
Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

# Abstraction-based model checking of linear temporal properties for critical systems

**Scientific Students' Association Report**

Author:

Milán Mondok

Advisor:

dr. András Vörös
Vince Molnár

2019

# Contents

# Kivonat

A technológia fejlődésével egyre több kritikus területen alkalmaznak szoftver alapú megoldásokat, ilyen esetekben azonban a szoftver hibás működése akár katasztrofális következményekhez is vezethet. Fontos feladat tehát a helyesség ellenőrzése és a hibák megtalálása lehetőleg már a fejlesztés korai fázisaiban is.

Kritikus rendszerek tervezése során gyakran használnak magas szintű modellezési nyelveket. A viselkedés leírására elterjedten használt az állapottérkép formalizmus, amely különösen alkalmas kritikus reaktív rendszerek tervezésének támogatására.

Modellek ellenőrzésére többféle megközelítés is ismert, amelyek közül a formális verifikáció alkalmas a modellek összes viselkedésének felderítésére és ellenőrzésére. Többféle formális temporális logikai nyelv is rendelkezésünkre áll a követelmények megfogalmazására, azonban kevés eszköz támogatja ezeknek a kifejező – ezáltal mérnökök által is könnyen használható – temporális logikáknak a használatát. A lineáris idejű temporális logikai specifikációk ugyan nagyon kifejezők, de ellenőrzésük különösen számításigényes feladat, emiatt az ellenőrzési módszerek hatékonyságának növelése állandó kihívás. A formális verifikáció alkalmazásának másik kihívása, hogy alacsony szintű matematikai modelleken működik, és ez megnehezíti a mérnöki alkalmazást.

Munkám célja, hogy egy olyan formális verifikációs algoritmust adjak, amely támogatja a lineáris idejű temporális logika használatát a verifikáció során. A hatékonyság érdekében egy absztrakció alapú algoritmust fejlesztettem, amely lehetővé teszi adatvezérelt reaktív rendszerek hatékony verifikációját is. Megközelítésemet egy mérnöki tervező eszközbe is integráltam, ezáltal lehetővé téve az új megközelítés mérnöki alkalmazását. Az elkészült algoritmust az egyetemen fejlesztett formális verifikációs keretrendszerben valósítottam meg, így támogatva a keretrendszer összes többi bemeneti formalizmusát is - többek között szoftverek forráskód alapú verifikációját is.

Az elkészült algoritmus hatékonyságát a szoftverellenőrzés területének egyik szabványos benchmarkján vizsgáltam meg.

# Abstract

With the rapid advancements of technology, more and more critical areas see the introduction of software-based solutions, where faulty behavior of the software can have catastrophic consequences. This means identifying faults and proving correctness are crucial parts of the development of such systems and should be done as early in the process as possible.

High-level modeling languages are often used when designing critical systems. The statechart formalism is a commonly used way of describing behavior and is suitable for the modelling of reactive systems. Several approaches exist to verification, from which formal verification is capable of fully exploring and verifying the behavior of a system. Various temporal logical languages exist, but not many tools support the use of expressive – and thus easy-to-use – linear temporal logics. Linear temporal logical specifications, while being particularly expressive, lead to a computationally demanding verification task, making the improvement of the performance of such techniques a permanent challenge.

My work aims at creating a formal verification algorithm that supports the use of linear temporal logic. My algorithm is based on abstraction for efficiency, enabling the verification of data-driven reactive systems as well. I integrated my solution into a modelling tool, creating a way for engineers to utilize my approach in real-world projects. I implemented my tool using a model verification framework developed at the University, supporting all input formalisms - for example, source code based analysis - of the framework.

I evaluated the efficiency of the implemented algorithm using a standard software verification benchmark.

# Chapter 1

# Introduction

With the rapid advancements of technology, more and more critical areas see the introduction of software-based solutions, where faulty behavior of the software can have catastrophic consequences. This means identifying faults and proving correctness are crucial parts of the development of such systems and should be done as early in the process as possible.

High-level modeling languages are often used when designing critical systems. The statechart formalism is a commonly used way of describing behaviour and is suitable for the modeling of reactive systems. Several approaches exist to verification, from which formal verification is capable of fully exploring and verifying the behavior of a system. It seems however, that high-level modeling tools that are suited for engineering work and efficient formal verification tools rarely intersect. Various temporal logical languages exist, but not many tools support the use of expressive – and thus easy-to-use – linear temporal logics. Linear temporal logical specifications, while being particularly expressive, lead to a computationally demanding verification task, making the improvement of the performance of such techniques a permanent challenge.

In this thesis I present my formal verification algorithm that supports the use of linear temporal logic. My algorithm is based on counterexample-guided abstraction refinement for efficiency, enabling the verification of data-driven reactive systems as well. I integrated my solution into a modeling framework, creating a way for engineers to utilize my approach in real-world projects. I implemented my tool using a formal verification framework developed at the University, supporting all input formalisms - for example, source code based analysis - of the framework.

I evaluated the efficiency of the implemented algorithm using a standard software verification benchmark and engineering models.

## 1.1 The structure of this thesis

The structure of this thesis is the following. Chapter 2 presents the theoretical concepts on which I based my solution. Chapter 3 details the theoretical advancements I made. In Chapter 4 I provide details about my implementation. Chapter 5 presents benchmarking results of running various model checking tasks. In Chapter 6 I draw the conclusions of my work and lay down possible paths to continue.

# Chapter 2

# Background

In this chapter I briefly present the theoretical concepts on which I based my solution. In section 2.1 I introduce *model checking*.

Section 2.2 is a summary of the *formalisms* I use to model the systems I intend to verify. *Kripke structures* (Section 2.2.1) provide a strong foundation on top of which many of the model checking methods are built. *Statecharts* (Section 2.2.2) offer an intutive, but at the same time strict and well-defined way of representing reactive systems. *Control flow automata* (Section 2.2.3) are a commonly used way of representing computer programs in a formal manner.

In Section 2.3 I present how abstraction can be used to tackle the performance challenges of model checking. *Counterexample-guided abstraction refinement*2.3.1 is a commonly used way of calculating the desired abstraction-level when analyzing a system's behaviour. Its two main components , the *Abstractor* and the *Refiner* are discussed in Sections 2.3.2 and 2.3.3.

Section 2.4 is a summary of the convential way of linar temporal model checking. Section 2.4.1 introduces *linear temporal logic*, a formalism for describing correctness requirements. Section 2.4.2 gives an introduction to *automata theory*, and shows (Section 2.4.2.2) how linear temporal logical formulas can be expressed using automata. Section 2.4.3 discusses how automata theoretic approaches can be used to check linear temporal properties.

## 2.1 Model checking

Formal verification consists of numerous methods for proving the correctness of systems with mathematical certainty, one of which is *model checking* [13]. It aims to exhaustively analyze all possible behaviours of a system to check if it meets a given correctness specification.

Formally, given a model $M$ and a specification $\varphi$ determine whether or not the behavior of $M$ meets the specification $\varphi$. The result of model checking can be either *true* if the specification holds, or a *counterexample* if it doesn't.

Model checking of *temporal logical properties*[20] is a specific class of model checking problems and is the primary focus of this thesis. In case of such problems the correctness property $\varphi$ is specified using a *temporal logical formula*.

The biggest challenge of model checking is dealing with the problem of *state space explosion*[6]. As the number of state variables in a system increases, the size of the system's

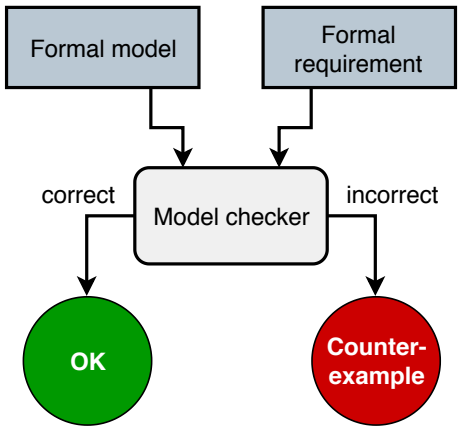


**Figure 2.1:** An illustration of model checking

state space grows exponentially, which makes its exploration impossible in practice. Various approaches have been developed to tackle this problem, including *bounded model checking*[3], *symbolic model checking*[4] and *abstraction*[5].

## 2.2 Modeling formalisms

In order to be able to reason about the correctness of a system using formal verification techniques the model of the system needs to be defined with mathematical precision [19]. A model can be called formal if it has a well-defined syntax and precise semantics.

Choosing a suitable abstraction-level for the model is a crucial question when designing a formal verification method. Lower-level models are usually easier to handle mathematically, but can be harder to comprehend for humans, meaning they are impractical for direct modeling of a system. A good example of low-level models are *Kripke structures* which are often used in model checking. Higher level models on the other hand allow engineers to model more efficiently by abstracting away less important details. Some examples of these models are *Control flow automata* and *Statecharts*. These higher level models can be transformed to lower-level ones, enabling efficient lower-level algorithms to be used when verifying them. This process is called *state space generation* or *exploration* and can easily lead to *state space explosion*.

### 2.2.1 Kripke structures

Kripke structures are finite directed graphs whose vertices are labeled with sets of atomic propositions [7][18]. We call the nodes of these graphs *states* and the edges *transitions*. Paths in these graphs (alternating sequences of states and transitions) represent possible behaviours of our system.

**Definition 1 (Kripke structure).** Given a set of atomic propositions $AP = \{p, q, ...\}$, a (finite) Kripke structure is a 4-tuple $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, L \rangle$, where:

- $\mathcal{S} = \{s_1, ..., s_n\}$ is the (finite) set of states;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation consisting of state pairs $(s_i, s_j)$;

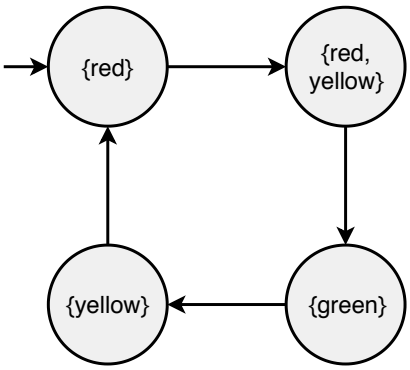


**Figure 2.2:** A Kripke structure modeling a traffic light.

- $L : \mathcal{S} \to 2^{\mathrm{AP}}$ is the labeling function that maps a set of atomic propositions to each state. ▪

For a state $s \in \mathcal{S}$, the set $L(s)$ represents the set of atomic propositions that are true when the system is in state $s$, and the set $AP \setminus L(s)$ contains the propositions that are false in state s. We assume the transition relation $\mathcal{R}$ is defined as left-total, meaning for all $s_i \in \mathcal{S}$, there exists $s_j \in \mathcal{S}$ such that $(s_i, s_j) \in \mathcal{R}$, i.e., that all states have non-zero outdegree. This ensures that all finite paths can be extended to an infinite path. A deadlock state can be modeled by a state having a single outgoing edge back to itself. We define a *path* in $M$ as an infinite sequence $\rho \in \mathcal{S}^\omega$ with $\rho(0) \in \mathcal{I}$ and $(\rho(i), \rho(i+1)) \in \mathcal{R}$ for every $i \geq 0$.

### 2.2.2 Statecharts

The defining characteristic of reactive systems is that they are event-driven, they have to continuously react to outer and inner events [16]. These systems appear everywhere in our daily lives, we interact with them without even knowing they belong in this category. For example operating systems, avionics systems, ATMs or even microwave ovens can be categorized as reactive systems. As these examples show, reactive systems often appear in areas, where safety-critical operation is crucial, as even the slightest misbehaviour can have catastrophic consequences. This makes the verification of these systems an important part of their design process.

Reactive systems can only be verified using model checking techniques if they are represented by mathematically precise models. *Statecharts*[17] provide an intuitive way for describing the behaviour of reactive systems, and are at the same time formal and rigorous. Statecharts are an extension of finite state machines, introducing hierarchy, orthogonality and broadcast communication.

Figure 2.3 shows the statechart representation of a traffic light. There are two states at the upmost level of the state hierarchy, *Off* and *On*. *On* is a composite state, it contains 3 substates, *Red*, *Green* and *Yellow*. *Switch* and *toggle* are outer events. The black dots represent entry points, their single outgoing edge leads to the initial state of their regions.

This thesis proposes an efficient way of verifying temporal logical properties of models created in the Gamma Statechart Composition Framework[1], and thus models of the YAKINDU Statechart Tools[2] as well. Gamma statecharts can be transformed to Extended symbolic transition systems (Section 2.2.4), a lower-level formalism.

[1] http://gamma.inf.mit.bme.hu
[2] https://www.itemis.com/en/yakindu/state-machine/

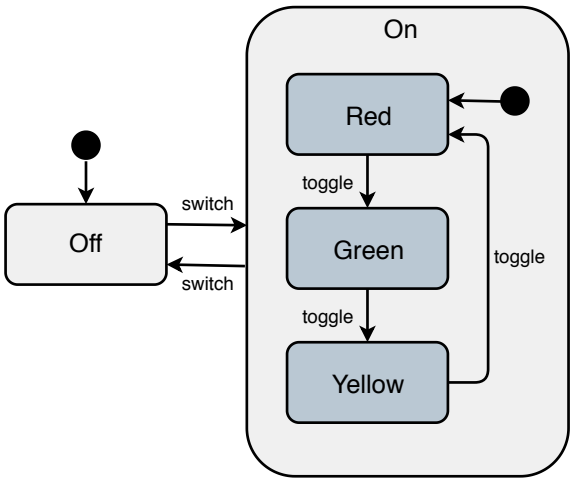


**Figure 2.3:** The statechart model of a traffic light.

### 2.2.3 Control flow automata

The main focus of this thesis is the verification of reactive systems, but the constructed algorithm, while being optimized for this domain, can be used to check the correctness of program code as well. Programming languages are usually designed to be human-readable and are easy to use for programmers, but are practically impossible to process mathematically. To allow program code to be reasoned about using methods of formal verification, it needs to be transformed to a formal model, in our case a *Control flow automaton* (CFA)[2]. This transformation can be carried out automatically, for example by the verification compiler proposed by Gyula Sallai[21], which is capable of transforming a subset of the *C* programming language to *Control flow automata*.

**Definition 2 (Control flow automaton).** We define a *Control flow automaton* as a 4-tuple $\langle V, L, l_0, l_{end}, E \rangle$, where:

- $V = \{v_0, v_1, ...\}$ is the set of *variables*. $\forall v_i \in V$ variable has a $D_{v_i}$ domain;
- $L = \{l_0, l_1, ...\}$ is the set of *control locations*, which model the program counter;
- $l_0 \in L$ is the initial location;
- $l_{end} \in L$ is the ending location;
- $E \subseteq L \times Ops \times L$ is the set of *transitions* ▪

A *state* is a tuple $s = (l, d_1, d_2, ..., d_n)$, where $l \in L$ is a control location and $d_i \in D_{v_i}$ is the value assigned to variable $v_i \in V$. A *transition* is a directed arc between two *locations* labeled with statement. A statement can be:

- An *assumption* statement ($[x == 0]$). A transition labeled with an assumption statement can only fire if the condition of the assumption statement evaluates to *true* in the source location;
- An *assignment* statement ($x := y + 1$). An assignment statement consist of a reference to a variable and an expression that will be assigned to it;
- A *havoc* statement (havoc $x$). A havoc statement is a non-deterministic assignment, the variable is assigned a random value from its domain.

A transition can be labeled with multiple statements, these statements are executed consecutively. Ending locations allow for finite paths to exist in our model. However, we only want to reason about infinite paths, so we extend these finite paths to infinite ones by adding the ending state infinitely many times to the end of any path that reached the ending location, i.e., by assuming that an assumption transition with the condition $[true]$ exists that starts and ends in the ending location and only this transition fires after the ending location is reached.

CFA can be visualized as directed graphs, whose nodes represent the control locations and whose edges represent the transitions. The ending location has a dashed border. The starting location has an incoming edge without a source node. See Figure 5.2 for an example a CFA.

Control flow automata can be illustrated with directed graphs, whose nodes and edges correspond to the locations and transitions of the automata. Edges are labeled with statements. The initial location is denoted with an incoming edge that has no source node. In graphical representations the ending node has double borders.

### 2.2.4 Extended symbolic transition systems

The *extended symbolic transition system* (xSTS) formalism is an extension to symbolic transition systems [15].

**Definition 3 (Extended symbolic transition system).** An *extended symbolic transition system* is a tuple $T = \langle V, Tran, Init \rangle$, where:

- $V = v_1, v_2, ..., v_n$ is the set of variables, with domains $D_{v_1}, D_{v_2}, ..., D_{v_n}$;
- $Tran$ is the set of transitions;
- $Init$ is the initial transition ▪

A *state* is a tuple $s = (d_1, d_2, ..., d_n)$, where $d_i \in D_{v_i}$ is the value assigned to variable $v_i \in V$. A *transition* is built up of *actions*. Actions can be defined as follows:

- An *assignment action* is an atomic action describing the assignment of a value to a variable;
- An *assumption action* is an atomic action describing the expectation of a certain condition. It contains a single expression that has to evaluate to *true*, otherwise that certain action branch (see nondeterministic actions for the description of branches) containing the false assume action cannot be executed;
- A *sequential action* is a composite action that contains one or more actions that are executed in order, one after another;
- A *parallel action* is a composite action that contains one or more branches. Upon execution one of the branches is selected nondeterministically.

Transitions are atomic in the sense that either all their actions are executed or none.

## 2.3 Abstraction-based model checking

The two most well-known abstraction based methods tackle the problem of state stace explosion differently:

- *Explicit value abstraction*[2] reduces the state space by marking a subset of the variables *untracked*. For example if our model has the variables $x$ and $y$, then we might only track $x$ and mark $y$ as *untracked*. This means $y$ can take any value from its domain.
- *Predicate abstraction* works by only tracking if the *predicates* defined by the current abstraction precision evaluate to *true* of *false*. These predicates are logical expressions, which can contain variables of the model, for example $(x > 0)$, $(x > y)$ or $(true)$.

These methods work by analyzing the behaviour of abstract, simpler models, which contain less information than the original ones. One would think that losing information leads to incorrect analyses. However, both of these methods are *over-approximations*, meaning they only add behaviours to the model and do not remove any. When looking for counterexamples *false positives* can occur, but no *false negatives*. If no counterexample is found in the abstract model, then the original model doesn't contain one either. On the other hand, finding a counterexample in the abstract model isn't enough to be certain about the existence of one in the original model. The verification algorithm has to check if the counterexamples obtained through abstraction are concretizable, i.e., if the path they describe can be traversed in the original model.

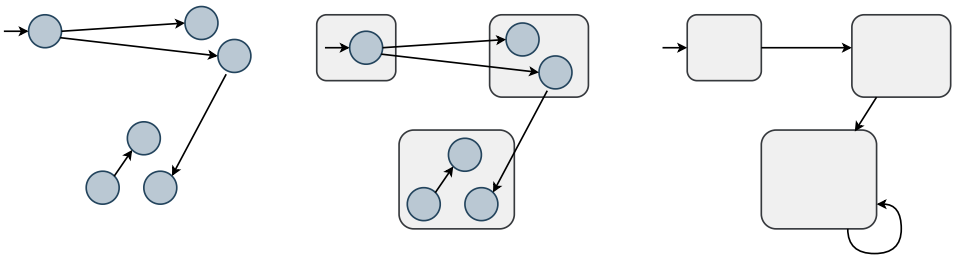

**Figure 2.4:** An illustration of abstraction

When analyzing the abstract model we explore the abstract *state space*, which is a set of *abstract states* and *abstract transitions*. An abstract state can contain multiple (even an infinite number) of concrete states. All concrete states belong to at least one abstract state, and each concrete state can belong to at most one abstract state.

### 2.3.1 The CEGAR-algorithm

*Counterexample-guided abstraction refinement* (CEGAR) [5] is an abstraction-based model checking algorithm. The CEGAR-loop (Figure 2.5) is the heart of this algorithm. The verification process starts with an initial *precision*. Depending on the type of abstraction we use, a precision can either be a list of tracked variables (explicit value abstraction), a list of tracked predicates (predicate abstraction), or something else in case of a different abstraction-method. In every iteration of the loop the *abstractor* component checks if an *abstract counterexample* can be found in the abstract model with the current precision. There are two possibilities after this:

- If no counterexamples are found, then the model is deemed *correct*, as *over-approximation* means the original model doesn't contain one either.
- If a counterexample is found, the *refiner* component checks if it's concretizable:
  - If it is, then the counterexample is valid in the original model as well, so the model is deemed *incorrect*.
  - If not, then the refiner returns a *refined precision*, and the loop starts again. Ideally, this refined precision adds just enough detail to the abstract model, that the same false counterexample can't surface again.

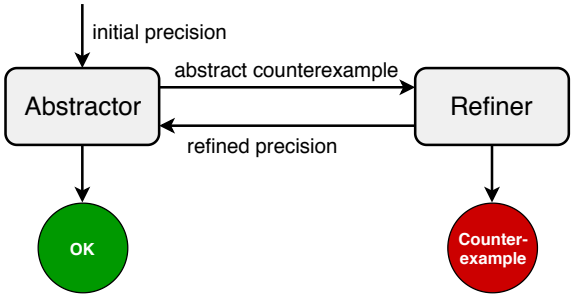


**Figure 2.5:** The CEGAR-loop

The loop keeps running until the model is deemed either *correct* or *incorrect*. In extreme cases the abstract state space can become so refined, that each abstract state contains only one concrete state, i.e., the abstract state space becomes equal to the concrete state space.

### 2.3.2 The abstractor

The verification algorithm presented in this thesis uses a predicate abstraction based CEGAR-algorithm for state space exploration. When using predicate abstraction we only track if certain expressions that contain the variables of the model evaluate to true of false. These expressions are called *predicates*.

*Predicates* are Boolean formulas over $V$, for example $(x > 0)$, $(x + 2 == y)$. We call the set of tracked predicates *precision*, $P = \{p_0, ..., p_n\}$. The abstract state space is denoted with $\widehat{S}$. An abstract state contains all concrete states in which the associated predicates evaluate to *true*:

- In case of an xSTS, an abstract state is an $\widehat{s} = (\widehat{p_0}, ..., \widehat{p_k}) \in \widehat{S}$ n-tuple.
- In case of a CFA, an abstract state is an $\widehat{s} = (l_i, \widehat{p_0}, ..., \widehat{p_k}) \in \widehat{S}$ n-tuple, where $l_i \in L$ is the control location. Abstract states can only contain concrete states of the same control location.

$\widehat{p_i} = p_i$ if $p_i$ evaluates to *true* in the concrete states, $\widehat{p_i} = \neg p_i$ if $p_i$ evaluates to *false*, and $\widehat{p_i} = true$ if $p_i$ cannot be evaluated (for example if a variable hasn't been initialized yet). Given an abstract state $\widehat{s} \in \widehat{S}$, let its label be $Label(\widehat{s}) = \bigwedge_{p \in \widehat{s}} p$, i.e., the conjunction of predicates (or their negations) in $\widehat{s}$. A concrete state $s$ is mapped to $\widehat{s}$ if $s| = Label(\widehat{s})$.

CEGAR is usually used for safety checking. In case of such problems a set of states are marked as *erroneous*, and the model checking task is to determine if any of the erroneous

states are reachable from the initial state. A counterexample is a path starting in the initial state and leading up to an erroneous state in this case.

### 2.3.3 The refiner

The refiner component has two main tasks:

- To determine whether the abstract counterexample is a valid path in the concrete model;
- To return a new predicate to refine the precision in case the abstract counterexample is not concretizable.

In my work I used *interpolation-based* refinement [14][15]. This refinement method uses Craig interpolation to extend the set of tracked predicates.

**Definition 4 (Craig interpolant).** Let $A$ and $B$ be first-order logic formulas such that $A \wedge B$ is unsatisfiable. A formula $I$ is a Craig interpolant (or simply an interpolant) for $(A, B)$ if the following properties hold [10]:

- $A \implies I$;
- $I \wedge B$ is unsatisfiable;
- $I$ refers only to common symbols of $A$ and $B$ (excluding the symbols of the logic). ▪

In model checking Craig interpolants are usually used as explanations of why a trace can't be concretized. They can be used to split abstract states into two parts, if the first set of concrete states can be described with the formula $A$ and the second set with the formula $B$. When an interpolant between $A$ and $B$ is added to our precision, the concrete states that $A$ and $B$ characterize will be mapped to different abstract states.

Let $\widehat{\pi} = (\widehat{s}_1, t_1, \widehat{s}_2, t_2, \widehat{s}_3, ..., t_{n-1}, \widehat{s}_n)$ be the abstract counterexample, where $\widehat{s}_i \in \widehat{S}$ is an abstract state and $t_i$ is the transition between $\widehat{s}_i$ and $\widehat{s}_{i+1}$. In this case the erroneous state is $\widehat{s}_n$.

In order to be able to refine the counterexample trace using interpolation it needs to be transformed to a list of constraints. This transformation consists of two steps. First, the states and transitions need to be expressed as logical formulas, denoted with $Label(\widehat{s})$ in case of states and $Label(t)$ in case of transitions. In case of states this expression is the conjunction of the predicates that apply to a state, as defined in Section 2.3.2. In case of transitions, we need to transform each statement to an expression $Label(t)$ the following way:

- An assignment statement can be expressed with an expression stating the next value of the left operand variable is equal to the assigned value. The next value of a variable can be referenced using the *prime* operator. For example the assignment $x := x + 1$ can be expressed as $x' = x + 1$;
- Assumption statements can be expressed using the conditions (as a formula). For example, the assumption statement $[x > 0]$ is expressed as a formula $x > 0$;
- A havoc statement can be expressed as an empty expression.

After all states and transitions are expressed as logical formulas, their labels need to be unfolded to the *static single assignment form*. In the static single assignment form each variable is assigned exactly once and every variable is defined before it is used. This can be achieved by replacing each variable with an indexed form of itself. Each state has its own indexing. All indices start at 0 and only get incremented after *assignment* and *havoc* statements. A primed variable is substituted using an increment of the current index of the variable. We denote with $Label(t)_i$ the indexing of state $\widehat{s}_i$ applied to the label of the transition $t$. Similarly, $Label(\widehat{s})_i$ denotes $Label(\widehat{s})$ with the indexing of state $\widehat{s}_i$ applied to it. Figure 2.6 illustrates the transformation process using an example.

```
x := 0                    x_0 = 0
y := 0                    y_0 = 0
[y = x + 2]               y_0 = x_0 + 2
x := x + 1                x_1 = x_0 + 1
[x > y]                   x_1 > y_0
```

**Figure 2.6:** An illustration of statement to constraint transformation.

Next, the refiner determines what is the furthest reachable state in the counterexample starting from the initial state. Let this state be $\widehat{s}_f$. This state is reachable, meaning $Label(\widehat{s}_1)_1 \wedge Label(t_1)_1 \wedge Label(\widehat{s}_2)_2 \wedge ... \wedge Label(t_{f-1})_{f-1} \wedge Label(\widehat{s}_f)_f$ is satisfiable. However, $Label(\widehat{s}_1)_1 \wedge Label(t_1)_1 \wedge Label(\widehat{s}_2)_2 \wedge ... \wedge Label(t_{f-1})_{f-1} \wedge Label(\widehat{s}_f)_f \wedge Label(t_f)_f \wedge Label(\widehat{s}_{f+1})_{f+1}$ isn't as $\widehat{s}_f$ is the furthest reachable state of the counterexample. In this case we call the state $\widehat{s}_f$ a *failure* state since a concrete path leads there but it cannot be extended any further. Please note that the *erroneous* state and the *failure* state do not refer to the same thing (even though they can be the same state). The *erroneous* state refers to the last state of the counterexample, the state about which we want to decide if it's reachable or not, while the *failure* state refers to the furthest reachable state of the counterexample. The set of concrete states mapped to the failure state $\widehat{s}_f$ are partitioned into the following three groups: states that can be reached from an initial state are *dead-end*, states having a transition to $\widehat{s}_{f+1}$ are *bad*, while other states are *irrelevant*. It is clear that a state cannot be dead-end and bad at the same time since then $\widehat{s}_f$ would not be a failure state.

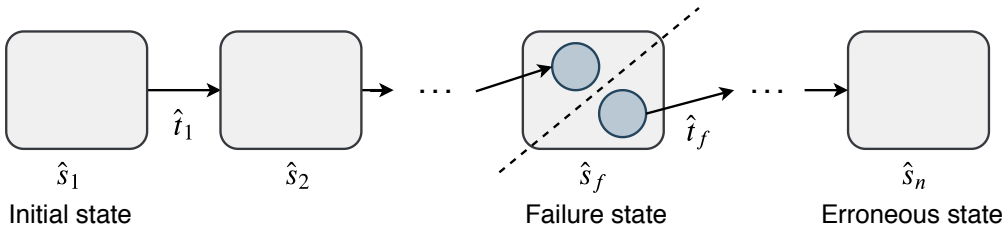


**Figure 2.7:** Illustration of the refinement. Lighter rectangles denote abstract states, while darker circles denote concrete states.

The purpose of abstraction refinement is to map dead-end and bad states to different abstract states so that the spurious counterexample cannot occur in the next iteration. Predicate abstraction uses predicate refinement to obtain new predicates.

Dead-end and bad states can be characterized with formulas $D$ and $B$ respectively in the following way.

$$D = Label(\widehat{s}_1)_1 \wedge Label(t_1)_1 \wedge Label(\widehat{s}_2)_2 \wedge ... \wedge Label(t_{f-1})_{f-1} \wedge Label(\widehat{s}_f)_f$$
$$B = Label(t_f)_f \wedge Label(\widehat{s}_{f+1})_{f+1}$$

It is clear that $D \wedge B$ is unsatisfiable, otherwise a longer prefix could be found or $\widehat{\pi}$ would be concretizable. Consequently, Craig interpolation can be applied, yielding an interpolant $I$ with the following properties.

- $D \rightarrow I$, i.e., $I$ is a generalization of dead-end states,
- $I \wedge B$ is unsatisfiable, i.e., bad states cannot satisfy $I$,
- $I$ refers only to common symbols of $D$ and $B$, which are variables with indexing of state $\widehat{s}_f$.

Therefore, removing the indices from the variables in $I$ yields a new predicate that separates dead-end and bad states mapped to $\widehat{s}_f$.

We need to remove the indices from the interpolant we got so that it isn't in static single assignment form. This means that we have to replace variables in their indexed form with their unindexed form. We denote with $a_v$ the index of variable $v \in V$ in state $\widehat{s}_f$. The fact that past values of variables can't be referenced in our expressions (only future using the prime operator), and the characteristic of Craig interpolation that states the interpolant can only contain symbols that are present both in $D$ and $B$, mean that for every variable $v \in V$, the index with which $v$ appears in the interpolant is equal to or greater than $a_v$. When folding in the interpolant if variable $v$ appears with an index $a$, then:

- if $a = a_v$, then we simply replace it with its unindexed form;
- if $a > a_v$, then we apply the prime operator to it $a_v - a$ times.

For example if we have the variables $x$ and $y$ with corresponding indices $a_x = 2$ and $a_y = 3$ state $\widehat{s}_f$, then the interpolant $x_2 + y_4 > x_4$ is going to be unfolded to $x + y' > x''$.

## 2.4 Linear temporal logic model checking

Temporal logics [13] were developed by philosophers and linguists but later found use in model checking. They describe the ordering of events in time without introducing time explicitly. A formula can specify for example that a state has to occur some time in the future, or that it can never occur. Temporal logics describe these relations using temporal operators, which can be nested in each other or combined with logical operators. Depending on the structure of time *linear* and *branching* time temporal logics are distinguished. This thesis focuses on linear temporal logic (LTL), which is detailed below.

### 2.4.1 Linear temporal logic

LTL has a linear time model, meaning that it is interpreted over a single realized future behavior of a system, i.e., over a linear sequence of states.

In my work I used the following temporal operators [7][11][20]:

- **X** $\varphi$ ("ne**X**t"): $\varphi$ holds in the next state
- **G** $\varphi$ ("**G**lobally"): $\varphi$ holds somewhere on the subsequent path
- **F** $\varphi$ ("**F**uture"): $\varphi$ holds in all states of the subsequent path
- $\varphi$ **U** $\psi$ ("**U**ntil"): $\psi$ holds at some state in the future, and $\varphi$ holds at all states until $\psi$ holds
- $\varphi$ **R** $\psi$ ("**R**elease"): $\psi$ has to be true until $\varphi \wedge \psi$ becomes true ($\psi$ should stay true if $\varphi \wedge \psi$ never becomes true)

The syntax of LTL expressions is defined as follows [7]:

- All $p \in AP$ atomic propositions are LTL-formulas
- If $\varphi$ and $\psi$ are LTL-formulas, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{X}\varphi$, $\mathbf{G}\varphi$, $\mathbf{F}\varphi$, $\varphi\mathbf{U}\psi$ and $\varphi\mathbf{R}\psi$ are LTL-formulas

The *negation normal form* of LTL-formulas is usually used when transforming them to automata. In negation normal form:

- all negations appear only in front of the atomic propositions;
- only the logical operators $true$, $false$, $\wedge$ and $\vee$ can appear;
- only the temporal operators **X**, **U** and **R** can appear.

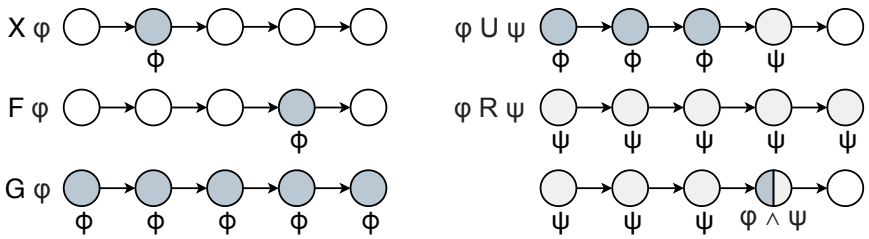


**Figure 2.8:** Illustration of temporal operators

### 2.4.2 Automata theory

Automata are simple mathematical models that model computations over sequences of input symbols. In formal language theory they are usually used as finite representations of infinite languages.

**Definition 5 (Finite automaton).** A finite automaton is a 5-tuple $\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, q_0, F)$, where:

- $\mathcal{Q}$ is the set of states;
- $\Sigma$ is the finite alphabet, i.e. the set of possible symbols;
- $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation, a set of state-symbol-state triples;
- $q_0 \in \mathcal{Q}$ is the initial state;

- $F \in \mathcal{Q}$ is the set of accepting states. ▪

For a given *word* $w = a_1 a_2 ... a_n \in \Sigma^*$, the behaviour of the system if defined as follows:

**Definition 6 (Run).** A $\rho_w = r_0, r_1, ... r_n$ $(r_i \in \mathcal{Q})$ sequence of states is a *run* reading the word $w = a_1 a_2 ... a_n$, if $r_0 = q_0$ and $(r_{i-1}, a_i, r_i) \in \delta$ for all $i = 1, ..., n$. ▪

In simpler terms, the automaton starts in the $q_0$ state, and steps according to the transition function for each symbol of the input word. A word $w$ of length $|w| = n$ is *accepted* if the run reading it ends in an accepting state, i.e., if $r_n \in F$. If the run doesn't end in an accepting state, or the automaton reads a symbol for which no transitions are defined in the current state, then the word is *rejected*. The *language* $L(\mathcal{A}) \subseteq \Sigma^*$ is the set of words that the automaton $\mathcal{A}$ accepts.

An automaton can be visualised using an edge-labeled directed graph, whose nodes represent elements of $\mathcal{Q}$ and whose edges represent elements of $\delta$.

Finite automata can only recognize finite words, but the systems we would like to analyze usually do not stop during their normal operation. The different possible behaviours of these systems can be modeled using infinite paths in Kripke structures, which can be considered as *infinite words* in terms of automata theory.

#### 2.4.2.1 Büchi automata

*Büchi automata* are finite automata that work on infinite words[7]. They consist of the same components as the finite automata we discussed above, the difference lies in the acceptance condition. Büchi automata read $w \in \Sigma^\omega$ infinite words. Let $inf(\rho_w)$ denote the set of states that appear infinitely often in $\rho_w$. A run $\rho_w$ is considered accepting, if it visits at least one accepting state infinitely often. Formally, if $inf(\rho_w) \cap F \neq \emptyset$. A word $w$ is accepted by an automaton $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on $w$.

An extension to Büchi automata are *generalized Büchi automata*. Such automata can have multiple acceptance sets. A run is accepted if at least one member of each acceptance set appears infinitely often in it. Generalized Büchi automata can be transformed to Büchi automata.

Büchi automata can be used to recognize so called $\omega$-regular languages, just like LTL-expressions[24]. Nondeterministic Büchi automata can recognize every $\omega$-regular language. In contrast linear temporal formulae can only recognize so called star-free languages, a strict subset of $\omega$-regular languages, meaning Büchi automata are more expressive than LTL-expressions.

#### 2.4.2.2 Transforming LTL-expressions to Büchi Automata

As LTL-expressions can only characterize a strict subset of $\omega$-regular languages, all LTL-expressions can be transformed to equivalent Büchi automata. The algorithm of Gerth, Peled, Vardi and Wolper [12] can transform LTL-expressions in negation normal form to generalized Büchi automata. I do not detail this algorithm here, as I only used it as a black box in my work.

In my work I used the LTL to Büchi Automata translator implementation of the Spot framework[3] [11]. The translator does not require the LTL-formula to be given in negation

[3] https://spot.lrde.epita.fr

normal form. The translator has an input flag ("-B"), with which a Büchi automaton is marked as the desired output instead of a generalized Büchi automaton. If this is used the translator degeneralizes the output automaton.

### 2.4.3 Automata theoretic model checking

Automata theoretic model checking provides a way of making LTL model checking computationally easier by reducing it to a language emptiness problem [19].

Given a Kripke structure $M$ and an LTL-formula $\varphi$ using the same atomic propositions $AP$, let $L(M)$ and $L(\varphi)$ denote the language that the Kripke structure can produce and the language that the LTL-formula specifies. $L(M)$ contains all possible behaviours of our model and $L(\varphi)$ contains all behaviours that comply with the correctness specification. The ltl model checking problem can now be restated as follows: is the set of provided behaviours a subset of the valid behaviours, does $L(M) \subseteq L(\varphi)$ hold?

An equivalent formalization is $L(M) \cap \overline{L(\varphi)} \stackrel{?}{=} \emptyset$, where $\overline{L(\varphi)}$ is the complement of the language $L(\varphi)$. Complementation is computationally hard, but it can avoided in case of LTL model checking, by utilizing that the complement of the language of an LTL-formula is the language of the negated formula: $\overline{L(\varphi)} \equiv L(\neg\varphi)$. This allows the model checking problem to be reduced to language intersection and language emptiness, both of which can be efficiently computed on Büchi automata.

#### 2.4.3.1 Algorithms for checking language emptiness

A possible way of checking the language emptiness of a Büchi automaton is checking whether at least one strongly connected component (SCC) that contains an accepting state is reachable from the initial state [22]. If such a strongly connected component is reachable, then the Kripke structure can produce words that contain infinitely many accepting states, fulfilling the acceptance condition of Büchi automata.

**Definition 7 (Strongly connected component).** A component of a directed graph is *strongly connected*, if a path exists in both directions between each pair of vertices of the component. ▪

Robert Tarjan proposed a linear time algorithm for finding SCCs in 1972 [23]. The algorithm is based on *depth-first search* (DFS) and determines which nodes belong to each SCC using clever indexing.

Algorithms based on *Nested DFS* offer a different approach to checking language emptiness. These algorithms usually conduct two depth-first searches, the former one to find and sort accepting states, and the latter one to find cycles around accepting states. The first Nested DFS algorithm (Algorithm 1) was proposed by Courcoubetis, Vardi, Wolper, and Yannakakis in 1992 [9]. Two bits per state, a red bit and a blue bit are used to keep track of which states have been visited by the two depth-first searches. Both bits start with the value $false$. The initial state is denoted by $q_0$ and $post(q)$ refers to the set of possible next-states (i.e. out-neighbours) of the state $q$. When a cycle is found a counterexample can be easily obtained from the call stack.

**Algorithm 1** Nested DFS

```
1:  procedure nested_dfs
2:      call dfs_blue(q_0);
3:  procedure dfs_red(q)
4:      q.red := true;
5:      for all t ∈ post(q) do
6:          if ¬t.red then
7:              call dfs_red(t);
8:          else
9:              if t = seed then
10:                 report cycle;
11: procedure dfs_blue(q)
12:     q.blue := true;
13:     for all t ∈ post(q) do
14:         if ¬t.blue then
15:             call dfs_blue(t);
16:     if q ∈ F then
17:         seed := q;
18:         call dfs_red(q);
```

## 2.5 Related work

In this section I briefly present related results in the domain of model checking.

### 2.5.1 Related theoretical domains

Bounded model checking[3] algorithms look for counterexamples of length equal to or shorter than $k$ at a time, and encode them as SAT-problems. The process is repeated with larger and larger values of $k$ until all possible violations are ruled out.

In symbolic model checking[19], states and state transitions are not simply enumerated, but represented symbolically by functions, encoded in efficient data structures such as decision diagrams.

Partial order reduction[8] is a state space reduction technique used in model checking of concurrent processes. It exploits the commutativity of actions, whose order of execution compared to each other does not matter.

### 2.5.2 Related model checking frameworks

The *SPIN*[4] model checker is a formal verification tool for multithreaded applications. The tool allows for verification of LTL properties on models defined in a language called Promela. SPIN uses explicit model checking techniques.

*NuSMV*[5] is a symbolic model checker based on binary decision diagrams that supports the analysis of specifications expressed as CTL and LTL formulae. NuSMV uses a different approach for LTL model checking, based on decision diagrams.

*PertiDotNet*[6] is a framework for the editing, simulation and analysis of Petri nets. It supports saturation-based techniques for LTL and CTL model checking and CEGAR-based reachability analysis on Petri nets. However, CEGAR-based analysis is not integrated with LTL model checking.

CEGAR-based approaches are used for software and also or high-level models. However, they are usually not capable of verifying more complex properties, such as LTL.

[4] http://spinroot.com
[5] http://nusmv.fbk.eu/
[6] https://inf.mit.bme.hu/research/tools/petridotnet

# Chapter 3

# LTL model checking of critical systems

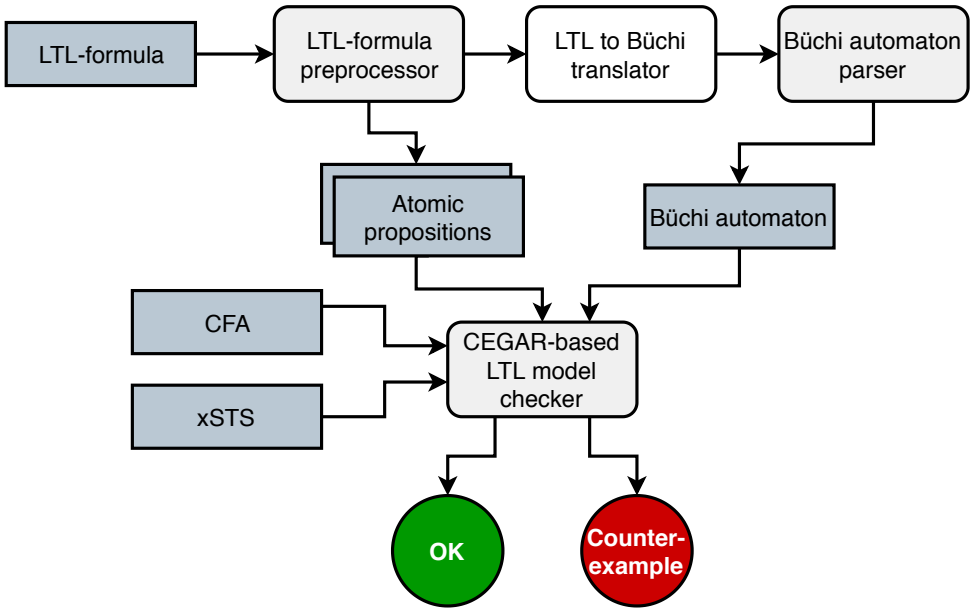


**Figure 3.1:** Overview of the approach

## 3.1 Overview of the approach

In my work, I developed a solution to support engineers in the verification of high-level engineering models. My work focused on two main aspects:

- Integration of the verification engine into high-level modeling tools using an intermediate formal representation and providing the algorithms for this representation. I also provide verification support for simple programs in the C programming language.
- I developed a new algorithm that exploits the efficiency of abstraction in LTL model checking.

During my work, several challenges had to be solved. First, theoretical and algorithmic developments were needed to support Linear Temporal Logic in CEGAR-based model checking. As LTL model checking is reduced to finding strongly connected components (cycles) in the state space, I had to develop new search and refinement strategies. Secondly, I had to adapt my algorithm to work on the intermediate models that the high-level engineering models can be transformed to. A result of this is that my novel algorithm can not only be used to verify low-level formal models, but also supports the verification of statecharts and C source code. In the following sections I detail my work.

## 3.2 Integration into the engineering workflow

Engineers use high-level engineering languages in system design, so my approach was designed to support the engineers to use the concept of hidden formal methods. The goal was to hide formal methods from the engineers in a way that they only have to be familiar with high-level modeling languages and they do not need knowledge about formal models/methods and verification. To achieve this high-level goal, I integrated my model checker into the Gamma Statechart Composition Framework[1]. The models of this framework are based on statecharts. The users can define components, which are the basic building blocks of Gamma models. Composite components can contain instances of other components and define channels between them for communication, enabling users to model highly complex systems using a standardised formalism. The framework already supports model checking, but only of CTL properties using the UPPAAL[2] model checker. The drawback of UPPAAL is that it does not use abstraction and it supports only a very limited subset of CTL. Despite the fact that LTL model checking is a more complex problem than CTL model checking, LTL can be used more efficiently by the engineers. [8]

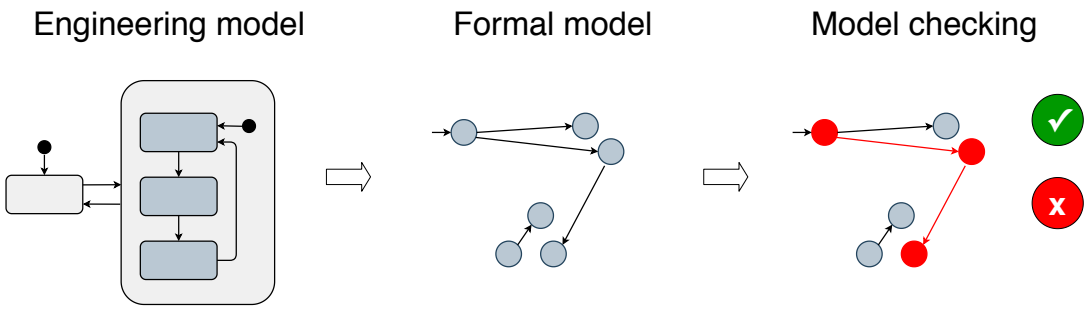


**Figure 3.2:** Integration into the engineering workflow

These composite statechart models are too complex to be checked efficiently using mathematical methods. However, Gamma models can be transformed to xSTS models, a formalism which is more suited for model checking. This is why I decided to fit my algorithm to xSTS models as well and enable engineers to check their complex models using my tool.

## 3.3 LTL-formula preprocessor

My goal was to provide the engineers using my tool with an intuitive interface, so that they can easily integrate model checking techniques into their development workflow. In

[1] http://gamma.inf.mit.bme.hu
[2] http://www.uppaal.org

order to achieve this I extended linear temporal logics in such a way that enables engineers to formulate relevant requirements about their systems.

Per definition linear temporal logics only allow us to reason about *atomic propositions*. In contrast, in my approach linear temporal logical formulas can be used to formulate requirements about any logical formula over $V$ (the set of variables in our model). An example of such a formula is $G(x > 5)$, which states that the value of the variable $x$ is always greater than 5. The task of the LTL-formula preprocessor is to transform these extended formulas to regular LTL-formulas to allow the use of existing LTL-modification methods on them. In our previous example this would mean transforming $G(x > 5)$ to $G(p)$, where $p$ is an atomic proposition that applies to a state if and only if in that state the logical formula $x > 5$ evaluates to true.

This extended formalism can be described with the following context-free grammar, where $n \in \mathbb{Z}$ is an integer and $v \in V$ is a variable:

$$\begin{aligned}\varphi ::= &\top \mid \bot \mid v \mid n \mid \neg\varphi \mid [\varphi \wedge \varphi] \mid [\varphi \vee \varphi] \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \mathbf{X}\varphi \mid [\varphi \mathbf{U} \varphi] \mid [\varphi \mathbf{R} \varphi] \mid \\ &[\varphi \Rightarrow \varphi] \mid [\varphi > \varphi] \mid [\varphi < \varphi] \mid [\varphi \geq \varphi] \mid [\varphi \leq \varphi] \mid [\varphi = \varphi] \mid \\ &[\varphi + \varphi] \mid [\varphi - \varphi] \mid \mid [\varphi * \varphi] \mid [\varphi / \varphi] \mid [\varphi \% \varphi] \mid (\varphi)\end{aligned}$$

Operator precedence can be seen in Table 3.1 (with increasing priority from top to bottom).

| Lower ↓ Higher | Operators |
|---|---|
| | $\Rightarrow$ |
| | $\vee$ |
| | $\wedge$ |
| | $\neg$ |
| | $\mathbf{U}, \mathbf{R}$ |
| | $\mathbf{F}, \mathbf{G}, \mathbf{X}$ |
| | $=$ |
| | $>, <, \leq, \geq$ |
| | $+, -$ |
| | $*, /, \%$ |
| | $v, n, \top, \bot$ |
| | $()$ |

**Table 3.1:** Operator precedence

A semantic analysis is required to detect invalid formulas, as not all words that can be generated using this grammar are valid. For example the word $G(true) > 5$ can be generated using this grammar, but is invalid, as an LTL-operator cannot be the operand of a relational operator. During this semantic analysis nodes of the syntax tree are assigned one of the three following types:

- *int-type* nodes can only have int-type children. The following nodes are always int-type: $n \in \mathbb{Z}$ integers, $+, -, *, /, \%$ operators. Variables $v \in V$ can be int-type depending on their declaration.
- *bool-type* nodes can belong to two subcategories:
  - *relational operators* can only have int-type children. These operators are the following: $>, <, =, \leq, \geq, \neq$.

- *boolean operators* are bool-type if they only have bool-type children. These operators are the following: $\wedge, \vee, \Rightarrow, \neg, \top, \bot$. Variables $v \in V$ can be bool-type depending on their declaration.

- *LTL-type* nodes can only have bool-type or LTL-type children.
  - The following operators are always LTL-type: $\mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}, \mathbf{R}$.
  - The boolean operators listed above ($\wedge, \vee, \Rightarrow, \neg$) are LTL-type if they have at least one LTL-type operand.

The parentheses operator can be of any type, it inherits the type of its child. Figure 3.5 shows an example of the types that get assigned to the nodes on the syntax tree of the formula $FG(x + 2 > 5) \wedge \bot$.

In order to be able to transform our extended formulas to LTL-formulas we need identify maximal bool-type subtrees (i.e. look for bool-type nodes, whose parent nodes are ltl-type) in our syntax tree, store them as logical formulas and replace them with different atomic propositions. Our previous example, $G(x > 5)$ would be transformed to the LTL-formula $G(p)$ this way, see Figure 3.3 for an illustration. Our other example, $FG(x + 2 > 5) \wedge \bot$ would be transformed to $FG(p_1) \wedge p_2$.

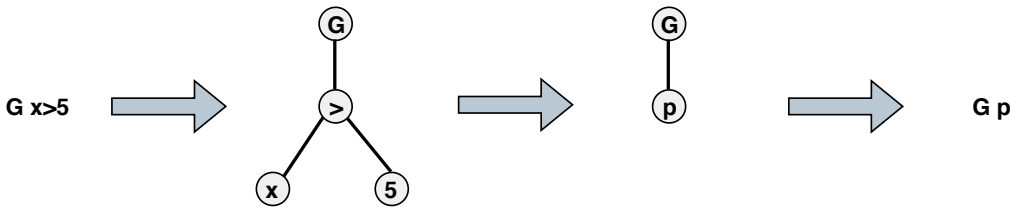


**Figure 3.3:** Illustration of the transformation process

The replaced logical formulas need to be stored and then evaluated in each state to determine if the corresponding atomic propositions are valid in them.

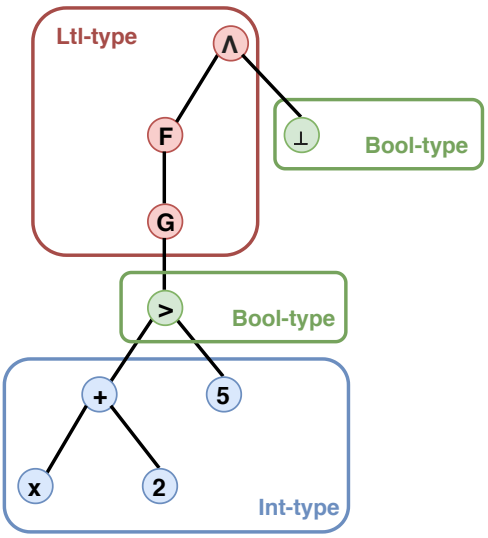


**Figure 3.4:** Illustration of the type system in action

## 3.4 CEGAR-based LTL model checking

In this section, I introduce the novel, CEGAR-based LTL model checking algorithm. This algorithm I developed is based on the combination of two model checking techniques that weren't used for this purpose together before, *counterexample-guided abstraction refinement* and *automata theoretic LTL model checking*. LTL model checkers have always struggled with performance. I propose that introducing counterexample-guided abstraction refinement can significantly increase performance in most cases of LTL model checking. The algorithm allows for "on the fly" model checking, meaning that usually only a portion of the state space has to be explored, further improving performance.

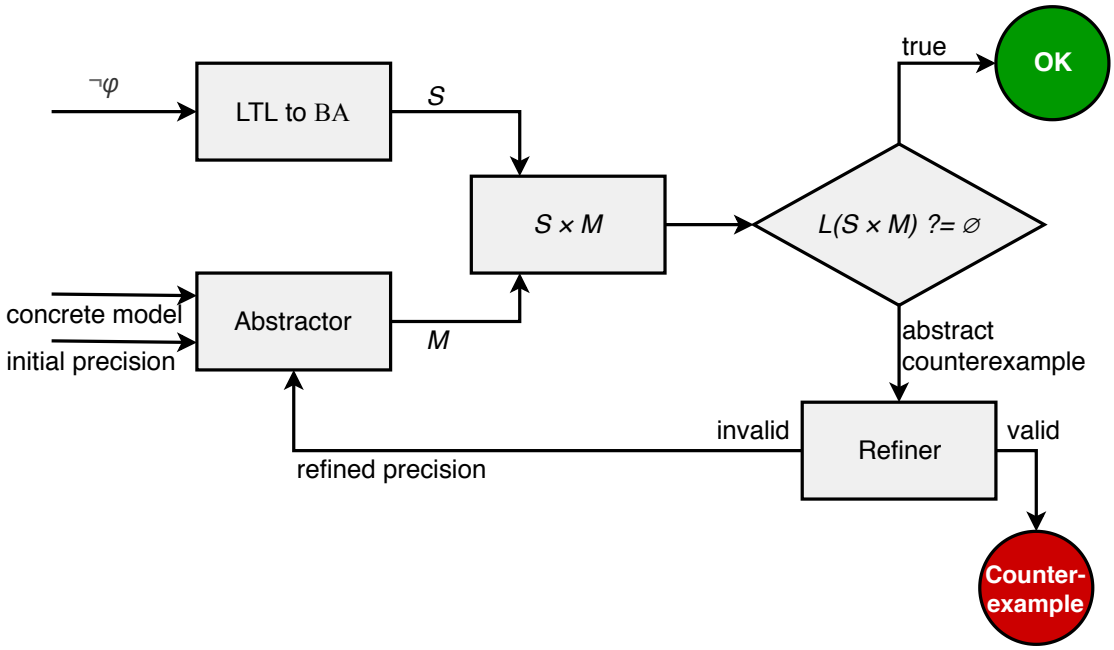


**Figure 3.5:** Overview of the algorithm

The algorithm has the following steps:

1. The requirement specification is given in the form of an LTL-formula $\varphi$. Negate this formula and transform it to an equivalent Büchi automaton $S$;
2. Apply abstraction to the concrete model with a given initial precision, calculate the abstract state space and represent it with an automaton $M$;
3. Calculate the product of the two automata $S \times M$. In this product the specification automaton steps based on the target state of the model automaton;
4. Check the language emptiness of the product automaton $S \times M$;
   - If the language of the automaton is empty, then the model meets the correctness specification as no counterexamples were found;
   - If a counterexample is found in the abstract state space, then verify whether it is valid in concrete state space as well;
     - If it is, then the model does not meet the correctness specification;
     - If it isn't, then then refine the abstract model with a new precision, calculate the abstract state space again, and jump to the 3rd step.

The basic challenges that had to be solved when constructing this algorithm are the following:

- Generation of the synchronous product: I needed to determine how the synchronous product of the abstract model and the Büchi automaton are to be calculated.
- Language emptiness checking in the abstract state space: I had to devise a way to search for accepting runs in the abstract state space.
- New refinement strategy: I also needed to construct a way to refine the abstraction if the abstract counterexample isn't concretizable.

Some steps of this algorithm can be executed simultaneously. For example, state space generation, calculation of the product automaton and language emptiness checking can be conducted together, which can result in a significant increase in performance. If these three tasks are carried out at the same time, then the model checking is said to happen "on-the-fly".

I'm going to demonstrate the steps of the algorithm using a CFA, which can be seen in Figure 3.6, and the LTL expression $F(x = 1)$. The preprocessor transforms this LTL expression to $F(p_0)$, and returns $(x = 1)$ as the atomic proposition $p_0$. This LTL expression can be transformed to the Büchi automaton in Figure 3.7. When illustrating Büchi automata I replace the atomic propositions with the corresponding logical formulas, so for example $x = 1$ appears instead of $p_0$ in figures.

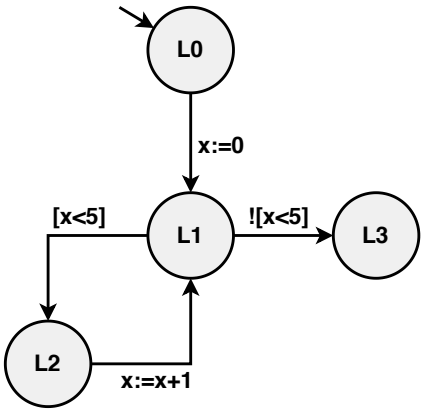


**Figure 3.6:** The example CFA

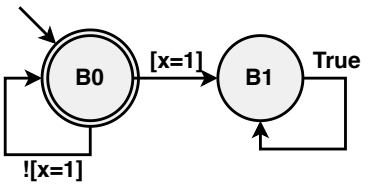


**Figure 3.7:** The Büchi automaton generated from $F(x = 1)$

### 3.4.1 The abstract state space

The main task of the abstractor component is the generation of the abstract state space. This state space is an over-approximation of the concrete state space, compared to the

concrete state space it can contain additional behaviour, but has to contain all behaviours of the concrete model. Abstract states can contain multiple concrete states. Each concrete state belongs to exactly one abstract state.

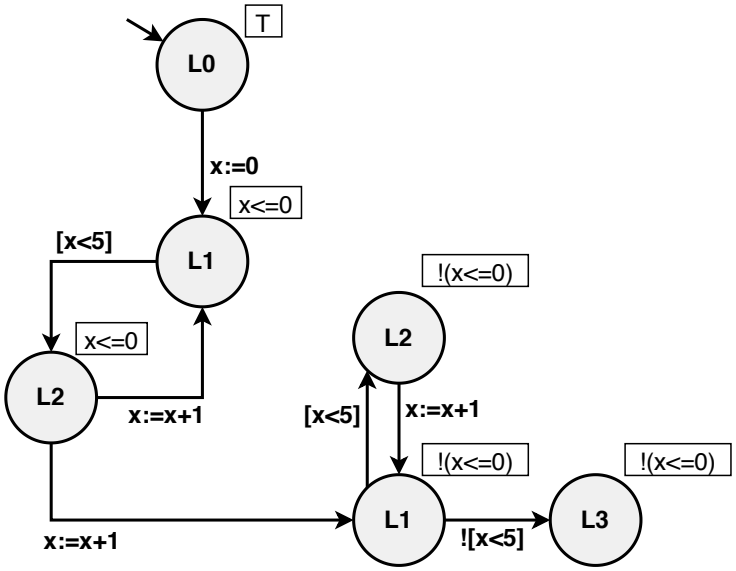


**Figure 3.8:** The abstract state space of the CFA presented in Figure 3.6 using predicate abstraction with the precision $[(x <= 0)]$.

The algorithm can work with multiple abstraction methods, such as explicit value abstraction, predicate abstraction, or a mix of the two. The appropriate abstraction method can only be selected based on the desired application domain. I chose to build my implementation on predicate abstraction, because it is more suited for reactive systems as variables in such systems usually only get assigned a few different values, not their whole domain.

Figure 3.8 shows the abstract state space of the CFA in 3.6 when applying predicate abstraction to it with the precision $[(x <= 0)]$. The set of active predicates is displayed in a white rectangle next to each state. Please note that this figure illustrates the abstract state space, not a CFA, thus nodes of this graph represent abstract states of the abstract model and not locations. This means that a location can appear multiple times in this graph, whereas in a CFA it can only appear once. Each node is labeled with the set of predicates that apply to the corresponding abstract state.

### 3.4.2 The product state space

The algorithm then generates the state space of the product of the abstract model and the Büchi automaton. A state in this product state space is an $\langle s, q \rangle$ pair, where $q \in Q$ is a state of the Büchi automaton, and:

- if the model is an xSTS, then $s = (\widehat{p_0}, ..., \widehat{p_k})$ is a state of the the abstract model;
- if the model is a CFA, then $s = (l_i, \widehat{p_0}, ..., \widehat{p_k})$ is a state of the the abstract model.

The initial state of the product consists of the initial state of the abstract model and the initial state of the Büchi automaton. A product state is called accepting if the corresponding Büchi state is accepting. At each step of the product the abstract model steps first.

After this the atomic propositions are evaluated based on the target state of the abstract model, and lastly the Büchi automaton steps based on the atomic propositions.

The product of the abstract model from Figure 3.8 and the Büchi automaton from Figure 3.7 can be seen in Figure 3.9.

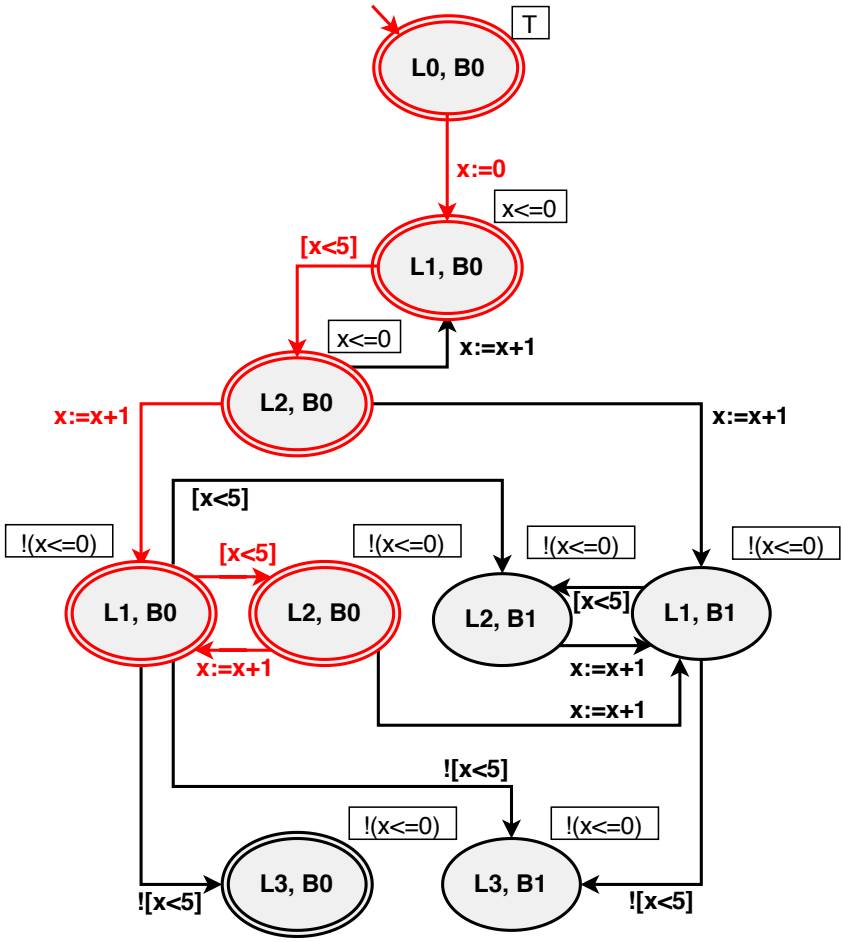


**Figure 3.9:** The product state space. A possible counterexample is denoted with *red* colour.

### 3.4.3 Counterexamples

In case of language emptiness checking of Büchi automata our counterexamples have a ”lasso”-like form. The first half of the counterexample is a path leading up to an accepting state and the second half is a cycle which starts and ends in said accepting state. If such a counterexample is found, then an accepting run is possible, because by repeatedly traversing the cycle an accepting state can be entered infinitely many times. In extreme cases it is possible that either the cycle, the path leading up to it, or both only contain one state.

An abstract counterexample can be seen in Figure 3.9 with red colour. In this example $\langle(l_0, \top), b_0\rangle, x := 0, \langle(l_1, x <= 0), b_0\rangle, [x < 5], \langle(l_2, x <= 0), b_0\rangle, x := x + 1, \langle(l_1, !(x <= 0)), b_0\rangle$ is the path leading up the the accepting state $\langle(l_1, !(x <= 0)), b_0\rangle$, and $\langle(l_1, !(x <= 0)), b_0\rangle, [x < 5], \langle(l_2, !(x <= 0)), b_0\rangle, x := x + 1, \langle(l_1, !(x <= 0)), b_0\rangle$ is the cycle. If we

traverse this cycle infinitely many times, then the accepting state $\langle (l_1, !(x <= 0)), b_0 \rangle$ is entered infinitely many times, fulfilling the acceptance condition of the Büchi automaton.

One of the challenges I had to face when constructing this algorithm is checking the feasibility of abstract counterexamples. Counterexample-guided abstraction refinement is usually used for safety checking, i.e. checking if any error states are reachable from the initial state. The counterexamples in these cases are alternating sequences of states and transitions, the only thing that has to be verified about them is whether they are traversable or not. However, when checking the feasibility of our "lasso"-like counterexamples traversability is required but not enough to prove that the counterexample is valid in the concrete model. A "lasso"-like counterexample has to be traversable in the concrete model in a way that the starting state of the cycle is the same concrete state as the ending state. This is required because as abstract states can contain multiple concrete states it is possible that an alternating sequence of states and transitions appears as a cycle in the abstract model, however -while being traversable- isn't a cycle in the concrete model.

Figure 3.10 shows this problem on the cycle of the counterexample from Figure 3.9. Abstract states are denoted with rectangles, while concrete states are denoted with circles. As can be seen in the figure, abstract states can contain multiple concrete states, for example $\langle (l_1, !(x <= 0)), b_0 \rangle$ contains all states, where the CFA is in location $l_1$, the Büchi automaton is in state $b_0$, and where $x$ is assigned such a value, that $!(x <= 0)$ holds. In our case the counterexample is traversable, however, what appeared as a cycle in the abstract model is not a cycle in the concrete model. This is easy to see, as the value of $x$ gets increased in every iteration, making it impossible to return to the same state. The abstract state $\langle (l_1, !(x <= 0)), b_0 \rangle$ has to be split in two in order to separate the cycle's starting and ending state from each other. This is the task of the *refiner* component.

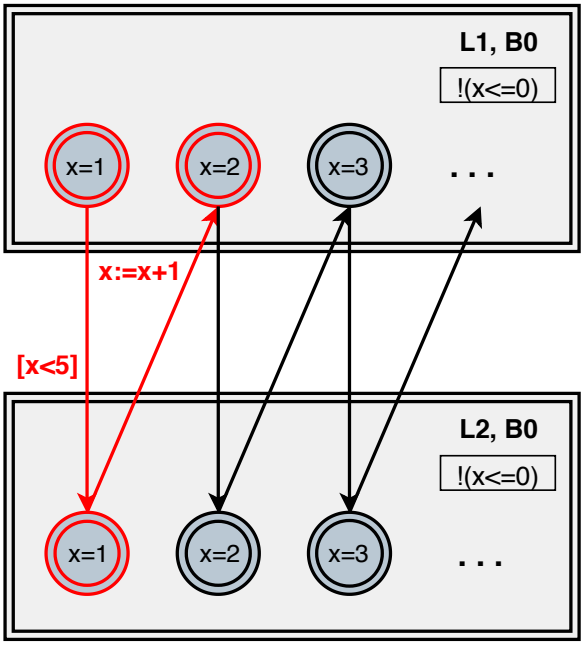


**Figure 3.10:** Illustration of the cycle validity problem. The cycle is denoted with red colour.

### 3.4.4 Refinement

The refinement algorithm discussed in 2.3.3 has to be extended to be able to handle "lasso"-shaped counterexamples. The refiner first checks the traversability of the counterexample the same way the original algorithm does. If the counterexample is found to be untraversable, i.e. there is at least one state in the counterexample that can't be reached using the given path, then the precision is refined as discussed in 2.3.3.

The refiner receives an alternating sequence of states and transitions $\widehat{\pi} = (\widehat{s}_1, t_1, \widehat{s}_2, t_2, \widehat{s}_3, ..., t_{n-1}, \widehat{s}_n)$ and an index $1 <= c <= n$ indicating the start of the cycle in this sequence as input.

The next step is checking whether the abstract cycle is a valid cycle in the concrete model as well. We already verified traversability, so we know that all constraints can be satisfied together, formally: $Label(\widehat{s}_1)_1 \wedge Label(t_1)_1 \wedge Label(\widehat{s}_2)_2 \wedge ... \wedge Label(t_{n-1})_{n-1} \wedge Label(\widehat{s}_n)_n$ is satisfiable (we can reach from an initial state to a final, goal state). Please note that in this case the last state of the counterexample, namely $\widehat{s}_n$ is not an erroneous state, simply the end of our cycle. In order to define a "lasso", we have to define identical states: two product states are identical if both their states in the abstract model and their Büchi states are identical. In an xSTS, two concrete states are identical if the variable assignments are the same. In case of CFA, the locations have to be the same as well (locations are also stored in a variable, so the definition is practically the same). Büchi states and CFA locations are tracked explicitly, meaning a state of the product state space can only contain concrete states that have the same Büchi state and the same location. This means that we only have to consider the values of the variables to determine if two concrete states inside an abstract state are the same.

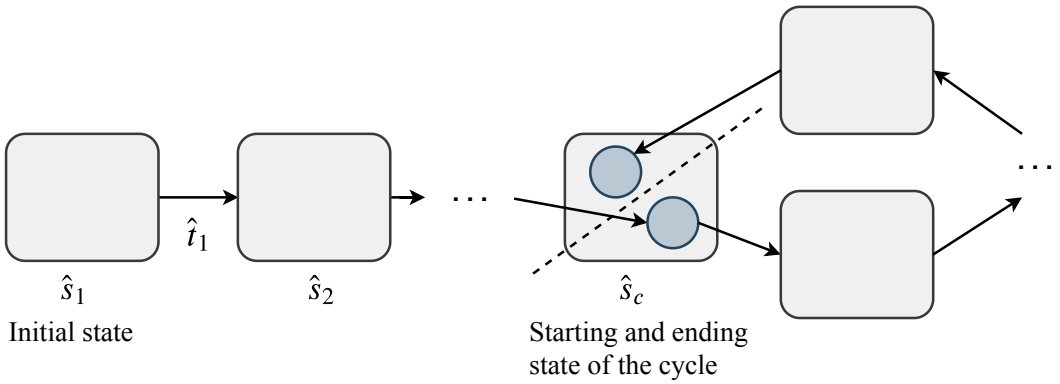


**Figure 3.11:** Illustration of the cycle refinement.

They key point in symbolic lasso detection is the symbolic encoding. To check this we are going to add a new constraint to the already checked ones and see if it is satisfiable together with them. This constraint is going to express that each variable has the same value assigned to it in the starting state and the final (recurrent) state of the cycle. If we have $n$ variables and denote the index of variable $v \in V$ in the starting state with $a_{v,s}$ and in the ending state with $a_{v,e}$, then our constraint is $\bigwedge_{v \in V} (v_{a_{v,s}} = v_{a_{v,e}})$. For example if variables $x$ and $y$ had the indices 2 and 1 in the starting, and the indices 3 and 1 in the ending state of the cycle, then our constraint would be $x_2 = x_3 \wedge y_1 = y_1$. If this new constraint is consistent with the earlier constraints, then our cycle is realizable in the concrete model as well, meaning the counterexample is valid and the model does not satisfy the requirements.

However, if it isn't consistent with the earlier ones, then we need to refine our precision. In the previous steps we constructed logical formulas that express that the counterexample has to be traversable such that the starting and ending state of the cycle is the same state, but it seems that such a concrete path is not realizable. By feeding our constraints to an interpolating SMT-solver, we can obtain an interpolant that clarifies why our path couldn't be concretized. If we create a new predicate by folding this interpolant and extend our list of tracked predicates with the newly gained predicate, then we eliminate the spurious counterexample and the algorithm continues the search.

How we can fold this interpolant is also different from the foldin procedure discussed in 2.3.3. This interpolant can only contain indexed variables from either the starting or the ending state of the cycle. One of the options would be to replace the starting ones with the variables themselves and the ending ones with primed versions of the variables. However, predicates that contain variables that have multiple prime operators applied to them (for example $(x > x''')$) are very inefficient to process during state space generation, because they require determining what can be the value of a variable multiple states from now. I chose a different option, replacing the ending ones with the unprimed variable and replacing the starting ones with a value that they can have in the starting state. If a set of constraints is satisfiable, then we only need to select one of the possible valuations that satisfy it and extract the value it assigns to our variable from it. In the example from 3.10 a possible value for the variable $x$ in the starting state of the cycle is 1, so assuming the variable $x$ had the index 1 in the starting and the index 2 in the ending state of the cycle, the interpolant $x_1 >= x_2$ would be folded in as $1 >= x$. We add this new predicate to our precision and start a new iteration of the CEGAR-loop.

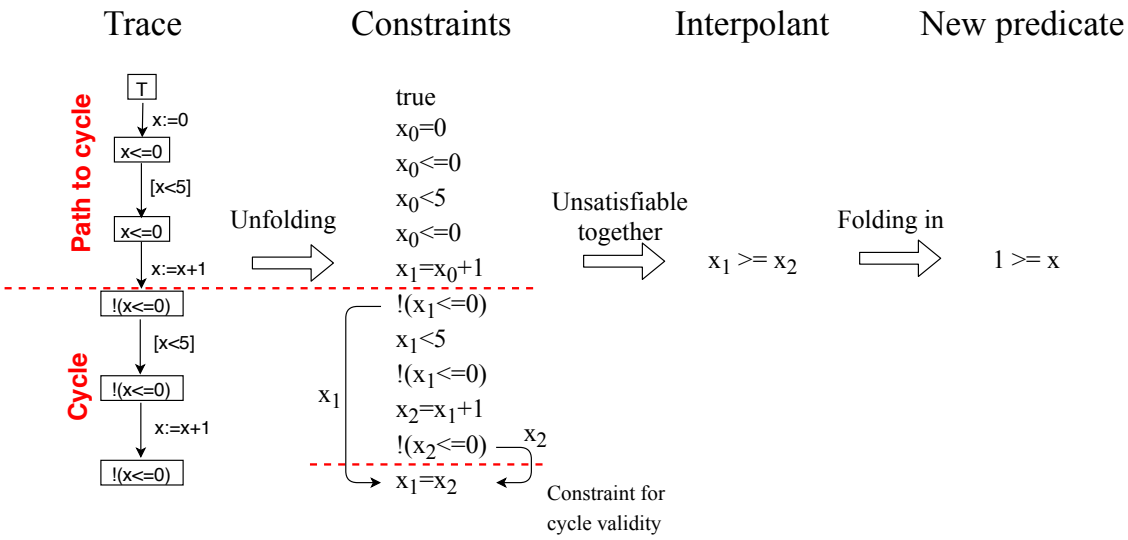


**Figure 3.12:** Illustration of the refinement process of the counterexample from Figure 3.9

# Chapter 4

# Implementation

## 4.1 External frameworks

I used the services of some frameworks when implementing my solution, in this section I present them briefly.

### 4.1.1 Theta model checking framework

The *Theta model checking framework*[1] [15] is a modular, configurable model checking framework developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics, which aims to support the development of abstraction-refinement based model checking solutions. It provides a set of building blocks for formal verification, various formalisms, analysis tools and SMT solvers.

I used the following components of this framework:

- Core: these classes provide general building blocks for any checking algorithm, for example classes representing expressions, statements, traces.
- CEGAR-tools: these classes provide basic building blocks for CEGAR-based model checking, for example classes that help when using predicate abstraction.
- Solvers: the Theta framework provides an interface to the Z3 SMT-solver, I used this when implementing my refinement algorithm.

### 4.1.2 Spot LTL manipulation framework

The *Spot LTL and $\omega$-automata manipulation framework*[2] [11] provides various LTL and $w$-automaton manipulation algorithms, accessible through C++ or Python libraries, or command line tools. I used the framework's command line LTL to Büchi automata translator in my work.

[1] https://github.com/FTSRG/theta
[2] https://spot.lrde.epita.fr/

### 4.1.3 Antlr

*Antlr*[3] is a powerful tool for text processing. It can generate lexers, parsers and other useful classes from grammars.

## 4.2 Input formalisms

In this section I present the supported input formalisms and their implementations.

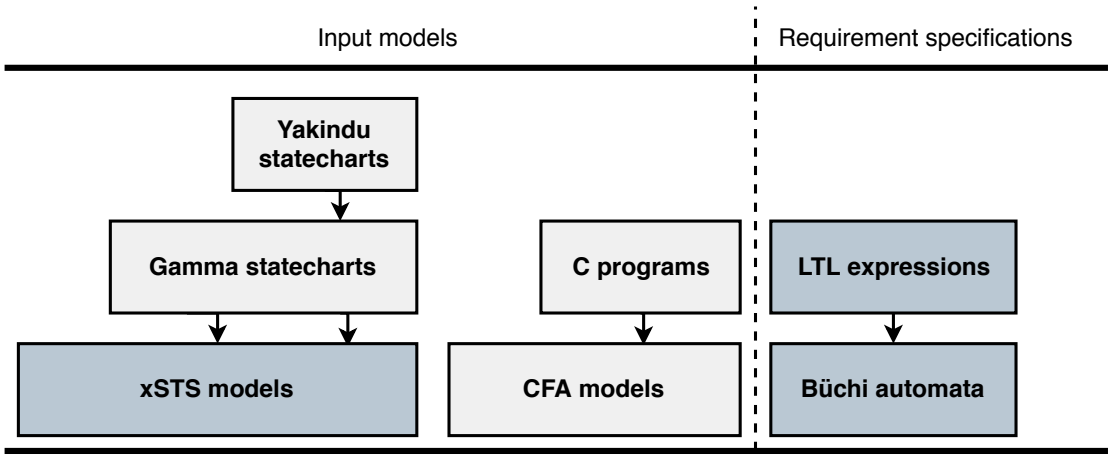


**Figure 4.1:** An overview of the supported input formalisms

### 4.2.1 Control flow automata

I used the CFA implementation of the Theta framework to represent control flow automata. This implementation allows the user to define error locations to define reachability checking tasks, but I didn't use this feature as I was only interested in LTL model checking. The framework provides a parser using which textually represented control flow automata can be imported.

### 4.2.2 Extended symbolic transition systems

I created my own implementation of extended transition systems and integrated it into the Theta framework. A pleasant side effect of this integration is that not only my tool can be used to check these models, but all analysis tools of the Theta framework.

### 4.2.3 C programs

I used the C to CFA compiler implementation of Gyula Sallai [21] to transform C programs to control flow automata. This compiler also uses the CFA implementation of the Theta framework, meaning no further transformations are required to process these models. The compiler only supports a subset of the C language at the moment:

- Only the following control structures are supported: if-then-else, do-while, switch, while-do, break, continue, goto and non-recursive functions;

[3]https://www.antlr.org/

- Variable types are restricted to integers and booleans;
- Arrays and pointers are not supported.

### 4.2.4 Yakindu statecharts

Yakindu statechart tools[4] is a tool for the specification and development of reactive, event-driven systems using the statechart formalism invented by David Harel [17]. The Gamma framework can transform Yakindu statecharts to Gamma statecharts, which can then be transformed to xSTS models, and be verified using my tool.

### 4.2.5 Gamma statecharts

The Gamma framework can transform Gamma models to xSTS models. At the moment Gamma only supports this for single component systems, sadly composite models can't be verified yet.

### 4.2.6 LTL formulas

I extended the LTL formalism to be able to reason about the variables of the verified models (see Section 3.3). I created a parser using Antlr to build a syntax tree of these extended formulas. After the systax tree is built I process the tree as detailed in Section 3.3, then negate the resulting LTL formula.

I use the *ltl2tgba* command line tool of the Spot framework to transform these preprocessed LTL-formulas to Büchi automata. This command line tool doesn't require the formula to be given in negation normal form. By default the tool outputs a generalized Büchi automaton, but using the "-B" flag it will degeneralize it for us. The tool supports multiple automaton output formats, I chose the Hanoi Omega-Automata Format (HOA) [1], which is a a flexible and robust mechanism for exchanging $\omega$-automata between different tools.

### 4.2.7 Büchi automata

When implementing Büchi automata I strived to achieve the best integration into Theta that I could to later be able to use the predicate abstraction related classes of the Theta framework when exploring the product state space. This was achieved by using multiple core building blocks of Theta when constructing the classes representing Büchi automata.

I implemented a parser component as well using the Java-based Hanoi Omega-Automata Format parsing tool called *jhoafparser*[5] to be able to parse Büchi automata generated by the LTL to Büchi automata translator.

## 4.3 The model checker

I implemented my model checker in a modular way so that it can be used to verify any formalism that is implemented using the building blocks provided by Theta and conforms to the interface I defined.

[4] https://www.itemis.com/en/yakindu/state-machine/
[5] https://automata.tools/hoa/jhoafparser/

I defined an interface called **Model**. Product states of this model have to implement the **ProductState** interface. Any model that correctly implements this interface can be verified using my model checker. The **Model** interface has two methods:

- **getInitialStates** has to return the set of initial product states;
- **getEnabledActionsFor** gets a product state as input and returns the set of enabled actions in that state. These actions have to implement the **ExprAction** interface of Theta. This interface has two methods that have to be implemented:
  - **toExpr** returns the action in form of a logical expression. For example in case of an assignment action $x := x + 1$ this method should return $x' = x + 1$;
  - **nextIndexing** returns how the index of the variables should change after this action is executed. The index of a variable should only increase if it's assigned a new value. For example in case of $x := y + 1$ the index of $x$ should be incremented with 1 and the index of $y$ should stay the same.

The interface **TransFunc** also has to be implemented. This interface has a method called **getSuccStates**, which returns the possible successor states if a given action is executed in a given state at a given precision.

### 4.3.1 The abstractor

I chose to implement my model checking algorithm in an on-the-fly manner, meaning model checking can happen simultaneously with state space generation. In my case this means that the calculation of the abstraction, the generation of the product state space and the search for accepting cycles happens at the same time.

I use the NDFS algorithm discussed in Section 2.4.3.1 to find accepting states and then search for cycles around them. When the NDFS algorithm calls the *post* method with a state $q$, I first determine where the model could step with the current precision. After this I evaluate the atomic propositions and calculate where the Büchi automaton could step in each possible successor state of the model.

If my algorithm finds an accepting cycle, then it passes the current call stack as a trace and a number indicating where the cycle starts in said state to the refiner.

### 4.3.2 The refiner

The refiner gets a trace (an alternating sequence of states and transitions) and a number indicating the start of the cycle in the trace as input. It checks if the abstract counterexample is realizable and returns a refined precision in case it isn't.

# Chapter 5

# Benchmarks

In this section I present the results of the verification tasks I ran to evaluate the performance of my model checker. In Section 5.1 I present the results of checking C programs using my tool. Section 5.2 and 5.3 list the results I had when verifying statecharts and control flow automata. In Section 5.4 I present cases for which my solution isn't optimal. The following values were measured in during each benchmark:

- **Time**: the time it took to verify the model. The measured values also include the time required to parse the models and initialize the required data structures.
- **Memory**: the amount of memory that was required to perfome the verification task.
- **ref**: the number of refinements that were needed to reach the final abstraction level.
- **blue**: the amount of product states visited by the blue DFS with the last precision.
- **red**: the amount of product states visited by the red DFS with the last precision.

All benchmarks were run on a laptop with the following configuration:

- Intel Core i5-3337U 4x2.7GHz
- 8 GB RAM
- Manjaro with Linux 4.14.141-1-MANJARO kernel
- Java 8

## 5.1 SV-Comp verification tasks

The following tasks are from the 2019 Competition on Software Verification(SV-Comp)[1] task set. All of these tasks are reachability tasks, they are described with the LTL-formula G(!(error)), where error is a boolean variable that is true in all erroneous states and false in all other states. These tasks don't utilize the full potential of LTL model checking as they only ask whether a certain set of states is reachable from the initial state or not. Reachability checkers perform better in these tasks than complete LTL checkers, but I chose to include them as the SV-Comp tasks are a commonly used way of checking the efficiency of model checking algorithms.

[1] https://sv-comp.sosy-lab.org/2019/

These tasks are given in the form of C programs. I used the verification compiler of Gyula Sallai[21] to transform them to control flow automata. The measured verification times do not include the time required to transform C programs to CFA. The verification process returned the expected output in all cases.

| Task | Time(s) | Memory(MB) | ref | blue | red |
|---|---|---|---|---|---|
| cggmp2005 | 1.263 | 7.88 | 10 | 6 | 0 |
| afnp2014 | 0.846 | 26.76 | 4 | 2 | 0 |
| bhmr2007 | 1.233 | 15.02 | 8 | 5 | 0 |
| sum03-2 | 0.74 | 26.15 | 2 | 2 | 0 |
| sum04-1 | 1.227 | 6.69 | 3 | 4 | 0 |

## 5.2 Statechart models

In this section I present the benchmarking results of checking various LTL expressions on Yakindu statecharts.

### 5.2.1 sc1

This is a simple statechart containing a composite state with 2 orthogonal regions in it.

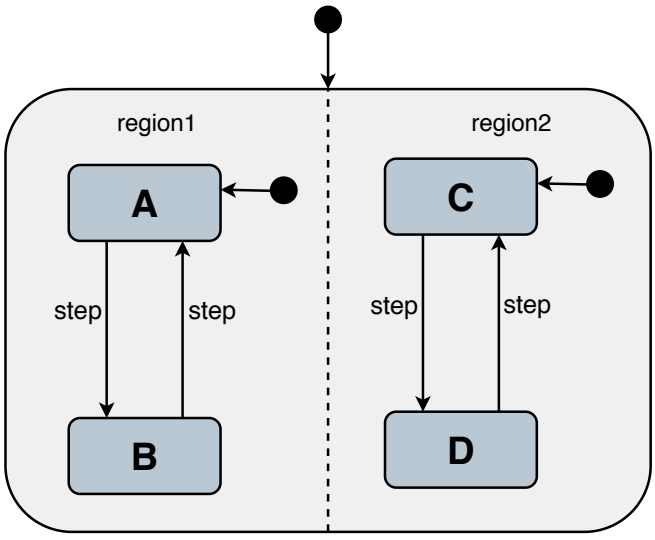


**Figure 5.1:** The **sc1** model

| Model | LTL-expression | Time(s) | Mem(MB) | ref | blue | red |
|---|---|---|---|---|---|---|
| sc1 | G(!(region1=A ∧ region2=D)) | 2.586 | 8.95 | 3 | 4 | 0 |
| sc1 | F(region1=B) | 1.181 | 6.40 | 1 | 4 | 3 |
| sc1 | GF(region1=A) | 1.281 | 10.29 | 2 | 6 | 3 |
| sc1 | G(region1=A ⇒ region2=C) | 1.309 | 10.92 | 4 | 4 | 0 |
| sc1 | GF(region1=A) ⇒ GF(region2=C) | 1.278 | 14.70 | 3 | 3 | 0 |

## 5.3 Control flow automata

In this section I present the benchmarking results of checking control flow automata with different LTL expressions.

### 5.3.1 counter5

This model is a simple control flow automaton which starts counting at 0 and counts up to 5. As no stuttering is allowed the counter will always reach 5.

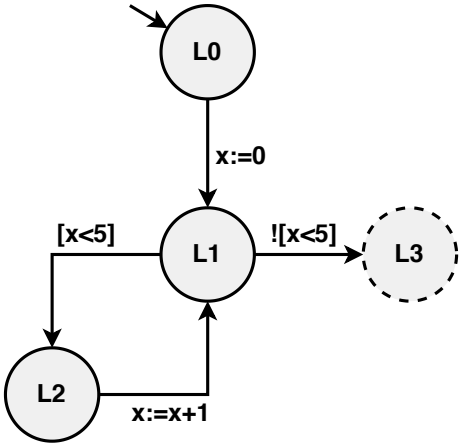


**Figure 5.2:** The **counter5** model

| Model | LTL-expression | Time(s) | Memory(MB) | ref | blue | red |
|---|---|---|---|---|---|---|
| counter5 | FG(x=5) | 0.903 | 10.72 | 6 | 14 | 14 |
| counter5 | F(x=1) | 0.981 | 29.37 | 4 | 11 | 11 |
| counter5 | G(x=0) | 0.768 | 28.72 | 3 | 7 | 0 |
| counter5 | (x<=2) U (x=3) | 0.909 | 5.57 | 3 | 14 | 14 |
| counter5 | G(!(x=6)) | 0.997 | 4.97 | 6 | 15 | 0 |
| counter5 | G(!(x=-1)) | 0.668 | 27.62 | 2 | 6 | 0 |

### 5.3.2 gcd

This simple control flow automaton describes Euclid's algorithm for the calculation of the greatest common divisor of two natural numbers that are greater than 0, $a$ and $b$. The error state is reached if the greatest common divisor of $a$ and $b$ is found to be 0.

| Model | LTL-expression | Time(s) | Memory(MB) | ref | blue | red |
|---|---|---|---|---|---|---|
| gcd | G(!(error)) | 0.762 | 27.44 | 2 | 4 | 0 |

## 5.4 Drawbacks of predicate abstraction

Predicate abstraction can be really efficient in control-flow dominated domains, but can have serious disadvantages compared to explicit model checking solutions in case of loops that require a constant and relatively big number of iterations. I demonstrate this on a modified version of the *counter5* model, where I replaced the conditions $[x < 5]$ and

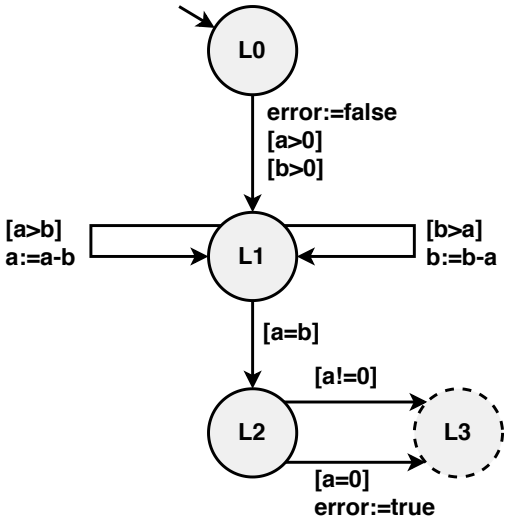


**Figure 5.3:** The **gcd** model

$![x < 5]$ with $[x < 100]$ and $![x < 100]$. I refer to this model as *counter100* in the followings.

| **Model** | **LTL-expression** | **Time(s)** | **Memory(MB)** | **ref** | **blue** | **red** |
|---|---|---|---|---|---|---|
| counter100 | GF(x=100) | 126.984 | 619.64 | 101 | 407 | 203 |
| counter100 | FG(x=100) | 130.341 | 270.27 | 101 | 204 | 204 |
| counter100 | F(x=100) | 131.125 | 552.54 | 101 | 204 | 204 |
| counter100 | (x < 50) U (G x >=50)) | 22.144 | 365.95 | 50 | 106 | 106 |

When checking this model all 100 iterations of the loop had to be unfolded to predicates, causing the CEGAR-loop to run 101 times. State space exploration slows down significantly with the introduction of more and more predicates. In cases like this explicit value abstraction or a mix of predicate abstraction and explicit value abstraction would likely perform better than a purely predicate based solution. The model checking raised the correct result in all of these cases as well.

# Chapter 6

# Conclusion

In this chapter I draw the conclusions of my work and lay down possible paths to continue.

## 6.1 Results

I started my work with two main goals in mind:

- To present an approach to LTL model checking that combines the advantages of multiple different model checking solutions. As LTL model checking is a computationally difficult task, my primary focus during development was efficiency.
- To integrate my model checker into a high-level modeling framework and thus enable engineers to use efficient and reliable model checking methods in their workflow.

### 6.1.1 Theoretical results

I constructed a novel LTL model checking algorithm, which combines the advantages of counterexample-guided abstraction refinement and automata theoretic model checking. I overcame multiple theoretic challenges:

- Generation of the synchronous product: I defined how the synchronous product of the abstract model and the Büchi automaton are to be calculated.
- Language emptiness checking in the abstract state space: I devised a searching strategy to find accepting runs in the abstract state space.
- New refinement strategy: I presented a way to check abstract counterexamples and refine the abstraction for lasso-shaped abstract counterexamples.

The benchmarks showed promising results about performance and suggest further examination of the topic.

### 6.1.2 Practical results

My approach is not only unique in its algorithmic contributions, but also in the sense that it bridges the gap between engineering models and formal verification:

- I integrated my model checker into the Gamma modeling framework and thus my work supports the engineers to verify LTL properties on their models. Currently my tool is the only one which supports LTL model checking of statechart models.
- I integrated my model checker into the Theta framework as well, extending the framework's already rich capabilities with LTL model checking. My tool already supports the verification of C programs, CFA and xSTS, but thanks to this integration and my simple interface this list can easily be extended.
- I created an intuitive engineering-centric extension of LTL to allow engineers to formulate relevant LTL expressions about their models.

## 6.2 Future work

Possible paths for the future:

- I plan to extend Gamma integration to support composite models as well. As Gamma has multiple composition semantics and even allows for asynchronous composition this seems quite challenging, but could mean a significant step forward in the model checking of distributed systems.
- I plan to introduce parallel actions into xSTS models. Model checking of parallel components can easily result in state space explosion, I intend to overcome this challenge by using partial order reduction when reasoning about parallel paths.
- I also plan to visualize the counterexamples in the high level models to make them easier to evaluate.

# Bibliography


[1] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 479–486, Cham, 2015. Springer International Publishing.

[2] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[4] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[6] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[8] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[9] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, Oct 1992.

[10] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[11] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and $\omega$-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, October 2016.

[12] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[13] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[14] Ákos Hajdu. A survey on CEGAR-based model checking, 2015. Master's Thesis, Budapest University of Technology and Economics.

[15] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components and Systems*, volume 9688 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2016.

[16] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag, Berlin, Heidelberg, 1985.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. ISSN 0167-6423.

[18] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[19] Vince Molnar. Advanced saturation-based model checking. Master's thesis, Budapest University of Technology and Economics, 2014.

[20] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[21] Gyula Sallai. Development of a verification compiler for C programs, 2016. Bachelor's Thesis, Budapest University of Technology and Economics.

[22] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–190, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[23] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2), 1972.

[24] Wolfgang Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on Infinite Objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.