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

Abstraction-based algorithms for configurable automata-theoretic model checking

Scientific Students' Association Report

Author:

Balázs Róbert Rippl

Advisors:

Milán Mondok
Dániel Szekeres

2023

Contents

Kivonat

A mindennapi életünk kritikus rendszereken alapszik: ebbe beletartoznak járművek, az energetikai és közlekedési hálózatok, digitális infrastruktúrák. Ezen rendszerek megbízhatósága és biztonsága létfontosságú, mivel még a legkisebb észrevétlen hibák is katasztrofális következményekkel járhatnak, szélsőséges esetekben emberéletekbe is kerülhetnek. A modellellenőrzés lehetővé teszi a tervezőmérnököknek, hogy szigorúan elemezzék és ellenőrizzék ezeket a rendszereket helyességük és megbízhatóságuk biztosítása érdekében. A kritikus rendszerek matematikai modellezésével és ezek kimerítő vizsgálatával a modellellenőrzés segít azonosítani a potenciális hibákat a fejlesztés korai szakaszában.

A Lineáris Temporális Logika (LTL) egy erős követelményspecifikációs nyelv a kritikus rendszerek számára. Az LTL nemcsak elérhetőségi feltételeket, hanem dinamikus tulajdonságokat is képes kifejezni, például összetett élőségi és stabilizációs követelményeket. Az LTL modellellenőrzés legelterjedtebb megoldása az automatelméleten alapul.

A modellellenőrzés gyakorlati alkalmazását az állapottér-robbanás problémája hátráltatja: a lehetséges állapotok száma akár exponenciális is lehet a változók számában. Egy gyakran alkalmazott megoldás az állapottér csökkentése absztrakció segítségével. Azonban az absztrakció potenciális alkalmazása az automatelméleti modellellenőrzés hatékonyságának növeléséhez még nincs kimerítően feltérképezve a szakirodalomban.

Dolgozatomban hatékony moduláris automatelméleti modellellenőrzési algoritmusokat és fogalmakat mutatok be, amelyeket alapvető építőelemként lehet felhasználni magasabb szintű absztrakció alapú algoritmusokhoz. (1) Bemutatok egy hatékony új *nyelvi ürességellenőrző* algoritmust, és összehasonlítom annak teljesítményét egy másik, a szakirodalomban megtalálható megközelítéssel. (2) Leírok egy új *absztrakció finomítási* algoritmust, és összehasonlítom azt egy, a szakirodalomban fellelhetővel. (3) Bővítem a nyílt forráskódú Theta modellellenőrzési keretrendszert egy új *kombinált formalizmussal*, amelyet használhatunk tetszőleges számú címkézett állapotátmeneti rendszer szorzatának előállításához. A bemutatott algoritmusok moduláris jellegét azzal demonstrálom, hogy egy kibővített ellenpélda-vezérelt absztrakció finomítási algoritmussá kombinálom őket, amely képes LTL tulajdonságok ellenőrzésére. Az algoritmusokat a nyílt forráskódú Theta modellellenőrzési keretrendszerben implementáltam, majd ipari partnerek által biztosított esettanulmányokon értékeltem ki, mely során teljesítményük ígéretesnek bizonyult.

Abstract

We base our everyday life on critical systems: they encompass vehicles, power grids, transportation networks and digital infrastructure. The reliability and safety of these systems are vital, as even the smallest of uncontrolled malfunctions might cost much, possibly human life. Model checking allows engineers and designers to rigorously analyze and verify these systems to ensure correctness and reliability. By creating mathematical models of critical systems and subjecting them to exhaustive checks, model checking helps identify potential flaws early in the development process.

Linear Temporal Logic (LTL) is a powerful requirement specification language for critical systems. LTL is able to express not only reachability conditions but also dynamic properties, like complex liveness and stabilization requirements. The most widely used solution for LTL model checking builds on automata theory.

The practical application of model checking is hindered by the state space explosion problem: the number of possible states can be exponential in the number of variables. One commonly used solution is shrinking the state space using abstraction. However, the application of abstraction to automata-theoretic model-checking has not yet been thoroughly explored. In this work, I present efficient modular automata-theoretic model-checking algorithms and concepts that can be utilized as foundational building blocks of higher-level abstraction-based algorithms. (1) I present an effective *language emptiness checking* algorithm and compare its performance to another approach from the literature. (2) I describe a novel *abstraction refinement* algorithm and benchmark it against one found in the literature. (3) I extend the open-source Theta model checking framework with a new *composite formalism* that can be used to generate the product of an arbitrary number of labeled transition systems. To demonstrate the modularity of these algorithms, I construct an extended counterexample-guided abstraction refinement algorithm that can verify LTL properties. I implemented the algorithms in the open-source Theta model checking framework and evaluated them on case studies provided by industrial partners and found the measured performance to be promising.

Chapter 1

Introduction

We base our everyday life on critical systems: they encompass vehicles, power grids, transportation networks and digital infrastructure. The reliability and safety of these systems are vital, as even the smallest of uncontrolled malfunctions might cost much, possibly human life. The model-based design of these systems enables the use of model checking. Model checking [7] is a formal method that allows engineers and designers to rigorously analyze and verify these systems to ensure correctness and reliability. By creating mathematical models of critical systems and subjecting them to exhaustive checks, model checking helps identify potential flaws early in the development process.

Linear Temporal Logic (LTL) [1] is a powerful requirement specification language for critical systems. LTL is able to express not only reachability conditions (e.g. that the system must not enter an erroneous state) but also dynamic properties, like complex liveness and stabilization requirements (e.g. that after any disturbance, the system must return to providing its service correctly). The most widely used solution for LTL model checking builds on automata theory. This approach represents both the model and the LTL requirement as automata and reduces the model checking problem to product automaton calculation and language emptiness checking.

The practical application of model checking is hindered by the state space explosion problem: the number of possible states can be exponential in the number of variables. One commonly used solution is shrinking the state space using abstraction, the main idea of which is to ignore some information about the model. An *abstract* model is constructed as a conservative approximation of the original, also called *concrete* model, by merging original concrete states into abstract states. With abstraction, the model checking can be carried out on a smaller state space, however, to be conservative, the abstract state space can contain extra behaviour that is not present in the original model. The counterexample-guided abstraction refinement (CEGAR) [6] algorithm aims to iteratively refine the abstraction precision until the optimum is found where the model contains just enough information for the examined property to be decided.

Although some preliminary work has been done toward it [22], the application of abstraction to automata-theoretic model-checking has not yet been thoroughly explored. In this work, I present efficient modular automata-theoretic model-checking algorithms and concepts that can be utilized as foundational building blocks of higher-level abstraction-based algorithms.

The contributions of this work are the following:

1. I present an effective *language emptiness checking* algorithm and compare its performance to another approach from the literature.

2. I describe a novel *abstraction refinement* algorithm and benchmark it against one found in the literature.

3. I extend the open-source Theta model checking framework with a new *composite formalism* that can be used to generate the product of an arbitrary number of labeled transition systems.

4. To demonstrate the modularity of these algorithms, I construct an extended counterexample-guided abstraction refinement algorithm that can verify LTL properties.

5. I implemented the algorithms in the open-source Theta model checking framework and evaluated them on case studies provided by industrial partners and found the measured performance to be promising.

The structure of my paper is as follows. Chapter 2 presents the notations I use and the theoretical background I built upon throughout my work. I also discuss works that are related to my approach in this chapter. Chapter 3 presents my theoretical contributions. Chapter 4 presents the practical evaluation of my approach, and in Chapter 5, I draw conclusions from my work.

Chapter 2

Background

In this chapter, I explain the notations I use and the theoretical foundation I built upon throughout my work.

2.1 Model checking

Model checking is a formal verification technique used to determine whether a given model of a system satisfies a specific set of properties or requirements. The model is a formal mathematical representation (e.g. a state transition system or logical formulas) of the system under evaluation, and the properties to be verified are expressed using a formal specification language. The model checker then automatically checks whether the model satisfies the given specifications, by exhaustively exploring all possible behaviors of the system and comparing them against the requirements. By doing so, if a property is found to be unsatisfied, the model checker returns a suitable counterexample, which shows how exactly the model behaves wrong. You can find this workflow visualized on Figure 2.1.

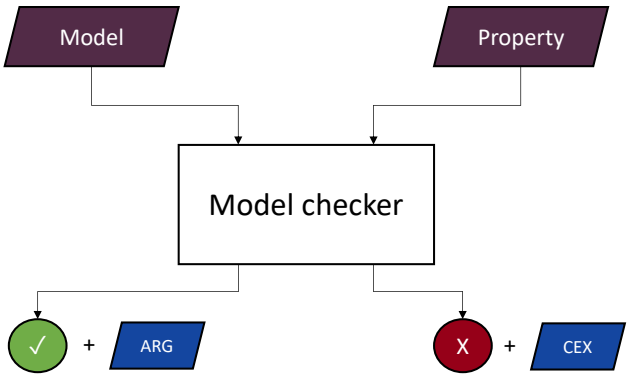


Figure 2.1: The model checking flow

Model checking has proven to be a powerful tool for verifying the correctness of complex systems, such as computer hardware, software, and protocols. It allows for the automatic and efficient detection of errors and vulnerabilities in the design of a system before it is implemented and deployed. This can save time and resources, and prevent costly errors and defects in the final product.

Model checking can also be used to optimize the performance and reliability of a system, by identifying and eliminating sub-optimal or unnecessary behaviors. It can be applied at

different stages of the development process, from early design and prototyping to testing and validation.

2.2 Modeling formalisms

Different modeling formalisms are used to represent systems with different characteristics. Model checking can be used on models that are defined with mathematical precision. Due to this constraint, a lot of models are quite low-level, resulting in simple definitions, but offering no complex design structure out of the box. Higher-level models usually provide possibilities for more human-friendly declarations, which are typically mapped to lower-level representations through a series of model transformations.

2.2.1 Kripke structures

Kripke structures [7] serve as the mathematical foundation of how we think about the state space of a system in a model checking problem. Kripke structures are not directly used for modeling systems, but the higher-level formalisms introduced below can all be transformed into Kripke structures.

Definition 1 (Kripke Structure). A Kripke structure is a $\langle S, I, R, L \rangle$ tuple defined over atomic propositions AP where:

- $S = s_1, s_2, ...$ is the set of states
- $I \subseteq S$ is the set of initial states
- $R \subseteq S \times S$ is the set of transitions, where $\forall s \in S \; \exists s' : (s, s') \in R$
- $L : S \to 2^{AP}$ is the labeling function ▪

A Kripke structure is a directed graph, whose nodes correspond to the states of the system and whose edges represent the potential state changes. A path in a Kripke structure that starts in one of the initial states corresponds to an execution trace in the system.

2.2.2 More complex data states using variables

While Kripke structures can represent more complex systems, it is rather inconvenient to define everything using simple atomic propositions. To get a step closer to higher levels, we introduce the concept of variables and First Order Logic (FOL) to modeling formalisms.

Suppose the set of variables in a given system is $V = \{v_1, v_2, \ldots, v_n\}$ with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$. A *data state* $s \in S \subseteq D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$ is an interpretation that assigns a value $s(v) \in D_v$ to each variable $v \in V$ of its domain D_v. A data state can also be regarded as a tuple of values $(s(v_1), s(v_2), \ldots, s(v_n))$. Given a FOL formula φ, let $s \models \varphi$ denote that assigning the variables in φ with the values in s evaluates to *true*. For example $(x = 0, y = 0) \models x \geq 0$. Similarly, let $s \not\models \varphi$ denote that φ in s evaluates to *false*.

Data states can always be mapped to atomic proposition representations. The most simple example would be to create an atomic proposition for each possible value for each variable. When a variable has a certain value, the atomic proposition bound to that value is considered to be true, while all other propositions for said variable are considered false. This allows us to reason about models using data states as we would about Kripke structures.

2.2.3 Symbolic Transition System

Symbolic Transition Systems (STSs) [16] aim to provide a simple way of representing the set of *states*, *transitions* and *initial states*. Besides data states, STS uses FOL formulas to represent transitions in the system too. A symbolic transition system is a tuple $STS = (V, Inv, Tran, Init)$, where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables
- Inv is the invariant formula over V, which must hold for every state
- $Tran$ is the transition formula over $V \cup V'$, which describes the transition relation between the current state (V) and the successor state (V')
- $Init$ is the initial state formula, which describes the values of the variables in the initial states

V' represents the primed version of variables. The prime notation allows for the reassigning of variables, thus allowing for modeling running systems. For example if $V = \{x, y\}$, then $Tran \equiv x' = x + y \wedge y' = y + 1$ describes a transition, which increases the value of x by the value of y, and the increases the value of y by 1.

The set of initial states is $S_0 = \{s \mid s \models Init \wedge Inv\}$. For example, if $Init \equiv x = 0 \wedge (y >= 0 \vee y < 10)$ and $Inv \equiv y < 2$, then $S_0 = \{(x = 0, y = 0), (x = 0, y = 1)\}$. A transition exists between two states s and s', if $(s, s') \models Inv \wedge Tran \wedge Inv'$. For example. if $Tran \equiv x' = x+1 \wedge y' = y$, then a transition exists between the states $s = (x = 0, y = 0)$ and $s' = (x = 1, y = 0)$.

A concrete path is a finite sequence of concrete states $\sigma = s_1, s_2, ..., s_n$, for which $(s_1^{\langle 1 \rangle}, s_2^{\langle 2 \rangle}, ..., s_n^{\langle n \rangle}) \models Init^{\langle 1 \rangle} \wedge \bigwedge_{1 \leq i \leq n} Tran^{\langle i \rangle} \wedge \bigwedge_{1 \leq i \leq n} Inv^{\langle i \rangle}$, i.e. the path starts in an initial state. The successor states satisfy the transition relation. A concrete state s is reachable if a path $\sigma = s_1, s_2, ..., s_n$ exists with $s = s_n$ for some n.

2.2.4 Operation-based formalisms

To be able to reason about systems even more efficiently, operations are introduced to control the transitions in the system. These help move formalisms closer to human intuition, allowing engineers to create straightforward models faster.

2.2.4.1 Operations

Operations $op \in Ops$ describe the transitions between the data states of the system, where Ops is the set of all possible transitions. An operation $op \in Ops$ can also be interpreted over $V \cup V'$ as a logical formula, denoted by $tran(\varphi)$. By applying the formula to a data state, the primed variables from the formula construct the following data state, but only if the formula evaluates to true. The execution of the operations is atomic in the sense that they either get executed in their entirety or not at all. In my work, I used the following *basic* operations defined in [21]:

- *Assignments* of the form $v := \varphi$ assign a deterministic value to a single variable. Here $v \in V$ is a variable, φ is an expression of type D_v, and $var(\varphi) \subseteq V$. The semantics of assignments are the following: $tran(v := \varphi) \equiv v' = \varphi \wedge \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$. This

formula expresses that the value of v in the successor state is φ, and all other variables stay unchanged. For example, if $V = \{x, y\}$, then $tran(x := 1) \equiv x' = 1 \wedge y' = y$;

- *Assumptions* of the form $[\varphi]$, where φ is predicate with $var(\varphi) \subseteq V$. Assumptions act as guards, as they can be evaluated to false, thus rendering the operation unusable. The semantics are the following: $tran([\varphi]) \equiv \varphi \wedge \bigwedge_{v \in V} v' = v$. In other words, assumptions check a condition and leave the values of all variables unchanged. For example, if $V = \{x, y\}$, then $tran([y < 0]) \equiv y < 0 \wedge x' = x \wedge y' = y$.

- *Havocs* of the form $\mathsf{havoc}(v)$, where $v \in V$. Havocs are nondeterministic assignments, where a variable v gets assigned to a nondeterministic value of its domain D_v. The semantics are $tran(\mathsf{havoc}(v)) \equiv \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$, which means that v can have any value, while all other variables keep their value. For example, if $V = \{x, y\}$, then $tran(\mathsf{havoc}(x)) \equiv y' = y$.

Composite operations can contain other operations and can be used to model even more complex control flows. Note that the execution of these operations is still atomic in the sense, that they either get executed in their entirety or not at all. There are two composite operations:

- *Sequences* of the form $op_1, op_2, ..., op_n$, where $op_i \in Ops$ are lists of operations that are executed in order after each other. Each operation of the sequence operates on the result of the previous operation. For example, $x := 2, [y < 0]; x := x + 1$ is a sequence that contains 3 inner operations.

- *Choices* model nondeterministic choices between multiple operations. One and only one branch is selected for execution, which cannot contain failing assumptions. This means that if all branches of the choice contain failing assumptions, then the choice operation fails as well. Choices have the following form: $\{op_1\}$ or $\{op_2\}$ or ... or $\{op_n\}$, where $op_i \in Ops$ are basic or composite operations. For example, $\{[y > 0], x := 1\}$ or $\{x := 0\}$ is a choice with 2 branches.

Refer to [21] for the detailed semantics of composite operations.

2.2.4.2 Structural and concrete states

To support complex control flow even more, modeling formalisms usually provide extra structural information. This is an arbitrary set B and its elements $b \in B$ are called *structural states*. This structural information adds depth to the formalism. The *concrete state* is the combination of the current structural state and data state. Thus the set of all possible concrete states for data states D is $S = D \times B$.

Operation-based models use their concrete states to reason about the exact possible behavior of the system. To do so, they also need to define the set of initial structural states, the *structural init function* $\iota_B \subseteq B$[1].

2.2.4.3 Actions

Even though structural states describe the structural information, they can't implicitly model the control flow. *Actions* are used that take these structural states into account

[1] Even though it is not strictly a function, it is called that to have a consistent naming scheme

and can decide which *operations* are allowed to happen to the data state, and how the structural state changes. So formalisms define an *action function* $\alpha : S \to 2^{Ops \times B}$ that:

- limits the possible mutation of the data state based on the structure of the model
- is capable of mutating the structural state

2.2.4.4 Interface of operation-based formalisms

Now putting together everything so far from Section 2.2.4, an interface can be defined for *Operation-based formalisms*.

Definition 2 (Operation-based formalism interface). A modeling formalism is an operation-based formalism with the tuple $(V, D, B, S, \iota_B, \alpha)$ where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$
- D are the *data states* over V
- B is the set of *structural states*
- $S = D \times B$ is the set of *concrete states*
- $\iota_B \subseteq B$ is the *structural init function*
- $\alpha : S \to 2^{Ops \times B}$ is the *action function*

if it can provide the tuple (V, B, ι_B, α). ▪

In the following parts, two formalisms that fulfill this interface are going to be introduced.

2.2.4.5 eXtended Symbolic Transition System

The eXtended Symbolic Transition System (XSTS) formalism was created to serve as the intermediate representation of component-based reactive systems, such as statechart compositions. XSTS partitions the transition relation into two parts: the *inner transition relation* models the behavior of the system, and the *environmental transition relation* models the system's environment. XSTS [21] is a tuple $\text{XSTS} = (V, V_C, Init, Tr, En)$, where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$. For example, $V = \{x, y\}$. The possible domains are *integers*, *booleans*, and custom *enums*. Integers correspond to mathematical integers, meaning that their domain is "unbounded": unlike to "machine integers", they cannot over- or underflow;
- $V_C \subseteq V$ is the set of *control variables*. These variables are always tracked explicitly if tracked;
- *Init* is the initial state formula, which describes the values of the variables in the initial states;
- $Tr \subseteq Ops$ is a set of operations representing the *inner transition relation*. This set describes the internal behavior of the system;

- $En \subseteq Ops$ is a set of operations representing the *environmental transition relation*. This is used to model the environment of the system.

XSTS uses a flag as structural information, that tells whether the last executed transition came from Tr or En, so $B = \{\text{LE}, \text{LT}\}$. Since the model alternates between environmental and inner transitions, this is the information used in the action function: $\alpha_{\text{XSTS}}(s) = Tr \times \{\text{LT}\}$ if $\text{LE} \in s$ and $\alpha_{\text{XSTS}}(s) = En \times \{\text{LE}\}$ if $\text{LT} \in s$. XSTS starts with its inner transitions, so $\iota_B = \{\text{LE}\}$.

Theorem 1 (XSTS is an operation-based formalism). An XSTS $X = (V, V_C, Init, Tr, En)$ fulfills the operation-based formalism interface with the tuple $(V, \{\text{LE}, \text{LT}\}, \{\text{LE}\}, \alpha_X)$. ▪

2.2.4.6 Control Flow Automaton

Control Flow Automaton (CFA) is a modeling formalism widely used in model checking of software systems. It provides a structured representation of the control flow within a program, enabling systematic exploration of all possible execution paths and verification of desired properties. We define CFA as the following [15]:

Definition 3 (CFA). A CFA is a $\text{CFA} = (V, L, l_0, T)$ tuple where

- $V = \{v_1, v_2, \dots\}$ is the set of variables appearing in the program
- $L = \{l_0, l_1, \dots\}$ is the set of control locations modeling the actual position of the program counter
- $l_0 \in L$ is the initial location representing the entry point of the program
- $T \subseteq L \times Ops \times L$ is a set of directed edges between the locations, annotated with operations over the variables that get executed when control flows from one location to another ▪

A CFA is a directed graph. Its nodes correspond to *locations*, representing the different values of the program counter register. Its edges capture the flow of control during program execution and the executed instructions. That means, for a CFA structural states consist of the locations, $B = L$, and the action function returns the operations of all the edges from the current location, i.e., $\alpha_{\text{CFA}}(s) = \{\{o, l_t\} | \{l, o, l_t\} \in T, l \in s\}$.

Theorem 2 (CFA is an operation-based formalism). A CFA $C = (V, L, l_0, T)$ fulfills the operation-based formalism interface with the tuple (V, L, l_0, α_C) ▪

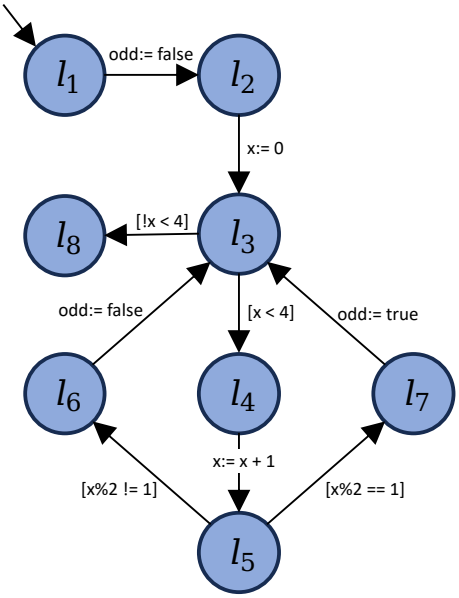
See the example on Figure 2.2, where a valid C algorithm is transformed into a CFA. The formal description would include $V = \{x, odd\}$, $L = \{l_0, l_1, \dots, l_6, l_f\}$. For each edge, there exists an element in T. The arrow with no source node on the top left corner identifies our initial location, l_0.

2.3 Requirement specifications

Apart from models with mathematically well-defined semantics, model checking also needs a mathematically precise requirement specification.

```
1 bool odd = false;
2 int x = 0;
3 while (x < 4) {
4   x = x + 1;
5   if (x % 2 == 1)
6     odd = true;
7   else
8     odd = false;
9 }
```

(a) Example C program



(b) CFA equivalent

Figure 2.2: Usage of CFA

2.3.1 Safety - reachability

Safety requirements specify unwanted states of a system. Checking such requirements is called *reachability* model checking, as the model checking problem in the case of safety properties is deciding whether such states can be reached from the initial states. For location-based models, such as the Control Flow Automaton, reachability requirements are usually specified by introducing extra locations, called *error locations*. Another possibility is to define an *invariant* property over the data variables, which is supposed to hold in every state of the model execution, like $battery > 0$ could mean that we need the battery always to be charged. In reachability model checking, if a requirement is deemed to not be met, the counterexample trace must be a sequence of concrete states, where the last state violates the requirement, i.e. :

- for invariant φ the trace is $(\{d_0, b_0\}, \{d_1, b_1\}, ..., \{d_n, b_n\})$ such that $0 \leq i < n : s_i \models \varphi$ and $s_n \not\models \varphi$
- for error location l_f the trace is $(s_0, s_1, ..., s_n)$ such that $0 \leq i < n : l_f \notin s_i$ and $l_f \in s_n$

2.3.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is used to reason about a (possibly infinite) sequence $(s_1, s_2 \dots)$ of states over a set AP of atomic propositions. For the sake of simplicity, let's make a state exactly equal to the set of propositions that hold in that state, so $s_n \subseteq AP$. LTL builds on top of propositional logic, so an atomic proposition and propositional logical formulas with operators $\neg, \wedge$ over these propositions are considered LTL formulas. Many more logical operators can be derived from these [1]. The LTL formula φ evaluates to true on state s and is written $s \models \varphi$, if swapping every proposition in φ that is $\in s$ to true, and the rest to false makes the formula true. Otherwise, $s \not\models \varphi$.

Being a temporal logic, LTL also uses temporal operators [1]. Let us look at the more common ones:

- $\mathbf{X}(\varphi)$ is an LTL formula and $s_n \models \mathbf{X}(\varphi) \iff s_{n+1} \models \varphi$. $\mathbf{X}$ is from the word "neXt".
- $\mathbf{F}(\varphi)$ is an LTL formula and is given by the recursive definition $s_n \models \mathbf{F}(\varphi) \iff (s_n \models \varphi \vee s_n \models \mathbf{X}(\mathbf{F}(\varphi)))$. $\mathbf{F}$ is from the word "Future", and formalizes the eventual occurrence of a state.
- $\mathbf{G}(\varphi)$ standing for "Globally" is an LTL operator, and $s_n \models \mathbf{G}(\varphi) \iff s_n \models \neg\mathbf{F}(\neg\varphi)$. It is used for properties that must hold throughout the whole trace.

As I mentioned earlier, LTL can be used to reason about a single run of a model. We can use it for whole models by stating we want all of the model's runs to satisfy the formula. We will see later what kind of problems that proposed and what solutions exist to tackle the challenges.

Let us see an example. Take the CFA from Figure 2.2. Let us say we have a single atomic proposition, conveniently the boolean variable of that model. So $AP = \{odd\}$, where *odd* holds $\iff$ *odd* evaluates to true in the CFA. We can state a requirement against our model: I want that after some time, odd to remain false forever. This requirement can be represented with the following LTL formula:

$$FG(\neg odd)$$

2.4 Automata-theoretic LTL checking

In automata-theoretic LTL checking, the system behavior is modeled as a finite-state automaton, and the LTL properties are translated into a second automaton called a *Büchi automaton*. The Büchi automaton represents the negation of the LTL property and its acceptance condition defines the set of system behaviors that violate the property.

Let us see first what a Büchi automaton looks like, then go into detail over how it can be used for LTL checking.

2.4.1 Büchi automata

Finite automata work great to check if certain words are part of a language. When we want to check infinite words over infinite alphabets, we need ω-automata, capable of handling infinitely long inputs. One such automaton is the Non-Deterministic Büchi Automaton (NDBA), which can describe ω-languages effectively [5].

Definition 4 (Non-Deterministic Büchi Automaton). A Non-Deterministic Büchi Automaton (NDBA) is a tuple of $\mathcal{A} = \langle Q, \Sigma, \Delta, I, F \rangle$, where

- $Q = \{q_0, q_1, \dots\}$ a set of the states of the automaton
- Σ is a finite set called the alphabet of the NDBA
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation
- $I \subseteq Q$ is the set of the initial locations

- $F \subseteq Q$ is the acceptance condition ▪

NDBAs can also have transition-based acceptance with a slight modification. The acceptance condition needs to become $F \subseteq \Delta$. For ease of handling, it is beneficial to define a $\text{ACC}(e = \{q_a, x, q_b\})$ predicate, that returns $e \in F$ for transition-based acceptance, and $q_b \in F$ for state-based acceptance.

To be able to reason about infinitely long words in writing, we can use the ω-notation. For example, if $\Sigma = \{a, b\}$, then $bb(ab)^\omega$ means the word that begins with a double b, then repeats ab forever. Σ^ω denotes all infinite words over Σ. Also, let w_i denote the i. letter of the word w.

In an NDBA $\mathcal{A} = \langle Q, \Sigma, \Delta, I, F \rangle$ the sequence $(e_1, e_2, ...), e_i \in \Delta$ is a run for the word $w \in \Sigma^\omega$ if

- $e_1 = \{q_0, w, q_1\} \implies q_0 \in I$
- $\forall i \in \mathbb{N} : w_i \in e_i$

An NDBA accepts an infinitely long word if there exists a run for it that causes the automaton to be in an accepting state or transition infinitely many times, i.e., $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega | (\exists \text{run}(e_1, e_2, ...) \text{for } w)(\nexists k \in \mathbb{N})[\bigvee_{i>k} \text{ACC}(e_i) = false]\}$.

Definition 5 (Lasso trace). For an NDBA $\mathcal{A} = \langle Q, \Sigma, \Delta, I, F \rangle$, $\sigma_L(w) = \langle T, H, L \rangle$ is a *lasso trace* for the word $w \in \Sigma^\omega$ with the run $(e_1, e_2, ..., (e_n, ..., e_m)^\omega), m \geq n$, where $T = (e_1, ..., e_{n-1})$ is called the *tail* of the trace, $L = (e_n, ..., e_m)$ is called the *loop* of the trace and if $e_n = \{q_a, x, q_b\}$, then $H = q_a$ is called the *honda*[2] of the trace. A lasso trace is an accepting lasso trace, if $\exists e \in L$ such that $\text{ACC}(e)$ evaluates to *true*. ▪

Theorem 3. For an NDBA $\mathcal{A} = \langle Q, \Sigma, \Delta, I, F \rangle$, the following two statements are equivalent:

(i) No accepting lasso trace can be shown

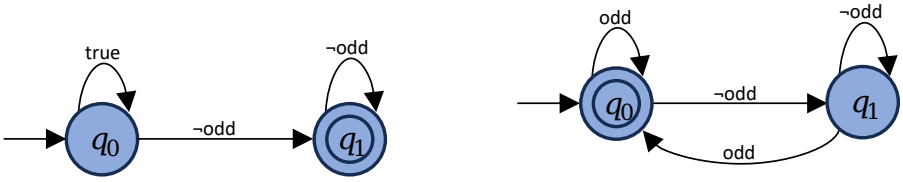
(ii) $\mathcal{L}(\mathcal{A}) = \emptyset$ ▪

Proof: The two statements are bound together by the definition of runs and both ways can be proven indirectly.
(i) → (ii) Suppose there exists an accepting lasso trace $\sigma_L(w) = \langle T, H, L \rangle$ in the NDBA $\mathcal{A} = \langle Q, \Sigma, \Delta, I, F \rangle$. From the definition, they repeat a transition infinitely many times, that either is accepting itself, or points to an accepting state (for transition- and state-based acceptance, respectively). That implies, that w is accepted by $\mathcal{A}$, which means $w \in \mathcal{L}(\mathcal{A}) \implies \mathcal{L}(\mathcal{A}) \neq \emptyset$.
(ii) → (i) It is the same as the other way, just mirrored backwards. Suppose the language is not empty, i.e., there is a $w \in \mathcal{L}(\mathcal{A})$. It means, that there is a run for w that repeats an accepting transition infinitely many times (or a transition pointing to an accepting state). Such run is a $\sigma_L(w)$. □

2.4.2 LTL checking with Büchi Automata

Since LTL is a subset of regular ω-languages, any formula can be represented as a NDBA. The transformation process might involve multiple steps [13] and it is out of the scope of this paper to go over it in detail.

[2] `https://en.wikipedia.org/wiki/Lasso`



(a) $FG(\neg odd)$ **(b)** $GF(odd)$

Figure 2.3: NDBA representation of different LTL formulae

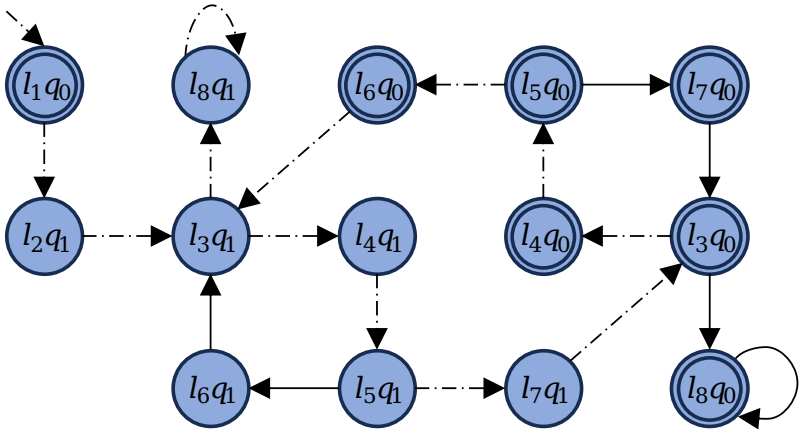


Figure 2.4: Product of the examples, with the only valid run annotated dashed

Rather just see the example formula from Section 2.3.2 displayed as an NDBA on Figure 2.3a.

The idea of model checking is to look for counterexamples, so not only the malfunction of the model can be proven, but aid can be provided on how to correct it. To find a suitable counterexample in our model $\mathcal{M}$ against a requirement specified in LTL, we need the following steps:

1. Negate the original formula
2. Create the Negational Normal Form (NNF) of it (most importantly no negation in front of temporal operators)
3. Transform the NNF into a NDBA, call it $\mathcal{A}$
4. Construct the product of our model and the resulting NDBA, $\mathcal{M} \times \mathcal{A} = \mathcal{P}$

Now if the resulting automaton accepts no word, $\mathcal{L}(\mathcal{P}) = \emptyset$, we can say that our original model conforms to the requirement. However, if there are runs that get accepted by the product, then $\forall \pi \in \mathcal{L}(\mathcal{P}), \pi$ is a counterexample.

Sticking to our example, we can check if our model on Figure 2.2 conforms to the requirement $FG(\neg odd)$. Let us do the steps in order.

1. $!FG(\neg odd)$
2. $GF(odd)$
3. See the NDBA on Figure 2.3b

4. The product can be seen on Figure 2.4

Notice that to make the visualization more compact, the steps of the two automata are merged. For example, the transition $l_5q_1 \to l_7q_1$ represents $\{l_5, [x\%2 == 1], l_7$ and $\{q_1, \neg odd, q_1\}$ (showing only the relevant part of the combined states).

Now if we were to check our product, we would find the single valid run seen on Figure 2.4. Converting it to a lasso trace, we would find that the tail is (omitting the words of the transitions) $(\{l_1q_0, \_, l_2q_1\}, \{l_2q_1, \_, l_3q_1\}, (\{l_3q_1, \_, l_4q_1\}, \ldots, \{l_6q_0, \_, l_3q_1\})^4, \{l_3q_1, \_, l_8q_1\})$, the loop is $\{l_8q_1, \_, l_8q_1\}$ and the honda is l_8q_1. Since $\text{ACC}(\{l_8q_1, \_, l_8q_1\})$ returns false, the lasso trace is not accepting. We can conclude that no accepting lasso traces can be shown, which means, the language of the product automaton is empty and the requirement is fulfilled.

2.5 Abstraction-based model checking

The state space of a system represents all possible configurations or states that the system can be in during its execution. In model checking, the state space is typically explored by exhaustively traversing all possible states to verify if a given specification holds for the entire system behavior. However, as systems become larger and more complex, the state space that needs to be explored during model checking grows exponentially, giving rise to a significant challenge known as the *state space explosion problem.*

To overcome the state space explosion problem, abstraction-based model-checking techniques have emerged as a promising solution. Abstraction involves creating a simplified, yet faithful, representation of the original system that captures only the essential aspects necessary for verifying the desired properties. By abstracting away irrelevant details, the state space can be significantly reduced, enabling more efficient model checking.

2.5.1 Abstraction

Abstraction can be defined over abstract domains, precisions, and transfer functions [15].

Definition 6 (Abstract domain). An abstract domain $\mathbb{D} = (D, \top, \bot, \sqsubseteq, \mathsf{expr})$ consists of the following:

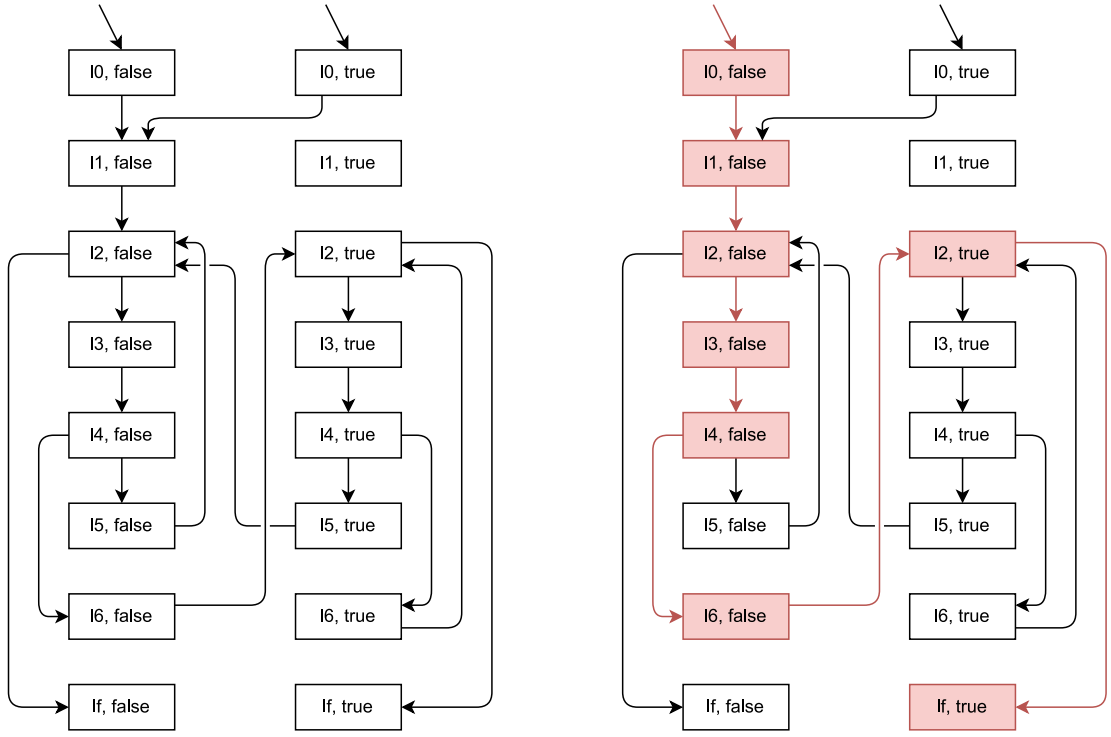
- the lattice D of abstract states
- $\top \in D$ top element
- $\bot \in D$ bottom element
- $\sqsubseteq \subseteq D \times D$ partial order conforming to the lattice
- $\mathsf{expr} : D \to$ FOL expression function concretizing the abstract states ▪

The arbitrary set Π holds the possible precisions, where an element $\pi \in \Pi$ defines the current precision of the abstraction. Defining precision so superficially keeps the definition of transfer functions simple.

Definition 7 (Transfer function). The transfer function T for an abstract domain $\mathbb{D} = (D, \top, \bot, \sqsubseteq, \mathsf{expr})$ is a function $T : D \times Ops \times \Pi \to 2^D$, calculating the successor abstract states for an abstract state given a precision and operation. ▪

There are two *abstract domains* I used throughout my work. Explicit value abstraction [2] tracks only a subset of the variables. Predicate abstraction [14] on the other hand tracks only the truth value of predefined predicates over the variables.

Recalling our example CFA from Figure 2.2, one can easily calculate how huge the state space would be. Considering the standard integer from C, the state space would contain $8 \cdot 2 \cdot 2^{16} = 1048576$ states. Applying explicit value abstraction on the variable *odd* however, would result in a state space sized a mere $8 \cdot 2 = 16$ states, which can be seen on Figure 2.5a.



(a) The state space of the CFA with abstraction

(b) An abstract counterexample for the reachability of $\langle l_f, true \rangle$

Figure 2.5: Abstract state space and an infeasible counterexample in it

2.5.2 Counterexample-guided abstraction refinement

Abstraction in itself does not solve our problems. By over-abstracting the model's state space and running model checking so, we can only be sure the model is good if we didn't find any counterexamples. However, if we do find one, it is going to be only an abstract counterexample. One can not be sure without further examination, whether the original model behaves the same way: as the abstract state space is an over-approximation of the original state space, it can contain extra behaviour.

The example in Section 2.5.1 demonstrates this problem. If we want to know, if *odd* can be true in the finishing location, a check with that abstraction would return an abstract counterexample as seen on Figure 2.5b.

CounterExample-Guided Abstraction Refinement (CEGAR) [6] is an algorithm, that tries to utilize abstraction to efficiently check models, while also solving the problem stated above. The main idea is that we start with a very inaccurate precision of abstraction. Once we find a counterexample, we need to check if it is feasible in the concrete state space. If it is not, we need to refine our precision, which would eliminate the counterexample.

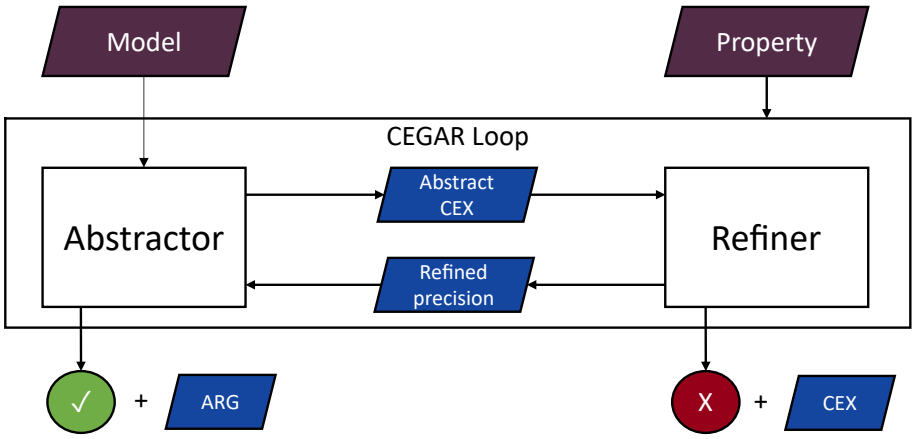


Figure 2.6: The CEGAR model checking flow

Once that is done, we can build a new abstract state space and go on in this loop until we either find a feasible counterexample and conclude the requirement to not hold, or find no counterexample at all. You can see the extended flow on Figure 2.6.

2.5.2.1 The abstractor

The abstractor part of the algorithm is responsible for applying a given precision of abstraction over the model and then looking for counterexamples in the generated abstract state space. Since most of the time CEGAR is used for reachability checks, the counterexample provided by the abstractor is usually a finite trace of abstract states, from which the last contains the concrete state desired to be unreachable.

2.5.2.2 The refiner

Once the abstractor finds an abstract counterexample, it is the refiner's job to determine if the counterexample defines any valid runs in the original model. In that case, the refiner finds the abstract counterexample feasible and returns the concrete trace as a valid counterexample.

If the abstract counterexample is found to be infeasible, the refiner needs to reason about why the run specified by the abstract counterexample can not be replicated in the concrete state space. Out of the many ways used in the literature to achieve an efficient behavior, my work relies on *interpolation-based refinement* [16].

Definition 8 (Craig interpolant [10]). Let A and B be FOL formulas, where $A \wedge B \models \bot$. A FOL formula I is an interpolant for A and B if:

- $A \implies B$, i.e. A implies B
- $A \wedge B \models \bot$, i.e. A and B are unsatisfiable together
- I only refers to variables that appear in both A and B .

It is proven [10], that there always exists at least one interpolant for two mutually exclusive formulas. Interpolants are used to invalidate an infeasible abstract counterexample, by providing a formula that appended to the current precision prevents this counterexample from occurring again.

Refinement is done with the help of Satisfiability Modulo Theories (SMT) solvers. SMT solvers are tools, that take a sequence of logical constraints over indexed variables as inputs, and output whether any valuation of the variables exists, such that the constraints are satisfied. If not, they are capable of providing an interpolant. The refiner converts the trace into a sequence of constraints, and sequentially checks with the solver, whether the trace is concretizable or not.

Given the following:

- an abstract domain $\mathbb{D} = (D, \top, \bot, \sqsubseteq, \mathsf{expr})$
- a trace $\sigma = (d_0, o_1, d_1, o_2, d_2, \ldots, o_n, d_n), d_i \in D, o_i \in \mathit{Ops}$
- an SMT solver solver, which has the following operations:
  - $\textsc{put}(\varphi)$: appends the FOL constraint φ to the constraints on the solver
  - $\textsc{check}$: returns whether the constraints currently on the solver are satisfiable
  - $\textsc{itp}(\varphi_1, \varphi_2)$: given two FOL formulas, φ_1 and φ_2, that are unsatisfiable together, returns a Craig interpolant explaining their unsatisfiability
  - $\textsc{model}$: returns a satisfying assignment (model) if the constraints on the solver are satisfiable
- an operation $\textsc{unfold}(\varphi, i)$ that returns an indexed FOL formula from the primed FOL formula φ, by substituting v unprimed variables with their indexed version v_i, and primed variables v' with their indexed version v_{i+1}
- we also define the $\textsc{unfold}$ operation for paths. Given a trace $\sigma = d_0, o_1, d_1, o_2, d_2, \ldots, o_n, d_n$, the operation $\textsc{unfold}(\sigma)$ returns $\textsc{unfold}(d_0, 0) \wedge \textsc{unfold}(o_1, 1) \wedge \cdots \wedge \textsc{unfold}(o_n, n) \wedge \textsc{unfold}(d_n, n)$, i.e. the conjunction of applying the $\textsc{unfold}$ operation to all operations and states of the trace, with their index as the i parameter

the refinement algorithm is:

Algorithm 1 Refinement

```
1: solver.PUT(UNFOLD(expr(d_0), 0))
2: for i ← 1, n do
3:     solver.PUT(UNFOLD(o_i, i))
4:     solver.PUT(UNFOLD(expr(d_i), i))
5:     if ¬solver.CHECK then
6:         itp ← solver.ITP(UNFOLD(d_0, o_1, ..., o_{i-1}, d_{i-1}), UNFOLD(o_i, i) ∧ UNFOLD(d_i, i))
7:         refutation ← create refutation from itp:
8:             replace all v_i ∈ itp with v
9:         return {infeasible, refutation}
10: return {feasible, ⊥}
```

The algorithm returns whether the received trace if feasible or not, and if the trace was found unfeasible, then also a FOL formula called *refutation*, which is created from the interpolant returned by the solver by removing the indices from the variables. This is formula is then used to refine the precision: in the case of predicate abstraction, the refutation formula is introduced as a new predicate into the precision and the case of explicit value abstraction, the variables that appear in the refutation formula are added to the precision.

2.6 CEGAR algorithm for LTL checking

By combining the algorithms in Section 2.4.2 and Section 2.5.2, one can achieve effective LTL model checking on complex models [22]. This algorithm utilizes CEGAR on the product model, which of course requires some modifications to the abstractor and refiner components of the simple CEGAR algorithm.

2.6.1 LTL formula preprocessor

Since by definition LTL formulas reason about atomic propositions, more complex properties given by FOL formulas need to be converted, before being able to be translated into NDBAs. This is done[22] by traversing the syntax tree, non-ltl type formulas are all swapped to new atomic propositions. The mapping from these Atomic Propositions (APs) to the original FOL formulas is preserved for later reconstruction. Please refer to [22] for more details.

2.6.2 Abstractor

Algorithm 2 Nested DFS

```
1: procedure NESTED_DFS
2:     DFS_BLUE(q_0)

3: procedure DFS_BLUE(q)
4:     q.blue ← true
5:     if q ∈ F then
6:         DFS_RED(q, q)
7:     for all t ∈ POST(q) do
8:         if ¬t.blue then DFS_BLUE(t)

Require: q, seed ∈ Q
9: procedure DFS_RED(q, seed)
10:    q.red ← true
11:    for all t ∈ POST(q) do
12:        if ¬t.red then DFS_RED(t, seed)
13:        else
14:            if t = seed then
15:                report cycle
```

The abstractor's job becomes a little bit more complex than that of Section 2.5.2.1. It is the abstractor's responsibility to find an accepting lasso trace if such exists. If the state space is depicted as a graph, a lasso trace is essentially a strongly connected component that is reachable from one of the initial states. To find accepting states that are part of a strongly connected component, Nested Depth-First Search (NDFS)[9] can be used, by looking for an accepting state, and then searching for a way from it back into itself. This algorithm can be seen on Algorithm 2.

2.6.3 Refiner

The refiner also gains some extra complexity. It takes a lasso trace $\sigma_L(w) = \langle T, H, L \rangle$ as an input and returns it as a valid counterexample or a refined precision that the abstractor can use to avoid this trace. First, it needs to check if the trace would even exist in the concrete state space. For that, it works exactly as defined in Section 2.5.2.2, but on the flattened $T \cup L$. If it is found to be feasible, it is proven that the lasso as a trace is traversable in the concrete model.

The challenge is validating if the loop exists in the concrete state space as a cycle too. A direct approach [22] can be used, that checks whether the honda can be in the same

state before and after the loop. This is done by adding the cycle validity constraint to the loop. CYCVAL(i, o) returns the following formula: $\bigwedge_{v \in V} v_i = v_{i+o}$, expressing that in the i-th and $(i + o)$-th states all variables have the same values. The extended algorithm is as follows:

Algorithm 3 Refinement

```
1: solver.PUT(UNFOLD(expr(d_0), 0))
2: for i ← 1, n do
3:     solver.PUT(UNFOLD(o_i, i))
4:     solver.PUT(UNFOLD(expr(d_i), i))
5:     if ¬solver.CHECK then
6:         itp ← solver.ITP(UNFOLD(d_0, o_1, ..., o_{i-1}, d_{i-1}), UNFOLD(o_i, i) ∧ UNFOLD(d_i, i))
7:         refutation ← create refutation from itp:
8:             replace all v_j ∈ itp, j ∈ {0, ..., i} with v
9:         return {infeasible, refutation}
10: model ← solver.MODEL
11: solver.PUT(CYCVAL(|T|, |L|))
12: if ¬solver.CHECK then
13:     itp ← solver.ITP(UNFOLD(σ), CYCVAL(|T|, |L|))
14:     refutation ← create refutation from itp:
15:         replace all v_{|T|} ∈ itp with model(v)
16:         replace all v_{|T|+|L|} ∈ itp with v
17:     return {infeasible, refutation}
18: return {feasible, ⊥}
```

In this case, creating a refutation from the interpolant is a bit more tricky, because variables can appear with multiple different indices ($|T|$ and $|T| + |L|$) in the interpolant. We circumvent this by replacing variables that appear with the $|T|$ index with a concrete value from the satisfying assignment we obtained for the tail previously.

2.7 Theta framework

Theta [23] is "a generic, modular and configurable model checking framework developed at the Critical Systems Research Group of Budapest University of Technology and Economics, aiming to support the design and evaluation of abstraction refinement-based algorithms for the reachability analysis of various formalisms. The main distinguishing characteristic of Theta is its architecture that allows the definition of input formalisms with higher level language front-ends, and the combination of various abstract domains, interpreters, and strategies for abstraction and refinement. Theta can both serve as a model checking backend, and also includes ready-to-use, stand-alone tools" [12].

	Common	**CFA**	**STS**	**XTA**	**XSTS**
Tools		cfa-cli	sts-cli	xta-cli	*xsts-cli*
Analyses	*analysis*	cfa-analysis	sts-analysis	xta-analysis	*xsts-analysis*
Formalisms	*core*, common	cfa	sts	xta	*xsts*
Solvers	solver, solver-z3				

Table 2.1: An overview of the Theta architecture.

Theta is built from the ground up to be as modular as possible. As seen on Table 2.1, everything formalism-independent is abstracted to the common module, and the modules of certain formalisms can implement or use features from this common module. This makes the implementation of new algorithms easier, developers can reuse the implementation of lower-level concepts.

2.8 Related work

The Spin [17] model checker is a widely used LTL model checker for asynchronous systems with message-based communication. Spin accepts its input models in the Promela language and employs decision-diagram representations and partial order reduction techniques for efficient model checking.

Model checking algorithms and tools that only support the verification of reachability properties can also be used to verify liveness and temporal logic properties through the liveness to safety [4] transformation. In this case, the model is augmented with additional variables that encode the extra information that is required to decide the more complex properties.

In [20], automata-theoretic LTL-checking is combined with another symbolic model checking algorithm: saturation. In decision diagram-based symbolic model checking algorithms, the model states and the transition relation are represented with decision diagrams, and the fixpoint of repeatedly applying the transition relation to the reached set of states is calculated. The saturation algorithm can calculate this fixpoint particularly efficiently in the case of distributed systems whose behavior is mainly local.

In [11], automata-theoretic LTL checking is combined with abstraction in the domain of computer programs. They limit the scope of the verification to terminable programs and define an alternate version of LTL that is interpreted over finite paths. These alternate LTL formulas can be expressed using deterministic finite automata, which makes their verification computationally less demanding than regular LTL model checking.

Chapter 3

Abstraction-based algorithms for configurable automata-theoretic model checking

This chapter discusses my theoretical contributions in creating a modular abstraction-based automata-theoretic LTL checking algorithm.

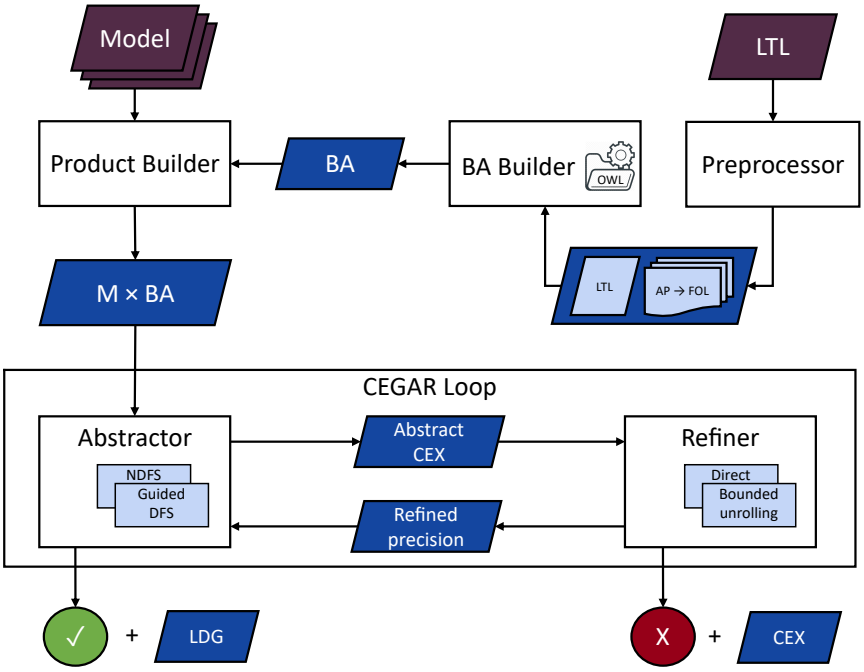


Figure 3.1: Overview

3.1 Overview of the presented abstraction-based LTL-checking approach

Figure 3.1 shows an overview of my configurable abstraction-based algorithm for LTL model checking. Automata-theoretic LTL model checking needs several components as building blocks, and I aimed to make each of these as generic as possible so that they

can be used in other use cases. The first step is the creation of a Büchi automaton from the LTL formula given as input. In my approach, the Büchi automaton is represented in a way that is compatible with the model formalisms. Then we need to represent the product of the original model and the Büchi automaton. For this, I utilized a generic *product formalism*, which can be used to create the product of models of any formalism. This formalism can also be used to create a composition of submodels, even when they are represented with different formalisms, and for representing the product with other kinds of automata used for other properties (e.g. tree automata for CTL, or timed Büchi automata for verifying timed systems). The resulting product is verified using a CEGAR-based language emptiness-checking algorithm, designed for finding lasso traces. This component is highly reusable as well. It can handle both state-based and transition-based acceptance criteria, and the input need not come from the LTL context. These can also be configured, as to which algorithms to use. In the abstractor component, I implemented the NDFS approach introduced in Section 2.6.2, but I also created a novel algorithm. In the refiner component, I included the algorithm shown in Section 2.6.3, but my refiner algorithm proposed in this work is available too.

3.2 Efficient language-emptiness checking

As stated in Theorem 3, the language of an NDBA is empty, if no lasso-shaped accepting runs can be shown. In the following parts, I propose a novel algorithm that provides a highly efficient solution for the problem. To maintain configurability, it supports both state-based and transition-based acceptance. For this purpose, I also propose a variant of NDFS so that it works on transition-based acceptance too.

3.2.1 Guided DFS

In the following part, I present the novel honda-guided search algorithm, aimed at efficiently finding accepting lasso traces in any kind of state graph. The basis of the algorithm is Depth-First Search (DFS), where we do only one search with a modified condition instead of two nested ones compared to NDFS. Let us call encountering an accepting state/transition an *acceptance*. In the DFS, while keeping an acceptance counter, look for a node that is already on the stack and has a smaller value on the counter than the top of the stack. The simple DFS Algorithm 4 showcases this. Here, q_0 denotes the initial node, $\langle q_1 \xrightarrow{e} q_2 \rangle$ denotes an edge e that goes from node q_1 to q_2, $\mathbb{F} = \{\langle q_1 \xrightarrow{e} q_2 \rangle |\ e \textit{ is accepting} \vee q_2 \textit{ is accepting}\}$ is the acceptance condition. $\textsc{out}(q)$ returns the set of all outgoing edges from q. Q is the collection holding all explored nodes.

Terminating the recursion

Algorithm 4 provides an elegant and clean DFS solution, leaving the $\textsc{stop}()$ predicate to be easily configured. Choosing the right termination condition for this recursive function is the key to making it efficient. First I examine two simple possibilities, then combine them with the idea of honda-guidance, to arrive at the final algorithm.

The simplest idea would be to use $\textsc{explored}(q) = q \in Q$, which checks if q is already explored. It would be fast, but it also would miss potential lassos. This problem can be seen on Figure 3.2 for both state- and transition-based acceptance. After reaching the

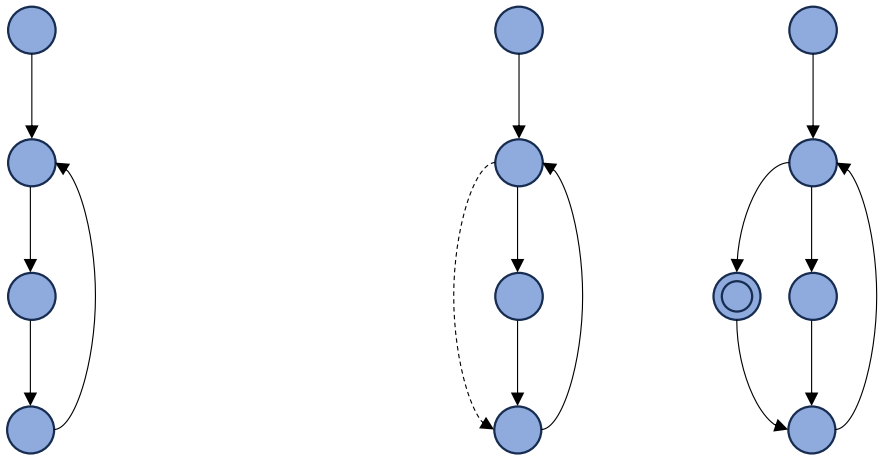
Algorithm 4 Simple DFS

Require: q_0 an initial node
Ensure: returns *safe* with $\emptyset$, or *unsafe* with a counterexample

1: **function** DFS(q_0)
2: ▷ *Construct a dummy initial edge* ◁
3: **return** SEARCH($\langle \_ \xrightarrow{e} q_0 \rangle, 0, \{\}$)

Require: $\mathfrak{a} \in \mathbb{N}$ current acceptance counter
path the nodes and acceptance counters on the stack

4: **function** SEARCH($\langle q_1 \xrightarrow{e} q_2 \rangle, \mathfrak{a}, path$)
5: **if** $e \in \mathbb{F}$ **then** $\mathfrak{a} \leftarrow \mathfrak{a} + 1$ ▷ *increment counter of acceptances if needed*
6: ▷ *Incrementation is done first, to find single element loops* ◁
7: **if** $\{q_2, i\} \in path \wedge i < \mathfrak{a}$ **then**
8: **return** $\{unsafe, path \cup q_2\}$
9: **if** STOP() **then return** $\{safe, \{\}\}$
10: $Q \leftarrow Q \cup \{q_2\}$
11: **for all** $e_p \in$ OUT(q_2) **do**
12: $result =$ SEARCH($e_p, \mathfrak{a}, path \cup \{q_2, \mathfrak{a}\}$)
13: **if** *result* is *unsafe* **then return** *result*
14: **return** $\{safe, \{\}\}$



(a) Graph before first backtrack **(b)** Accepting loops missed

Figure 3.2: Graph space progression showcasing the problem of STOP=EXPLORED

state shown on Figure 3.2a and backtracking to the second node, the left path is explored. However, once we reach the node pictured as the bottom one, EXPLORED would stop the algorithm, missing the obvious lasso.

Another possibility would be to use $\text{INPATH}(q, path) = q \in path$, which evaluates whether the node is already in the current path. It would be a sufficient criterion, however, it would not be in itself useful, as preliminary measurements have shown a huge lack of efficiency.

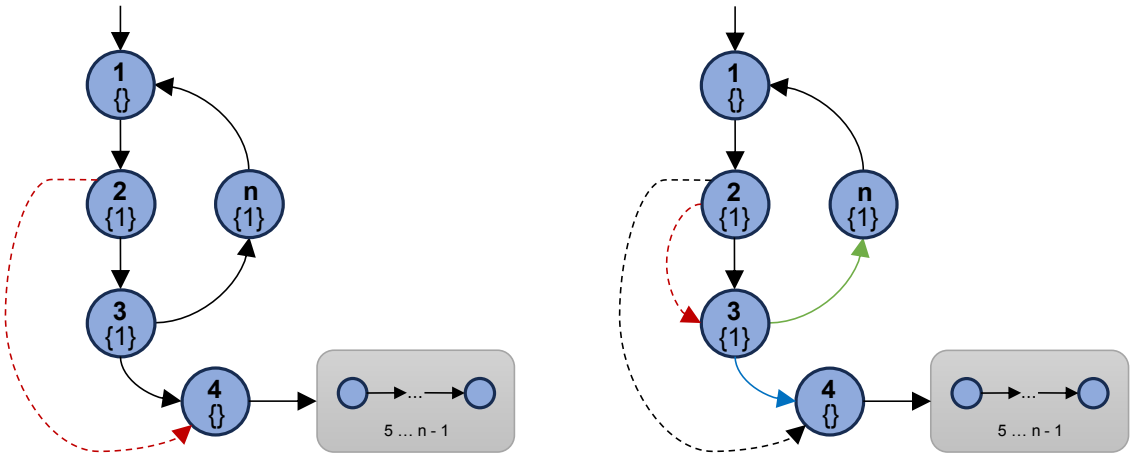
We need a way to tell upon reaching an already explored node, whether it is worth traversing the nodes behind it. We know for sure if we start traversing behind such nodes, we are only going to encounter nodes that have already been explored too, because of the DFS nature of the exploration algorithm. Such a node can lead back into our current path only if that loop is already known. We also know that such a loop can not contain any acceptance inside, since that would already have concluded our search. We just need the information about every explored node, which loops it is part of. Hence, by expanding Algorithm 4 with storing all hondas for every explored node, if we later encounter such a node, we just need to check if any of its registered hondas are on the current path. If not, the node is for sure not worth traversing through. The extension can be seen on Algorithm 5, where $\mathcal{H}_q$ denotes the set of marked hondas for node q.

Algorithm 5 Guided DFS

Require: q_0 an initial node
Ensure: returns *safe* with $\emptyset$, or *unsafe* with a counterexample
1: **function** DFS(q_0)
2: ▷ *Construct a dummy initial edge* ◁
3: **return** SEARCH($\langle \_ \xrightarrow{e} q_0 \rangle, 0, \{\}$)

Require: $\mathfrak{a} \in \mathbb{N}$ current acceptance counter
path the nodes and acceptance counters on the stack
4: **function** SEARCH($\langle q_1 \xrightarrow{e} q_2 \rangle, \mathfrak{a}, path$)
5: **if** $e \in \mathbb{F}$ **then** $\mathfrak{a} \leftarrow \mathfrak{a} + 1$
6: **if** $\{q_2, i\} \in path$ **then**
7: **if** $i < \mathfrak{a}$ **then**
8: **return** $\{unsafe, path \cup \{q_2, \mathfrak{a}\}\}$
9: **else**
10: $\mathcal{H}_{q_1} \leftarrow \mathcal{H}_{q_1} \cup \{q_2\}$
11: **return** $\{safe, \{\}\}$
12: **if** EXPLORED(q_2) $\wedge((\nexists h \in \mathcal{H}_{q_2})[\{h, i\} \in path \wedge i < \mathfrak{a}])$ **then**
13: **return** $\{safe, \{\}\}$
14: **for all** $e_p \in \text{OUT}(q_2)$ **do**
15: $result = \text{SEARCH}(e_p, \mathfrak{a}, path \cup \{q_2, \mathfrak{a}\})$
16: **if** $result \leftarrow unsafe$ **then return** $result$
17: **return** $\{safe, \{\}\}$

What makes this simple terminating check an efficiency boost to our search is exactly the already used search condition. We can not only tell, if traversing through a node leads back into our path forming a lasso: we can know for sure if such lasso *contains an acceptance* in its loop. We just simply need to check if the number of acceptances for said honda in our current trace is smaller than the acceptance counter at the top of the call stack, similar to the happy-path termination. Now that we know we can find the accepting lasso somewhere behind the node, we are still not blind trying to find the trace. The extra information on the nodes also guides the search right back into the current path, without taking any misdirection.



(a) Next edge (red) after the first backtrack to node 2

(b) Next edge (red) after the second backtrack to node 2

Figure 3.3: Example showing the strength of guided DFS

Take for example Figure 3.3. The search first explored the first $n-1$ nodes in order. Since it came to a dead-end, the algorithm backtracks to node 3. Here the next recursive call explores node n and then reaches node 1. There the stop condition terminates, as node 1 is on the current path, however, our acceptance counter is 0 both at node 1 and on the stack. As we are backtracking to node 2, we mark node 1 as a valid honda on every node. Now from node 2 the algorithm takes on the edge marked as red on Figure 3.3a. Node 4 is encountered, which is already explored, but not on the current path. After that, the algorithm checks, if any of the hondas marked on node 4 are on path. Since none, the execution here terminates, backtracking again to node 2.

Following that, the search continues with the red edge seen on Figure 3.3b. The algorithm finds, that the honda node 1 marked on node 3 is on the current path. Since the acceptance counter on the path for node 1 is 0, while on the stack it's 1, it is clear that traversing beyond node 3 would lead to a valid lasso trace. First, it takes the blue edge, but there it terminates for the same reason as before. Then it takes the green edge and using the same guidance would efficiently reach back to node 1 finding the valid lasso trace.

3.2.2 NDFS for transition-based acceptance

For simple graphs, the same NDFS algorithm could be used after some modification to the graph. For every accepting edge $\langle q_1 \xrightarrow{e} q_2 \rangle$, add a new accepting node q_n. Remove e from the graph, and add the edges $\langle q_1 \xrightarrow{e_{ln}} q_n \rangle$ and $\langle q_n \xrightarrow{e_{rn}} q_2 \rangle$ to the graph. Figure 3.2b is a great example of this, where the modification on the left graph would result in the right one.

However, considering that we want to look for lasso traces in a product of multiple state spaces, creating artificial states is not an option, we need to modify Algorithm 2. The idea is that once we find an accepting edge, we start the second search for the "from" node of the found edge. The modification can be seen on Algorithm 6 annotated with blue.

Algorithm 6 Nested DFS for transition-based acceptance

```
1: procedure NESTED_DFS
2:     ▷ Construct a dummy initial edge ◁
3:     DFS_BLUE(⟨_ --e--> q0⟩)

Require: e to be an edge
4: procedure DFS_BLUE(⟨q0 --e--> q⟩)
5:     q.blue ← true
6:     if e ∈ F then
7:         DFS_RED(q, q)
8:     for all ⟨q --ep--> q2⟩ ∈ OUT(q) do
9:         if ¬q2.blue then DFS_BLUE(ep)

Require: q, seed ∈ Q
10: procedure DFS_RED(q, seed)
11:     q.red ← true
12:     for all t ∈ POST(q) do
13:         if ¬t.red then DFS_RED(t, seed)
14:         else
15:             if t = seed then
16:                 report cycle
```

Algorithm 7 Guided DFS full exploration

Require: q_0 an initial node
Ensure: returns *safe* with $\emptyset$, or *unsafe* with all counterexamples

```
1: function DFS(q0)
2:     ▷ Construct a dummy initial edge ◁
3:     return SEARCH(⟨_ --e--> q0⟩, 0, {})

Require: 𝔞 ∈ ℕ current acceptance counter
         path the nodes and acceptance counters on the stack
4: function SEARCH(⟨q1 --e--> q2⟩, 𝔞, path)
5:     if e ∈ 𝔽 then 𝔞 ← 𝔞 + 1
6:     if {q2, i} ∈ path then
7:         if i < 𝔞 then
8:             ▷ return it as a one-element set
9:             return {unsafe, {path ∪ {q2, 𝔞}}}
10:        else
11:            H_q1 ← H_q1 ∪ {q2}
12:            return {safe, {}}
13:    if EXPLORED(q2) ∧((∄h ∈ H_q2)[{h, i} ∈ path ∧ i < 𝔞]) then
14:        return {safe, {}}
15:    C ← {}
16:    for all ep ∈ OUT(q2) do
17:        result ← SEARCH(ep, 𝔞, path ∪ {q2, 𝔞})
18:        if unsafe ∈ result then
19:            C ← C ∪ (paths from result)
20:    end for
21:    if |C| > 0 then
22:        return {unsafe, C}
23:    return {safe, {}}
```

3.2.3 Guided DFS for full exploration

Guided DFS can be easily configured to explore the whole state space, and return all lasso counterexamples. Since the algorithm is specifically designed to reduce aimless wandering in the state space, it should be able to efficiently return all lasso traces right after the full exploration is done. This possibility opens up the doorway to using multiple counterexamples for refinement [15] in a later work. Please find the modifications annotated in Algorithm 7. By not stopping upon finding an unsafe result, the algorithm guarantees to explore the whole state space before terminating. The introduction of a set $\mathcal{C}$ before the recursive calls allows the algorithm to collect all the counterexamples from the exploration.

3.3 Bounded unrolling for lasso traces

The idea of bounded unwinding comes from E. Clarke et al., who defined an algorithm [6] to detect and refine lasso-shaped traces in the abstract state-space. The idea was that even though we do not know if an abstract loop directly corresponds to a concrete loop, we can be sure that the abstract loop can be concretized at most m different ways, where m is the size of the smallest abstract state in the loop (if we think about abstract states as sets of concrete states). That is because, if the loop is run m times and is concretizable, the state that had the smallest number of concrete states has to repeat itself at least once. The only limitations of the original algorithm were that it was defined for deterministic operations only.

To slightly mitigate this limitation and be able to use the algorithm, we need to eliminate as many nondeterministic operations as possible. To achieve this, nondeterministic operations have to be unfolded: they are replaced with all their possible deterministic counterparts. In this work, we are currently supporting the unfolding of the "choice" operation.

Another limitation of the original algorithm in our context is that it I am working with possibly infinite domains, for which m could also potentially evaluate to infinite. To have a chance to avoid these infinite unwindings, it is worth noting that counting all the concrete states allowed by the abstract states in the loop is an overapproximation of the number of all possible different states the concrete loop can reach. If a variable is included in only such assignments (or no assignments at all) where the expression contains only literals, that variable has a fixed value throughout the loop. That means, for such variables, just one unwinding is enough.

To find all the variables that contribute towards the needed number of unwindings, I created Algorithm 8. It works on a lasso trace $\sigma_L(w) = \langle T, H, L \rangle$, where if o is an assignment operation, o.var returns the variable that is being assigned the expression o.expr. The function VARS(expr) returns the variables appearing in the expression expr.

Using the variables collected by the algorithm, I can find an even smaller m than defined by the original algorithm, and check the trace more efficiently. Since infinite bounds can still be encountered, there is a configurable maximum for the bound. If m would be greater than that, the refiner defaults to the direct approach introduced in Section 2.6.3.

Algorithm 8 Variable collector

```
1: function COLLECT_VARS
2:     return COLLECT_VARS({})
3: function COLLECT_VARS(𝒱)
4:     s ← |L|
5:     for all o ∈ L do
6:         if o is Assignment then
7:             𝒱_e ← VARS(o.expr)
8:             v ← o.var
9:             if o.var ∈ 𝒱_e ∨ 𝒱 ∪ 𝒱_e ≠ ∅ then
10:                𝒱 ← 𝒱 ∪ {o.var}
11:    if |L| > s then
12:        return COLLECT_VARS(𝒱)
13:    return 𝒱
```

3.4 Theory of the composite formalism

As stated in Section 2.4.2, for automata-theoretic model checking of LTL formulae one needs the product of the model and the Büchi automaton. However, certain use cases might include models that should already be products of two or more primitive ones. Also, the implementation behind the Büchi automaton might change, or for different types of algorithms, one might need different automata. To proactively tackle these challenges, I created a composite formalism that is:

1. Indefinitely nestable
2. Generic enough to support a wide variety of components

Even though I was aiming for item 2, composition in this work is limited to operation-based formalisms. To easily reason about composite models, I introduce the set of sides, which consists of two atoms for "left" and "right": $\mathsf{sides} = \{L, R\}$. I also introduce the complementer operator for sides, which just returns the other side than the operand, i.e., $\overline{L} = R$ and $\overline{R} = L$.

Theorem 4 (Composite formalism). Given the following:

- the left modeling formalism $(V_L, D_L, B_L, S_L, \iota_{BL}, \alpha_L)$
- the left composite member interface $(\mathcal{E}_L, \mathcal{B}_L, \mathcal{S}_L)$
- the right modeling formalism $(V_R, D_R, B_R, S_R, \iota_{BR}, \alpha_R)$
- the right composite member interface $(\mathcal{E}_R, \mathcal{B}_R, \mathcal{S}_R)$
- a next side function $Sd : B_L \times B_R \times D \to \{L, R\}$
- an init side $\mathfrak{s}_i \in \{L, R\}$

the left and right formalisms can be combined into an operation-based composite formalism $(V_C, D_C, B_C, S_C, \iota_{BC}, \alpha_C)$ such that:

- $V_C = V_L \cup V_R$

- $B_C = B_L \times B_R \times \{L, R\}$
- $\iota_{BC} = \iota_{BL} \times \iota_{BR} \times \{\mathfrak{s}_i\}$
- $\alpha_C(s = \{\{b_L, b_R, \mathfrak{s}\}, d)\} = \left\{\{o, \{b, b_{\overline{Sd(s)}}, Sd(s)\}\} | \{o, b\} \in \alpha_{Sd(s)}(\{b_{Sd(s)}, d\})\right\}$ ▪

Note, that we don't reduce the data state to the variables delegated by a model when we pass the data state to that model's action function. From the practical point of view, this means that if there is a variable reference occurring in only one of the models, it is considered ignored in the other model. This not only makes the definition more simple and intuitive but also comes in handy considering our use case.

3.5 Abstraction-based linear temporal logic model checking

Recalling Figure 3.1, I have now formally defined both components in the *CEGAR Loop* and the component *Product Builder*. However, composition only works on operation-based formalisms. We need an operation-based representation of NDBAs.

Definition 9 (BA). Operation-based BA is a modeling formalism given by the tuple $\text{BA} = (V, L, l_0, T, F_s)$ where

- $V = \{v_1, v_2, \dots\}$ is the set of variables appearing in the program
- $L = \{l_0, l_1, \dots\}$ is the set of control locations modeling the actual position of the program counter
- $l_0 \in L$ is the initial location representing the entry point of the program
- $T \subseteq L \times Ops_{\text{ASS}} \times L$ is a set of directed edges between the locations, annotated with assume operations over the variables
- $F \subseteq L$ is the state-bassed acceptance condition
- $F \subseteq T$ is the transition-based acceptance condition

BA fulfills the operation-based modeling formalism interface the same way as a CFA does in Theorem 2. ▪

Such a BA is created by the *BA Builder* component. A raw LTL string is first transformed by the preprocessor mentioned in Section 2.6.1. This yields a map from APs to FOL formulas, as well as the transformed LTL formula. This transformed formula is fed into a library called Owl [19], developed at the Technische Universität München. The library creates an NDBA in Hanoi Omega-Automata Format (HOAF). *BA Builder* then read this HOAF and creates a BA, by substituting every atomic proposition with the mapped FOL formula wrapped inside an assume operation.

Given any operation-based model, we can now effectively check requirements given with LTL formula on it. The formula is converted to a BA as described above. Then a composition is created from the model and the BA. If we have multiple models describing synchronous systems, they can be composited together, and then a new composition can be created from the composite submodels and the BA. After that, a configurable CEGAR loop can check, whether the requirement holds for the model(s).

Chapter 4

Evaluation

This chapter presents the experimental evaluation of my theoretical and algorithmic contributions using benchmark models. I have implemented the algorithms and the product formalism described in this work as a part of the Theta model checking framework [1].

4.1 Experiment design

In this section I introduce all the model and parameter sets I used to benchmark my implementation.

4.1.1 Benchmark models

I used the following models with LTL properties to run LTL-only benchmarks:

- **simple**: Simple handcrafted models;
- **Weather**: Handcrafted model describing a person learning weather patterns to optimize clothing choices for their day;
- **TrafficLight**: Generated from a composite statechart network with 2 traffic light components and a controller component;
- **Crossroad**: Generated from the model used in the tutorial of the Gamma Statechart Composition Framework[2];
- **Mission**: Generated from a composite SysML statechart model modeling a spacecraft that communicates with a ground station;
- **COID**: A composite statechart network modeling components of railway safety equipment. Provided by a confidential partner of the university;

The following models were created for verifying reachability properties, which I automatically transformed to equivalent LTL queries (property φ to LTL formula G(φ):

- **simple**: Simple handcrafted models that cover all language constructs and features;

[1] https://github.com/ftsrg/theta
[2] `http://inf.mit.bme.hu/en/gamma`

- **TrafficLight**: Generated from a composite statechart network with 2 traffic light components and a controller component;
- **Spacecraft**: Generated from the SysML statechart model modeling a spacecraft from [18];
- **INPE**[3]: A composite system modeling two components of a communication protocol inside a nanosatellite;
- **COID**: A composite statechart network modeling components of railway safety equipment. Provided by a confidential partner of the university;
- **PIL**: A composite statechart network that models components of railway safety equipment. Models in this set are more complex than the **COID** set. Provided by a confidential partner of the university;
- **mcaas**: These models are generated from the open-source model of the Thirty Meter Telescope [8]. The models were modified by hand to remove language elements currently not supported by the mcaas framework;
- **mcaas-sliced**: These models were created by hand from the models in the **mcaas** set by splitting up the single monolithic transition into 80 smaller transitions. The models express equivalent behaviour.

4.1.2 Research questions

To evaluate my apprach and its implementation, I proposed the following research questions:

RQ1: How do Nested DFS and Guided DFS compare regarding CPU time? Does any of them outperform the other?

RQ2: How do the direct and bounded unrolling refinement strategies compare regarding CPU time? Does any of them outperform the other?

RQ3: Which config suits LTL checking the best? Does that depend on specific model types?

RQ4: What is the overhead of using the LTL-checking algorithm for reachability analysis compared to the dedicated reachability algorithm?

4.1.3 Configuration parameters - combinations for specific research questions

For the benchmarks, I used the *XstsCli* tool of the Theta framework. It already had a lot of configuration options, from which I reused all that were applicable to my algorithms. So I only needed to define three new parameters, two regarding the algorithm selections for the LTL CEGAR loop, and one to actually select LTL CEGAR checking instead of regular reachability. A summary can be seen on Table 4.1[4]. The searchLTL value NDFS refers to Algorithm 2 in Section 2.6.2, while the value DFS is the guided DFS from Algorithm 5.

[3]INPE (Instituto Nacional de Pesquisas Espaciais) refers to the brazilian National Institute for Space Research, who provided the **INPE** model set, see http://www.inpe.br

[4]For more detailed description of the options, see: `https://github.com/ftsrg/theta/blob/master/doc/CEGAR-algorithms.md`

The refinementLTL value MONDOK is the counterpart of Algorithm 3 from Section 2.6.3 and BOUNDED_UNROLLING is the algorithm detailed in Section 3.3.

Parameter	Possible values	Description
algorithm	CEGAR, LTLCEGAR	Define whether to use LTL or reachability checking
searchLTL	NDFS, DFS	Define which language emptiness checking algorithm to use
refinementLTL	MONDOK, BOUNDED_UNROLLING	Select LTL refinement algorithm from direct approach or bounded unrolling
domain	EXPL, PRED_CART, PRED_SPLIT, PRED_BOOL	Choose an abstract domain
refinement	BW_BIN_ITP, SEQ_ITP	Define to use binary or sequence interpolation
predSplit	WHOLE, ATOMS	Define whether to split new predicates after refinement

Table 4.1: Configuration parameters used, new ones above horizontal line

4.1.3.1 Parameter sets for LTL only benchmarks

Bound parameters:

- algorithm = LTLCEGAR
- domain = PRED_SPLIT
- predSplit = ATOMS

Free parameters:

- searchLTL: {NDFS, DFS}
- refinementLTL: {BOUNDED_UNROLLING, MONDOK}

4.1.3.2 Parameter sets for reachability benchmarks

Free parameters:

- algorithm: {CEGAR, LTLCEGAR}
- domain: {EXPL, PRED_CART, PRED_SPLIT, PRED_BOOL}
- predSplit: {WHOLE, ATOMS}
- refinement: {SEQ_ITP, BW_BIN_ITP}
- searchLTL: {NDFS, DFS}
- refinementLTL: {BOUNDED_UNROLLING, MONDOK}

4.1.4 Shortened configuration names

All configurations are encoded into a short name using the first few letters of the parameters.

For LTL-only benchmarks, configuration names are of the form: L_{M | BU}_{D | N}. For the middle part, BU denotes that the refinement strategy selected was bounded unrolling, whereas M denotes the direct refinement approach, adapted from the work of *Mondok et. al.* [22]. The last letter is D for Guided Depth-First Search (GDFS) and N for NDFS.

4.1.5 Benchmark environment

The measurements were carried out using virtual machines in the BME cloud. BenchExec [3] was used to ensure the reliability of the results.

LTL-only benchmarks were executed on a single machine with the following configuration:

- OS: Ubuntu 18.04
- CPU: 4 cores
- RAM: 8192 MB
- Set timeout: 1800 seconds (30 minutes)

As the reachability benchmark set contained much more models, the related measurements were performed on a distributed benchmark environment in the cloud, where BenchExec was used to constrain the RAM to 15 GB and the number of CPU cores to 3. The timeout for these measurements was set to 900 seconds (15 minutes).

4.2 The results of the benchmarks

I used LTL-only benchmarks to find the answers to research questions 1-3. Then I used the results of reachability checking benchmarks to answer research question 4.

4.2.1 LTL-only benchmarks

Research question 1

As the pairwise plot on Figure 4.1 shows, GDFS was able to outperform NDFS in many model categories. This shows that my algorithm is viable.

Research question 2

As seen on the pairwise plot Figure 4.2, the direct approach and the new bounded unrolling do not show real differences. I suspect this is because bounded unrolling has a hard limit on the number of unwinding and when that is reached, the refinement defaults to the direct approach.

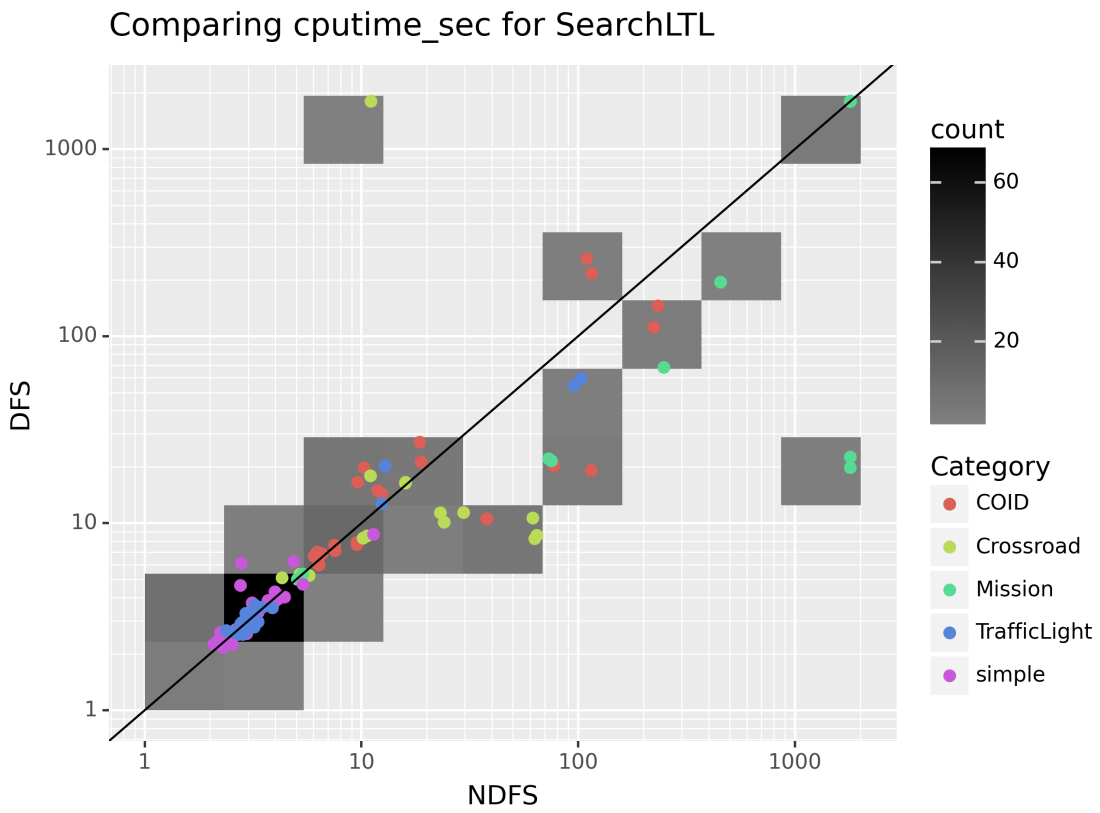


Figure 4.1: Measurement results for RQ1

Research question 3

Based on the heatmap seen on Figure 4.3, we can see that there are some model categories in which guided DFS could outperform NDFS. There are also categories where the performance is almost identical. The quantile plot seen on Figure 4.4 also shows the same, that guided DFS is slightly better performing. The pairwise plot on Figure 4.5 shows huge potential with the explicit value abstraction domain.

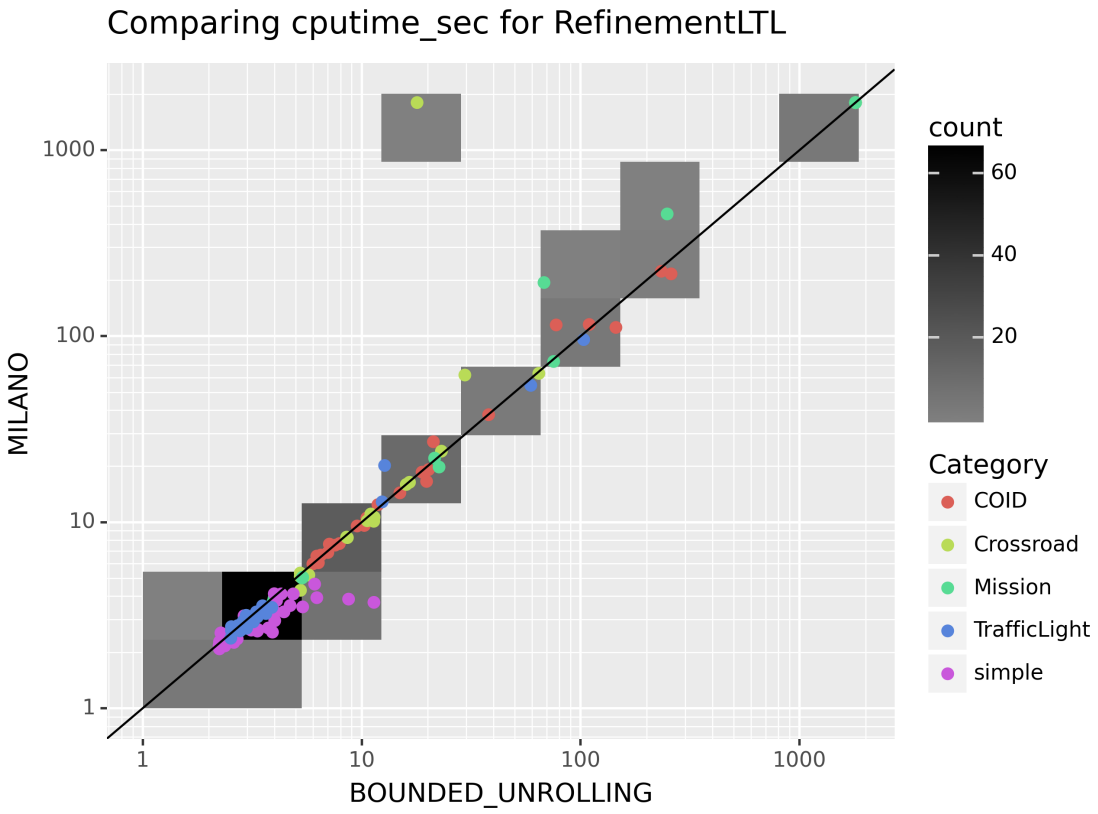


Figure 4.2: Measurement results for RQ2

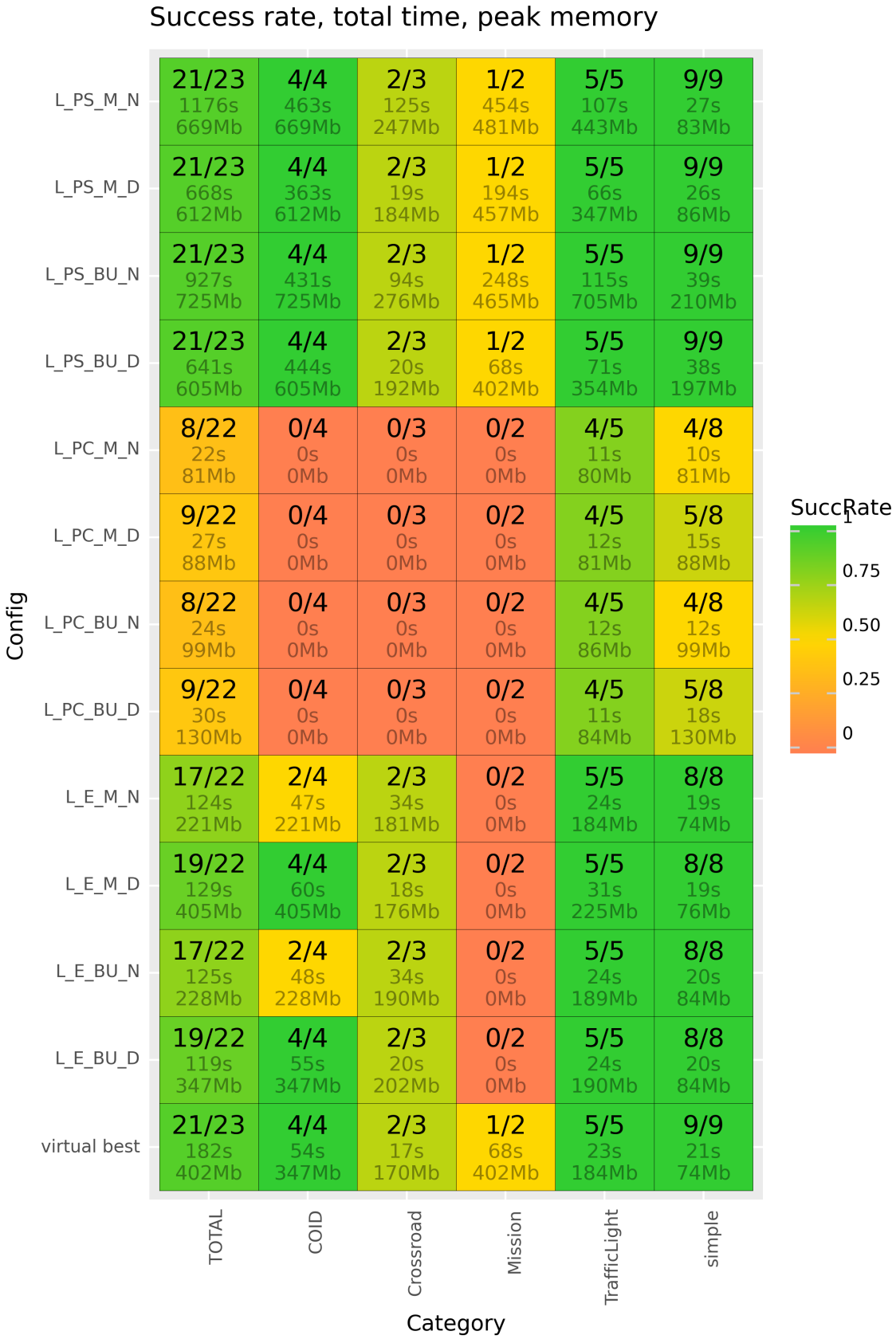


Figure 4.3: Heatmap illustrating measurement results for RQ3

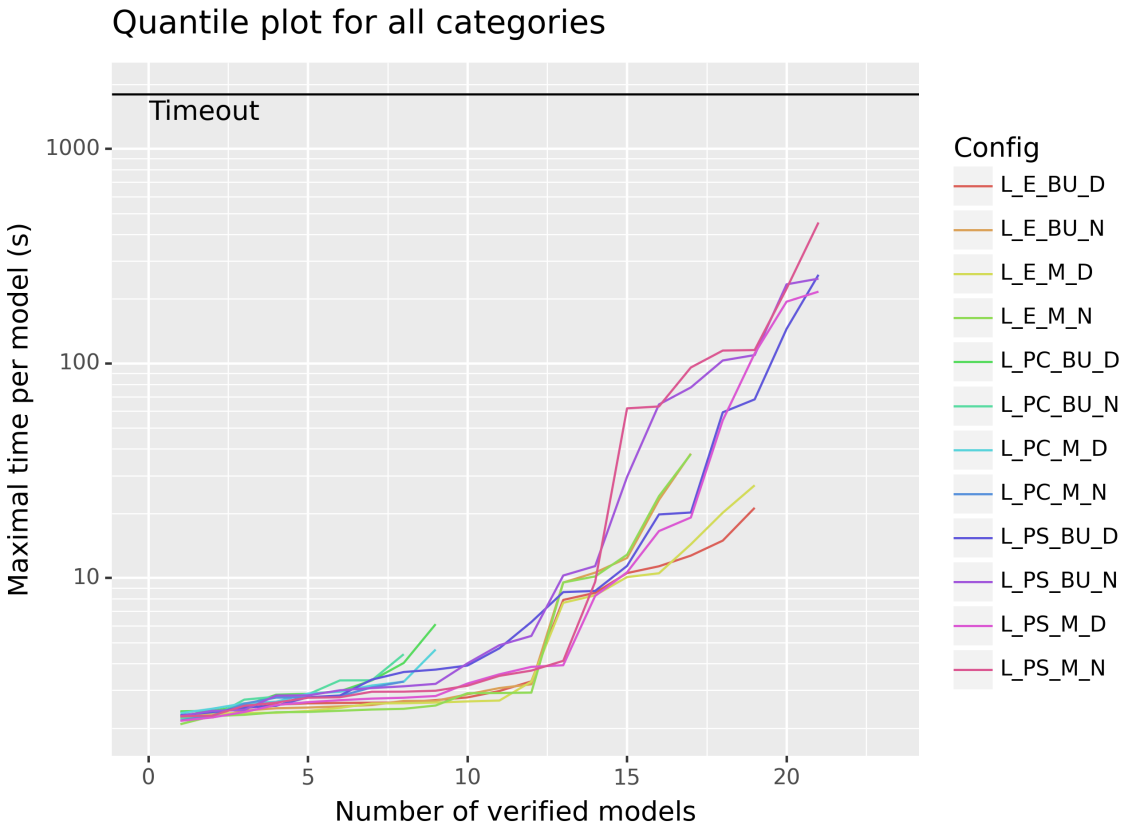


Figure 4.4: Quantile plot illustrating measurement results for RQ3

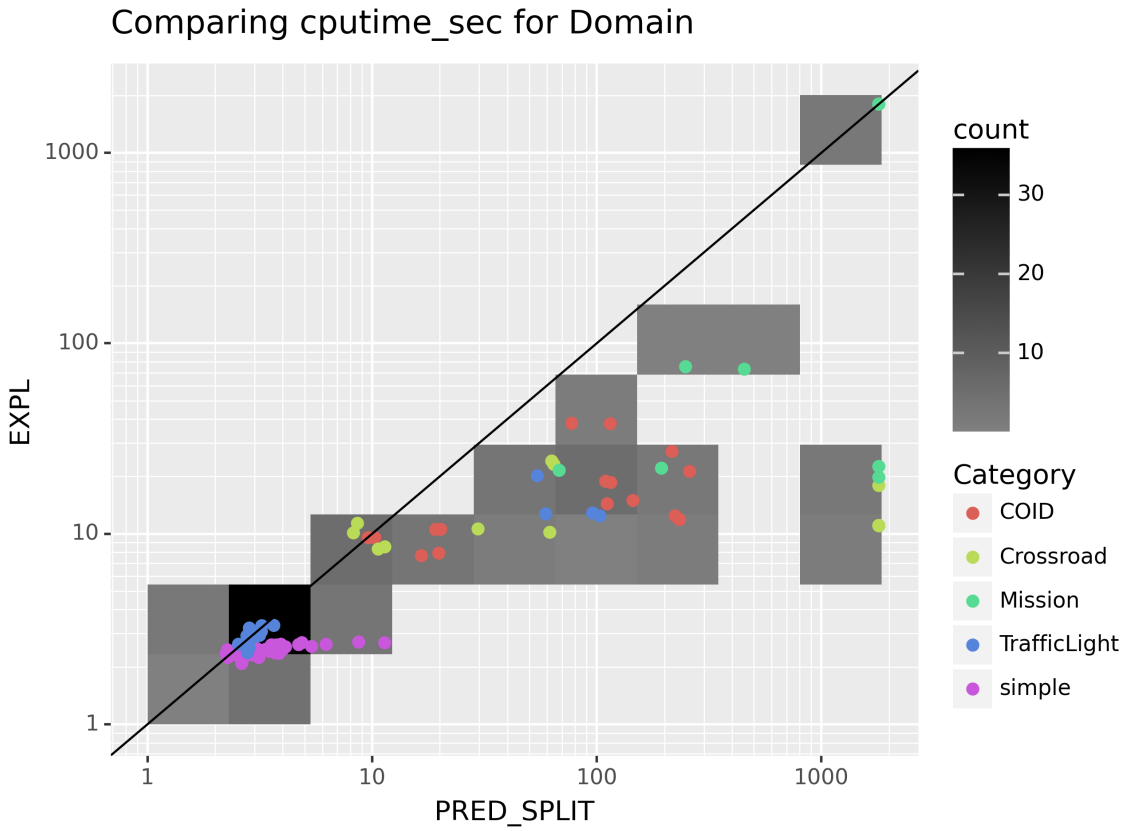


Figure 4.5: Pairwise plot illustrating measurement results for RQ3

4.2.2 Reachability benchmarks - Research question 4

The quantile plot on Figure 4.6 shows what was expected, that for reachability the classic CEGAR algorithm is way better and more efficient, being able to solve more problems. However, the pairwise plot on Figure 4.7 suggests that there are certain models where

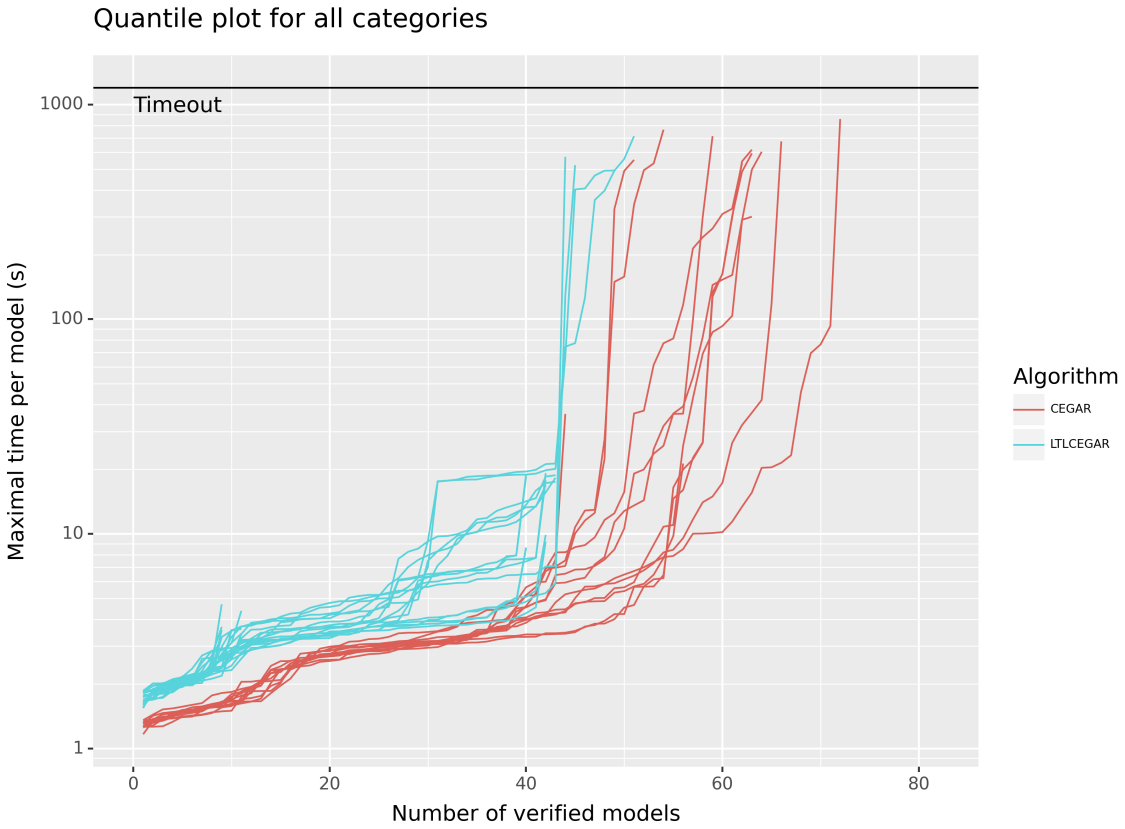


Figure 4.6: Quantile plot illustrating measurement results for RQ4

4.2.3 Threats to validity

Since I had access to only few models with well-defined LTL properties, the sample size and the representativity of the models limits the generalizability of my evaluations.

The results might have been impacted by the fact that the benchmarks were run in cloud configurations. I made every effort to counteract it by using BenchExec, and running the benchmarks in periods when the general usability of the cloud was the lowest, mainly outside of business hours.

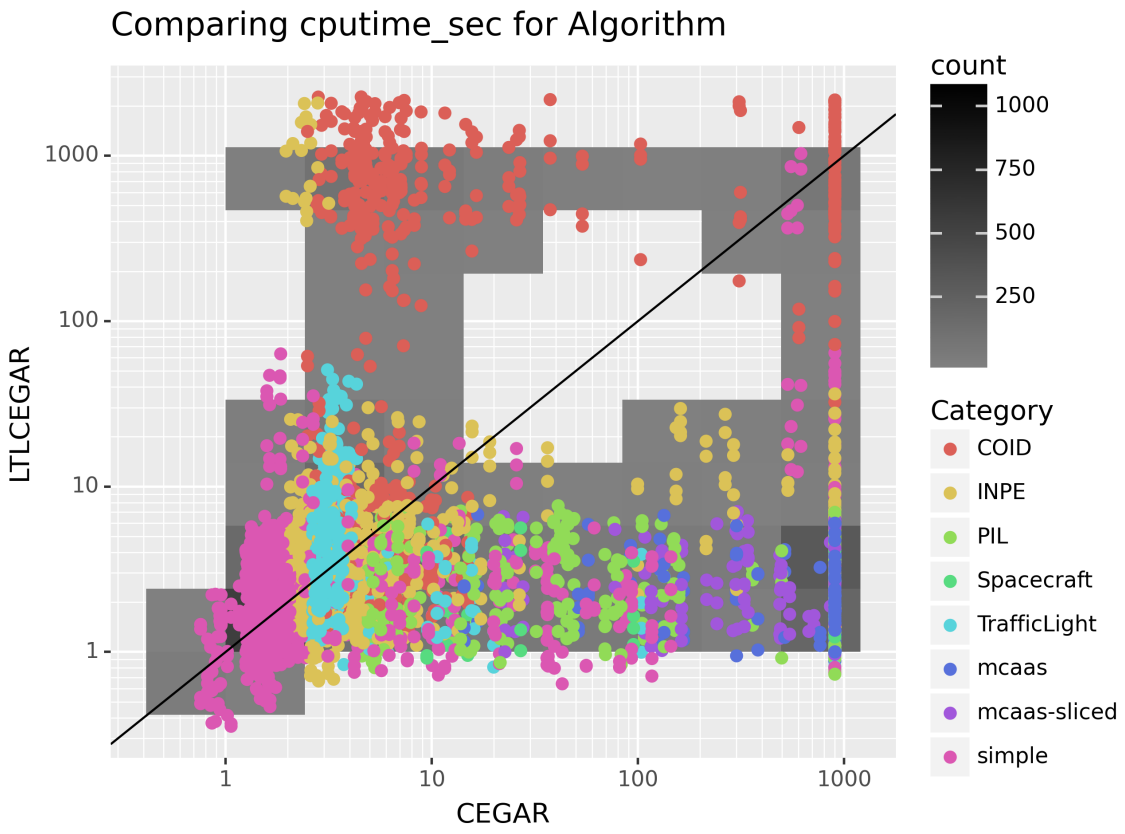


Figure 4.7: Pairwise comparison of LTL-based reachability with the dedicated reahcability algorithm for RQ4

Chapter 5

Conclusions

In this chapter, I draw conclusions from my work and outline some possible future directions it can be expanded in.

Theoretical contributions

To create a flexible LTL model checking workflow, I introduced the following concepts that can be used as building blocks of abstraction-based automata-theoretic algorithms:

- I defined what operation-based formalisms are, based on which I was able to give a formal definition of a *composite formalism* that supports the composition of an arbitrary number of nested models. I also defined how a Büchi automaton can be regarded as a state-based formalism. Using all these, products of Büchi automata and other models can be created;
- I introduced a novel *language-emptiness checking* algorithm called guided DFS;
- I presented a novel *refinement algorithm* for lasso-shaped traces based on the bounded unwinding algorithm proposed by Clarke in [6].

Combining the aforementioned algorithms and the composite formalism, I proposed an extension of the CEGAR algorithm that is capable of finding lasso-like traces. I then explained, how the lasso searching CEGAR algorithm can be used on the product to verify requirements given in the LTL formal specification language.

Practical contributions

I implemented my approach in the open source Theta framework. I specifically paid attention to making most of my contributions highly configurable and reusable, so that they could be part of any other type of workflow. The generic nature of my product formalism opens the door to many different approaches, such as modeling concurrent systems as standalone components. The implementations of the two lasso abstracting algorithms and the two refinement algorithms allow us to look for loops in any model, not just a product for LTL checking.

I made the following further modifications to the Theta framework to support my approach:

- I implemented an efficient data structure for the abstract state space used in lasso searching.
- In the Theta framework previously enum-like custom types were transformed integers. I introduced a new implementation of these types that uses the sort type of the SMT solver.

I evaluated my approach on a diverse set of models that included both handcrafted examples and real-life models that were provided by industrial partners of the university. The benchmarking yielded promising results, and also helped my identified areas where my approach could be further improved.

Future work

The most important would be to run the benchmarks on a wider set of models and configurations, to truly get a whole picture on how the algorithms perform. The bounded unrolling algorithm could be generalized to work on nondeterministic operations.

There are many directions the concepts in this work could be evolved. It is clear, that the "counter" problem is not solved. Given predicate abstraction and an integer slowly increasing in value in the concrete state space, the CEGAR algorithm needs multiple whole cycles to come to a precision that eliminates the false counterexample for that loop. To solve this issue, one could use the concept of weakest preconditions, to eliminate such loops earlier.

A new refiner algorithm could also be introduced, that is capable of taking advantage of the multiple lasso traces the full exploration can yield. I would also like to introduce generalized Büchi automata and compare their performance to NDBAs. An interesting idea would be to achieve slight optimizations, to use some of the Büchi automatons transitions as initial precisions, since most of the first few iterations in the CEGAR loop usually return those.

Acknowledgements

Many many special thanks to both of my advisors, Milán and Dániel. They sacrificed countless hours to introduce me into the depth of formal verification methods, helped me understand every part of the theta framework, and were there generally at any given time to aid me when I was lost. If I didn't understand something, they made sure to stay with me until I got grasp of the theme. It was a pleasure being advised by them.

Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the partial support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X.

[2] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37057-1.

[3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019. DOI: 10.1007/S10009-017-0469-Y. URL https://doi.org/10.1007/s10009-017-0469-y.

[4] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In Rance Cleaveland and Hubert Garavel, editors, *FMICS 2002, ICALP 2002 Satellite Workshop*, volume 66 of *Electronic Notes in Theoretical Computer Science*, pages 160–177. Elsevier, 2002. DOI: 10.1016/S1571-0661(04)80410-9.

[5] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. 1990. ISBN 978-1-4613-8928-6. DOI: 10.1007/978-1-4613-8928-6_23.

[6] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 2000. ISBN 978-3-540-45047-4.

[7] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. 01 2001. ISBN 978-0-262-03270-4.

[8] TMT Observatory Corporation. Thirty Meter Telescope SysML model. URL https://github.com/Open-MBEE/TMT-SysML-Model. Last accessed on 2023-10-31.

[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, 1992.

[10] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22, 1957. DOI: 10.2307/2963594.

[11] Zhao Duan, Cong Tian, and Zhenhua Duan. Verifying temporal properties of c programs via lazy abstraction. 2017. ISBN 978-3-319-68689-9. DOI: 10.1007/978-3-319-68690-5_8.

[12] FTSRG. Theta framework GitHub repository. https://github.com/ftsrg/theta, 2023. Accessed: 2023-10-31.

[13] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. 1996. ISBN 978-0-387-34892-6. DOI: 10.1007/978-0-387-34892-6_1.

[14] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, 1997. ISBN 978-3-540-69195-2.

[15] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64, 2020. ISSN 1573-0670. DOI: 10.1007/s10817-019-09535-x.

[16] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. volume 9688 of *Lecture Notes in Computer Science*, 2016. DOI: 10.1007/978-3-319-39570-8_11.

[17] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23, 1997. ISSN 0098-5589. DOI: 10.1109/32.588521.

[18] Benedek Horváth, Bence Graics, Ákos Hajdu, Zoltán Micskei, Vince Molnár, István Ráth, Luigi Andolfato, Ivan Gomes, and Robert Karban. Model checking as a service: Towards pragmatic hidden formal methods. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, 2020. ISBN 9781450381352. DOI: 10.1145/3417990.3421407.

[19] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. Owl: A library for ω-words, automata, and LTL. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, 2018. DOI: 10.1007/978-3-030-01090-4_34.

[20] Vince Molnár, András Vörös, Dániel Darvas, Tamás Bartha, and István Majzik. Component-wise incremental ltl model checking. *Form. Asp. Comput.*, 28, 2016. ISSN 0934-5043. DOI: 10.1007/s00165-015-0347-x.

[21] Milán Mondok. Formal verification of engineering models via extended symbolic transition systems, 2020. Bachelor's Thesis, Budapest University of Technology and Economics.

[22] Milán Mondok and András Vörös. Abstraction-based model checking of linear temporal properties. In *Proceedings of the 27th PhD Mini-Symposium*, 2020.

[23] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.

Appendix

A.1 Acronyms

FOL First Order Logic . . . 4
AP Atomic Proposition . . . 17
CFA Control Flow Automaton . . . 8
LTL Linear Temporal Logic . . . 9
STS Symbolic Transition System . . . 5
XSTS eXtended Symbolic Transition System . . . 7
NDBA Non-Deterministic Büchi Automaton . . . 10
CEGAR CounterExample-Guided Abstraction Refinement . . . 14
NNF Negational Normal Form . . . 12
DFS Depth-First Search . . . 21
NDFS Nested Depth-First Search . . . 17
GDFS Guided Depth-First Search . . . 32
SMT Satisfiability Modulo Theories . . . 16
HOAF Hanoi Omega-Automata Format . . . 28

A.2 Reachability performance of configurations

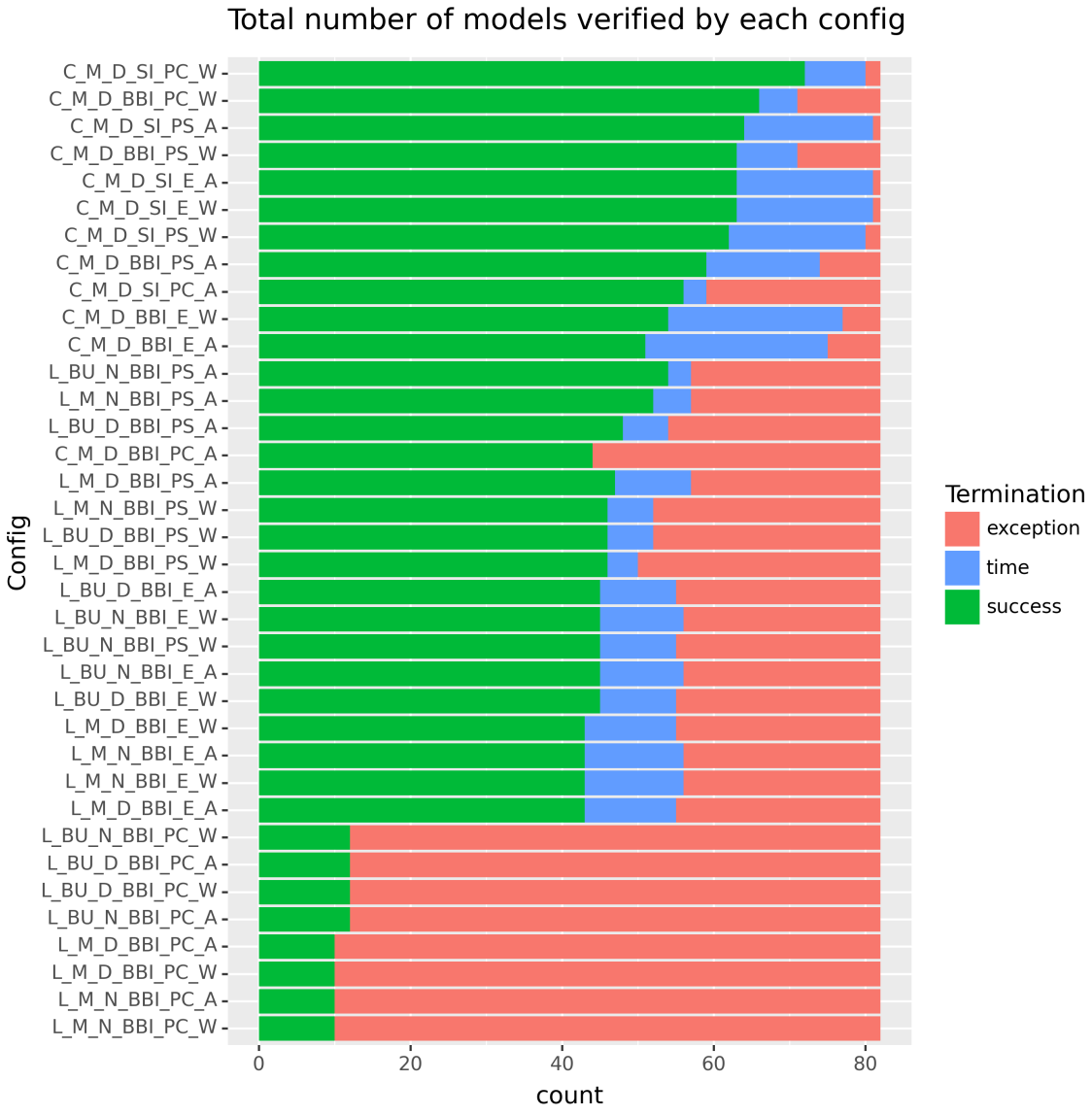


Figure A.2.1: The performance of the different configurations on reachability problems