MŰEGYETEM 1782

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

Extended symbolic transition systems: an intermediate language for the formal verification of engineering models

Scientific Students' Association Report

Author:

Milán Mondok

Advisor:

Ákos Hajdu

2020

Contents

Kivonat

A modellvezérelt fejlesztési folyamatban a formális verifikáció korai visszacsatolást tud adni a fejlesztés alatt álló rendszer helyességéről. A formális módszerek gyakorlati alkalmazását azonban számos akadály hátráltatja. Egyrészt a mérnöki modellek általában magasabb szintű modellezési nyelveken vannak megfogalmazva, míg a formális módszerek alacsony szintű matematikai formalizmusokon képesek működni. Másrészt a verifikációs algoritmusok komoly erőforrásigénnyel rendelkeznek, főleg a komplexebb mérnöki modellek esetében. A Theta egy általános, konfigurálható verifikációs keretrendszer, ami ezeket a kihívásokat különböző alacsony szintű formalizmusok és hatékony, absztrakcióalapú algoritmusok segítségével igyekszik leküzdeni. A létező formalizmusok azonban általánosságban vagy túlságosan alacsony szintűek vagy túlságosan domén specifikusak a modellvezérelt fejlesztéshez.

Ebben a dolgozatban bemutatok egy új köztes formalizmust, a kiterjesztett szimbolikus tranzíciós rendszereket (eXtended Symbolic Transition System, XSTS). Az XSTS formalizmus magasabb szintű nyelvi elemeket, illetve egy szöveges reprezentációt kínál a mérnöki modellek könnyebb transzformációja érdekében. Ezek mellett tiszta és jól definiált szemantikával rendelkezik különböző absztrakt domének felett és alkalmazkodik a létező verifikációs algoritmusok interfészéhez. Továbbá XSTS specifikus algoritmikus kiegészítéseket és stratégiákat is megalkottam a teljesítmény javítása érdekében.

A munkám integrálásra került a Gamma modellező keretrendszerbe, lehetővé téve, hogy megközelítésemet ipari partnerek által biztosított valós példákon szisztematikusan kiértékeljem. Az eredmények rávilágítanak a különböző algoritmuskonfigurációk erősségeire és gyengeségeire, és igazolják az XSTS formalizmus alkalmazhatóságát és hatékonyságát.

Abstract

In a model-driven development workflow, formal verification can give early feedback on the correctness of the system under development. However, formal methods face various challenges in practice. First, engineering models are typically developed in higher-level modeling languages, whereas formal methods usually operate on low-level mathematical formalisms. Second, verification algorithms are resource-intensive, especially on complex engineering models. Theta is a generic and configurable verification framework that aims to tackle these challenges by providing different low-level formalisms and efficient, abstraction-based algorithms. However, existing formalisms are either too low-level or domain-specific for model-driven development in general.

In this work I propose a novel intermediate representation, the eXtended Symbolic Transition System (XSTS) formalism. The XSTS formalism offers higher-level constructs and a textual domain-specific language for easier translation from engineering models. In the meantime, it also has clear and well-defined semantics under different abstract domains, and adapts a standard interpreter interface towards existing verification algorithms. Furthermore, I developed XSTS-specific extensions and strategies that can improve the performance.

My work was integrated into the Gamma Statechart Composition Framework, allowing me to perform an experimental evaluation on use cases provided by industrial partners. I evaluated the strengths and weaknesses of the different algorithm configurations and the results confirmed that the XSTS formalism is both effective and efficient.

Chapter 1

Introduction

Model-driven development allows early evaluation and feedback about different properties of the system under development. This is especially important for critical domains (e.g., transportation, industrial controllers) as an error can lead to serious damages. Formal verification can detect errors or rigorously prove correctness with respect to interesting properties, typically described by assertions or the reachability of an erroneous state. When applied in early phases of development, this can lead to increased confidence in the system as well as reduced implementation costs. Engineering models are typically developed in higher level modeling languages, such as hierarchical statecharts. In contrast, formal methods are available on low-level mathematical formalisms with clear semantics, e.g., logical formulas or automata. Bridging this gap is a key to applying verification for real-world engineering problems.

Formal verification tasks are computationally hard and usually come with heavy time and memory consumption. The systematic exploration of all possible states often makes the verification of even simple models impossible in practice (often termed “state space explosion”). Abstraction-based methods, such as CEGAR (counterexample-guided abstraction refinement) [14] tackle state space explosion by performing the verification task on simpler, abstract models, which are constructed by leaving out unnecessary details about the behavior of the system. The appropriate precision of abstraction is automatically reached with iterative refinements. Theta [42] is a generic and configurable model checking framework, which supports different low-level models using CEGAR-based algorithms. However, the existing formalisms are not sufficient for high-level engineering application. Symbolic transition systems (STS) [23] are too low-level and control-flow automata (CFA) [22] are too software specific.

In this work I propose the novel XSTS - eXtended Symbolic Transition System - formalism for the Theta framework. The new XSTS formalism serves as an intermediate language between low-level logic solvers and high-level engineering models. I define their formal semantics to bridge the semantic gap between efficient symbolic model checkers and high-level engineering models. I implemented the new formalism in Theta, along with a textual domain-specific language for easy model parsing. I adapted the existing abstraction-based algorithms of Theta to the new XSTS formalism and extended the framework with additional core components required for complete integration. Furthermore, I extended the algorithms and introduced algorithmic improvements to exploit the constructs of the XSTS language. My contributions form an integral part of the framework since its v2.0.0 release.

In addition, I also cooperated with the developers of the Gamma Statechart Composition Framework [38] who implemented a translation from their statechart language to the XSTS formalism. This way we achieved an automated, end-to-end verification workflow between the two tools, completely hiding the details of formal methods from the end-user. I demonstrated the applicability of my formalism and algorithms on real-world examples provided by industrial partners. I conducted a systematic and exhaustive benchmarking campaign with numerous different input models and algorithm configurations to identify the strengths and weaknesses of the approach. Results confirm that the XSTS formalism and the related algorithms are both effective and efficient.

Chapter 2

Background

In this chapter I address the theoretical foundations of this work. In Section 2.1 I introduce *model checking*, which is a mathematically rigorous verification method. After this I present the relevant *modeling formalisms* in Section 2.2. *Symbolic transition systems* (Section 2.2.1) offer a compact way of representing transition systems, while *statecharts* (Section 2.2.2) are a high-level formalism that can conveniently model reactive systems. Section 2.3 presents *abstraction*-based model checking and an efficient model checking algorithm based on abstraction, namely *counterexample-guided abstraction refinement* (CEGAR). Section 2.4 briefly presents Theta, the model checking framework I integrated my work into, while Section 2.5 contains a short introduction to the related work in this area.

In my work, I will be describing states and transitions of a system using first-order logic (FOL) formulas. I assume that there is an SMT (satisfiability modulo theories) solver [11] available (such as [18]) that can answer various queries, e.g., is a formula satisfiable, does a formula imply an other one, etc. I use the following notation [22] from first-order logic (FOL) throughout my work. Given a set of variables $V = \{v_1, v_2, ...\}$ let $V' = \{v'_1, v'_2, ...\}$ and $V^{\langle i \rangle} = \{v_1^{\langle i \rangle}, v_2^{\langle i \rangle}, ...\}$ represent the primed and indexed version of the variables. I use V' to refer to successor states, i.e. the values of the variables in the successor state of a transition. I use $V^{\langle i \rangle}$ for paths, in each state of a path the variables appear with a different index. Given an expression φ over $V \cup V'$, let $\varphi^{\langle i \rangle}$ denote the indexed expression obtained by replacing V and V' with $V^{\langle i \rangle}$ and $V^{\langle i+1 \rangle}$ respectively in φ. For example if φ is $x' = x + 1$, then $\varphi^{\langle 5 \rangle}$ is $x_6 = x_5 + 1$. Given an expression φ let $var(\varphi)$ denote the set of variables appearing in φ, e.g., $var(x < y + 2) = \{x, y\}$.

2.1 Model checking

Formal verification consists of numerous methods for proving the correctness of systems with mathematical certainty, one of which is *model checking* [16]. It aims to exhaustively analyze all possible behaviours (states and transitions, i.e., the state space) of a system to check if it meets a given correctness specification. Formally, the purpose of model checking is the following: given a model M and a specification φ, determine whether or not the behavior of M meets the specification φ. This is illustrated by Fig. 2.1. The result of model checking can be either *safe* if the specification holds, or *unsafe* (and a *counterexample*) if it does not.

One of the biggest challenges of model checking is dealing with the problem of *state space explosion* [15]. As the number of state variables in a system increases, the size of

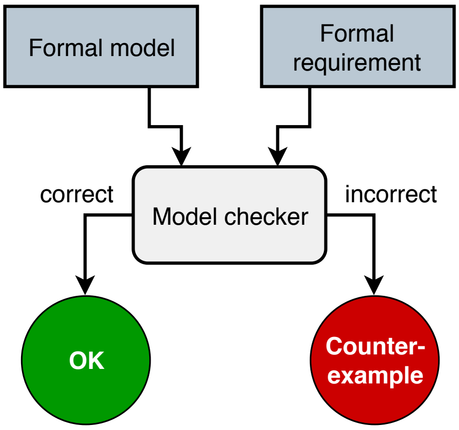


Figure 2.1: An illustration of model checking.

the system's state space grows exponentially (or even at a higher rate), which makes its explicit exploration impossible in practice. Various approaches have been developed to tackle this problem, including *bounded model checking* [12], *symbolic model checking* [13] and *abstraction* [14].

The target of model checking can be different kinds of properties. In this thesis I focus on *reachability analysis*, in which case the *safety* property φ describes a logical invariant that a correct model cannot violate. For example if the property φ is $x > 0$, then the model checking task is to decide whether a state of the system is reachable, where $x > 0$ does not hold. A counterexample in this case would be a path starting in the initial state and ending in said erroneous state.

2.2 Modeling formalisms

In order to be able to reason about the correctness of a system using formal verification techniques, the system needs to be modeled with mathematical precision [37]. A model can be called formal if it has a well-defined syntax and precise semantics, usually relying on mathematical concepts.

Choosing a suitable level of abstraction for the model is a crucial question when designing a formal verification method. The lowest-level formalisms used in model checking are *Kripke structures* [31], which are directed graphs with their states labeled. In Kripke structures, all states and transitions of a system are explicitly represented in a graph. They aren't generally used to model systems directly, but rather as a mathematical formalization of the state space of the verified system.

Model checkers usually operate on intermediate-level languages like *control flow automata* (CFA) [7], *symbolic transition systems* (STS) [23], or the *extended symbolic transition system* formalism presented in this work. These models use varying levels of abstraction and different constructs like FOL formulas or statements to encode the possible states of the system in a compact, symbolic manner, while being low-level enough to be easily verifiable using satisfiability modulo theory (SMT) [3] solvers. These formalisms are used as inputs of model checkers, but generally not as primary development languages.

High-level models allow engineers to model more efficiently by offering high-level constructs to abstract away less important details. These formalisms are often domain-specific in order to allow engineers and developers to use the language most suited for their domain.

Some examples of these models are *programming languages* (e.g., C) or *statecharts* [25], which are widely used by engineers. These higher level models can be transformed to lower-level ones, enabling efficient lower-level algorithms to be used when verifying them.

2.2.1 Symbolic transition systems

Symbolic transition systems (STS) [23] offer a compact way of representing the set of states, transitions and initial states using variables and FOL formulas. This formalism provides a solid ground for the introduction of the CEGAR algorithm in Section 2.3.

Definition 1 (Symbolic transition system). A *symbolic transition system* is a tuple $STS = (V, Tran, Init)$, where:

- $V = \{v_1, v_2, ..., v_n\}$ is the set of variables with domains $D_{v_1}, D_{v_2}, ..., D_{v_n}$;
- *Tran* is a FOL formula over $V \cup V'$, representing the *transition relation*;
- *Init* is a FOL formula over V, representing the *initial states.* ▪

A concrete state $c \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$ is an assignment of the variables. Thus, c can also be denoted by enumerating the values of the variables, i.e., $c = (d_1, d_2, \ldots, d_n)$, where $d_i \in D_{v_i}$. In this work I focus on Boolean and integer variables, but the presented approaches can work on any type as long as the underlying SMT solver supports it. Given a FOL formula φ over the variables V let $c \models \varphi$ denote, that assigning the variables in φ with the values of c evaluates to true. Analogously, let $c \not\models \varphi$ denote that φ evaluates to false.

The state space (C, R, I) of a symbolic transition system can be obtained from a symbolic transition system $STS = (V, Tran, Init)$ in the following way:

- The set of concrete states C is defined by the domains $D_{v_1}, D_{v_2}, ..., D_{v_n}$ in the following way: $C \subseteq D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$. Informally, C is the set of all possible assignments of the variables, e.g., if $V = \{x, y\}$ then a possible state c is $(x = 0, y = 1)$, or in short, $(0, 1)$.
- The set of transitions R is defined by *Tran*. In the transition formula, variables have a non-primed $(v_1, v_2, ..., v_n)$ and a primed $(v'_1, v'_2, ..., v'_n)$ version, corresponding to the actual and successor states respectively. R is then defined in the following way: $R = \{(c, c') \in C \times C \mid (c, c') \models Tran\}$. Informally, c' is a successor of c if assigning values from c to the non-primed variables and values from c' to the primed variables evaluates to true. For example if *Tran* is $x' = x + 1 \wedge y' = y$ then $(x = 1, y = 0)$ is a successor of $(x = 0, y = 0)$.
- Finally, the set of initial states I is defined by *Init* in the following way: $I = \{c \in C \mid c \models Init\}$. Informally, I is the subset of C for which the initial formula holds. For example if $V = \{x, y\}$, and the initial formula *Init* is $x = y$, then I contains all states in which x equals y, i.e. $(x = 1, y = 1), (x = 2, y = 2), (x = 3, y = 3), \ldots$.

2.2.2 Statecharts

Reactive systems appear everywhere in our daily lives: in operating systems, avionics systems, ATMs or even microwave ovens. As these examples show, reactive systems frequently appear in areas, where safety-critical operation is crucial, as even the slightest

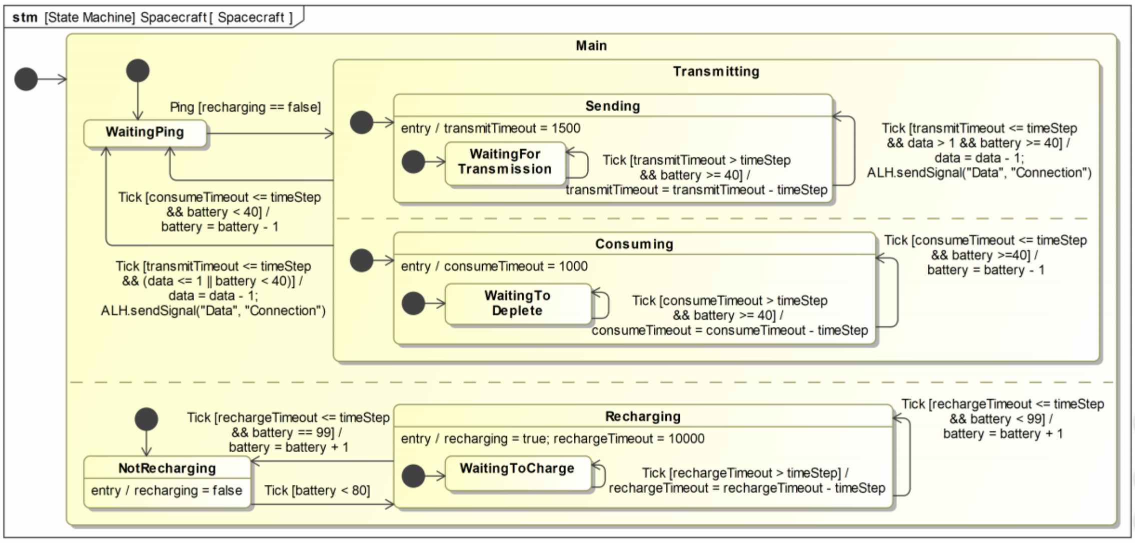

stm [State Machine] Spacecraft[Spacecraft]
Main
Transmitting
Ping [recharging == false]
WaitingPing
Sending
entry / transmitTimeout = 1500
WaitingFor Transmission
Tick [transmitTimeout > timeStep && battery >= 40] / transmitTimeout = transmitTimeout - timeStep
Tick [transmitTimeout <= timeStep && data > 1 && battery >= 40] / data = data - 1; ALH.sendSignal("Data", "Connection")
Tick [consumeTimeout <= timeStep && battery < 40] / battery = battery - 1
Tick [transmitTimeout <= timeStep && (data <= 1 || battery < 40)] / data = data - 1; ALH.sendSignal("Data", "Connection")
Consuming
entry / consumeTimeout = 1000
WaitingTo Deplete
Tick [consumeTimeout > timeStep && battery >= 40] / consumeTimeout = consumeTimeout - timeStep
Tick [consumeTimeout <= timeStep && battery >=40] / battery = battery - 1
Tick [rechargeTimeout <= timeStep && battery == 99] / battery = battery + 1
Recharging
entry / recharging = true; rechargeTimeout = 10000
Tick [rechargeTimeout <= timeStep && battery < 99] / battery = battery + 1
NotRecharging
entry / recharging = false
Tick [battery < 80]
WaitingToCharge
Tick [rechargeTimeout > timeStep] / rechargeTimeout = rechargeTimeout - timeStep

Figure 2.2: The statechart model of a spacecraft component from [29].

misbehaviour can have catastrophic consequences. This makes the verification of these systems a crucial part of their design process.

The defining characteristic of reactive systems is their event-driven nature, which means that they continuously receive external stimuli (events), based on which they change their internal state and possibly react with some output [24]. Reactive systems can be verified using model checking techniques if they are represented by mathematically precise models. *Statecharts* [25] are a popular and intuitive language to capture the behaviour of reactive systems [29, 34, 41], and are at the same time formal and rigorous. Statecharts are an extension of finite state machines, introducing hierarchical state-refinement, orthogonality and broadcast communication. These advanced constructs make statecharts easy to use for engineers, but lead to the formal verification process being challenging.

Figure 2.2 from [29] shows the SysML[1] statechart representation of a spacecraft component (a larger version of the same figure is included in Appendix A.1 for better readability). The spacecraft can receive a Ping signal from the ground to start sending data in packets. The data transmission consumes battery power, and if the battery level falls below 80%, the spacecraft has to start recharging. If the battery level falls below 40%, ongoing data transmission is paused until a full recharge. Rectangles represent states and directed edges denote the possible transitions between states. Black dots point to the initial states of their containing regions. In the example, recharging and data transmission are handled in orthogonal regions, this is denoted with the dashed line that separates the two regions. Transitions can be labeled with guard conditions (conditions are surrounded with square brackets) and upon execution can raise events or assign values to variables. States can contain inner states, this is called hierarchical state refinement.

Statechart compositions

Some modeling frameworks (for example the Gamma Statechart Composition Framework [38]) allow users to define networks of communicating statecharts. The users can define interfaces and components implementing the same interfaces can be connected through

[1] The Systems Modeling Language (SysML) [20] is a general-purpose modeling language for systems engineering applications.

event channels. The statechart components can send and receive signals on their connected channels. Components can be reused to build complex composite components. This additional layer of abstraction allows engineers to efficiently model complex distributed systems, but also makes verification more difficult (due to for example the large number of interleavings). Therefore, efficient verification methods are needed, like CEGAR, which is presented next.

2.3 CEGAR

The key idea behind abstraction-based model checking methods is to analyze the behaviour of abstract, simpler models, which contain less information than the original ones, but in exchange can be explored faster. One would think that losing information leads to incorrect analyses. However, both of the methods presented in the following are *over-approximations*, meaning they only add behaviours to the model and do not remove any. When looking for counterexamples, *false positives* can occur, but no *false negatives.* If no counterexample is found in the abstract model, then the original model doesn't contain one either. On the other hand, finding a counterexample in the abstract model isn't enough to be certain about the existence of one in the original model. The verification algorithm has to check if the counterexamples obtained through abstraction are concretizable, i.e., if the paths they describe are reproducible in the original model. If the abstract counterexample is found *spurious*, i.e., it isn't present in the concrete model, then the abstraction has to be refined and the abstract state space has to be explored again. This iterative repetition of abstraction and refinement phases until the precision is sufficient to come to a conclusion regarding the safety of the model is called counterexample-guided abstraction refinement (CEGAR) [14].

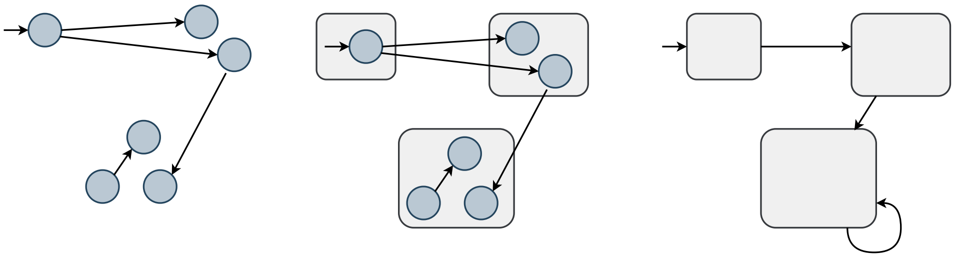


Figure 2.3: An illustration of abstraction.

When analyzing the abstract model, we explore the abstract *state space*, which is a set of *abstract states* and *abstract transitions.* An abstract state can contain multiple (even an infinite number) of concrete states. All concrete states belong to at least one abstract state, and each concrete state can belong to at most one abstract state. An abstract transition is present between two abstract states if there is at least one concrete transition between the contained concrete states. This is illustrated in Fig. 2.3.

2.3.1 Abstraction

I define abstraction based on an *abstract domain* D, a set of *precisions* Π, a *transfer function* T and an *init function* I, using notations from [7, 22].

Definition 2 (Abstract domain). An *abstract domain* is a tuple $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$, where:

- S is a lattice of abstract states;
- $\top \in S$ is the top element;
- $\bot \in S$ is the bottom element;
- $\sqsubseteq \in S \times S$ is a partial order conforming to the lattice. This is used to express that an abstract state covers another abstract state;
- $\mathsf{expr} : S \mapsto FOL$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) using a FOL formula. ▪

By abusing the notation we will allow abstract states $s \in S$ to appear as FOL expressions by automatically replacing them with their meaning, i.e., $\mathsf{expr}(s)$. This will allow us to use cleaner notation for execution paths later.

Elements $\pi \in \Pi$ in the set of precisions define the current precision of the abstraction, which depends on the domain. The transfer function $T : S \times \Pi \mapsto 2^S$ calculates the successors of an abstract state with respect to a target precision. The init function $I : \Pi \mapsto 2^S$ calculates the initial states for the current precision.

The two most well-known abstract domains – namely predicate abstraction and explicit value analysis – tackle the problem of state stace explosion differently, which I introduce in the following.

Predicate abstraction

Predicate abstraction works by only tracking wheter a set of *predicates* (defined by the current precision) evaluate to *true* of *false*. All concrete states where the same predicates hold are represented by a single abstract state. The motivation for this is that in many cases in order to prove the correctness of a system we don't necessarily need to know the precise value of a variable, only whether some predicate applies to it or not. For example in case of a 32 bit integer variable if the property can be decided by knowing whether the value of the variable is positive or not (the predicate $x > 0$), then the state space has to contain only 2 abstract states instead of 2^{32} concrete. These predicates are logical expressions, which can contain variables of the model, for example $(x > 0)$, $(x > y)$ or (true).

In *Boolean predicate abstraction* [2, 22] an abstract state $s \in S$ is an arbitrary Boolean combination (negation, disjunction, conjunction, etc.) of FOL predicates. The top and bottom elements are $\top \equiv \mathit{true}$ and $\bot \equiv \mathit{false}$ respectively. The partial order corresponds to implication, i.e., $s_1 \sqsubseteq s_2$ if $s_1 \Rightarrow s_2$ for $s_1, s_2 \in S$ (s_1 implies s_2). For example if s_1 is $x > 0 \wedge y = 2$ and s_2 is $x > 0$, then $s_1 \sqsubseteq s_2$, i. e. the abstract state s_2 covers s_1. Informally, this means that s_2 represents all the states and behaviors that s_1 does. Therefore, if all outgoing transitions of s_2 were explored, then all subsequent behaviors after s_1 were explored as well. Such queries can be decided by satisfiability modulo theories (SMT) solvers [11]. The expression function is the identity function as abstract states are formulas themselves, i.e., $\mathsf{expr}(s) = s$.

A precision $\pi \in \Pi$ is a set of FOL predicates that are currently tracked by the algorithm. For example $\pi = \{(x > 0), (x > 1)\}$. The result of the transfer function $T(s, \pi)$ is the strongest Boolean combination of predicates in the precision that is entailed by the source state s and the transition formula $Tran$. The result of the init function $I(\pi)$ is the strongest Boolean combination of predicates in the precision that is consistent with the init formula *Init*. This can be calculated by SMT solvers [22].

In *Cartesian predicate abstraction* [2, 22] an abstract state $s \in S$ is a conjunction of FOL predicates, i.e., it is more restricted than Boolean predicate abstraction. Only the transfer function and init function are defined differently than in Boolean predicate abstraction. The transfer function yields the strongest conjunction of predicates from the precision π that is entailed by the source state s and the transition formula *Tran*. The init function yields the strongest conjunction of predicates from the precision that is consistent with the init formula *Init*. Cartesian predicate abstraction is not as precise as Boolean, but it is more efficient [22].

Explicit value abstraction

Explicit value abstraction [5] reduces the state space by marking a subset of the variables *untracked*. For example if our model has the variables x and y, then we might only track x and mark y as *untracked*. This means y can take any value from its domain, which is represented by a single $\top$ value. The motivation behind this domain is that in order to prove a property, not all variables need to be known. The goal when using this domain is to find the smallest subset of the variables (via transitive dependencies), which is sufficient to prove or refute the property.

In explicit-value abstraction [5, 22] an abstract state $s \in S$ is an abstract variable assignment, mapping each variable $v \in V$ to an element from its domain extended with top and bottom values, i.e., $D_v \cup \{\top_{d_v}, \bot_{d_v}\}$. The top element $\top$ with $\top(v) = \top_{v_d}$ holds no specific value for any $v \in V$ (i.e., it represents a state where no variables are known). The bottom element $\bot$ with $\bot(v) = \bot_{v_d}$ means that no assignment is possible for any $v \in V$. For example if $V = \{x, y, z\}$, then $(x = 1, y = 2, z = \top)$ is an abstract state that contains all concrete states where x is 1 and y is 2. If the value of z is not relevant for deciding whether the property holds, then this representation allows us to process a significantly smaller state space. The partial order $\sqsubseteq$ is defined as $s_1 \sqsubseteq s_2$ if $s_1(v) = s_2(v)$ or $s_1(v) = \bot_{d_v}$ or $s_2(v) = \top_{v_d}$ for each $v \in V$. Informally, a state covers another state if it defines less or the same variables and the value of the defined variables match. For example, $(1, 0, \top)$ covers $(1, 0, 2)$, but not $(2, 0, \top)$. The expression function is $\mathsf{expr}(s) \equiv \mathit{true}$ if $s = \top$, $\mathsf{expr}(s) \equiv \mathit{false}$ if $s(v) = \bot_{d_v}$ for any $v \in V$, otherwise $\mathsf{expr}(s) \equiv \bigwedge_{v \in V, s(v) \neq \top_{d_v}} v = s(v)$. Informally, the expression function takes the defined values and creates a conjunction of equality expressions. For example if $s = (x = 1, y = 2, z = \top)$, then $\mathsf{expr}(s)$ is $x = 1 \wedge y = 2$.

A precision $\pi \in \Pi$ is a subset of the variables $\pi \subseteq V$ that is currently tracked by the analysis. The transfer function is $T(s, \pi) = \{s' \mid s \wedge \mathit{Tran} \Rightarrow \bigwedge_{v \in \pi, s'(v) \neq \top_{d_v}} v' = s'(v)\}$. Informally, s' is a successor of s regarding the precision π if assigning values in $s \wedge \mathit{Tran}$ from s to the non-primed variables and assigning values from s' to primed variables that are contained by the precision results in a satisfiable formula. The init function is $I(\pi) = \{s \mid \mathit{Init} \Rightarrow \bigwedge_{v \in \pi, s(v) \neq \top_{d_v}} v = s(v)\}$. Informally, s is an initial state if assigning values in *Init* from s to the non-primed variables contained by the precision results in a satisfiable formula. For example if we have the precision $\{x\}$ and the init formula $\mathit{Init} \equiv (x = 0 \wedge y = 1)$, then the init function $I(\{x\})$ returns all states where $x = 0$, without regards to the value of y, i. e. $(x = 0, y = 1), (x = 0, y = 2), (x = 0, y = 3), \ldots$, represented by $(x = 0, y = \top)$, even though the *Init* formula states that $y = 1$.

Product abstraction

Each abstract domain has its strengths and limitations, there is no single domain that fits all scenarios [22, 1]. For example there are models which cannot be verified using explicit-value abstraction because of the amount of nondeterminicity present in their behaviour. On the other hand, predicate abstraction struggles for example with integer variables that get assigned a big portion of their domains as values, because it has to introduce a separate predicate for each assigned value, causing state space exploration to be impossible in practice.

An abstract domain that could leverage the strengths of multiple abstract (sub-)domains could widen the set of verifiable models and lead to improved model checking performance. The approaches of Beyer et al. [8, 9] combine explicit-value abstraction and predicate abstraction in algorithms that choose between the domains dynamically. Bajkai et al. propose an approach in [1] that tracks all variables explicitly by default and changes to predicate abstraction if a variable is assigned a specified number of different values. I'm not going to present either of these approaches in their entirety here, only the relevant parts of them on which I built my solution that I introduce in Section 4.3.

In a product abstraction domain with the explicit-value and predicate abstraction subdomains, the abstract state space is the product of the state spaces of the subdomains [1]: $S = S_E \times S_P$, an abstract state $s = (s_E, s_P) \in S$ is a pair of an explicit state $s_E \in S_E$ and a predicate state $s_P \in S_P$, where S_E and S_P are the set of abstract states according to the explicit-value and predicate abstraction domains, respectively. The partial order is defined as follows: $(s_{E_1}, s_{P_1}) \sqsubseteq (s_{E_2}, s_{P_2})$ if $s_{E_1} \sqsubseteq s_{E_2}$ and $s_{P_1} \sqsubseteq s_{P_2}$. The expression function yields the conjunction of the expression functions of the subdomains, i.e. $\mathsf{expr}((s_E, s_P)) \equiv \mathsf{expr}_E(s_E) \land \mathsf{expr}_P(s_P)$, where expr_E and expr_P are the expression functions of the explicit-value and predicate abstraction domains, respectively. A precision $\pi = (\pi_E, \pi_P) \in \Pi_E \times \Pi_P$ is a pair an explicit-value precision $\pi_E \in \Pi_E$ and predicate abstraction precision $\pi_P \in \Pi_P$, where Π_E and Π_P are precision sets of the explicit-value and predicate abstraction domains, respectively.

The transfer function $T(s, op, \pi)$ uses the transfer functions $T_E(s_E, op, \pi_E)$ and $T_P(s_P, op, \pi_P)$ as black boxes to form a joint transfer function. The transfer function calculates the Cartesian product of the results of the subdomains' transfer functions. Formally, $T((s_E, s_P), op, (\pi_E, \pi_P)) = T_E(s_E, op, \pi_E) \times T_P(s_P, op, \pi_P)$ then removes invalid product states using the so-called *strengthening operator*. This filtering is required because product states obtained through the Cartesian product can be invalid states. For example the product state $((x = 0), (x > 5))$, where the explicit state is $s_E = (x = 0)$ and the predicate state is $s_P = (x > 5)$, is an invalid state, because the semantics of the two substates contradict each other (x cannot be greater than 5 if x is 0).

Abstract reachability graph

The abstract state space can be represented by an *abstract reachability graph* (ARG) [6, 22].

Definition 3 (Abstract reachability graph). An *abstract reachability graph* is a tuple $ARG = (N, E, Cov)$, where:

- $N \subseteq S$ is the set of nodes, each corresponding to an abstract state in some domain;
- $E \subseteq N \times N$ is a set of directed *edges*. An edge $(s_1, s_2) \in E$ is present if s_2 is a successor of s_1 with regards to the transfer function T;

- $Cov \subseteq S \times S$ is the set of covered-by edges. A covered-by edge $(s_1, s_2) \in Cov$ is present if $s_1 \sqsubseteq s_2$. ▪

A node $s \in N$ is *expanded* if all of its successors are included in the ARG with respect to the transfer function and *covered* if it has an outgoing covered-by edge $(s, s') \in Cov$ for some $s \in N$. A node that violates the safety property φ is called *unsafe*. This can be decided by checking if $s \not\models \varphi$ holds. A node that is not expanded, covered or unsafe is called *unmarked*. When building the ARG, expanded and covered nodes can be disregarded, only unmarked nodes have to be processed (and thus expanded, covered or marked unsafe). An ARG is *complete* if no nodes are unmarked.

An abstract path $\sigma = (s_1, s_2, \ldots, s_n)$ is an alternating sequence of abstract states. An abstract path is *feasible* if a corresponding concrete path $(c_1, c_2, \ldots, c_n)$ exists, where each c_i is mapped to s_i , i.e., $c_i \models \mathsf{expr}(s_i)$. In practice, this can be decided by querying an SMT solver [11] with the formula $s_1^{\langle 1 \rangle} \wedge Tran^{\langle 1 \rangle} \wedge s_2^{\langle 2 \rangle} \wedge Tran^{\langle 2 \rangle} \wedge \ldots \wedge Tran^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$. A satisfying assignment to this formula corresponds to a concrete path.

Abstraction algorithm

The input of the abstraction algorithm [22] is a partially constructed ARG (with possibly unmarked states), a FOL safety property φ, an abstract domain D, a current precision π, a transfer function T and an init function I. The result of the algorithm is whether the model satisfies the safety property φ, i.e. all states reachable from the initial state satisfy the formula.

In the first iteration, the ARG only contains the initial states returned by the init function I, the precision π is usually empty (or is constructed from the property φ).

The algorithm (Alg. 1, based on [22]) initializes the reached set with all states from the ARG and the waitlist with all unmarked states. The algorithm removes and processes states from the waitlist based on some search strategy (e.g., BFS or DFS). If the current state does not satisfy the property φ, the abstraction terminates with an unsafe result and an unsafe ARG. Otherwise, we check if some already reached state covers the current with respect to the partial order. If not, we calculate successors with the transfer function making the node expanded. If there are no more nodes to explore and no unsafe states were found, the abstraction concludes with a safe result and a complete ARG.

2.3.2 Refinement

Abstract counterexamples found during the abstraction phase are not necessarily present in the concrete model as well, which means their concretizability has to be verified. Abstract counterexamples that are not present in the concrete model are called *spurious*. The occurrence of a spurious counterexample means the abstraction has to be refined to ensure that the same counterexample isn't found again, and thus the abstraction converges towards a precision which is sufficient to verify the requirement property. The refinement algorithm verifies the satfisfiability of abstract counterexamples and either returns that the model is unsafe (if the counterexample is concretizable), or returns a refined precision and prunes back the ARG to exclude the spurious path described by the counterexample.

The Refinement algorithm (see details in [22]) receives the ARG and the current precision as argument. It constructs an SMT formula from the path described by the counterexample and queries an SMT solver with it. If the solver finds that the formula is satisfiable, then

Algorithm 1 Abstraction algorithm (based on [22])

```
Input: $ARG(N, E, Cov)$: partially constructed abstract reachability graph
       $\varphi$: FOL safety property
       $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$: abstract domain
       $\pi$: current precision
       $T$: transfer function
Output: ($safe$ or $unsafe$, $ARG$)
reached $\leftarrow N$
waitlist $\leftarrow$ unmarked nodes from $N$
while waitlist $\neq \emptyset$ do
    $s \leftarrow$ remove from waitlist
    if $s \not\models \varphi$ then
        // $s$ is unsafe
        return ($unsafe$,$ARG$)
    if $s \sqsubseteq s'$ for some $s' \in$ reached then
        // $s$ is covered
        $Cov \leftarrow Cov \cup \{(s, s')\}$ // Add covered-by edge
    else
        // $s$ is expanded
        for all $s' \in T(s, \pi) \setminus \bot$ do
            reached $\leftarrow$ reached $\cup \{s'\}$
            waitlist $\leftarrow$ waitlist $\cup \{s'\}$
            $N \leftarrow N \cup \{s'\}$ // Add new node
            $E \leftarrow E \cup \{(s, s')\}$ // Add successor edge
return ($safe$,$ARG$)
```

the counterexample is conretizable and the model is unsafe. However, if the formula is unsatisfiable, then the algorithm queries the solver for a Craig interpolant [35], which is a formula that "explains" the reason for the unsatisfiability. This interpolant is then either used as a predicate (in case of predicate abstraction), or its variables are added to the set of tracked variables to construct a new precision. The ARG is then pruned back until the longest satisfiable prefix of the counterexample. The algorithm in this case returns the pruned ARG and the new precision. From the viewpoint of my work the refinement algorithm can be regarded as a black box, if the reader is interested, then [22] is recommended for further reading.

2.3.3 The CEGAR algorithm

Counterexample-guided abstraction refinement (CEGAR) [14] is an abstraction-based model checking algorithm, that applies abstraction and refinement iteratively. The CEGAR-loop (Fig. 2.4, Alg. 2) is the heart of this algorithm. The verification process starts with an initial *precision*. Depending on the kind of abstraction we use, a precision can either be a list of tracked variables (explicit value abstraction), a list of tracked predicates (predicate abstraction), or something else in case of a different abstraction-method. In every iteration of the loop the abstraction algorithm (Alg. 1) checks if an *abstract counterexample* can be found in the abstract model with the current precision. There are two possibilities after this:

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

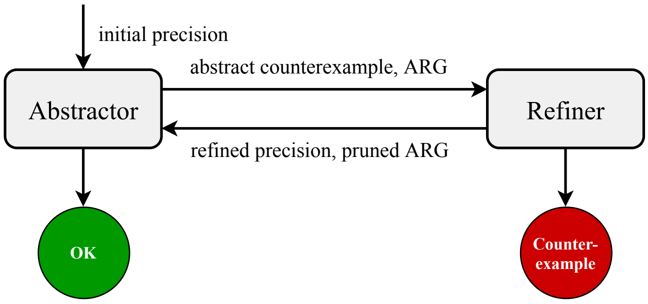


Figure 2.4: The CEGAR-loop.

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

Algorithm 2 CEGAR loop (based on [22])

```
Input: $\varphi$: FOL safety property
       $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$: abstract domain
       $\pi_0$: initial precision
       $T$: transfer function
       $I$: init function
Output: safe or unsafe
$\pi \leftarrow \pi_0$
$ARG \leftarrow (N \leftarrow I(\pi_0), E \leftarrow \emptyset, Cov \leftarrow \emptyset)$
while true do
    $\mathsf{result}, ARG \leftarrow \textsc{Abstraction}(ARG, \varphi, \pi, T)$
    if $\mathsf{result} = \mathit{safe}$ then return safe
    else
        $\mathsf{result}, \pi, ARG \leftarrow \textsc{Refinement}(ARG, \pi)$
        if $\mathsf{result} = \mathit{unsafe}$ then return unsafe
```

2.4 Theta

Theta[2] [42] is a “generic, modular and configurable model checking framework developed at the Fault Tolerant 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.” [21].

	Common	**CFA**	**STS**	**XTA**
Tools		cfa-cli	sts-cli	xta-cli
Analyses	analysis	cfa-analysis	sts-analysis	xta-analysis
Formalisms	core, common	cfa	sts	xta
SMT solvers	solver, solver-z3			

Table 2.1: An overview of the Theta modules.

One of the strengths of Theta is its modular architecture, which is illustrated in Table 2.1. The *core* module contains various general building blocks for model checking, e.g. variable declarations, expressions, statements, which are used in the *control flow automata* (CFA), *symbolic transition system* (STS) and *timed automata* (XTA) formalisms. Each formalism has a corresponding domain-specific language for easy parsing. The analysis backend resides in the *analysis* module and contains constructs for the CEGAR loop, supporting multiple abstract domains such as *predicate abstraction* and *explicit-value abstraction.* Each formalism has a corresponding analysis module that contains formalism-specific concretisations of the CEGAR algorithm. In order to enable the algorithms of the *analysis* module to operate on a specific formalism, formalism-specific implementations have to be provided of the generic interfaces defined in the *analysis* module. These implementations can also provide formalism-specific extensions and optimizations to the algorithm that can make the analysis more efficient by utilizing extra information present in the models. The CEGAR algorithm relies on the SMT solver accessible through the *solver* interface and

[2] `https://github.com/ftsrg/theta`

its implementation in *solver-z3* [18]. Each formalism has a formalism-specific command line interface in its corresponding *cli* module.

2.5 Related work

Czipó Bence et al. present a CEGAR-based statechart verification algorithm in [17] that exploits the hierarchy of statecharts by using clever encoding that preserves information about state hierarchy and enables for SMT-based verification. There are multiple similarities between their approach and mine, such as tracking control variables explicitly, using an SMT solver to verify concretizability and using the CEGAR-algorithm, but the two approaches are fundamentally different in that theirs is specific to statecharts, while mine is a generic formalism that supports other engineering models as well.

Yael Meller et al. present a CEGAR-based statechart verification approach in [36] that operates on UML statechart models. The approach defines an abstract domain that over-approximates the behaviour of interfaces of some (or all) state machines of a system. They abstract part of the interface's variables, which may change the number and order of the generated events on a particular interface. This abstraction method is similar to the one I use in explicit-value abstraction in the sense that it also only tracks a subset of the variables. The statechart-specific nature of this approach enables the verification process to utilize the additional information that is present in the structure of a statechart model, but comes with drawbacks in terms of genericity and flexibility, as it can only be applied to statecharts.

Nafiseh Kahani et al. present an approach for statechart verification in [30] that is based on bounded model checking [12]. Their algorithm takes a depth bound as input, generates all execution paths of the system to the specified bound and encodes them as SMT formulas to check their satisfiability with an SMT solver. Their approach reduces the state space by considering only a subset of the paths of the system that are shorter than a given bound. My approach on the other hand reduces the state space by applying either explicit-value abstraction, predicate abstraction or a combination of the two.

As we can see, several different approaches exist to formal verification of statechart models. However, all approaches presented above are specific to the statechart formalism. It is important to highlight that the XSTS formalism presented in this work is on the other hand a general intermediate model checking formalism that was carefully designed to support the transformation of several high-level engineering formalisms as well as multiple model checking paradigms. In the following paragraphs I give a brief introduction into other intermediate model checking languages.

BoogiePL [19] is a generic intermediate language for program verification with a focus on object oriented languages. The language is coarsely typed and offers constructs such as procedures and arrays. BoogiePL is used as the input formalism of the Boogie model checker. The main distinguishing characteristic of BoogiePL compared to XSTS is that it was specifically tailored to fit programming languages and offers constructs for this specific domain (like the aforementioned procedures).

The Spin [27, 28] model checker is a tool for verifying distributed systems, specifically data communication protocols. The systems are described in the Promela language, which offers high-level constructs such as processes and message channels with synchronous and asynchronous semantics. The Spin model checker can check for the absence of deadlocks, unspecified receptions, unexecutable code and also supports the verification of linear temporal properties.

PVS [40] is a specification and verification language that based on classical, typed higher-order logic and can be used for theorem proving. This is a lower-level approach that can be used to construct and maintain large formalizations and logical proofs.

Chapter 3

Extended symbolic transition systems

The high-level nature of engineering models means they are easy-to-use for engineers, but leads to difficulties during the formal verification process. In case of statecharts for example, orthogonal regions, hierarchical state-refinement or broadcast communication are all high-level constructs that make the modeling workflow more intuitive and enable the modeling of significantly more complex systems. They are however difficult to process using formal methods that are defined on low-level mathematical formalism and verified using SMT solvers. In this work I propose the *extended symbolic transition system* (XSTS) formalism, which aims to bridge the aforementioned gap between engineering models and formal methods.

3.1 Design decisions

When designing XSTS several questions had to be decided. The XSTS is intended as a general formalism that is expressive enough to allow not only statecharts, but other higher-level formalisms (such as Petri nets or control flow automata for example) to be able to be transformed to XSTS. In this work I focus on abstraction-based reachability analysis, but XSTS was designed with multiple model checking properties and methods in mind.

The STS formalism (Section 2.2.2) used FOL expressions to describe the transitions of the system. While these offer great flexibility, they are not necessarily intuitive, which is why higher-level formalisms such as statecharts or programming languages offer higher-level constructs like operations or statements. For example the operation $x := x + 1$ can be much more intuitive for an engineer than the FOL formula $x' = x + 1$. This is why I opted for *operation*-based transitions in XSTS, which are high-level enough to allow for easy understanding, while also being easy to describe using logical formulas, thus allowing for SMT-based model checking. Even though expression-based transitions offer more flexibility, the higher-level nature of XSTS operations allows for the transformation process of engineering models to XSTS to be simpler and easier, while still being able to express the behaviour of the engineering models. To achieve this the traditional list of operations (assign and assume [22]) had to be extended with composite operations such as non-deterministic choices and sequences.

The XSTS formalism includes two separate initialization-related constructs, an initialization formula (*init formula*) described through the *initial value function* and a set of initialization operations (*init transitions*). The former denotes a FOL formula that describes the initial states of the model, the latter denotes a set of operations that can only be invoked once, at the start of the execution. The init formula describes the very first state so that the execution doesn't start with unknown variable values. This is required because else we would start with an infinite state space which would restrain us from utilizing certain model checking methods such as saturation-based model checking [37] and linear temporal logic model checking [39]. As XSTS is intended as a general formalism that can support several model checking paradigms, an init formula can be used in XSTS to describe the very first state. This FOL formula on the other hand is unintuitive and inconvenient for describing complex behaviour, which is why XSTS includes the init transition that can fire only once, at the beginning. The init transition fires from the initial states described by the init formula. The init transitions can include high-level operations, which makes them suitable to describe complex initialization behaviour. For example in case of statecharts the init formula can be used to initialize the variables (for example by assigning 0 to the int variables), while an init transition can set the appropriate state configuration (which might depend on complex logic).

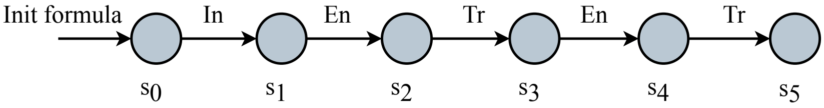


Figure 3.1: An illustration of the ordering of the transitions.

XSTS models have 3 different sets of transitions, namely *Init* (In), *Tran* (Tr) and *Env* (En). We discussed *Init* in the previous paragraph. *Tran* models the internal behaviour of the system and *Env* models the effect of the environment on the system and vice-versa (for example incoming and outgoing events). This can be used to model reactive systems, which constantly react to outer stimuli depending on their internal states. During each execution of the system transitions from the *Tran* and *Env* set strictly follow each other in an alternating manner to ensure that the system reacts to all incoming stimuli. Fig. 3.1 illustrates the order in which the different transition sets follow each other along paths of a system (the grey circles denote states of the path while the labels on the arrows denote single transitions from the denoted sets).

It is important to make a distinction between transitions and transition sets. Transition sets are In, Tr and En, they can contain multiple operations. The behaviour of a single transition is described by a single operation (either basic or composite). When a transition set is executed, then a single operation is chosen from the set non-deterministically and executed. When we say that an In transition is executed in the initial state, what we mean is that an operation is selected from the In set non-deterministically and executed. This means that not all operations of the In transition get executed, only a single one.

3.2 Formal definition

In order to offer mathematical precision, formal verification methods require formally defined models with clear semantics. In this section I present the formal definition of the novel XSTS formalism.

Definition 4 (Extended symbolic transition system). An *Extended symbolic transition system* is a tuple $XSTS = (V, V_C, IV, Tr, In, En)$, where:

- $V = \{v_1, v_2, ..., v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, ..., D_{v_n}$. For example $V = \{x, y\}$. In this work I use the *integer*, *bool* and *enum* domains (integers are mathematical integers[1], bool and enum domains are equivalent to the corresponding types in common programming languages);

- $V_C \subseteq V$ is a set of variables marked as *control variables.* For example $V_C = \{x\}$. The set of control variables is used for algorithmic optimizations, see Section 4.3;

- $IV \in D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$ is the *initial value function* used to describe the initial state. The initial value function IV assigns an initial value $IV(v) \in D_v$ to variables $v \in V$ of their domain D_v. For example $IV(x) = 0$, $IV(y) = 1$. It is also possible to omit the initial value of a variable, in that case $IV(v) = \top$. The initial value function is used to construct the *init formula*, a FOL formula that characterises the initial states. The init formula IF is defined as follows: $IF \equiv \bigwedge_{v \in IV, IV(v) \neq \top} v = IV(v)$.

In case of STS the transition relation was described with a single FOL formula. While this offered great flexibility, the complex behaviour of engineering models was hard to translate to such a low-level formalism. XSTS uses high-level, expressive operations to describe behaviour and divides transitions into 3 different sets with different semantics:

- $Tr \subseteq Ops$ is a set of operations, representing the *internal transition relation.* This set describes the internal behaviour of the system;

- $In \subseteq Ops$ is a set of operations, representing the *initialization transition relation.* This set is used to describe more complex initialization that cannot be described using the initialization vector. This set is executed once and only once, at the very beginning.

- $En \subseteq Ops$ is a set of operations, representing the *environmental transition relation.* This set is used to model the system's interactions with its environment. ▪

In any state of the system a single operation is executed, which is selected from the sets we introduced above (Tr, In and En). The set from where the operation can be selected depends on the current state. In the initial state (which is described by the initialization vector IV) only operations from the In set can be executed. Operations from the In set can only fire in the initial state and nowhere else. After that, En and Tr can be fired in an alternating manner (Fig. 3.1).

Operations $op \in Ops$ describe the transitions between states of the system, where Ops is the set of all possible transitions. An operation $op \in Ops$ can also be regarded as a transition formula $tran(op)$ defining its semantics. Transition formulas are interpreted over $V \cup V'$. In other words transition formulas $tran(op)$ are logical formulas that contain the non-primed and primed versions of the variables of the XSTS and describe the values of the variables after the execution of the operation. All operations are atomic in the sense that they are either executed in their entirety or none at all. XSTS defines the following operations:

- *Basic* operations contain no inner (nested) operations. We define the following basic operations:

[1] Mathematical here means that these integer variables are unbounded, i.e. unlike "machine integers" they cannot overflow, there are no unsigned variables, etc.

- *Assignments* assign a value to a single variable. Assignments have the form $v := \varphi$, where $v \in V$, φ is an expression of type D_v and $var(\varphi) \subseteq V$. For example the operation $x := y + 2$ assigns the value $y + 2$ to the variable x. For an assignment operation, the transition formula is defined as $tran(v := \varphi) \equiv v' = \varphi \wedge \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$. Informally, this means that the value of v is overwritten with φ, while all other variables remain the same in the next state. For example if $V = \{x, y\}$, then $tran(x := y + 2)$ is $x' = y + 2 \wedge y' = y$;
- *Assumptions* check a condition and their transition can only be taken if their condition evaluates to *true*. Assumptions have the form $[\psi]$, where ψ is a predicate with $var(\psi) \subseteq V$. For example the operation $[x > 0]$ can only be executed if the value of x is greater than 0. For an assume operation the transition formula is $tran([\psi]) \equiv \psi \bigwedge_{v \in V} v' = v$. Informally, this means that the condition ψ is checked, but all variables remain the same in the next state. For example if $V = \{x, y\}$, then $tran([x > 0])$ is $x > 0 \wedge x' = x \wedge y' = y$.
- *Havocs* are non-deterministic assignments of the form $havoc(v)$, where $v \in V$. The havoc operation $havoc(x)$ means that the resulting state will contain no restrictions on the value of the variable x. This can be used to simulate unknown user input or a nondeterministic value. The transition formula is defined as $tran(havoc(v)) \equiv \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$, i.e. all other variables will keep their value, and v can hold any value. For example if $V = \{x, y\}$, then $tran(havoc(x))$ is $y' = y$.

- *Composite* operations contain other operations, and can be used to describe complex control stuctures. Note that while these are composite operations, their execution is still atomic. XSTS defines the following composite operations:
  - *Sequences* are essentially multiple operations executed after each other. Each operation of the sequence operates on the result of the previous operation, Sequences have the form $op_1, op_2, ..., op_n$, where $op_i \in Ops$ are basic or composite operations, For example: $x := 2, [x > 0], x := x + 1$ is a sequence containing 3 operations;
  - *Choices* model non-deterministic choices between multiple operations. One and only one branch of the choice operation is selected for execution and only if said choice executes in its entirety, i.e., a single failing assume operation means that a branch can not be executed. Non-deterministic choices have the form $\{op_1\}$ *or* $\{op_2\}$ *or* ... *or* $\{op_n\}$, where $op_i \in Ops$ are basic or composite operations.

By abusing the notation, we allow operations $op \in Ops$ to appear as FOL expressions by automatically replacing them with their semantics, i.e., $tran(op)$. I already defined the semantics of the basic operations above, I define the semantics of composite operations in the following paragraphs.

Sequences are operations executed after each other, each operation using the result of the previous operation. To represent the results of the inner suboperations in the transition formula, let V^* denote the starred versions of the variables. Let $star(V, n)$ denote the set of variables we get by applying the star operator to each variable $v \in V$ n times and let $star(\varphi, n)$ denote the FOL expression we get by replacing the primed versions of the variables with their starred versions and then applying the star operator to each variable $v \in var(\varphi)$ in φ (both primed and unprimed versions) n additional times. For example if $V = \{x, y\}$, then $V^* = \{x^*, y^*\}$, and if $\varphi \equiv x' = x + 1$, then $star(\varphi, 0) \equiv x^* = x + 1$, and

$star(\varphi, 2) \equiv x^{***} = x^{**} + 1$. For a sequence operation we define the transition formula over $V \cup V'$ as $tran(op_1, op_2, ..., op_n) \equiv \bigwedge_{v \in V} v^* = v \wedge \bigwedge_{i=1}^{n} star(tran(op_i), i) \wedge \bigwedge_{v \in V} v' = star(v^*, n)$. This formula consists of 3 parts:

- In the first part, $\bigwedge_{v \in V} v^* = v$, we store the values of the non-starred variables (corresponding to the source state) in the starred versions of the variables;
- In the second part, $\bigwedge_{i=1}^{n} star(tran(op_i), i)$, we form the conjunction of the transition formulas of the suboperations, each having the star operator applied to it once more than the previous one, so that in each operation v will refer to v' of the previous operation;
- In the third part, $\bigwedge_{v \in V} v' = star(v^*, n)$, we store the successor values of the last transition formula in the primed versions of the variables, so that v' will have the value of v' from the last transition formula.

To demonstrate the transition formula of sequences, let's consider the sequence $x := 2, [x > 0], x := x + 1$ with $V = \{x\}$. The three parts will be the following:

- First part: $x^* = x$;
- The second part: $x^{**} = 2 \wedge (x^{**} > 0 \wedge x^{***} = x^{**}) \wedge x^{****} = x^{***} + 1$;
- The third part: $x' = x^{****}$.

All three parts together:

$$x^* = x \wedge x^{**} = 2 \wedge (x^{**} > 0 \wedge x^{***} = x^{**}) \wedge x^{****} = x^{***} + 1 \wedge x' = x^{****}.$$

For a non-deterministic choice operation of the form $\{op_1\}$ *or* $\{op_2\}$ *or ... or* $\{op_n\}$, the transition formula over $V \cup V'$ is defined as follows: $tran(\{op_1\}\ or\ \{op_2\}\ or\ ...\ or\ \{op_n\}) \equiv \bigvee_{i=1}^{n}(temp = i \wedge tran(op_i))$, where *temp* is a temporary variable that is used to express that only one branch of the choice operation is executed. To demonstrate this on an example, let's consider the choice operation $\{x := 2\}$ *or* $\{[x > 0]\}$ *or* $\{x := x + 1\}$. The transition formula will consist of three parts, one for each suboperation:

- $temp = 1 \wedge x' = 2$ for $x := 2$;
- $temp = 2 \wedge (x > 0 \wedge x' = x)$ for $[x > 0]$;
- $temp = 3 \wedge x' = x + 1$ for $x := x + 1$.

All three together:

$$temp = 1 \wedge x' = 2 \vee temp = 2 \wedge (x > 0 \wedge x' = x) \vee temp = 3 \wedge x' = x + 1.$$

During any execution the *temp* variable is assigned a single value (non-deterministically by the SMT solver), ensuring that only one branch of the choice is executed.

3.2.1 State space of XSTS

In the preceding section I formally defined what an XSTS model consists of. In this section I precisely define how the state space of an XSTS is constructed. I present among other things how a state is defined, what the possible transitions are, what a path is. The precise definition of these constructs is crucial for model checking.

A *concrete data state* $c \in D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$ assigns a value $c(v) = d \in D_v$ to each variable $v \in V$ of its domain D_v. For example if $V = \{x, y\}$, then $(x = 1, y = 2)$ is a possible concrete data state. States with a prime (c') or an index ($c^{\langle i \rangle}$) assign values to V' or $V^{\langle i \rangle}$ respectively. A *concrete state* (c, i, e) is a 3-tuple, where $c \in D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$ is a concrete data state, $i \in \{\textsc{Ini}, \textsc{Un}\}$ is a flag denoting whether the initialization transition has been executed yet (Ini and Un referring to 'initialized' and 'uninitialized', respectively), and $e \in \{\textsc{Lt}, \textsc{Le}\}$ is a flag denoting whether the last executed transition was from the Tr or the En set (Lt and Le standing for 'last Tr' and 'last En', respectively). The set of initial states is $\{\ (c, i, e) \mid c \models \mathit{IF}\ \ \land i = \textsc{Un} \land e = \textsc{Le}\}$. In other words, the initial state of the system satisfies the init formula and has the Un and the Le flags. As we will see in the next paragraph, the intention behind these flags is to limit the possible transitions of the system and to ensure that the different transition sets follow each other in the desired order (which is illustrated in Fig. 3.1).

The set of available transitions in any state depends on the values of the i and e flags. For a pair of source flags i and e and target flags i' and e' let $\mathit{Ops}(i, e, i', e')$ denote the set of available transitions. The value of $\mathit{Ops}(i, e, i', e')$ is defined as follows:

- If the source flags are $(\textsc{Un}, \textsc{Le})$ and target flags are $(\textsc{Ini}, \textsc{Lt})$, then $\mathit{Ops}(i, e, i', e') = In$. Informally, In transitions can fire in states where $i = \textsc{Un}$ and $e = \textsc{Le}$ and result in a state where $i' = \textsc{Ini}$ and $e' = \textsc{Lt}$. Setting the e flag to Lt in the target state ensures that the first transition after the In transition will be an environmental transition. This is important because else the system would ignore external stimuli that is received right after the initialization.

- For the source flags $(\textsc{Ini}, \textsc{Lt})$ and target flags $(\textsc{Ini}, \textsc{Le})$, $\mathit{Ops}(i, e, i', e') = \mathit{En}$. Informally, if the source e flag is Lt, then En transitions can fire and will result in a state with the Le flag.

- For the source flags $(\textsc{Ini}, \textsc{Le})$ and target flags $(\textsc{Ini}, \textsc{Lt})$, $\mathit{Ops}(i, e, i', e') = Tr$. Informally, if the source e flag is Le, then Tr transitions can fire and will result in a state with the Lt flag.

- For any other value of (i, e, i', e'), $\mathit{Ops}(i, e, i', e') = \emptyset$, i.e. no transition is possible. Note that no transitions exist that start from a state with the Ini flag and result in a state with the Un flag, which ultimately means that In transitions can fire once and only once.

A transition exists between two states (c, i, e) and (c', i', e') if an operation $\mathit{op} \in \mathit{Ops}(i, e, i', e')$ exists with $(c, c') \models \mathit{op}$, i.e., c and c' satisfy the semantics of op. To summarize, any path in the system starts with an operation from the initialization set and continues with the alternation of environmental and internal operations $(In, \mathit{En}, Tr, \mathit{En}, Tr, \mathit{En}, Tr, \ldots)$ as illustrated in Fig. 3.1.

A *concrete path* is a finite, alternating sequence of concrete states and operations $\sigma = ((c_1, i_1, e_1), \mathit{op}_1, \ldots, \mathit{op}_{n-1}, (c_n, i_n, e_n))$ if the transitions exist between (c_i, i_i, e_i) and $(c_{i+1}, i_{i+1}, e_{i+1})$ for every $1 \leq i < n$ and $(c^{\langle 1 \rangle}, c^{\langle 2 \rangle}, \ldots, c^{\langle n \rangle}) \models \bigwedge_{1 \leq i < n} \mathit{op}_i^{\langle i \rangle}$, i.e. there is a sequence of transitions starting from the initial state and the interpretations satisfy the semantics of the operations. A concrete state (c, i, e) is *reachable* if a concrete path $\sigma = ((c_1, i_1, e_1), \mathit{op}_1, \ldots, \mathit{op}_{n-1}, (c_n, i_n, e_n))$ with $c = c_n$, $i = i_n$ and $e = e_n$ for some n.

3.3 Domain-specific language

In order to make the transformation process of higher-level models easier I created a domain-specific language (DSL) for the XSTS formalism which allows for the easy definition of XSTS models.

3.3.1 Language constructs

In this section I give a brief introduction into the syntax of the basic constructs of the XSTS DSL.

Types

XSTS contains two default variable types, logical variables (*boolean*) and mathematical integers (*integer*). XSTS also allows the user to define *custom types*, similarly to enum types in common programming languages.

A custom type can be declared the following way:

```
type <name> : { <literal>, . . . , <literal> }
```

Example:

```
type Color : { RED, GREEN, BLUE }
```

Variables

Variables can be declared the following way, where `<value>` denotes the value that will be assigned to the variable in the initialization vector:

```
var <name> : <type> = <value>
```

If the user wishes to declare a variable without an initial value, this is possible as well:

```
var <name> : <type>
```

A variable can be tagged as a control variable with the keyword `ctrl`:

```
ctrl var <name> : <type>
```

Examples:

```
var a : integer
var b : boolean = false
var c : Color = RED
ctrl var x : integer = 0
```

Expressions

Expressions φ of the XSTS DSL can be described with the following context-free grammar, where $n \in \mathbb{Z}$ is an integer and $v \in V$ is a variable:

$$\varphi ::= \top | \bot | v | n | \neg\varphi | [\varphi \land \varphi] | [\varphi \lor \varphi] | [\varphi \Rightarrow \varphi] | [\varphi > \varphi] | [\varphi < \varphi] | [\varphi \geq \varphi] | \\ [\varphi \leq \varphi] | [\varphi = \varphi] | [\varphi + \varphi] | [\varphi - \varphi] || [\varphi * \varphi] | [\varphi / \varphi] | [\varphi \% \varphi] | (\varphi)$$

Operator precedence can be seen below with a short description of each operator:

Lower	Operators	Description
↓	$\Rightarrow$	implies
	$\lor$	logical or
	$\land$	logical and
	$\neg$	logical negation
	$=$	equals
	$>, <, \leq, \geq$	relational operators
	$+, -$	addition, subtraction
	$*, /, \%$	multiplication, division, modulo
	$v, n, \top, \bot$	variable, integer literal, top, bottom
	$()$	parentheses
Higher		

Table 3.1: Operator precedence.

Operations

The behaviour of XSTS can be described using basic and composite operations. Basic operations include assignments, assumptions and havocs. An assumption operation has the following syntax, where `<expr>` is a boolean expression:

```
assume <expr>
```

Assignments have the following syntax, where `<varname>` is the name of a variable and `<expr>` is an expression of the same type:

```
<varname> := <expr>
```

The syntax of havocs is the following, where `<varname>` is the name of a variable:

```
havoc <varname>
```

Composite operations are either non-deterministic choices or sequences. Non-deterministic choices have the following syntax, where `<operation>` are arbitrary basic or composite operations:

```
choice { <operation> } or { <operation> }
```

Sequences have the following syntax:

```
<operation>
<operation>
<operation>
```

An example showing the usage of all operations presented above:

```
x := 1
choice {
    assume y<2
    x := x+y
} or {
    choice {
        havoc y
        assume true
    } or {
        assume x>2
    }
    x := x-1
}
y := y*2
```

3.3.2 Transitions

Each transition is a single operation (basic or composite). We distinguish between three sets of transitions, *Tran*, *Init* and *Env*. Transitions are described with the following syntax, where `<transition-set>` is either `tran`, `env` or `init`:

```
<transition-set> {
    <operation>
} or {
    <operation>
} or
...
or {
    <operation>
}
```

If the user does not intend to use a transition set, then they can also be left empty, this translates to a set containing a single *skip operation*, with the semantics $v' = v$ for all $v \in V$, i.e. all variables keep their value. Example:

```
env {}
```

3.3.3 Structure of an XSTS model

An XSTS model begins with the type declarations, followed by the variable declarations. After this come the transition sets in the following order: *Tran*, *Init*, *Env*:

```
<type-declarations>
<variable-declarations>
```

```
<tran-set>
<init-set>
<env-set>
```

3.4 Examples

In this section I demonstrate the use of the domain-specific language, how the formal definitions are applied in practice and the state space generation process through examples.

Simple example

Below is a simple example given in the textual representation of the XSTS DSL. The XSTS has two variables, x and y, both with the inital value 0. The init transition sets both variables' values to 1. After this, the environment repeatedly increments the value of y, to which the systems reacts by either incrementing x, or leaving the value of x unchanged.

```
var x: integer = 0
var y: integer = 0

tran {
    x:=x+1
} or {
    x:=x
}

init {
    x:=1
    y:=1
}

env {
    y:=y+1
}
```

Formally, the set variables is $V = \{x, y\}$. Both have the initial value 0, so the initial value function is $IV(x) = 0, IV(y) = 0$. The initial formula formulated from the initial value function is $IF \equiv x = 0 \wedge y = 0$. The single initial state is $((x = 0, y = 0), \textsc{Un}, \textsc{Le})$. Figure 3.2 depicts a part of the concrete state space of the system (the entire state space cannot be depicted as it is infinite, because x and y can take any value). The grey rectangles represent concrete states of the system. The edge that only has a target state but no source state points to the initial state of the system. The edges connecting the states are denoted with the transitions that were required to get from their source state to their target state.

As we can see, the single outgoing edge of the initial state is labeled with the In transition. We can also see that transitions from the Tr and En sets alternate along paths of the system. When En fires, y is incremented. When Tr fires, there are always two opportunities: x is incremented or remains the same. This is a non-deterministic choice,

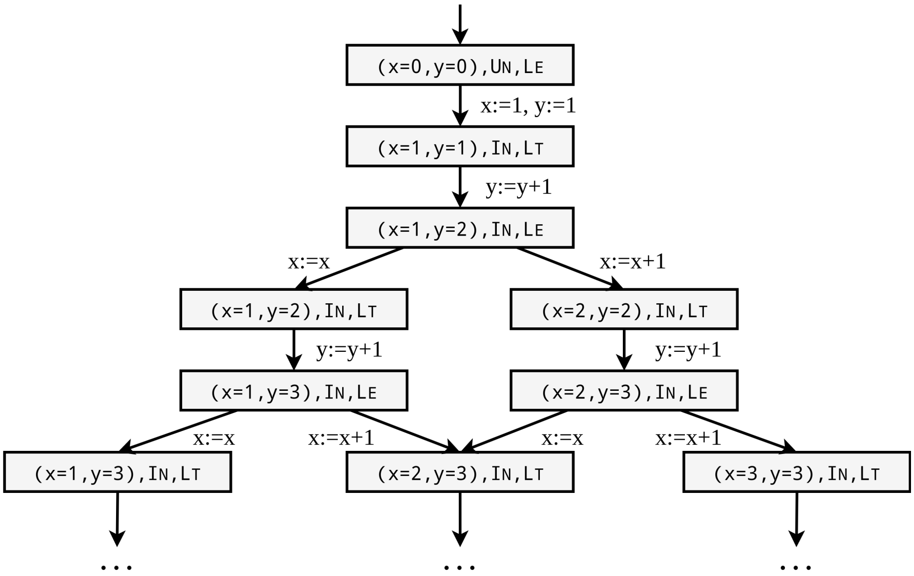


Figure 3.2: The concrete state space of the example XSTS.

in which case the model checker explores all possibilities. We can also see that the In set was only active once, at the beginning with its single transition $x := 1, y := 1$.

To demonstrate the reachability analysis process, let's now consider the safety property $\varphi \equiv y = x$. The analysis starts in the initial state, where the property holds as both x and y are 0. In the next state both x and y are 1, the property still holds. As we progress to the next state however, the property is violated as y gets assigned the value 2, while x stays 1. As a state that violates the property is reachable from the initial state, the model is deemed unsafe.

To demonstrate a case where the model is safe, consider the property $\varphi \equiv y >= x$. Simply by looking at the possible transitions of the system we can determine that the model satisfies the property: the variables start with the same value, and any time x gets incremented y was definitely incremented with the same amount in the previous step. On the other hand, deciding this same property by simple exploration of the concrete state space is impossible in practice. The state space has an infinite number of reachable states and all reachable states have to be enumerated to be certain that none of them violate the property. This is the reason we introduce abstraction in an attempt to avoid generating the entire concrete state space.

Statechart example

Below is a more complex example, which describes the simple Yakindu[2] statechart shown in Fig. 3.3:

```
type Main_region : { __Inactive__, Normal, Error}
var signal_alert_Out : boolean = false
var signal_step_In : boolean = false
var main_region : Main_region = __Inactive__
```

[2] Yakindu Statechart Tools is a statechart modeling framework, see https://www.itemis.com/en/yakindu/state-machine/

```
    tran {
        assume (main_region == Normal && signal_step_In == true)
        main_region := Error
        signal_alert_Out := true
    } or {
        assume (main_region == Error && signal_step_In == true)
        main_region := Normal
    } or {
        assume (!(main_region == __Inactive__) \\
        && !((main_region == Normal && signal_step_In == true) \\
        || (main_region == Error && signal_step_In == true)))
    }

    init {
        main_region := Normal
    }

    env {
        choice {
         signal_step_In := true
        } or {
         signal_step_In := false
        }
        signal_alert_Out := false
    }
```

Note how incoming and outgoing events are described as boolean variables and handled in environmental transitions. A type (`Main_region`) is intruduced to describe the main region and the states inside the region are represented by the literals of this custom type.

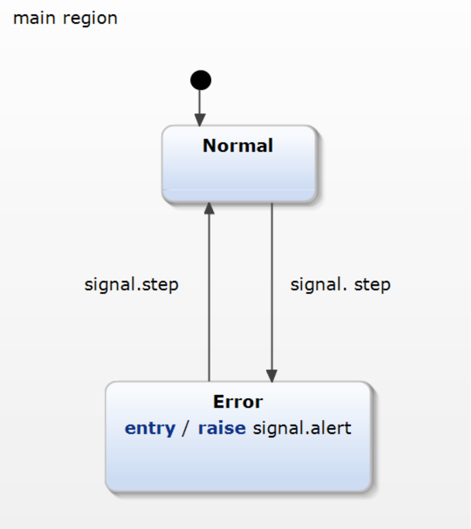


Figure 3.3: A simple Yakindu statechart.

Chapter 4

Model checking of XSTS

In this chapter I present the extensions that I made to the CEGAR algorithm to enable the verification of XSTS models. I also present a novel approach to combining multiple abstract domains along with some minor extensions.

4.1 Extending the CEGAR algorithm for XSTS

To define the CEGAR algorithm for XSTS, I introduce XSTS-specific extensions of concepts defined in Chapter 2, like the transfer function or the abstract reachability graph. I denote the extensions with a subscript X. The extended constructs I introduce in this section all build upon (wrap) the previously defined constructs of the abstract domains introduced in Chapter 2. This wrapping approach is also standard in Theta to support various formalisms with the same algorithmic backend.

Abstract domain

Given an abstract domain $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$, let $D_X = (S_X, \bot_X, \sqsubseteq_X, \mathsf{expr}_X)$ denote its extension for XSTS. Extended XSTS abstract states wrap abstract states of a given domain and explicitly encode the i and e flags. Formally, abstract states $S_X = S \times \{\textsc{Un}, \textsc{Ini}\} \times \{\textsc{Lt}, \textsc{Le}\}$ are 3-tuples consisting of a state $s \in S$ (which can be an explicit or a predicate state for example), and two flags $i \in \{\textsc{Un}, \textsc{Ini}\}$ and $e \in \{\textsc{Lt}, \textsc{Le}\}$. The bottom element becomes a set $\bot_X = (i, e, \bot \mid i \in \{\textsc{Un}, \textsc{Ini}\}, e \in \{\textsc{Un}, \textsc{Ini}\})$. Informally, this is a set of all possible combinations of the i and e flags and the bottom value. The partial order between two states (s_1, i_1, e_1) and (s_2, i_2, e_2) is extended as $(s_1, i_1, e_1) \sqsubseteq_X (s_2, i_2, e_2)$ iff $s_1 \sqsubseteq s_2$, $i_1 = i_2$ and $e_1 = e_2$, i.e. the values of the flags have to match and the sub-state has to be covered by the other sub-state for a state to be covered by another state. The expression function stays the same: $\mathsf{expr}_X \equiv \mathsf{expr}$, the flags are not included in the expression (the transfer function will handle them separately).

Transfer function

Informally, the extended transfer function selects the appropriate set of transitions based on the flags (using the *Ops* function) and applies the transfer function of the wrapped domain to it. Formally, the extended transfer function $T_X : S_X \times \Pi \mapsto 2^{S_X}$ is the following: $T_X((s, i, e), \pi) = \{(s', i', e') \mid op \in \mathit{Ops}(i, e, i', e'), s' \in T(s, op, \pi)\}$, i.e. (s', i', e') is a successor of (s, i, e) if an operation *op* exists that is included in $Ops(i, e, i', e')$, and s' is a successor of s with regards to the inner transfer function T, *op* and π. The function $Ops(i, e, i', e')$ (as defined in Section 3.2) returns the set of available transitions for a pair of source and target flags.

Init function

The extended init function $I_X : \Pi \mapsto 2^{S_X}$ is the following: $I_X(\pi) = \{(s, i, e \mid s \in I(\pi),\ i = \textsc{Un},\ e = \textsc{Le})\}$, i.e. the states returned by the inner init function I, extended with the Un and Le flags. Informally, the extended init function takes the result of the wrapped init function and attaches the appropriate flags (Un and Le) to it to ensure the proper ordering of the transition sets (as depicted in Fig. 3.1).

Abstract reachability graph

The extended abstract reachability graph ARG_X is a tuple $ARG_X = (N_X, E_X, C_X)$ where

- $N_X \subseteq S_X$ is the set of nodes, where an extended node $(s, i, e) \in N_X$ represents an extended state;
- $E_X \subseteq N_X \times Ops \times N_X$ is a set of directed edges labeled with operations. An edge $(s_1, i_1, e_1, op, s_2, i_2, e_2) \in E_X$ is present if s_2 is a successor of s_1 with regards to *op*, and $op \in Ops(i_1, e_1, i_2, e_2)$;
- $C_X \subseteq S_X \times S_X$ is the set of covered-by edges. An edge $(s_1, i_1, e_1, s_2, i_2, e_2) \in C_X$ is present if $(s_1, i_1, e_1) \sqsubseteq (s_2, i_2, e_2)$.

Model checking

The XSTS model checker awaits two inputs, an XSTS model and a safety property φ. The model checker then constructs the initial precision π_0 (optionally by using the optimizations introduced in Section 4.4). The model checker then invokes the CEGAR algorithm (see Alg. 2) with the safety property, the initial precision and the extended abstract domain, transfer function and init function that were defined in the previous sections. Formally: $\text{CEGAR}(\varphi, D_X, \pi_0, T_X, I_X)$.

4.2 XSTS model checking example

In this section I demonstrate the XSTS model checking process through an example. Let us consider the simple XSTS example model from Section 3.4.

To demonstrate the state space generation process, let's first consider explicit-value abstraction with the precision $\pi = \{x\}$, i.e., only tracking x. In this case the init function returns a single initial state, $((x = 0, y = \top), \textsc{Un}, \textsc{Le})$.

Fig. 4.1 depicts part of the ARG (this state space is infinite, so it cannot be depicted in its entirety) in the situation outlined earlier. The grey rectangles represent nodes of the ARG $s_X = (s_E, i, e) \in N_X$, where $s_E \in S_E$ are explicit states. The node with the incoming edge that only has a target but no source contains the initial state of the system. The edges are labeled with the transitions that were executed to get from their nodes' source state

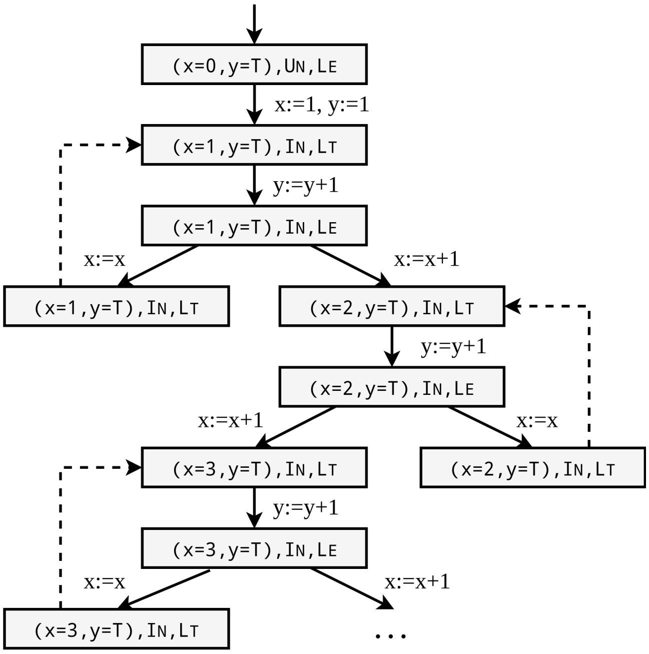


Figure 4.1: Part of the ARG of the example XSTS with the precision $\pi = \{x\}$ using explicit-value abstraction.

to their nodes' target state. Dashed edges represent a covered-by relation between nodes. As we can see, transitions from the *Tr* and *En* sets alternate along paths of the system. When *En* is fired, y is incremented, but its value remains $\top$ as it is not tracked. When *Tr* fires, there are always two opportunities: x is incremented or remains the same. As x is tracked explicitly, the model checker explores both possibilities. Just like in case of the concrete state space, it is also clearly visible in the ARG that the *In* set was only active once, at the beginning.

Let's now consider the predicate abstraction domain with the precision $\pi = \{y > x, y < 1\}$. Fig. 4.2 denotes the ARG in this case. The result of the init function is the state $((!(y > x) \wedge y < 1), \text{Un}, \text{Le})$, as the first predicate $y > x$ contradicts the initial formula $x = 0 \wedge y = 0$, and the second predicate $y < 1$ is consistent with it. In the next step, when applying *In*, $(y < 1)$ changes to $!(y < 1)$ after setting y to 1 because the formula is now false. After that, *En* is applied. Note that $!(y < 1)$ does not change when incrementing y. However, $y > x$ might hold or not, so there are two branches (Cartesian abstraction would merge these two states, Boolean keeps the disjunction).

As opposed to the previous case, this ARG is finite as the state space is finite (the two predicates and the two Boolean flags can take 2^4 total combinations). The analysis quickly comes to point where all successor states are covered and can thus terminate.

4.3 Product abstraction with information exchange

In this section I present a static combination of explicit-value abstraction and predicate abstraction. This approach builds upon the concepts defined in Section 2.3.1. In my approach, a fixed subset of the variables is tracked explicitly, other variables are tracked through predicates. This subset of explicitly tracked variables is the set of *control variables* introduced in Section 3.2. Control variables can for example be integer variables that get

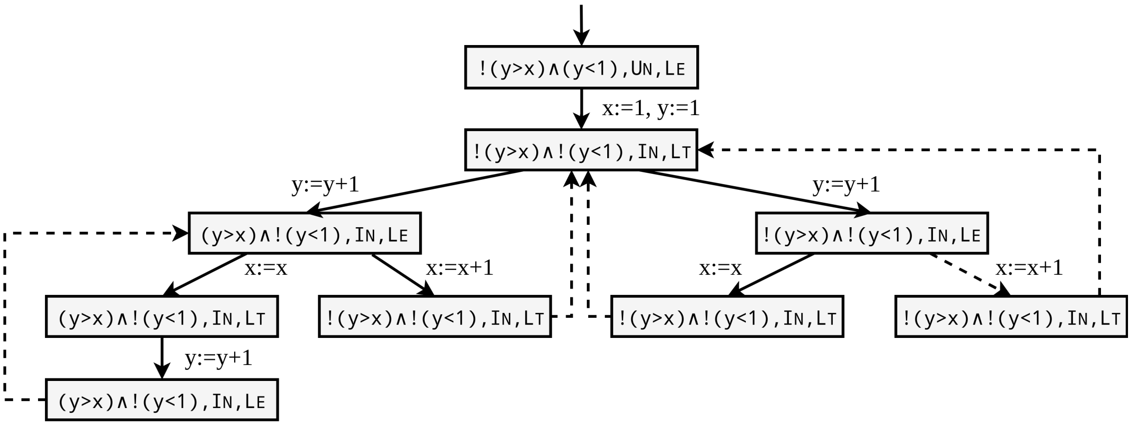
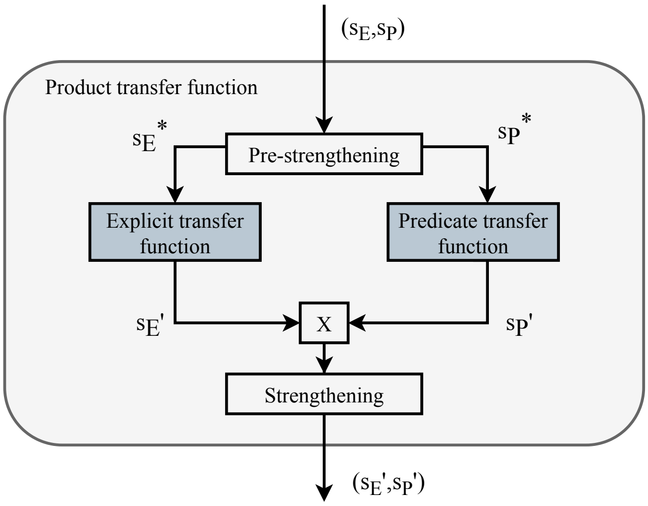


Figure 4.2: The ARG of the example XSTS with the precision $\pi = \{y > x, y < 1\}$ using predicate abstraction.

assigned a big portion of their domains or the variables storing the state configuration of a statechart. In this case there would be too many predicates. More importantly, I also present a novel operator which allows for information exchange between the domains to supply the individual transfer functions of the subdomains with additional information.



(sE,sP)
Product transfer function
sE*
sP*
Pre-strengthening
Explicit transfer function
Predicate transfer function
sE'
X
sP'
Strengthening
(sE',sP')

Figure 4.3: An overview of the modified product transfer function.

In order to allow the transfer functions of the subdomains to utilize information that is present in the other domain, I introduce a novel operator, the *pre-strengthening operator*. The pre-strengthening operator is a function $PS : S \mapsto S_E \times S_P$ that takes a product state as an input and returns the pair of a strengthened explicit state and a strengthened predicate abstraction state, that can then be used as inputs of the individual transfer functions. The original behaviour of the transfer function can be simulated with the following pre-strengthening operator: $PS((s_E, s_P)) \equiv (s_E, s_P)$, i.e. by returning the input states unmodified. Instead of this, I propose the following pre-strengthening operator: $PS((s_E, s_P)) \equiv (s_E, \{s_P^* \mid \mathsf{expr}(s_P^*) = \mathsf{expr}(s_E) \land \mathsf{expr}(s_P)\})$. Informally, this pre-strengthening operator returns the unchanged explicit state and a predicate domain state that is constructed by forming the conjunction of the expressions of the two input states and interpreting it as a predicate domain state. For example if (s_E, s_P) is $((x = 1), (y >= 2))$, then $PS(((x = 1), (y >= 2)))$ is $((x = 1), (x =$

$1 \land y >= 2))$. The transfer function extended with the pre-strengthening operator PS is $T_{PS}((s_E, s_P), op, (\pi_E, \pi_P)) = \{T_E(s_E^*, op, \pi_E) \times T_P(s_P^*, op, \pi_P) \mid PS(s_E, s_P) = (s_E^*, s_P^*)\}$. Informally, the transfer function feeds the strengthened states returned by the pre-strengthening operator as input to the individual transfer functions of the subdomains, then calculates the products of their returned states. The advantage of this approach is that the analysis can be more precise by using the information from the other subdomain. Note that this pre-strengthening operator is not XSTS specific and can be applied to other formalisms as well.

4.4 Initial precision optimization

When starting the CEGAR loop with an empty precision we leave some information unused that is present in the model checking question itself. For example, if the safety property is $x > 0$, then knowing something about the value of x is in most cases necessary to decide whether or the property holds. I propose an optimization to the CEGAR algorithm that uses the safety property φ to construct the initial precision and can thus save the algorithm some refinement iterations that would be spent introducing the variables of the property into the precision. The intuition behind this is that in order to decide whether or not the property holds, the algorithm has to be able to track the variables that appear in it. In case of predicate abstraction the property φ can be used as a predicate in the precision, i.e., $\pi_0 = \{\varphi\}$. In case of explicit-value abstraction the variables of the property are tracked explicitly from the start, i.e., $\pi_0 = var(\varphi)$. These optimizations are not XSTS-specific, they can be used in any safety analysis. An additional, XSTS-specific optimization in case of explicit-value abstraction would be to start tracking the set of control variables V_C explicitly from the start. When both optimization are applied together, the initial precision is constructed the following way: $\pi_0 = var(\varphi) \cup V_C$, i.e. by forming the union of the variables of the property and the set of control variables.

Chapter 5

Evaluation

5.1 Implementation

I implemented the XSTS formalism along with a domain-specific language and a command-line model checking tool in the Theta[1] model checking framework [42], in three separate modules, *xsts*, *xsts-analysis* and *xsts-cli*. Figure 5.1 illustrates the way these new modules integrate into the architecture of Theta. I also had to extend and modify the existing *core* and *analysis* modules of the framework to implement lower-level constructs such as sequence or non-deterministic choice operations, or the product abstraction domain. My additions were thoroughly reviewed by the developers of the Theta framework and included in the v2.0.0 release[2] of the Theta framework. As of October 2020, the XSTS and its related modules are developed and maintained by me, and are a core part of the Theta framework.

	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 5.1: An overview of the Theta architecture as of October 2020. *Italic* entries represent modules where I contributed.

5.1.1 Additions to existing Theta modules

This section gives a brief summary on my additions to existing Theta modules. I implemented the high-level composite operations defined in Section 3.2 in the *core* module. In Theta, operations are called statements, I created the *SequenceStmt* and *NondetStmt* statements and modified the necessary statement processing classes (e.g classes that interpret statements, transform statements to expressions, or utility classes that collect variables) to be able to handle the new statements. I implemented product abstraction and the related algorithms (Section 4.3) in the *analysis* module, this involved among other things the modified product transfer function and the pre-strengthening operator.

[1] `https://github.com/ftsrg/theta`

[2] `https://github.com/ftsrg/theta/releases/tag/v2.0.0`

5.1.2 XSTS modules

This section briefly summarizes the new XSTS modules I created.

xsts

The *xsts*[3] module contains the classes representing XSTS models and the domain-specific language described in Section 3.3. An XSTS model is represented by the *XSTS* class, that contains the variables and statements of the system. In this implementation I could reuse many of the building blocks Theta offers (such as classes representing variables, statements, expressions). I implemented the domain-specific language in Antlr[4].

xsts-analysis

The *xsts-analysis*[5] module contains classes required for model checking of XSTS models. XSTS-specific adjustments of the CEGAR algorithm described in Section 4.1, such as the extended transfer function, extended init function or the extended abstract states are all implemented in this module.

xsts-cli

The *xsts-cli*[6] module contains a command-line model checker that provides an easy-to-use user interface for configurable model checking of XSTS models. This program parses the input model, input property and the algorithm configuration and instantiates the analysis accordingly. The model checking algorithm can be configured using the following command-line parameters [21]:

- `-domain`: The abstract domain. Possible values are:
  - `PRED_CART`: Cartesian predicate abstraction, see Section 2.3.1;
  - `PRED_BOOL`: Boolean predicate abstraction, see Section 2.3.1;
  - `PRED_SPLIT`: Boolean predicate abstraction, but states are split into sub-states along disjunctions. See Section 2.3.1;
  - `EXPL`: Explicit-value abstraction, see Section 2.3.1;
  - `PROD`: Product abstraction using two subdomains, `EXPL` and `PRED_CART`. See Section 4.3.
- `-initprec`: The initial precision of the algorithm. Possible values:
  - `EMPTY`: Start with an empty initial precision;
  - `PROP`: Track all variables appearearing the the safety property in case of `EXPL` domain. Construct predicates from the safety property in case of `PRED_*`. See Section 4.4;
  - `CTRL`: Track all control variables explicitly in case of `EXPL` or `PROD` domain. See Section 4.4.

[3] `https://github.com/ftsrg/theta/tree/master/subprojects/xsts`

[4] `https://www.antlr.org/`

[5] `https://github.com/ftsrg/theta/tree/master/subprojects/xsts-analysis`

[6] `https://github.com/ftsrg/theta/tree/master/subprojects/xsts-cli`

- `-search`: Search strategy in the abstract state space. Determines the order in which abstract states are processed:
  - `BFS`: Breadth-first search;
  - `DFS`: Depth-first search.
- `-maxenum`: Maximal number of states to be enumerated when performing explicit-value analysis and an expression cannot be deterministically evaluated [22], i.e. the SMT solver returns multiple satisfying assignments as result of the operation. If the limit is exceeded no more states are enumerated and $\top$ is returned as the successor state. This allows the explicit-value domain to better handle non-deterministic assignments. Possible values are integers, as a special case, 0 stands for infinite;
- `-refinement`: Strategy for refining the precision of the abstraction, i.e., inferring new predicates or variables to be tracked. Detailed introduction into the different refinement strategies is outside the focus of this thesis, but below is a short informal summary of each option (see [22] for further reading):
  - `FW_BIN_ITP`: Forward binary interpolation. Searches for the first state where the counterexample becomes infeasible starting from the initial state;
  - `BW_BIN_ITP`: Backward binary interpolation. Searches backwards for the first state where the counterexample becomes infeasible starting from the target state;
  - `SEQ_ITP`: Sequence interpolation, which refines the whole path at once;
  - `MULTI_SEQ`: Sequence interpolation with multiple counterexamples.
- `-predsplit`: Available if `-domain` is `PRED_*`. Determines whether splitting is applied to new predicates that are obtained during refinement. See [22] for details. Possible values:
  - `WHOLE`: Keep predicates as a whole, no splitting is applied;
  - `CONJUNCTS`: Split predicates into conjuncts.
  - `ATOMS`: Split predicates into atoms.
- `-prunestrategy`: The pruning strategy controls which portion of the abstract state space is discarded during refinement. Possible values:
  - `FULL`: The whole abstract reachability graph (ARG) is pruned and abstraction is completely restarted with the new precision;
  - `LAZY`: The ARG is only pruned back to the first point where refinement was applied, i.e. until the first unconcretizable step of the abstract counterexample [26].

The command-line tool accepts the following additional parameters:

- `-model`: Path of the input XSTS model (mandatory).
- `-property`: Input property as a string or a file (*.prop) (mandatory).
- `-cex`: Output file where the counterexample is written (if the result is unsafe). If the argument is not given (default) the counterexample is not printed.

- `-loglevel`: Detailedness of logging. Possible values (from the least to the most detailed): `RESULT`, `MAINSTEP`, `SUBSTEP` (default), `INFO`, `DETAIL`, `VERBOSE`.
- `-version`: Print version info (in this case `-model` and `-property` is of course not required).

The tool is deployed in form or a runnable JAR file, and a Docker[7] image is also offered for portability. The following command shows an example for starting the model checker with the default configuration, where `theta-xsts-cli.jar` is the JAR file containing the *xsts-cli* tool:

```
java -jar theta-xsts-cli.jar --model crossroad.xsts --property "x>1"
```

5.2 End-to-end hidden formal methods

Besides their high computational complexity, the other important hindering factor holding back formal methods from widespread engineering application is that they are cumbersome to use. Engineers and software developers who are often experts of other domains can't realistically be expected to know the fine details behind formal methods or be familiar with the low-level formalisms and concepts involved in efficient model checking techniques. A widespread model checking tool that is an essential and easy-to-use tool of any developer's toolbox cannot realistically require engineers to get familiar with domain-unrelated concepts such as CEGAR, SMT solvers or the XSTS formalism. Such a tool needs to accept inputs and provide feedback in a way that is relevant to the application domain, and has to be utilizable with minimal model checking knowledge, abstracting away such details from its user and leading to easy-to-use end-to-end hidden formal methods.

The Gamma Statechart Composition Framework [38] (as mentioned in Section 2.2.2), is a framework to model, verify and generate code for component-based reactive systems. It allows users to define complex statechart networks. Based on the defined models, its code generator can synthesize Java code. The statecharts can also be verified using the UPPAAL model checker [33].

In an effort to extend the Gamma framework's model checking capabilites, Theta, more specifically the XSTS formalism and my XSTS implementation, was integrated[8] into the Gamma framework as a verification backend (as an alternative to UPPAAL). This means that the complex statechart networks of the Gamma framework can be verified with the click of a button. In the background, the statechart networks get transformed to the XSTS formalism and the *xsts-cli* (see Section 5.1.2) tool gets invoked (parameterizing the model checker is not a trivial question, as we will see in the next section). The Gamma framework then interprets the output produced by the model checker and presents it in an intuitive manner. If the model checker returns a counterexample, then the Gamma tool also generates a test case from the trace, which can be used to verify that the counterexample applies to the generated Java code as well. This integration into the Gamma framework is in the largest part the contribution of Graics Bence (PhD student at BME MIT). During the process he consulted with me and we discussed the syntax and semantics of the XSTS constructs in multiple iterations. The XSTS formalism and command-line tool being applied as backend in the end-to-end hidden formal verification workflow of the

[7] `https://www.docker.com/`

[8] The XSTS-related plugins of the Gamma framework: `https://github.com/ftsrg/gamma/tree/master/plugins/xsts`

Gamma framework proves the practical applicability of my approach. It also allows me to evaluate my approach on complex statechart network models that were transformed to XSTS using Gamma. The transformation that maps statechart compositions to XSTS models proves that the XSTS formalism has the expressive power to encode the complex structure and behaviour of statechart models. It is also important to note that the ability of the Gamma framework to generate test cases from the counterexamples produced by my tool proves that my tool is capable of producing meaningful output that other tools can build upon.

5.3 Experimental evaluation

I evaluated my approach by formulating various research questions and answering them using benchmarks conducted in a controlled environment. In this section I present the questions and the results of the experiments.

5.3.1 Experiment planning

My goal during the evaluation process was to evaluate my approach on a broad set of input models from different sources with various different configurations. I also intended the find the set of configurations which perform well in general or in specific scenarios. I formulated the following research questions that I intended to answer during benchmarking:

RQ1 Which configuration performs best overall?

RQ2 Does changing the search strategy (`BFS` vs `DFS`) lead to a significant difference in performance?

RQ3 Which variant of predicate abstraction (`PRED_CART` vs `PRED_BOOL` vs `PRED_SPLIT`) has the best performance?

RQ4 Which enumeration limit of explicit states (`-maxenum` parameter) has the best performance?

RQ5 Which refinement strategy and prune strategy combination has the best performance?

RQ6 How does product abstraction perform compared to explicit value abstraction and product abstraction?

An important aspect of experiment planning is determining how to quantify the "goodness" of a test subject. In my benchmarks I selected the number of successfully verified models (within a time limit) as the primary measure of performance, i.e. if a configuration was able to verify more models, than another configuration, then it offered better performance. The second most important measure is the time required for verification. If two configurations verified the same amount of models, but one did so in shorter time, then it performed better.

5.3.2 Benchmarking environment

The measurements were performed in the BME Smallville Cloud[9]. The virtual machines each had 4 processor cores (the hosting service provides no further information about the CPUs of the virtual machines), 8192 MiB of memory and were running the Ubuntu 18.04 LTS Server operating system. Reliable bechmarking was ensured by BenchExec [10], a state-of-the-art benchmarking framework, which guarantees proper isolation of the measured process and independence from outer noises. BenchExec guarantees precise measurements, is capable of measuring the actual CPU-time (as opposed to the wall time), and can precisely track memory consumption even in case a swapping happens. BenchExec provides the measurements of the annual Competition on Software Verification (SV-Comp) [4]. Each measurement was run with a timeout limit of 120s.

5.3.3 Input models

I increased the validity of the measurements by using models from different and diverse sources, including industrial models. In this section I give a brief introduction of the different sets of models I used.

5.3.3.1 Artificial examples

The **simple** set contains 18 smaller models each constructed by me. The primary goal when constructing this test set was to provide complete coverage of all XSTS constructs and features, like the composite statements, the En and In transition sets and the operators of the DSL.

5.3.3.2 Industrial models

The following model sets contain real-world examples provided by different industrial partners of the university. This set contains 10 models that appear in 53 test cases overall.

- **INPE**[10]: A statechart composition modeling 2 components of a communication protocol inside a nanosatellite.
- **COID**: This statechart composition models components of railway safety equipment (an antivalence filter and an indicator component). Two antivalence filter components are connected to an indicator component, which emits signals for adjusting a release-timing parameter. This model set was provided by a confidential industry partner of the university;
- **PIL**: A statechart composition modeling components of railway safety equipment (a railroad switch and an indicator component). These components are more complex, their purpose is modeling and verifying railroad track blocking based on a safety equipment algorithm. This model set was provided by a confidential industry partner of the university;
- **TrafficLight**: A statechart composition modeling a crossroad traffic light with 2 traffic light components and a control component;

[9] `https://smallville.cloud.bme.hu/`

[10] 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/`

- **Spacecraft**: A statechart modeling a spacecraft component from [29], illustrated in Appendix A.1. The SysML model depicted in the figure was transformed to the statechart language of the Gamma framework, then transformed to XSTS using Gamma.

Table 5.2 contains metrics about the variables of the input model sets. The **Avg Vars** column contains the average number of variables present in models of the set. The **Max Vars** column contains the maximum number of variables present in any model of the set. Table 5.3 contains metrics about the statements present in the sets. Similarly to the previous table, **Avg Stmts** contains the average number of statements present in models of the set. **Max Stmts** contains the maximum number of statements present in models of the set. In both tables the **Model** column contains the number of distinct models present in the set, while the **Cases** column contains the total number of test cases present in the set (a single model can be present with multiple requirement properties and thus appear in multiple test cases).

Model set	**Models**	**Cases**	**Avg Vars**	**Max Vars**
simple	18	21	2.8	6
COID	1	12	43.0	43
INPE	1	18	31.0	31
PIL	6	8	227.6	372
Spacecraft	1	1	18.0	18
TrafficLight	1	14	29.0	29
Total	28	74		

Table 5.2: Metrics about the variables of input model sets.

Model set	**Models**	**Cases**	**Avg Stmts**	**Max Stmts**
simple	18	21	13.1	29
COID	1	12	940.0	940
INPE	1	18	152.0	152
PIL	6	8	1608.6	2310
Spacecraft	1	1	146.0	146
TrafficLight	1	14	216.0	216
Total	28	74		

Table 5.3: Metrics about the statements of the input model sets.

5.3.4 Configurations

In each research question except the first one (RQ1: Which configuration performs best overall?), I focus on either 1 or 2 parameters (these are the *free* parameters [43]) and leave all other parameters (these are the *bounded* parameters) at a fixed setting, usually one that performed well in previous experiments (such as [22]). For example in RQ2, the single free parameter is the `-search` parameter, the parameter I want to come to a conclusion about, all other parameters are fixed at the setting that was found the overall best in previous experiments.

The configuration names follow the following naming pattern. All parameter options are abbreviated to 1-3 characters:

- -domain: PC = PRED_CART, PB = PRED_BOOL, PS = PRED_SPLIT, E = EXPL, P = PROD;
- -prunestrategy: L = LAZY, F = FULL;
- -refinement: FBI = FW_BIN_ITP, BBI = BW_BIN_ITP, SI = SEQ_ITP, MS = MULTI_SEQ;
- -search: B = BFS, D = DFS;
- -predsplit: W = WHOLE, C = CONJUNCTS, A = ATOMS.

The configuration names are constructed from the abbreviations the following way (if a parameter is not present in the name, then it was not specified in the current configuration):

```
<domain><maxenum>_<prunestrategy>_<refinement>_<search><predsplit>
```

For example the configuration PC_L_SI_BW refers to PRED_CART domain with LAZY pruning strategy, SEQ_ITP refinement strategy, BFS search strategy, WHOLE predicate splitting strategy and an omitted maxenum parameter.

5.3.5 Benchmarking results

In this section I present my benchmarking results.

RQ1: Best configuration overall

Free parameters:

- -domain: {PRED_CART, PRED_BOOL, PRED_SPLIT, EXPL, PROD};
- -initprec: {EMPTY, PROP, CTRL};
- -search: {BFS, DFS};
- -refinement: {FW_BIN_ITP, BW_BIN_ITP, SEQ_ITP, MULTI_SEQ};
- -predsplit: {WHOLE, CONJUNTS, ATOMS};
- -prunestrategy: {FULL, LAZY}.
- -maxenum: {0, 10, 250, 1000}

This benchmark had no bounded parameters, but certain parameters were omitted in special cases. For example predicate splitting is not used in the explicit domain, so when -domain is EXPL, then the -predsplit parameter is omitted. Similarly -maxenum is not used when predicate abstraction domain is applied. Generating all possible combinations of the parameters would have resulted in 2880 separate configurations, netting 213120 total measurements, which would have been too much to execute in practice. To reduce the set of configurations, I employed the pairwise combination generator of PICT[11], which pairs parameters (for example -search with -refinement), and generates all possible

[11] PICT (Pairwise Independent Combinatorial Testing) is a test case generator tool, see: https://github.com/microsoft/pict

combinations of the pairs of values. Research shows that this is an effective alternative to generating all possible combinations [32].

There were 74 cases in this benchmark with 34 configurations, netting 2516 measurements, which ran for 1 day and 6 hours. All cases were verified by at least one configuration and none of the runs returned an incorrect result. 1728/2516 (69%) measurements were successful, 5 runs resulted in stack overflow, the rest reached the timeout limit.

Configuration	**SuccCount**	**TimeSum**	**TimeMax**
PC_F_MS_DW	71	507.98	56.08
PB_L_SI_DA	68	493.10	73.91
PS_F_BBI_DA	66	762.19	113.60
P1000_L_BBI_DW	64	523.34	52.01
P10_F_BBI_DW	64	559.08	55.23
PB_F_BBI_BC	63	731.58	110.55
PS_F_SI_DC	62	508.46	64.33
E10_L_UC_B	62	403.77	68.74

Table 5.4: RQ1: The 8 best results.

Table 5.4 shows the top 8 configurations regarding the number of verified cases. The **SuccCount** column contains the number of verified cases, **TimeSum** refers to the sum of the CPU times of the successful runs in seconds (i.e. timeouts are not included in this number). **TimeMax** is the maximum measured CPU time among successful runs in seconds. Of the 34 configurations included in this benchmark, the `PC_F_MS_DW` configuration performed the best, successfully verifying 71 cases.

A rowchart depicting the success rates measured in this benchmark is included in Appendix A.2. The more models a configuration verified, the wider the *blue* area is in its corresponding row. The *green* areas represent timeouts and the *red* areas represent exceptions. This diagram indicates that there are significant differences between configurations in their success rates.

The results show that `PROD` can compete with the other domains. The 4th and 5th best configurations used the `PROD` domain, and the best configuration that used `PROD` finished higher than the best configuration using `EXPL`, they successfully verified 64 and 62 cases, respectively. This benchmark alone is on the other hand not enough to come to conclusions about any of the parameters, because the 34 configurations cover only a small portion of the 2880 possible configurations. For example, the best configuration based on previous experience, `PC_L_SI_BW`, which is capable of verifying 73 cases of the 74, wasn't included in this set either.

RQ2: Search strategy

In this benchmark I constructed 2 different configurations, which only differ in their `-search` parameter.

Free parameters:

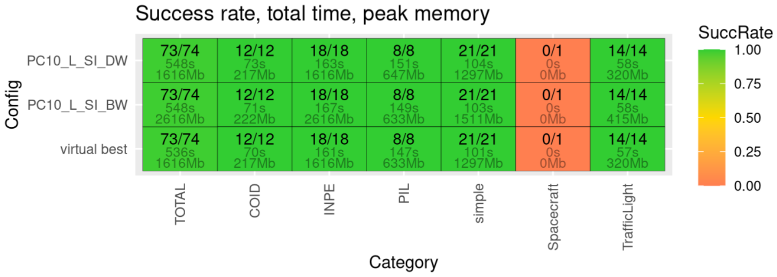
- `-search`: {`BFS`, `DFS`}.

Bounded parameters:

- `-domain`: `PRED_CART`;

- `-refinement`: `SEQ_ITP`;
- `-predsplit`: `WHOLE`;
- `-prunestrategy`: `LAZY`;

There were 74 cases in this run with 2 configurations, resulting in 148 measurements.



Config	TOTAL	COID	INPE	PIL	simple	Spacecraft	TrafficLight
PC10_L_SI_DW	73/74 548s 1616Mb	12/12 73s 217Mb	18/18 163s 1616Mb	8/8 151s 647Mb	21/21 104s 1297Mb	0/1 0s 0Mb	14/14 58s 320Mb
PC10_L_SI_BW	73/74 548s 2616Mb	12/12 71s 222Mb	18/18 167s 2616Mb	8/8 149s 633Mb	21/21 103s 1511Mb	0/1 0s 0Mb	14/14 58s 415Mb
virtual best	73/74 536s 1616Mb	12/12 70s 217Mb	18/18 161s 1616Mb	8/8 147s 633Mb	21/21 101s 1297Mb	0/1 0s 0Mb	14/14 57s 320Mb

Figure 5.1: RQ2: A heatmap comparing the success rates of the two search options.

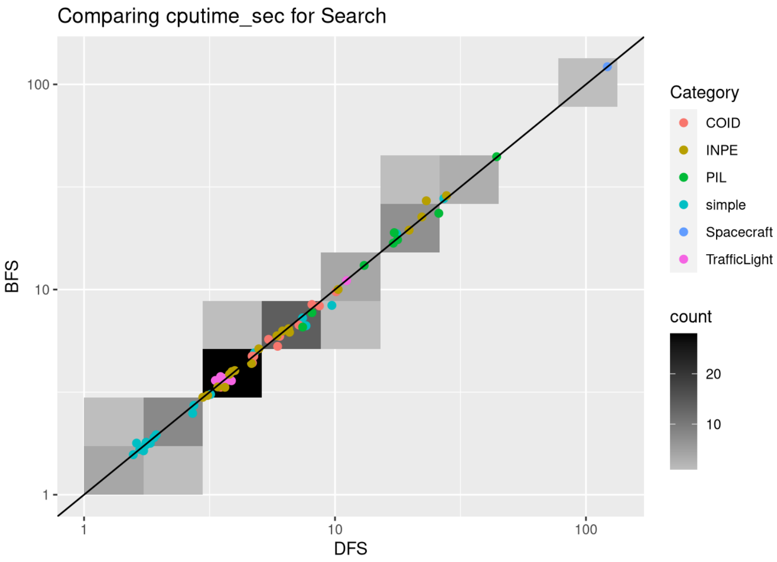


Figure 5.2: RQ2: Comparison of the two search options.

Fig. 5.2 compares the CPU times measured for each model by the two configurations. Each dot represents a single run, with its x-coordinate corresponding to its CPU time with `DFS` and similarly the y-coordinate indicating its CPU time with `BFS`. Points that are above the line indicate runs, where the the `DFS` configuration needed less time, and point above the line run, where the `BFS` domain was faster. As all points fall in close proximity of the line, it is clearly visible that there was no significant difference between the two search options in terms of measured CPU time. Fig. 5.1 depicts the result in a heatmap. Rows of the heatmap correspond to the different configurations and columns correspond to the different input model sets. Each cell contains the ratio of models from the corresponding set that the corresponding configuration was able to verify, and also the combined time and memory consumption of the successful measurements. The leftmost column contains information about the total set of inputs, while the downmost row corresponds to the virtual best configuration (taking the result of the best configuration for each individual model). The heatmap indicates that there was no difference between the two options in terms of success rate either, and minor differences in terms of memory consumption.

RQ3: Predicate abstraction variants

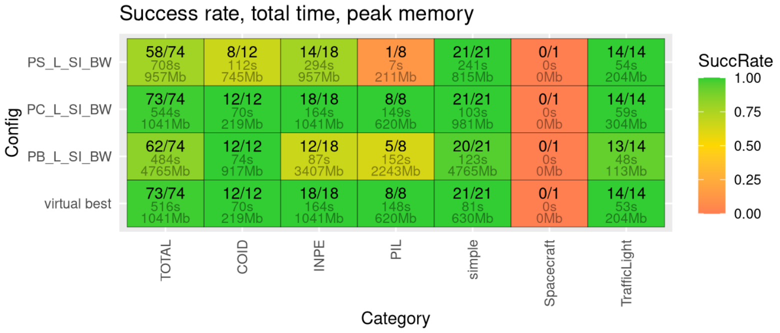
Free parameters:

- `-domain`: {`PRED_CART`, `PRED_BOOL`, `PRED_SPLIT`}.

Bounded parameters:

- `-search`: `BFS`;
- `-refinement`: `SEQ_ITP`;
- `-predsplit`: `WHOLE`;
- `-prunestrategy`: `LAZY`;

There were 74 cases in this run with 3 configurations resulting in 222 measurements.



Config	TOTAL	COID	INPE	PIL	simple	Spacecraft	TrafficLight
PS_L_SI_BW	58/74 708s 957Mb	8/12 112s 745Mb	14/18 294s 957Mb	1/8 7s 211Mb	21/21 241s 815Mb	0/1 0s 0Mb	14/14 54s 204Mb
PC_L_SI_BW	73/74 544s 1041Mb	12/12 70s 219Mb	18/18 164s 1041Mb	8/8 149s 620Mb	21/21 103s 981Mb	0/1 0s 0Mb	14/14 59s 304Mb
PB_L_SI_BW	62/74 484s 4765Mb	12/12 74s 917Mb	12/18 87s 3407Mb	5/8 152s 2243Mb	20/21 123s 4765Mb	0/1 0s 0Mb	13/14 48s 113Mb
virtual best	73/74 516s 1041Mb	12/12 70s 219Mb	18/18 164s 1041Mb	8/8 148s 620Mb	21/21 81s 630Mb	0/1 0s 0Mb	14/14 53s 204Mb

Figure 5.3: RQ3: A heatmap comparing the success rates of the different predicate abstraction variants.

The results (see Fig. 5.3) show that the `PRED_CART` domain performed significantly better than the two other variants. The leftmost cloumn contain the total success rates, here we can see that `PRED_CART` successfully verified 73 cases, while `PRED_SPLIT` and `PRED_BOOL` only verified 58 and 62, respectively.

RQ4: Explicit state enumeration limit

Free parameters:

- `-maxenum`: {0, 1, 10, 250, 500, 1000, 10000}.

Bounded parameters:

- `-domain`: `EXPL`;
- `-search`: `BFS`;
- `-refinement`: `SEQ_ITP`;
- `-prunestrategy`: `LAZY`;

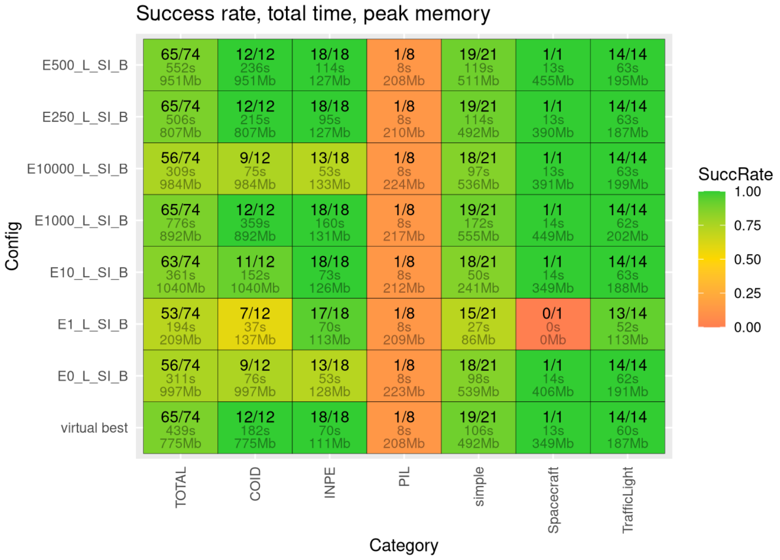


Figure 5.4: RQ4: A heatmap comparing the success rates of the different -maxenum options.

There were 74 test cases in this run with 7 different configurations, giving 518 measurements.

The results show that the 250 option performed the best. Looking at the leftmost column of the heatmap in Fig. 5.4 we can see that it verified the most amount of models, 65, in the least amount of time, 506s. Two other options, 500 and 1000 also verified 65 models, but required more time. Fig. 5.5 displays the measured results on a quantile plot [10]. The colored lines represent the different configurations, the more models a configuration was able to verify, the further to the right its corresponding line stretches. Maximum required CPU time is plotted on the y-axis, meaning the lower down the right end of a line is, the lower the worst case in terms CPU time was for the corresponding configuration. Note that the y-axis is not linear. As it can be seen in Fig. 5.5, the 250 option verified the highest amount of models and was fastest among the configurations which had the same amount of successful verifications. Increasing the -maxenum parameter to 500, 1000 or even 10000 did not increase the success ratio, but resulted in an increase in the measured CPU time. This is due to the reason that the increased limit meant that more explicit states were enumerated. Note that the optimal setting for the -maxenum parameter is highly dependant on the verified model. Even though the 250 option fits our set of models the best, other models might perform better with a different -maxenum value. The heatmap in Fig. 5.4 gives a brief overview of the performance of the other configurations. It is also interesting to see that the maxenum value 1 performed significantly worse than any other setting (this is also clearly visible in the quantile plot).

RQ5: Refinement and pruning strategy

Free parameters:

- -refinement: {FW_BIN_ITP, BW_BIN_ITP, SEQ_ITP, MULTI_SEQ};
- -prunestrategy: {LAZY, FULL}.

Bounded parameters:

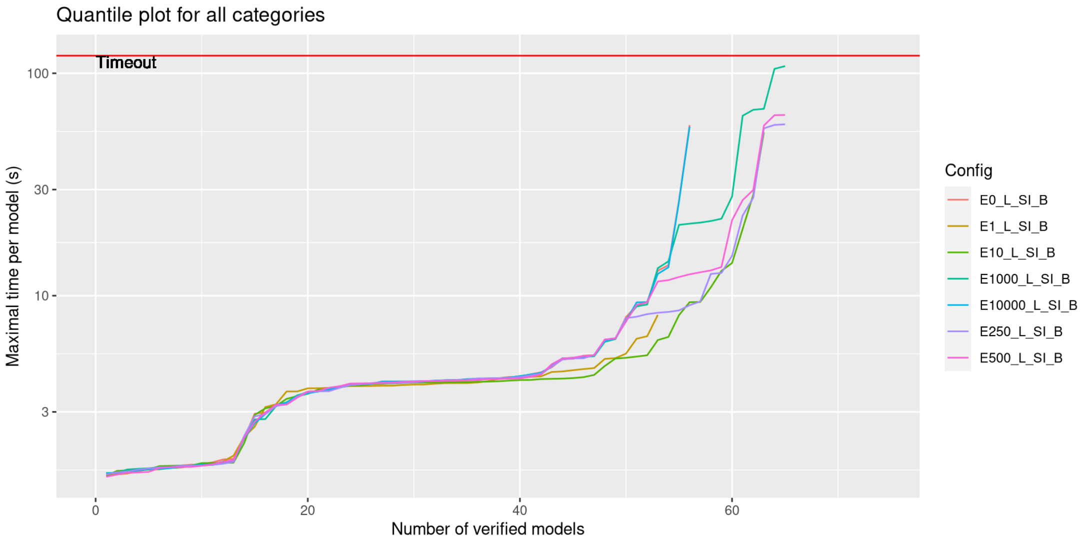
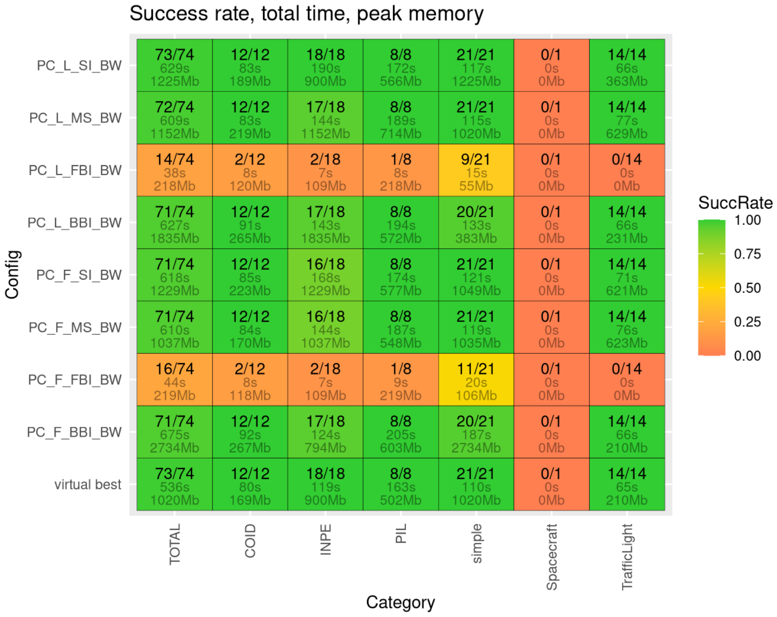


Figure 5.5: RQ4: Quantile plot comparing the different `-maxenum` options.

- `-domain`: `PRED_CART`;
- `-search`: `BFS`;
- `-predsplit`: `WHOLE`.

There were 74 test cases in this run with 8 different configurations resulting in 592 measurements.



	TOTAL	COID	INPE	PIL	simple	Spacecraft	TrafficLight
PC_L_SI_BW	73/74 629s 1225Mb	12/12 83s 189Mb	18/18 190s 900Mb	8/8 172s 566Mb	21/21 117s 1225Mb	0/1 0s 0Mb	14/14 66s 363Mb
PC_L_MS_BW	72/74 609s 1152Mb	12/12 83s 219Mb	17/18 144s 1152Mb	8/8 189s 714Mb	21/21 115s 1020Mb	0/1 0s 0Mb	14/14 77s 629Mb
PC_L_FBI_BW	14/74 38s 218Mb	2/12 8s 120Mb	2/18 7s 109Mb	1/8 8s 218Mb	9/21 15s 55Mb	0/1 0s 0Mb	0/14 0s 0Mb
PC_L_BBI_BW	71/74 627s 1835Mb	12/12 91s 265Mb	17/18 143s 1835Mb	8/8 194s 572Mb	20/21 133s 383Mb	0/1 0s 0Mb	14/14 66s 231Mb
PC_F_SI_BW	71/74 618s 1229Mb	12/12 85s 223Mb	16/18 168s 1229Mb	8/8 174s 577Mb	21/21 121s 1049Mb	0/1 0s 0Mb	14/14 71s 621Mb
PC_F_MS_BW	71/74 610s 1037Mb	12/12 84s 170Mb	16/18 144s 1037Mb	8/8 187s 548Mb	21/21 119s 1035Mb	0/1 0s 0Mb	14/14 76s 623Mb
PC_F_FBI_BW	16/74 44s 219Mb	2/12 8s 118Mb	2/18 7s 109Mb	1/8 9s 219Mb	11/21 20s 106Mb	0/1 0s 0Mb	0/14 0s 0Mb
PC_F_BBI_BW	71/74 675s 2734Mb	12/12 92s 267Mb	17/18 124s 794Mb	8/8 205s 603Mb	20/21 187s 2734Mb	0/1 0s 0Mb	14/14 66s 210Mb
virtual best	73/74 536s 1020Mb	12/12 80s 169Mb	18/18 119s 900Mb	8/8 163s 502Mb	21/21 110s 1020Mb	0/1 0s 0Mb	14/14 65s 210Mb

Figure 5.6: RQ5: A heatmap comparing the success rates of the different refinement strategy and pruning strategy combinations.

The results show that the `FW_BIN_ITP` refinement strategy performs poorly on XSTS models. This is reaffirms the benchmarking results presented in [22], where `FW_BIN_ITP` performed similarly poorly on CFA (control flow automaton) models. As visible in Fig. 5.7,

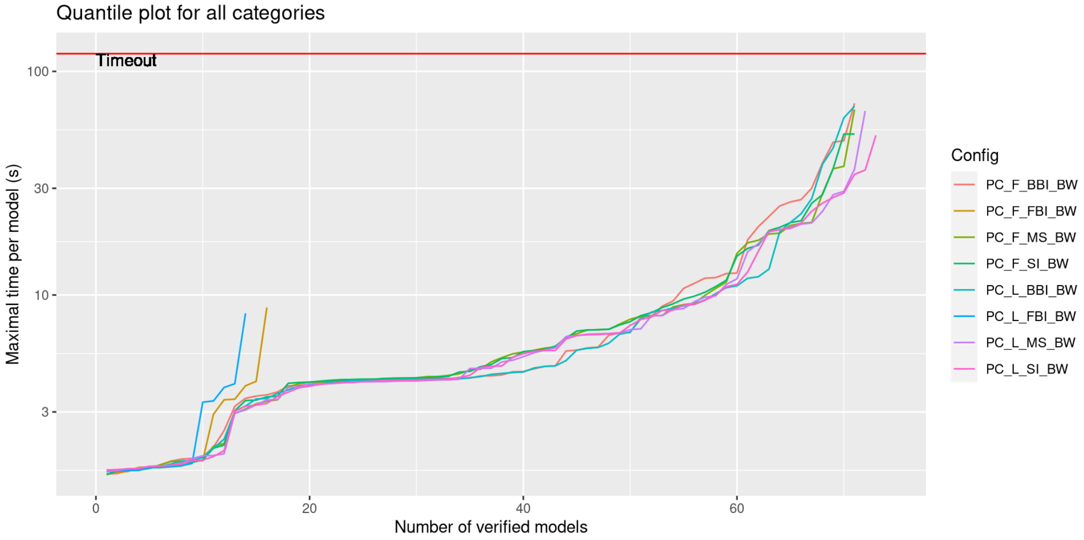


Figure 5.7: RQ5: A quantile plot comparing the different refinement strategy and pruning strategy combinations.

all other combinations performed roughly the same, except the two configurations with the FW_BIN_ITP refinement strategy.

RQ6: Product abstraction compared to other domains

In this benchmark I compared the best configuration that uses product abstraction to the best predicate abstraction configuration and the best explicit-value abstraction configuration. To achieve this, I picked the configuration for each domain that had the best overall performance, which resulted in three configurations:

- PC_L_SI_BW represents the predicate abstraction domain;
- P250_L_SI_BW represents the product abstraction domain;
- E250_L_SI_B represents the explicit-value abstraction domain.

There were 74 test cases in this run with 3 different configurations, resulting in 222 measurements.

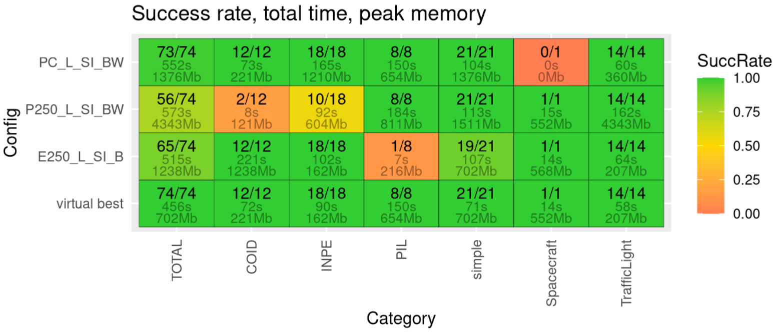


Figure 5.8: RQ6: A heatmap comparing the success rates of the different configurations.

The results shown in Fig. 5.8 indicate that there is no clear winner here. Based on the result of evaluating RQ1 it is no surprise that PC_L_SI_BW had the best overall success rate.

On the other hand, when we look at the different model sets, we can see that predicate abstraction wasn't able to verify the **Spacecraft** set, while both product and explicit-value abstraction could. Similarly, when we look at the **PIL** set, we can see that explicit-value abstraction could not verify its models (with the exception of 1 test case), while predicate and product abstraction could. What this means is that product abstraction was able to combine the strengths of its subdomains in such a manner in both directions, that allowed it to verify models that can only be verified by a single one of its subdomains. On the other hand it is also clearly visible that in certain situations, product abstraction can also perform poorer than both of its subdomains, which can be seen in the results of the **COID** and **INPE** sets. These sets call for further examination, which is future work.

Summary

The experimental evaluation of the XSTS formalism showed promising results. All models were successfully verified by at least one configuration and no run yielded an incorrect result. The benchmarks reaffirmed the belief that the performance of the analysis is highly dependant on the parameterization of the model checker, as there are significant differences in performance between the different configurations. The benchmarks also identified configurations that generally perform well, as well as configurations whose performance is poor overall and whom should be avoided. This helped us to determine a low number of possible configurations that for example Gamma should try to invoke. The benchmarks also showed that the `PROD` domain can compete with the other domains, and even perform better in certain cases.

Chapter 6

Conclusions

In this final chapter I draw the conclusions of my work and lay down possible future directions. The goal of this work was to develop a novel intermediate language that offers high-level constructs to make modeling easier, and to demonstrate its applicability on real-world industrial examples.

The results of the work are twofold. From the theoretical point of view, I developed the novel extended symbolic transition system (XSTS) formalism, an intermediate language between low-level SMT solvers and high-level engineering models that is suitable for model checkers to operate on. In the progress of developing the formalism, I evaluated semantic questions and design decisions to create a flexible formalism that can support several high-level engineering models and multiple model checking paradigms. Furthermore, I defined the formal semantics of XSTS models, adapted the CEGAR-based model checking algorithm to the formalism, and proposed novel variants that exploit the structure of XSTS.

From the practical point of view, I implemented the XSTS formalism in the Theta framework, along with a domain-specific language (using Antlr) for easy parsing of XSTS models. I also created a parameterizable command-line model checking tool that can run the aforementioned CEGAR-based analysis. My additions were included in the main development branch since the v2.0.0. release of the Theta framework. I ran an exhaustive benchmarking campaign with several models and configurations to evaluate the strenghts and weaknesses of my approach, including real-world examples provided by industry partners. My formalism and model checker was integrated as a verification backend to the Gamma Statechart Composition Framework, which offers an end-to-end hidden formal verification workflow for engineers, proving the applicability of my work.

Future work. While the XSTS formalism showed promising results, I have various plans for further improvements. I intend to introduce further composite statements into the XSTS formalism that would aid the modeling of concurrent, communicating systems. I also plan to prove the generic nature and flexibility of XSTS by defining mappings from other formalisms such as Petri nets and control flow automata. To increase the expressive power of the analysis, I intend to adapt temporal logic (e.g., LTL) model checking algorithms to the XSTS formalism. Furthermore, as part of a collaboration between several industrial partners, Gamma is already integrated in a scalable and automated cloud-based model checking service [29], currently relying on Uppaal [33] as its backend. However, we also plan to integrate Theta (via my XSTS-based analyses) as an alternate backend into the workflow.

Acknowledgements

I would like to express my acknowledgement to my supervisor, Ákos Hajdu, who has continuously provided me with guidance, valuable ideas and feedback during this work. I would also like to thank András Vörös, who has been supporting my studies and research since 2017, offered me numerous opportunities and his guidance. Furthermore, I would like to express my gratitude towards Vince Molnár and Graics Bence who helped my work with their ideas, comments and feedback.

The results presented in this work were established in the framework of the professional community of the Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the ”BME Balatonfüred Knowledge Center”, supported by EFOP 4.2.1-16-2017-00021.

Bibliography

[1] Viktória Dorina Bajkai and Ákos Hajdu. Software Model Checking with a Combination of Explicit Values and Predicates. In *Proceedings of the 26th PhD Mini-Symposium*, pages 4–7. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2019. DOI: `10.5281/zenodo.2597969`.

[2] Thomas Ball, Andreas Podelski, and Sriram Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001. DOI: `10.1007/3-540-45319-9_19`.

[3] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. DOI: `10.1007/978-3-319-10575-8_11`.

[4] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904. Springer, 2016. DOI: `10.1007/978-3-662-49674-9_55`.

[5] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013. DOI: `10.1007/978-3-642-37057-1_11`.

[6] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007. DOI: `10.1007/s10009-007-0044-z`.

[7] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007. DOI: `10.1007/978-3-540-73368-3_51`.

[8] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008. DOI: `10.1109/ASE.2008.13`.

[9] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In *Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2015. DOI: `10.1007/978-3-319-23404-5_3`.

[10] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21 (1):1–29, 2019. DOI: `10.1007/s10009-017-0469-y`.

[11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, NLD, 2009.

[12] Armin Biere, Alessandro Cimatti, Edmund M Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14.

[13] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[14] Edmund M Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. DOI: 10.1145/876638.876643.

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

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

[17] Bence Czipó, Ákos Hajdu, Tamás Tóth, and István Majzik. Exploiting hierarchy in the abstraction-based verification of statecharts using SMT solvers. In *Proceedings of the 14th International Workshop on Formal Engineering Approaches to Software Components and Architectures*, volume 245 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–45. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.245.3.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[19] Robert DeLine and K Rustan M Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[20] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[21] FTSRG. Theta framework GitHub repository. https://github.com/ftsrg/theta, 2020. Accessed: 2020-10-25.

[22] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: 10.1007/s10817-019-09535-x.

[23] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components and Systems*, volume 9688 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2016. DOI: 10.1007/978-3-319-39570-8_11.

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

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

[26] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002. DOI: 10.1145/3236950.3236969.

[27] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual.* 01 2004.

[28] Gerard J. Holzmann. The model checker Spin. 23(5):279–295, 1997. DOI: 10.1109/32.588521.

[29] 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*, 2020. DOI: 10.1145/3417990.3421407.

[30] Nafiseh Kahani and James R. Cordy. Bounded verification of state machine models. In *Proceedings of the 12th System Analysis and Modelling Conference*, SAM '20, page 23–32. ACM, 2020. DOI: 10.1145/3419804.3420263.

[31] Saul A Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[32] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. Sp 800-142. practical combinatorial testing. Technical report, Gaithersburg, MD, USA, 2010.

[33] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. DOI: 10.1007/s100090050010.

[34] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 166–182, Cham, 2014. Springer International Publishing.

[35] Kenneth L McMillan. Applications of Craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005. DOI: 10.1007/978-3-540-31980-1_1.

[36] Yael Meller, Orna Grumberg, and Karen Yorav. Verifying behavioral UML systems via CEGAR. In *Integrated Formal Methods*, pages 139–154. Springer, 2014.

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

[38] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 113–116. ACM, 2018. DOI: `10.1145/3183440.3183489`.

[39] Milán Mondok and András Vörös. Abstraction-based model checking of linear temporal properties. In Balázs Renczes, editor, *Proceedings of the 27th PhD Mini-Symposium*, pages 29–32. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2020.

[40] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[41] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. Who knows/uses what of the UML: A personal opinion survey. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 149–165, Cham, 2014. Springer International Publishing.

[42] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. DOI: `10.23919/FMCAD.2017.8102257`.

[43] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Springer, 2012. DOI: `10.1007/978-3-642-29044-2`.

Appendix

A.1 Spacecraft model

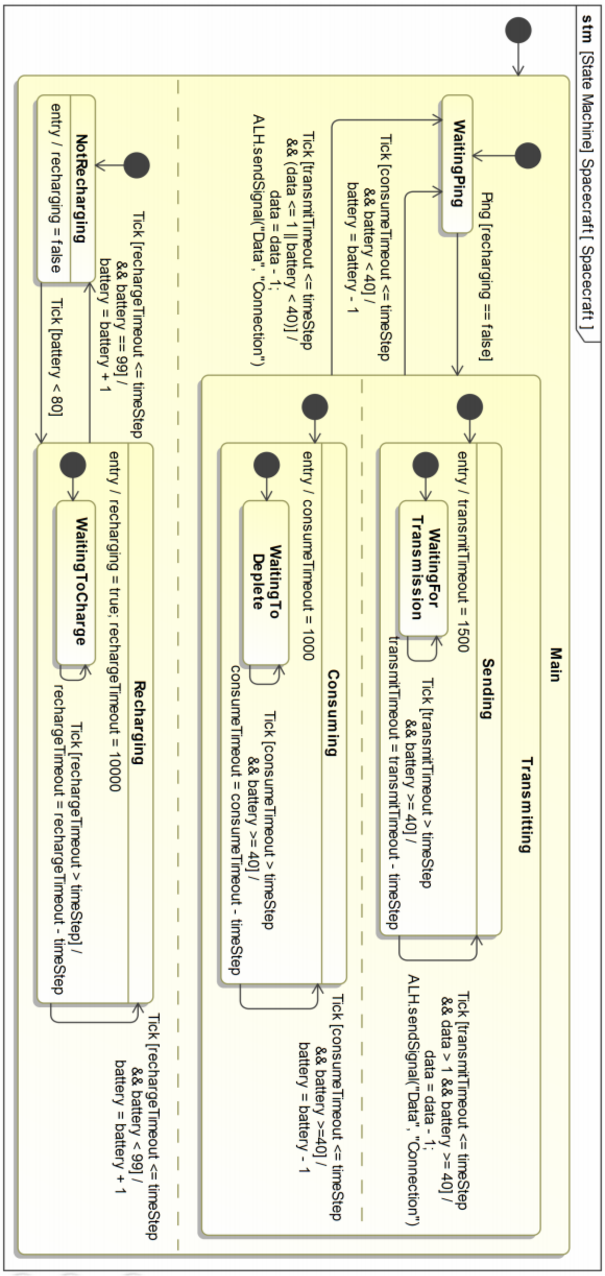

stm [State Machine] Spacecraft [Spacecraft]
Main
WaitingPing
Ping [recharging == false]
Tick [consumeTimeout <= timeStep && battery < 40] / battery = battery - 1
Tick [transmitTimeout <= timeStep && (data <= 1 || battery < 40)] / data = data - 1; ALH.sendSignal("Data", "Connection")
Transmitting
Sending
entry / transmitTimeout = 1500
WaitingFor Transmission
Tick [transmitTimeout > timeStep && battery >= 40] / transmitTimeout = transmitTimeout - timeStep
Tick [transmitTimeout <= timeStep && data > 1 && battery >= 40] / data = data - 1; ALH.sendSignal("Data", "Connection")
Consuming
entry / consumeTimeout = 1000
WaitingTo Deplete
Tick [consumeTimeout > timeStep && battery >= 40] / consumeTimeout = consumeTimeout - timeStep
Tick [consumeTimeout <= timeStep && battery >=40] / battery = battery - 1
NotRecharging
entry / recharging = false
Tick [rechargeTimeout <= timeStep && battery == 99] / battery = battery + 1
Tick [battery < 80]
Recharging
entry / recharging = true; rechargeTimeout = 10000
WaitingToCharge
Tick [rechargeTimeout > timeStep] / rechargeTimeout = rechargeTimeout - timeStep
Tick [rechargeTimeout <= timeStep && battery < 99] / battery = battery + 1

Figure A.1.1: SysML statechart model of a spacecraft component [29].

A.2 Overall success rates

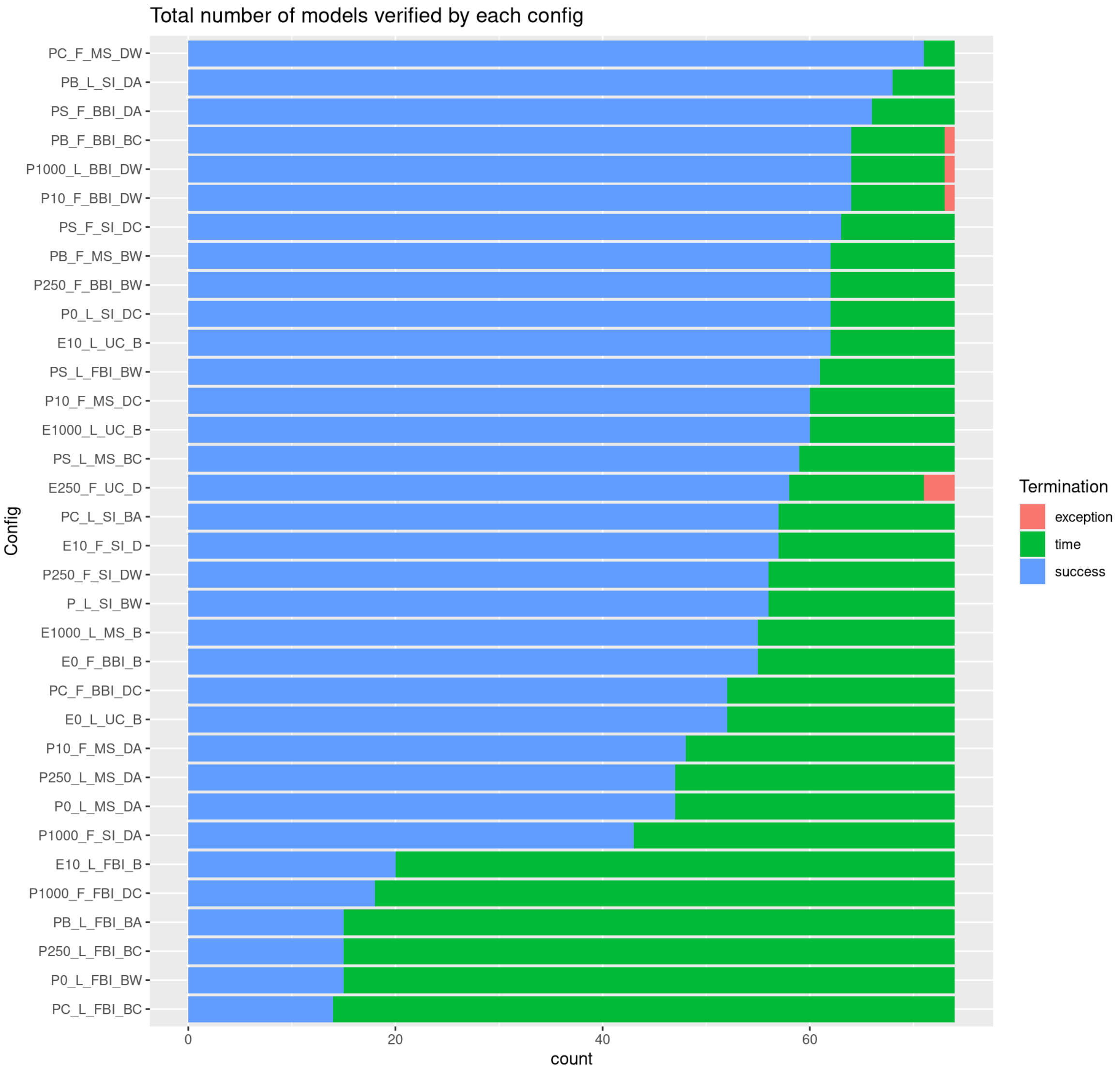


Figure A.2.1: A rowchart depicting the success rates measured in the benchmark of RQ1 (Section 5.3.5).