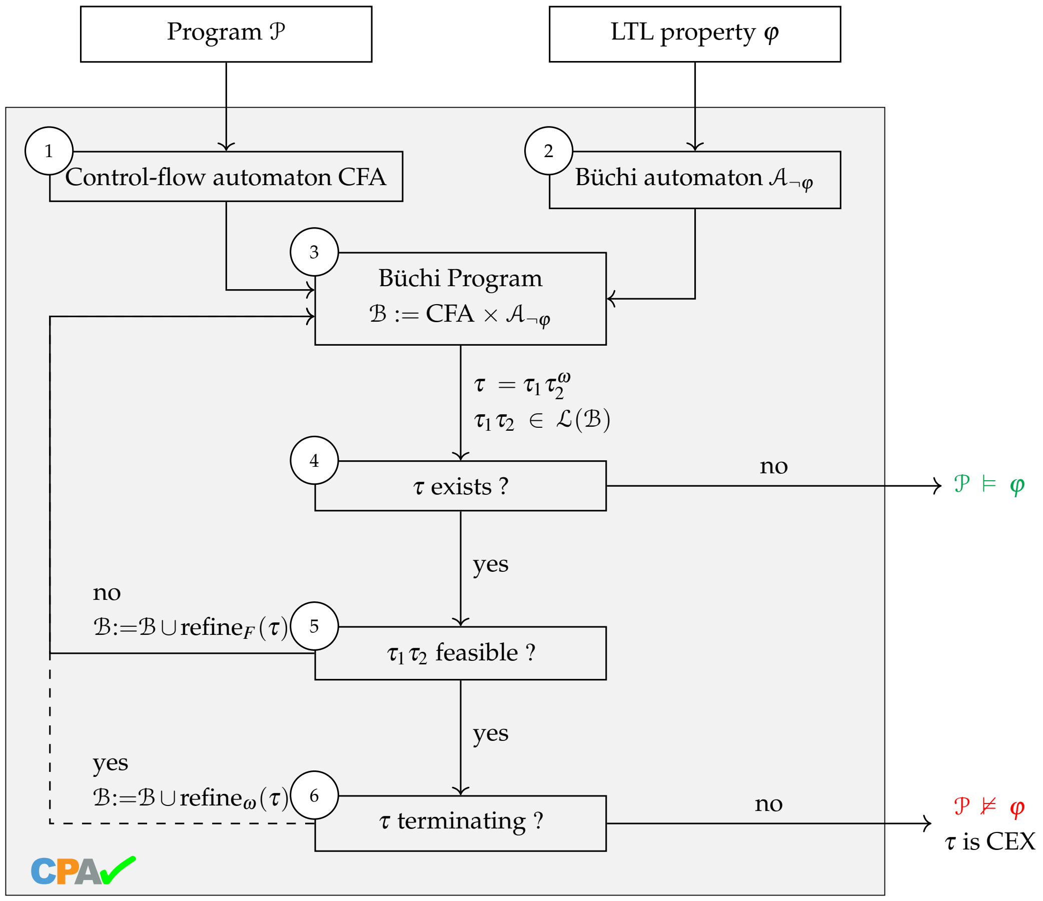
LTL Software Model Checking

This chapter describes the elements of the LTL software model checking algorithm and how they are implemented in CPACHECKER. The whole process is illustrated in Fig. 5.1. The overall idea is to build a product of the CFA and the LTL property, and to continuously refine the resulting ARG afterwards using counterexample-guided abstraction refinement (CEGAR).

In the first step, a program $\mathcal{P}$ is converted into a control-flow automaton (CFA), in which the set of states are the semantically reachable locations of $\mathcal{P}$ (c.f. box 1 in Fig. 5.1). All states are thereby implicitly assumed to be accepting, such that each infinite trace (i.e., the sequence of program statements) is accepted by the CFA. To reiterate from Sect. 4.4.1, if a state does not have a successor state, we simply add a selfloop that uses a *noop*-operation.
Now, let $\mathcal{B}_\varphi$ be a Büchi automaton that accepts the same language as the LTL property φ, such that $\mathcal{L}(\mathcal{B}_\varphi) = \mathcal{L}(\varphi)$. In order to check if the abstract model of the program $\mathcal{P}$ (i.e., the CFA) satisfies an LTL property φ, it is required that

$$\mathcal{L}(CFA) \subseteq \mathcal{L}(\mathcal{A}_\varphi) \tag{5.1}$$

The reason for this is that each (fragment of an) infinite trace must satisfy the given property φ. That is, each of these (fragments of) traces are required to be in the language of $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. However, language containment is rather hard to prove. In CPACHECKER, the LTL formula φ is instead negated and thereafter translated into a corresponding Büchi automaton $\mathcal{B}_{\neg\varphi}$ that accepts the same language, such that $\mathcal{L}(\neg\varphi) = \mathcal{L}(\mathcal{B}_{\neg\varphi})$. This is depicted in box 2 in Fig. 5.1 . Afterwards, the intersection of



Program $\mathcal{P}$
LTL property φ
1
Control-flow automaton CFA
2
Büchi automaton $\mathcal{A}_{\neg\varphi}$
3
Büchi Program
$\mathcal{B} := \text{CFA} \times \mathcal{A}_{\neg\varphi}$
$\tau = \tau_1 \tau_2^\omega$
$\tau_1 \tau_2 \in \mathcal{L}(\mathcal{B})$
4
τ exists ?
no
$\mathcal{P} \models \varphi$
yes
no
$\mathcal{B} := \mathcal{B} \cup \text{refine}_F(\tau)$
5
$\tau_1 \tau_2$ feasible ?
yes
yes
$\mathcal{B} := \mathcal{B} \cup \text{refine}_\omega(\tau)$
6
τ terminating ?
no
$\mathcal{P} \not\models \varphi$
τ is CEX
CPA

Figure 5.1: The LTL software model checking algorithm in CPACHECKER.

the CFA and the Büchi automaton $\mathcal{A}_{\neg\varphi}$ is built. It is much easier to perform automata intersection than checking for language inclusion, since then the model checking process can be reduced to checking whether the language of the intersection result is empty, i.e.:

$$\mathcal{L}(CFA) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset \tag{5.2}$$

The equation above (Eq. 5.2) is equivalent to the equation given in 5.1 . The intersection results in a so-called *Büchi program* $\mathcal{B}$ (c.f. box 3 in the above figure), that is the cross-product of the CFA and the Büchi automaton $\mathcal{A}_{\neg\varphi}$, denoted as $CFA \times \mathcal{A}_{\neg\varphi}$. The language $\mathcal{L}(CFA \times \mathcal{A}_{\neg\varphi})$ satisfies $\mathcal{L}(CFA) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$. In CPACHECKER, the Büchi program is represented by an *abstract reachability graph* (ARG), as this is the default used model that has all of the tools available in order to conveniently store all of the essential information. While the Büchi program is only a theoretical notion, it is in the following used synonymously with the term ARG.

The next step is now to check the Büchi program $\mathcal{B}$ for emptiness. The basic idea boils down to "looking" for a fair and feasible trace τ in the ARG with $\tau \nvDash \varphi$, since the existence of such a trace would disprove that $\mathcal{B} \vDash \varphi$. Recall that a trace being feasible means that it corresponds to some program execution, while a fair trace denotes the following of a path which visits accepting states infinitely often. If no such τ exists, the algorithm is done and it is concluded that the specification is in fact satisfied by the Büchi program. This step is shown in box 4 in Fig. 5.1 .

Each infinite trace τ in the ARG consists of a finite prefix τ_1 and an infinite suffix τ_2^ω, such that $\tau = \tau_1 \tau_2^\omega$. The latter includes thereby only statements, in which the corresponding locations of the ARG are visited infinitely many times. This means, that the respective statements are reachable from each other. Since the set of locations in the ARG is finite, this allows for computing a *strongly connected component* (SCC) that contains only the set of corresponding locations of the trace. In CPAchecker, this is done using Tarjan's algorithm [41], which is a depth-first-search where the respective procedure is called only once for each node. The time complexity is linear in the number of nodes and edges, i.e. $\mathcal{O}(|n| + |e|)$.

The LTL software model checking implementation only considers traces that are reachable from the initial location of the ARG and where the suffix τ_2 contains at least one accepting state, otherwise this would already contradict the requirement of the trace being both fair and feasible. The trace would then not be part of the language of $\mathcal{B}$. As the SCCs can be nested, i.e. each SCC might be part of a bigger SCC, the implementation in CPAchecker extracts in a next step for each SCC the contained cycles within. The algorithm used for this was developed by Donald B. Johnson in [33]. He has proven in this work that all elementary cycles can be found in time bounded by $\mathcal{O}((|n| + |e|)(|c| + 1))$, with n being the nodes, e the edges, and c the number of cycles found. The algorithm itself is however quite involved, and thus not further elaborated in this thesis.

Subsequently, the algorithm checks the trace τ for the feasibility of finite prefixes. This step is depicted in box 5 in Fig. 5.1 . This is done by checking first only the stem τ_1, then the loop τ_2, and finally the concatenation $\tau_1 \tau_2$. The algorithm always proceeds in this order. Should neither of these finite traces prove to be feasible, a trace abstraction $\mathcal{B}{:=}\mathcal{B} \cup \text{refine}_F(\tau)$ is performed. In this process, an *interpolation automaton* is created that is afterwards used to refine the ARG. The concept is based on [30] and is explained in greater detail in Sect. 6.2 . In general, the refined ARG excludes all fair traces of the current Büchi program that are infeasible for the same reason for which trace τ is infeasible.

Otherwise, if all three finite prefixes were proven to be satisfiable, a check for termination for the full infinite trace will be performed afterwards. This is shown in box 6 in Fig. 5.1, and is executed in CPAchecker using the components of the termination analysis that was implemented by Ott in [36]. The algorithm for LTL software model checking makes in particular use of both the "LassoBuilder" framework in CPAchecker, as well as the tool LassoRanker which is part of the Ultimate software analysis framework. The former is thereby used to create a *lasso*-object from the current analyzed trace, which will then be used as input for the LassoRanker. A lasso consists of a starting state, a stem τ_1, a loop τ_2, and a so-called *honda* state inbetween them that marks the ending of the stem and both the beginning and ending of the cycle. Using this as input, the LassoRanker then tries to find a ranking function that proves the eventual termination of the loop. When non-termination can be proven, this relates to the trace being in fact executable by the program $\mathcal{P}$, and is hence a concrete counterexample that witnesses a violation of the property φ. Otherwise, if LassoRanker proves the trace to be non-terminating, it is spurious and the algorithm continues with the next step.

Conceptually, this next step would now be a so-called omega-refinement (refine$_\omega$) , in which another form of automaton is constructed, that is then used to refine the ARG. The approach is quite similar to that of the trace abstraction for finite prefixes, and has its concept explained extensively in [31]. However, this work was out of scope for this thesis and needs yet to be implemented in CPAchecker. For this reason, the line in Fig. 5.1 is drawn dashed. The implementation of this in CPAchecker should however be rather straightforward, in particular after the trace abstraction of finite prefixes is fully working. For the time being, the implementation in CPAchecker checks each lasso individually for its termination, i.e. without performing any ω-refinements of the ARG at all.

In contrast, CEGAR is used for the process of finite-trace refinement in CPAchecker. The implementation is currently such that each time a complete ARG is built from anew. Only afterwards is the above described analysis then performed. This approach in CPAchecker is also called *global refinement*.

For the checks of both the finite prefixes and termination, it should be considered that these are – in general – based on undecidable methods. Thus, it is possible that the algorithm (or more specifically, either the SMT-solver or LassoRanker) does not terminate and instead runs into a timeout. The analysis from CPAchecker will then return *unknown* as result.

As for the specification being satisfied by the program, it needs to also be emphasized that the initial location of the CFA – and hence also in the ARG – is not checked for its satisfiability, unlike any other location. Not having this restriction allows additionally programs that satisfy the LTL property $\Box(x = 0)$. An example for such a program would be one that has its first statement set the value of x to 0, and then to never modify it again in the remainder of the program flow.

CHAPTER 6

Implementation

The CPACHECKER framework uses SVN[1] for versioning and revision control. Additionally, a read-only GIT mirror[2] is provided that contains a detailed description on how to install and utilize CPACHECKER. In order to execute the program, it requires both a Java SDK, which is at least Java 8 compatible, and *Apache Ant* on the working machine in order to compile the Java code and build an executable target binary.

After everything has been set up, LTL software model checking can be performed in CPACHECKER using the following command[3] from within the directory:

```
scripts/cpa.sh -ltl <path/program> -spec <path/specification>
```

The <program> is required to be a program written in C, while the <specification> is the property that is to be verified. The specification needs to be given as a file with the name "path/filename.prp", and it may consist of only one line that includes the LTL property. More precisely, the property within the specification file is required to be precisely in the following format:

```
CHECK ( init(<init_function>) , LTL(<property>) )
```

The <init_function> is the main-function,i.e. the entry point of the program, and is most often declared as `main()`. However, any other function name that is valid in C is also appropriate. The property needs to be specified as a syntac-

[1]https://svn.sosy-lab.org/software/cpachecker/trunk/ (last accessed on April 06, 2019)
[2]https://gitlab.com/sosy-lab/software/cpachecker (last accessed on April 06, 2019)
[3]This has been tested using the operating system Ubuntu 18.04 LTS

tically and semantically correct LTL property, such as defined in Sects. 4.1.1 and 4.1.2.

As an example, Sect. 3.2 has shown a C-program (c.f. Fig. 3.2a) that has an entry point function `main()`, and which is to be checked for an LTL property $\square(\text{"}x > 0\text{"} \Rightarrow \Diamond(\text{"}y = 0\text{"}))$, which is depicted in Fig. 3.3a. Now, in this example, the specification file would therefore require to have the following line:

```
CHECK( init(main()), LTL([]("x > 0" ==> <>("y == 0"))) )
```
(6.1)

Deviations in the LTL property are possible, such as e.g. writing `F("y==0")` instead of using the diamond symbol <>, or alternatively, adding additional whitespace characters for a better readability. This is explained further in Sect. 6.1.

Examples for C-programs that are destined to be checked for LTL properties can be found in the GitHub repository from ULTIMATE[4]. The latter is a program analysis framework developed by the *Software Engineering* chair of the University of Freiburg. Note that in this repository the LTL properties are directly written as comments into the program files. I.e., to be able to execute these in CPACHECKER, the properties must be first extracted into specification files and afterwards formatted as described above. The example program from Sect. 3.2 was also taken from this repository[5], and has been used predominantly in order to test the implementation of LTL software model checking in CPACHECKER.
The full implementation of the algorithm in CPACHECKER is versioned in the official SVN-repository[6] and can be found in the branch *ltl-model-checking* as of revision 30987. Though, it is intended to have it merged into trunk within the foreseeable future.

6.1 Parsing LTL Formulas in CPACHECKER

For this thesis, it is required to parse LTL properties from the input specification file and transform them into automata from the CPACHECKER framework. However, the actual transformation to Büchi automata is performed by third party tools that are publicly available, and which are in particular *Spot* [24] and *LTL3BA* [4].

[4]`https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL` (last accessed on April 06, 2019)

[5]`https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL/simple/cav2015.c` (last accessed on April 06, 2019)

[6]`https://svn.sosy-lab.org/software/cpachecker/branches/ltl-model-checking/` (last accessed on April 06, 2019)

For the transformation process, while it would be theoretically possible to take the raw LTL statement from the input specification and simply hand that over to either of the tools, this used in practice to not be applicable for various reasons. Consider e.g. the following property, which is a real example from the repository of ULTIMATE[7] and demonstrates how a specification file can look like:

```
CHECK( init(main()), LTL( (G (! "input == 4" || (F "output == 22")))) )
```

To simply take the raw string within the parentheses of `LTL( ... )` and passing it on to the external tools used to not work in the past:

- Most of the available open-source tools for LTL to Büchi translation were not able to cope with quotation-marks. *Spot*, for example, can only deal with such quotes since around last summer[8].
- The *Spin*-syntax[9] for LTL formulas uses the symbolic form of LTL operators instead of their textual form (e.g., '$[\,]a$' for the globally operator instead of '$G\ a$'). An example for such a specification can be found on the previous page in 6.1. Again, most of the tested tools used to only be able to deal with one of the forms (i.e., either symbolical or textual).

The tested tools next to *Spot* and *LTL3BA* have been *LTL2BA* [28] and *Rabinizer4* [26], among others. The above mentioned restrictions have existed during the time the framework was written in CPACHECKER. In fact, this issue has been the driving force for implementing the framework in CPACHECKER in the first place. The two shortcomings listed above could be simply circumvented by using a string visitor, in which an output is created that is suited for the respective tools. For example, the content within the quotation marks can be simply substituted by a temporary placeholder variable. However, as of April 2019, most of the aforementioned tools seem to have been updated, as these restrictions were no longer be experienced. Some of them still remain though, e.g. for the tools *LTL2BA* and *Rabinizer4*. Nonetheless, parsing raw LTL formulas into strongly typed objects by oneself provides several more advantages, which are described in the remainder of this section.

[7]https://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/LTL/svcomp17format/ltl-eca/Problem14_prop_002_true-valid-ltl.c.i.prp (last accessed on April 06, 2019)

[8]Tested in June, 2018, using the official online tool at https://spot.lrde.epita.fr/app/ (last accessed on April 06, 2019)

[9]For more information regarding the syntax of Spin LTL formulas, c.f. http://spinroot.com/spin/Man/ltl.html (last accessed on April 06, 2019)

In order to parse and extract LTL properties, a parser generator has been used that allows to build hierarchical structures of an incoming stream of tokens. The name of the tool is *ANTLR4* [37], and is short for "*Another Tool For Language Recognition*". It is an open-source tool published under a 3-clause BSD License[10]. The main reasons for taking this particular tool were mostly the nonrestrictive license, its support for Java, that it is still maintained and under active development as of the date of submission, and lastly the fact that the setup is intuitive and hence both easy to understand and implement.

The approach of ANTLR4 is to first do a lexical analysis, in which all words are grouped into tokens, and then to perform the actual parsing, where the tokens are analyzed for their structure such that a parse tree can be built. The grammar in ANTLR4 is specified by an *Extended Backus-Naur-Format* (EBNF). While the grammatical rules are too large to be listed here, these are indicated in Fig. 6.1 on the right. The exemplary LTL formula 'a -> F "$x > 0$"' is parsed such, that at the end all its different atoms can be unambiguously assigned according to the parser rules. Should the submitted LTL formula in CPACHECKER be syntactically incorrect, or in any other way malformed, this would then be spotted here, and an appropriate (checked) exception is thrown. After a raw LTL string has been parsed and evaluated by ANLTR4, it is afterwards stored in a strongly typed *LabelledFormula*-object in CPACHECKER, in which the formula is represented again in the form of a tree-like structure.

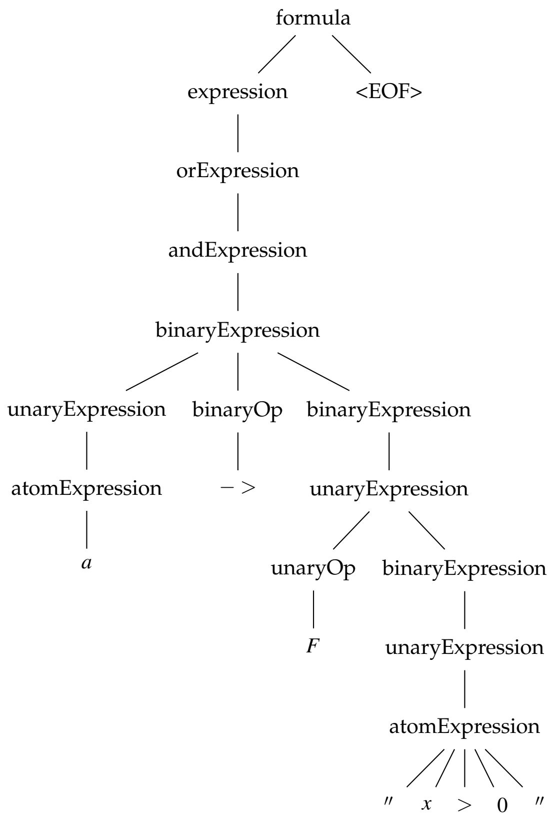


Figure 6.1: ANTLR4 syntax tree for the arbitrary LTL formula a -> F "$x > 0$" .

Regarding the LTL lexer, a high emphasis was put on being able to deal with all prevalent forms of propositional and temporal modal operators. This is depicted in Table 6.1, which shows all valid options on how the operators in an LTL formula

[10]`https://www.antlr.org/license.html` (last accessed on April 06, 2019)

Table 6.1: The lexer rules for the ANTLR4 parser.

// LOGIC	
TRUE	'TRUE' | 'True' | 'true' | '1'
FALSE	'FALSE' | 'False' | 'false' | '0'
// Logical Unary	
NOT	'!' | 'NOT'
// Logical Binary	
IMP	'->' | '–>' | '=>' | '==>' | 'IMP'
EQUIV	'<->' | '<=>' | 'EQUIV'
XOR	'^' | 'XOR'
// Logical n-ary	
AND	'&&' | '&' | 'AND'
OR	'||' | '|' | 'OR'
// Modal Unary	
FINALLY	'F' | '<>'
GLOBALLY	'G' | '[]'
NEXT	'X'
// Modal Binary	
UNTIL	'U'
WUNTIL	'W' | 'WU'
RELEASE	'R' | 'V'
SRELEASE	'S'

may be denoted. Taking the *implication* symbol for example, the implemented lexer is able to cope with multiple different notations. That is, all of the following options are allowed for this operator: '->', '–>', '=>','==>', and 'IMP'. Furthermore, a conjunction consisting of two atoms *a* and *b* can be expressed by using the logical *and* in several ways. That is, all of the following options are valid: '*a*&*b*', '*a*&&*b*', and '*a* AND *b*'. The same also holds for modal operators, e.g. an atomic propositions *a* can be expressed by a globally operator using either its symbolical form [ ] *a*, or alternatively the textual form *G a*.

Regarding the variables, there are two options on how to express these. That is, either by specifying them as a lowercase identifier followed by any letter or number, like e.g. 'a' or 'myInput42', or alternatively by using quotation marks, in which all numbers, letters (both lower- and uppercase), and a wide range of mathematical operators and comparators are allowed, such as e.g. "x > 0" or "myParam += 5". Note that anything which is written between the quotation marks is not actually parsed by ANTLR4, but instead it is taken as raw string and passed on to the existing *CParserUtils*-framework in CPACHECKER. If there is a syntactical error within the quotes, this would then only be noticed there.

Having the parsing process of an LTL property available in CPACHECKER provides furthermore the advantage of being able to perform transformations that additionally simplify and optimize the LTL formula. For example, redundant symbols can be removed, such as multiple consecutive parentheses as in $(((\varphi)))$ for a temporal formula φ, or double negatives as in $!!\varphi$. Beyond that, trivial identities are applied in CPACHECKER which are stated below.

$$\begin{array}{ccc} !0 \equiv 1 & 1 \Rightarrow \varphi \equiv \varphi & \varphi \Rightarrow 1 \equiv 1 \\ !1 \equiv 0 & 0 \Rightarrow \varphi \equiv 1 & \varphi \Rightarrow 0 \equiv !\varphi \\ !!\varphi \equiv \varphi & & \varphi \Rightarrow \varphi \equiv 1 \end{array}$$

Note that in CPACHECKER the implication is implicitly converted into a disjunction, i.e. '$1 \Rightarrow \varphi$' is first converted into '$!1 \mid\mid \varphi$', and only afterwards will this term be simplified to 'φ'. This is also true for any equivalency- and xor-operations in an LTL formula, i.e. these are also first converted into their respective logical connectives, and only then checked for possible trivial identities. Below are denoted several more of such trivial identities, with respect to the propositional operators. As the operators thereof are commutative, the identities are hence also valid with their arguments swapped.

$$\begin{array}{cc} 0 \ \&\& \ \varphi \equiv 0 & 0 \mid\mid \varphi \equiv \varphi \\ 1 \ \&\& \ \varphi \equiv \varphi & 1 \mid\mid \varphi \equiv 1 \\ \varphi \ \&\& \ \varphi \equiv \varphi & \varphi \mid\mid \varphi \equiv \varphi \end{array}$$

Regarding the temporal modal operators, there are also several trivial identities that are simplified in CPACHECKER. These are all the ones stated below.

$$\begin{array}{ccc} X0 \equiv 0 & F0 \equiv 0 & G0 \equiv 0 \\ X1 \equiv 1 & F1 \equiv 1 & G1 \equiv 1 \\ & FF\varphi \equiv F\varphi & GG\varphi \equiv G\varphi \end{array}$$

$$\begin{array}{llll}
\varphi\, U\, 1 \equiv 1 & \varphi\, WU\, 1 \equiv 1 & \varphi\, S\, 0 \equiv 0 & \varphi\, R\, 1 \equiv 1 \\
0\, U\, \varphi \equiv \varphi & 0\, WU\, \varphi \equiv \varphi & 0\, S\, \varphi \equiv 0 & \varphi\, R\, 0 \equiv 0 \\
\varphi\, U\, 0 \equiv 0 & 1\, WU\, \varphi \equiv 1 & 1\, S\, \varphi \equiv \varphi & 1\, R\, \varphi \equiv \varphi \\
\varphi\, U\, \varphi \equiv \varphi & \varphi\, WU\, \varphi \equiv \varphi & \varphi\, S\, \varphi \equiv \varphi & \varphi\, R\, \varphi \equiv \varphi
\end{array}$$

As a final point regarding the parsing process of LTL properties, the goal of this thesis is to check programs for liveness properties. For the time being, the input LTL formula is hence simply assumed to be given as liveness property. However, the presented algorithm in Chapter 5 is theoretically also able to check programs for safety properties, although some minor modifications would be required for this in the current implementation. The availability of such a strongly typed LTL formula makes it however easy to implement a query that checks properties for their type.

6.2 Trace Abstraction

Trace abstraction is a refinement method for counterexample-guided abstraction refinement. In Chapter 5, it was shown how in each iteration loop of the CEGAR algorithm a fair trace is analyzed. That is, the trace is checked (among others) for finite prefixes, in which the statements possibly contradict each other. This is done by an SMT solver, which is in case of this work SMTInterpol [16]. Should the solver prove the trace to be unsatisfiable, the general idea is to refine the ARG afterwards, such that in the new abstraction the trace is excluded from the language. The concept of the trace abstraction now allows to additionally generalize this single trace to a set of traces, such that in the next refinement abstraction all traces can be excluded from the language that are infeasible for the same reason for which the analyzed trace is infeasible. This is introduced in the two papers in [29, 30]. These are also the sources on which the implementation in CPAchecker is based upon. They describe the approach of not only refining the current examined trace, but instead to refine an over-approximation of the set of possible traces. Thus, in each iteration a new finite *interpolation automaton* is created that recognizes a set of infeasible traces. This can be done automatically by using interpolants. In [22], a *Craig interpolant* is defined as follows. Let A and B be formulas, such that $A \Rightarrow B$. An interpolant C is then a formula, in which the following holds:

1. $A \Rightarrow C$
2. $C \Rightarrow B$
3. C contains only atoms which occur both in A and B

The interpolant C is thus an assertion that contains just enough information for A to conclude on B.

In the algorithm for LTL software model checking, the interpolants are generated by the infeasibility proof for the error trace. The interpolants are then used to create the interpolation automaton, that not only accepts the error trace, but instead recognizes (usually) a much larger set of traces that "share" the same reason of infeasibility as the error trace.
In Chapter 5 in Eq. (5.2), it is shown how model checking is reduced to the emptiness problem. In the first iteration of CEGAR, in which no refinement has been made yet, the CFA and the Büchi automaton $\mathcal{A}_{\neg\varphi}$ are intersected and afterwards analyzed for whether the language of the intersection result is empty. This is now extended to trace abstraction. A *trace abstraction* is given by a tuple of automata $(\mathcal{A}_1,...,\mathcal{A}_n)$, in which each automaton $\mathcal{A}_i$ for $i = 0...n$ recognizes a subset of infeasible traces. Now, let $\mathcal{P}$ be a program that is represented by a CFA, and let $\mathcal{L}(\mathcal{A}_\varphi)$ be the specification automaton for the LTL property φ. The trace abstraction $(\mathcal{A}_1,...,\mathcal{A}_n)$ is said to not admit an error trace, if the language recognized by the automaton $(CFA \cap \overline{\mathcal{A}_\varphi} \cap \overline{\mathcal{A}_1} \cap ... \cap \overline{\mathcal{A}_n})$ is empty, i.e.:

$$\mathcal{L}(CFA \cap \overline{\mathcal{A}_\varphi} \cap \overline{\mathcal{A}_1} \cap ... \cap \overline{\mathcal{A}_n}) = \emptyset$$

The proof that the above equation is both sound and complete can be found in Sect. 4 in [29]. Regarding the negated Büchi automaton, note that in the above equation $\overline{\mathcal{A}_\varphi}$ is equal to $\mathcal{A}_{\neg\varphi}$. In the next section, it is shown by means of an example how this concept is working out in CPAchecker.

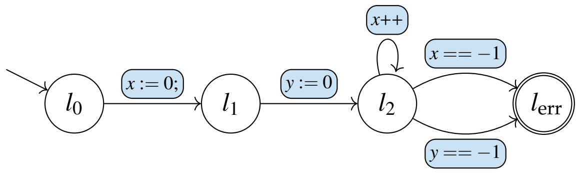
6.2.1 Example

In Fig. 6.2a, a C-program $\mathcal{P}$ is given as pseudo-code. In its sequence of program statements, it first sets the value of x and y to 0, increments afterwards the value of x for an arbitrary often, and finally checks that none of the two variables x and y has its value set to -1. The depicted automaton in Fig. 6.2b shows the corresponding CFA of program $\mathcal{P}$. The initial trace abstraction is the empty tuple, therefore the resulting restriction in this CEGAR iteration is the CFA itself. Note that for the sake of simplicity, the LTL property is omitted in this example.
Now, the language of the CFA is clearly not empty, as it is possible to reach the error state l_{err} from the starting location l_0 with e.g. the trace π_1 given below:

$$\pi_1 = \boxed{x := 0}\boxed{y := 0}\boxed{x++}\boxed{x == -1}$$

```
l0 x := 0;
l1 y := 0;
l2 while (nondet) {x++;}
   assert(x != -1);
   assert(y != -1);
```

(a) Program $\mathcal{P}$.



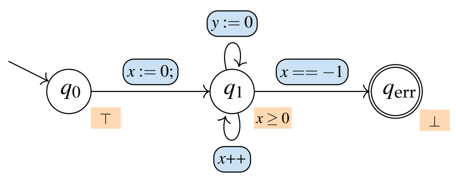
(b) The CFA for $\mathcal{P}$.

Figure 6.2: Example program $\mathcal{P}$ for the trace abstraction. The program statements are shown in (a) on the left. The automaton in (b) is the corresponding CFA. $\mathcal{P}$ is correct iff all words accepted by the CFA are unsatisfiable.

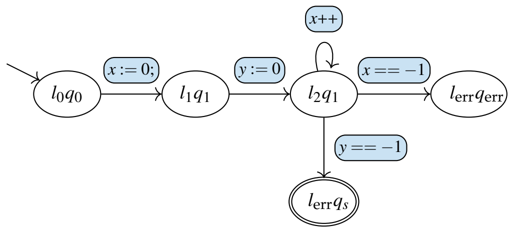
This trace is a spurious counterexample. In the algorithm of CPACHECKER, an SMT solver is used to confirm that this trace is indeed unsatisfiable, and returns as proof a sequence of interpolants. These are required in order to construct the interpolation automaton $\mathcal{A}_1$, which is depicted in Fig. 6.3a. More precisely, the states and transitions in this automaton are determined by Hoare triples, in which the interpolants are used as invariants. The four Hoare triples below are sufficient in proving that all possible paths from the initial location to the error location using the statement $x == -1$ are unsatisfiable in the CFA. They state that after the statement $x := 0$ the assertion $x \geq 0$ holds, that it is an invariant during the execution of $y := 0$ and $x{+}{+}$, and finally that it prevents the statement $x == -1$ from ever being executed.

$$\begin{array}{ccc}
\{\ \mathit{true}\ \} & x := 0 & \{\ x \geq 0\ \} \\
\{\ x \geq 0\ \} & y := 0 & \{\ x \geq 0\ \} \\
\{\ x \geq 0\ \} & x{+}{+} & \{\ x \geq 0\ \} \\
\{\ x \geq 0\ \} & x == -1 & \{\ \mathit{false}\ \}
\end{array}$$

These four Hoare triples translate exactly to the automaton as given in Fig. 6.3a. It has three states, one for each interpolant. That is, the initial state q_0 for the assertion *true*, the state q_1 for $x \geq 0$, and the final state q_{err} for *false*. Note that these assertions are however only depicted in the figure for the sake of demonstration, i.e. they are used solely for building the automaton. Afterwards, they have fulfilled their purpose and are no longer used. The automaton $\mathcal{A}_1$ has furthermore four transitions, namely one for each Hoare triple. The resulting interpolation automaton come always in this form, the only thing that is changing are the transitions. In general, the automaton generalizes to any set of Hoare triples. Thus, any program statement can be added as a transition to the automaton for as long as the respective Hoare Triple is valid.



(a) Interpolation automaton $\mathcal{A}_1$



(b) Resulting automaton of CFA $\cap\, \overline{\mathcal{A}_1}$

Figure 6.3: Fig. (a) on the left shows the interpolation automaton that is created from the interpolants computed during the feasibility check of the error trace π_1. In Fig. (b) is the automaton from CFA $\cap\, \overline{\mathcal{A}_1}$ depicted, which is the result of the first trace abstraction refinement.

Automaton $\mathcal{A}_1$ accepts each trace that has the same sequence of interpolants as the trace π_1. This also holds for subsequences of interpolants, and is the actual reason why each of these traces have the same "reason of infeasibility". In the second CEGAR iteration, a new abstraction is created, which is shown in Fig. 6.3b. This is the intersection result of the CFA and the tuple $(\overline{\mathcal{A}_1})$, which consists in this iteration step of only one element. In CPACHECKER, this is the resulting ARG after the first refinement. As can be seen here, the counterexample π_1 is no longer in the language of this result. It is restricted due to the intersection of the CFA with the complement of the derived interpolation automaton $\mathcal{A}_1$.

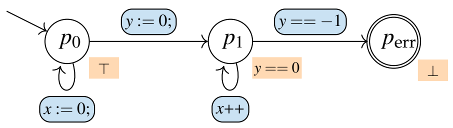
Yet, the error location is still reachable by another trace π_2, which can e.g. be as follows:

$$\pi_2 = \boxed{x := 0}\,\boxed{y := 0}\,\boxed{x{+}{+}}\,\boxed{y == -1}$$

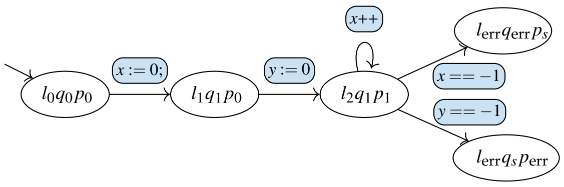
This trace is unsatisfiable again. Using an SMT solver for checking this counterexample yields the assertion $y = 0$ as part of the infeasibility proof. The interpolant is then used again as invariant in order to create a second interpolation automaton $\mathcal{A}_2$, which accepts all traces that reach the error location using the statement $y == -1$. This is proven by the four Hoare triples below.

$$\begin{array}{ccc}
\{\ \mathit{true}\ \} & x := 0 & \{\ \mathit{true}\ \} \\
\{\ \mathit{true}\ \} & y := 0 & \{\ y = 0\ \} \\
\{\ y = 0\ \} & x{+}{+} & \{\ y = 0\ \} \\
\{\ y = 0\ \} & y == -1 & \{\ \mathit{false}\ \}
\end{array}$$

The next step is to create the interpolation automaton $\mathcal{A}_2$. This is done in the same fashion as described above for $\mathcal{A}_1$. The resulting automaton can be seen in Fig. 6.4a.



(a) Interpolation automaton $\mathcal{A}_2$



(b) Resulting automaton of CFA $\cap\, \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2}$

Figure 6.4: Fig. (a) shows the interpolation automaton that results from the infeasibility proof of the error trace π_2. Fig. (b) on the right depicts the result from the second refinement, which is created from the intersection of CFA $\cap\, \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2}$.

Afterwards, a new abstraction is computed, based on the derived abstraction from the last iteration together with the interpolation automaton $\mathcal{A}_2$ that is constructed in the current iteration of CEGAR. The result is shown in Fig. 6.4b. It is the intersection of the CFA and the trace abstraction tuple $(\mathcal{A}_1, \mathcal{A}_2)$. The resulting automaton CFA $\cap\, \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2}$ excludes not only the traces π_1 and π_2, it in fact accepts no word at all. The result accepts thus the empty language, which proves the program $\mathcal{P}$ to be correct.

Discussion of the trace abstraction concept in CPAchecker

As a closing point regarding the trace abstraction, all of the above points have been implemented in the framework of CPAchecker. However, at the time of submission of this thesis, the implementation does unfortunately not work, which is most likely caused by a conceptual error in the setup of the interpolation automata. Technical errors in CPAchecker can be ruled out with almost absolute certainty, as the implementation has been debugged extensively for several weeks, where it was repeatedly double-checked that the code itself is indeed working as intended. The current status is hence that technically everything is working correctly, while during the execution of LTL program verification, the ARG is somewhat malformed during the refinement. This leads to the ARG currently suffering from a state space explosion after roughly 25 CEGAR iterations. The fixing of this issue is hence the most critical thing in CPAchecker, and is what needs to be addressed next. This is the sole reason which currently prevents LTL software model checking from working correctly.

CHAPTER 7

Conclusion and Future Work

The LTL software model checking problem can be solved in general by modeling both the program and the specification as automata on infinite words. In CPACHECKER, this is done using a CFA that models the program, while an LTL property is transformed into an automaton in the already existing framework. Afterwards, the model checking is then performed by verifying that the language of the product of the CFA together with the specification automaton is empty. In practical, this is however only applicable to small programs with non-complex structures, as the model checking process requires a check for each infinite fair path for its feasibility in order to prove program correctness. It is evident that this does not scale well to large programs.

Thus, it was decided early in this work to additionally implement the concept of trace abstraction for finite prefixes. The approach is to define a sequence of semi-tests for infinite paths and have them checked first for their feasibility, before performing the actual check for termination of the full infinite path. Should one of these semi-tests prove that the path is unsatisfying, the trace abstraction algorithm then automatically derives an interpolation automaton which is used thereafter with the current abstraction in order to perform a *refinement for finite prefixes*. Using this procedure has two major advantages: First, should the semi-tests prove a path to be indeed infeasible, this would then render the full test for termination redundant, thus making it possible to avoid the costly computation for a ranking function. Secondly, the trace abstraction allows to (usually) exclude a large set of traces from the language in the next refinement iteration, namely all of those that have the same reason of infeasibility. These two techniques lead to a huge improvement in terms of efficiency, and allow the LTL software verification for larger programs where the increasing complexity is no longer an insurmountable obstacle.

There are several ways how LTL software model checking can be further improved with respect to CPAchecker. First and foremost, the integration of trace abstraction for termination proofs comes to mind. The approach is similar to the trace abstraction of finite prefixes that is indicated above. The only thing that truly changes is the way the interpolation automaton is constructed.
Another improvement is the increase of the used block size in the analysis. The current implementation uses single-block encoding (SBE), however, different analyses in CPAchecker have shown that making use of adjustable-block encoding (ABE) provides a further significant boost to the efficiency of the performed analyses.
Lastly, while the performance of the external tools that translate LTL formulas to Büchi automata is already quite high, it would be interesting to see if it makes a noticeable difference for LTL software model checking in CPAchecker when the different tools are exchanged with each other (e.g., *LTL3BA*, *SPOT*, or *Rabinizer4*).

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012.

[5] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[6] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 51–62. ACM, 2013.

[7] D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 622–640. Springer, 2015.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[9] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 29–38. IEEE Computer Society, 2008.

[10] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. *CoRR*, abs/0902.0019, 2009.

[11] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 189–197. IEEE, 2010.

[12] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[13] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2005.

[14] J. R. Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[15] J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[16] J. Christ, J. Hoenicke, and A. Nutz. Proof tree preserving interpolation. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2013.

[17] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[18] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

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

[20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001.

[21] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 265–276. ACM, 2007.

[22] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[23] D. Dietsch, M. Heizmann, V. Langenfeld, and A. Podelski. Fairness modulo theory: A new approach to LTL software model checking. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[24] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, Oct. 2016.

[25] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 127–140. ACM Press, 1983.

[26] J. Esparza and J. Kretínský. From LTL to deterministic automata: A safraless compositional approach. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2014.

[27] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 163–173. ACM Press, 1980.

[28] P. Gastin and D. Oddoux. Fast LTL to büchi automata translation. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[29] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2009.

[30] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proc. CAV*, LNCS 8044, pages 36–52. Springer, 2013.

[31] M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.

[32] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[33] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[34] A. Lal and S. Qadeer. Reachability modulo theories. In *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, volume 8169 of *Lecture Notes in Computer Science*, pages 23–44. Springer, 2013.

[35] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[36] S. Ott. Implementing a termination analysis using configurable software analysis. Master's Thesis, University of Passau, Software Systems Lab, 2016.

[37] T. Parr and K. Fisher. Ll(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436. ACM, 2011.

[38] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[39] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[40] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[41] R. E. Tarjan. Depth-first search and linear graph algorithms (working paper). In *12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971*, pages 114–121. IEEE Computer Society, 1971.