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

Efficient abstraction-based model checking using domain-specific information

Scientific Students' Association Report

Author:

Milán Mondok

Advisor:

Dr. Vince Molnár

2021

Contents

Kivonat

A biztonságkritikus rendszerek helyes működésének szavatolása kulcsfontosságú, mivel az azokban jelenlévő legkisebb hiba is súlyos anyagi kárral járhat, szélsőséges esetekben akár emberi életekbe is kerülhet. A formális verifikáció képes a rendszerek helyességének matematikailag precíz bizonyítására és a rejtett hibák megtalálására is, széleskörű elterjedését azonban hátráltatja nagy számításigénye. A beágyazott fejlesztők és a hardvertervező mérnökök számára mára kiterjedt formális verifikációs eszközkészlet áll rendelkezésre, ugyanez azonban a rendszertervező mérnökökről sajnos nem mondható el.

A rendszertervező mérnökök jellemzően magas szintű, összetett, heterogén modelleken dolgoznak. Egy ilyen rendszermodell ellenőrzése más jellegű kihívást jelent, mint szoftverek vagy hardvermodellek analízise - az ezeknél használt reprezentációk, algoritmusok más jellegű problémákhoz lettek kifejlesztve. Egy rendszerterv általában több heterogén komponenst is tartalmaz, így a teljes rendszer formális leírásához egy olyan közös nyelvre van szükség, amely jó kompromisszumot jelent a különböző paradigmák között. Egy ilyen nyelv megtervezése során az általánosság és a hatékonyság között egyensúlyozunk: hogyan tudunk minél több problémaspecifikus információt kihasználni az ellenőrzés során úgy, hogy az általánosságot - és ezáltal a különböző heterogén komponensek közös leírásának lehetőségét - ne veszítsük el?

Korábbi kutatási munkám során kidolgoztam a kiterjesztett szimbolikus rendszer (röviden XSTS) formalizmust, mely az első lépés volt egy ilyen nyelv megalkotásának irányába. Jelen dolgozatomban olyan kiegészítéseket és optimalizációkat mutatok be, melyekkel az XSTS nyelv és infrastruktúra képessé vált nagyméretű, ipari modellek ellenőrzésére. A nyelv kifejezőerejének növelésével szélesítettem az ellenőrizhető magas szintű modellek körét. A szorzat absztrakciós domén továbbfejlesztésével létrehoztam egy olyan dedikált absztrakt domént, mely hatékonyan képes együtt kezelni a vezérlési és általános információkat. Kidolgoztam olyan algoritmikus optimalizációkat is, melyek az adott absztrakciós szinten rendelkezésre álló információ leghatékonyabb felhasználását segítik, ezzel egyszerűsítve az algoritmus mögötti kényszermegoldónak átadott logikai kifejezéseket. Emellett javaslok olyan modellezési, illetve leképzési gyakorlatokat, melyek a legjobb teljesítményhez vezetnek adott forrásmodell esetén. A dolgozatban bemutatott kiegészítések egy jelentős és határozott lépést jelentenek a heterogén rendszertervek ellenőrzéséhez optimális absztrakciós szint megtalálásában. Eredményeim alátámasztására dolgozatomban egy olyan, SysML modellezési nyelven készült esettanulmányt is bemutatok, amely az XSTS nyelv segítségével vált verifikálhatóvá és demonstrálja a nyelv gyakorlati alkalmazhatóságát. Megközelítésem alaposabb kiértékeléséhez egy kiterjedt mérési kampányt folytattam, melynek eredményei alátámasztják az XSTS nyelv kiegészítéseinek hatékonyságát.

Az Innovációs és Technológiai Minisztérium ÚNKP-21-2 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

Az Európai Bizottság és Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal támogatásával az Arrowhead Tools projekt keretében készült (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

Abstract

Ensuring the correct functioning of safety critical systems is essential, as even the smallest mistakes can lead to significant material damage or - in extreme cases - cost human lives. Formal verification is capable of proving the correctness of the system and uncovering hidden faults. As opposed to testing, formal verification can give a mathematically sound proof of correctness, but its resource intensiveness hinders widespread application. Embedded developers and hardware design engineers can already make use of an extensive set of formal verification tools, but the same cannot be said about systems engineers sadly.

Systems engineers typically use high-level, complex, heterogeneous models. Verifying such systems poses a different challenge than the verification of software or hardware - the representations and algorithms used for these two have been developed to solve problems of a different nature. A system model typically contains multiple heterogeneous components, which means that in order to verify the complete system, a common language is required that is a good compromise between the different paradigms. When designing such a language, there is a trade-off between generality and efficiency. How can we take advantage of as much domain specific information as possible without losing generality - and thus the ability to describe the different heterogeneous components with the same language?

During my previous research I developed the Extended Symbolic Transition System (XSTS) formalism, which was the first step towards creating such a language. In this work, I present extensions and optimizations, which enabled the XSTS language and infrastructure to efficiently verify large-scale industrial models. Increasing the expressive power of the language widened the scope of verifiable high-level engineering models. By improving the product abstraction domain, I created a dedicated combined abstract domain, which is capable of efficiently tracking control and general information together. I also developed algorithmic optimizations that help make the most efficient use of the information available at a given level of abstraction, resulting in a significant simplification of the logical formulas passed to the constraint solver underlying the algorithm. In addition, I propose modeling and transformation best practices that lead to better performance in certain source models. The extensions presented in this work constitute a significant and definite step towards finding the most suitable level of abstraction for the verification of heterogeneous system models. To prove the practical applicability of my approach, I present an industrial case study created in the SysML modeling language, which became verifiable thanks to the additions made to the XSTS formalism and model checker. I evaluated my approach in an extensive benchmarking campaign, which further underlined the efficiency and necessity of the extensions presented in this work.

Supported by the ÚNKP-21-2 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund.

This research was funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

Chapter 1

Introduction

The correct functioning of safety critical systems is essential, as even the smallest mistakes present in them can lead to significant material damage or cost human lives in extreme cases. Ensuring safe behaviour is a challenging task, engineers often and easily make errors that remain unnoticed for years or even decades, to which the ever growing complexity of the designed systems only adds. Formal verification is capable of proving the system's correctness and uncovering hidden faults. As opposed to testing, formal verification can give a mathematically sound proof of the correctness of the system. Embedded developers and hardware design engineers can already make use of an extensive set of formal verification tools, the same cannot be said about systems engineers sadly.

Systems engineers typically use high-level, complex, heterogeneous models, the like of which we can come across in the automotive, aerospace, and the space industries. Verifying such systems poses a different challenge than the verification of software systems or hardware models - the representations and algorithms used for the latter two were developed to solve problems of a different nature. A system model typically contains multiple heterogeneous components, requiring a common language that is a good compromise between the different paradigms. The widespread use of formal methods is also hindered by its high computational complexity, and the fact that its application requires deeper mathematical knowledge. Thus, the primary unsolved problems of the field remain improving the performance of the verification algorithms and developing tools that are capable of so-called "end-to-end hidden" verification, i.e. can work with the click of a button, integrated into widespread modeling tools, requiring no special knowledge, automatically.

During my previous research I developed the Extended Symbolic Transition System (XSTS) formalism, which was the first step towards creating such a language. In this work, I present extensions and optimizations, which enabled the XSTS language and infrastructure to efficiently verify large-scale industrial models. Increasing the expressive power of the language widened the scope of verifiable high-level engineering models. By improving the product abstraction domain, I created a dedicated combined abstract domain, which is capable of efficiently tracking control and general information together. I also developed algorithmic optimizations that help make the most efficient use of the information available at a given level of abstraction, resulting in a significant simplification of the logical formulas passed to the constraint solver underlying the algorithm. In addition, I propose modeling and transformation best practices that lead to better performance in certain source models. The extensions presented in this work constitute a significant and definite step towards finding the most suitable level of abstraction for the verification of heterogeneous system models. To prove the practical applicability of my approach, I present an industrial case study created in the SysML modeling language,

which became verifiable thanks to the additions made to the XSTS formalism and model checker. I evaluated my approach in an extensive benchmarking campaign, which further underlined the efficiency and necessity of the extensions presented in this work.

The structure of this work is the following. In Chapter 2, I present the theoretical and practical foundations that my work builds upon. In Chapter 3, I present new language constructs that bring the XSTS models closer to the high-level engineering models, which makes both the transformation process easier, and allows for more efficient model checking by exploiting the additional information that is encoded in the model using the algorithm presented in Chapter 4. In Chapter 5, I present optimizations of the product abstraction domain. In Chapter 6, I evaluate my work, and in Chapter 7, I draw the conclusions.

Chapter 2

Background

In this chapter, I describe the theoretical and practical foundations that this work builds upon. In Section 2.1, I present model checking, and in Section 2.2, I describe the high-level SysML, and the low-level STS formalisms. In Section 2.3, I give a brief summary of the XSTS formalism, an intermediate language I created during my earlier research. In Section 2.4, I introduce the CEGAR algorithm, which is an abstraction-based iterative model checking algorithm. In Section 2.5, I present the Theta framework, an open source model checking framework that I implemented my model checker in. In Section 2.6, I give a brief summary about related work.

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 [8] available (such as [13]) that can answer various queries, e.g., is a formula satisfiable, does a formula imply another formula, etc. I use the following notation [16] 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

Model checking [11] is a formal verification method that aims to exhaustively explore all possible behaviours of an input model to see if any of them violate a given requirement. It is formal in the sense that both the formalisms and the algorithms used must have precise mathematical definitions. In exchange for this the results are mathematically precise and definite (as opposed to *testing* for example). An overview of model checking is seen in Fig. 2.1. A model checker receives a formal model and a formal requirement as inputs. If the model violates the requirement, then the model checker returns *unsafe* and a counterexample demonstrating what actions are required to violate the property. Otherwise, the model checker returns *safe* as a result.

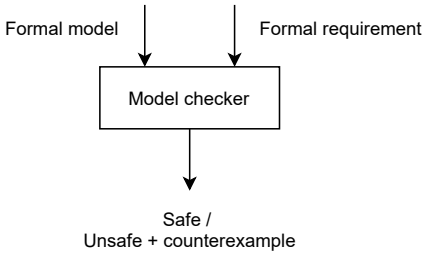

Figure 2.1: An overview of model checking.

Input models Only formalisms that have precise formal syntax and semantics can be used for model checking. Model checking algorithms are usually defined on lower-level mathematical formalisms, such as *Kripke structures* [11], *control flow automata* (CFA) [16] or *extended symbolic transition systems* (XSTS) [23]. The verification of higher-level formalisms such as *statecharts* [18], *activity diagrams* [25], or programming languages like *C* is usually done by defining a translation to a lower-level formalism (e.g. *C* programs are usually transformed to *control flow automata* for model checking [1]). The focus of this work is the verification of higher-level engineering models through a translation to the XSTS intermediate language.

Requirement properties Various different types of requirement properties are used in model checking depending on the domain, from simple *assertions* or *null-division properties* to complex *temporal logic* (typically LTL or CTL [11]) expressions. The focus of this work are *safety properties*, which specify a set of erroneous states. If any of the erroneous states can be reached from the initial state, then the model is *unsafe*, otherwise it is *safe*.

State space explosion One of the biggest challenges of model checking is the so-called *state space explosion* [10] effect: as the number of state variables in a system grows, the size of its state space grows at least exponentially, quickly reaching a level, where explicitly exploring all states of a system becomes infeasible. To combat this, model checkers usually have to employ some sort of clever representation or simplification. One such simplification technique is *abstraction*, which is the focus of this work.

2.2 Formalisms

In this section, I present two modeling formalisms that are in stark contrast of each other. The SysML formalism presented in Section 2.2.1 is a high-level formalism that is easy-to-use for systems engineers. The STS formalism presented in Section 2.2.2 that only consists of two logical formulas is one of the simplest model checking formalisms that has practical use.

2.2.1 SysML

The *OMG Systems Modeling Language* (SysML) [24] is a general purpose modeling language for systems engineering. Having originally started out as a dialect of the *Unified Modeling Language* (UML), SysML is an extension of a subset of the UML language. SysML provides various easy-to-use constructs for the modeling of not only software sys-

tems, but hardware, cyber-physical and even less technical systems like processes, personnel or facilities.

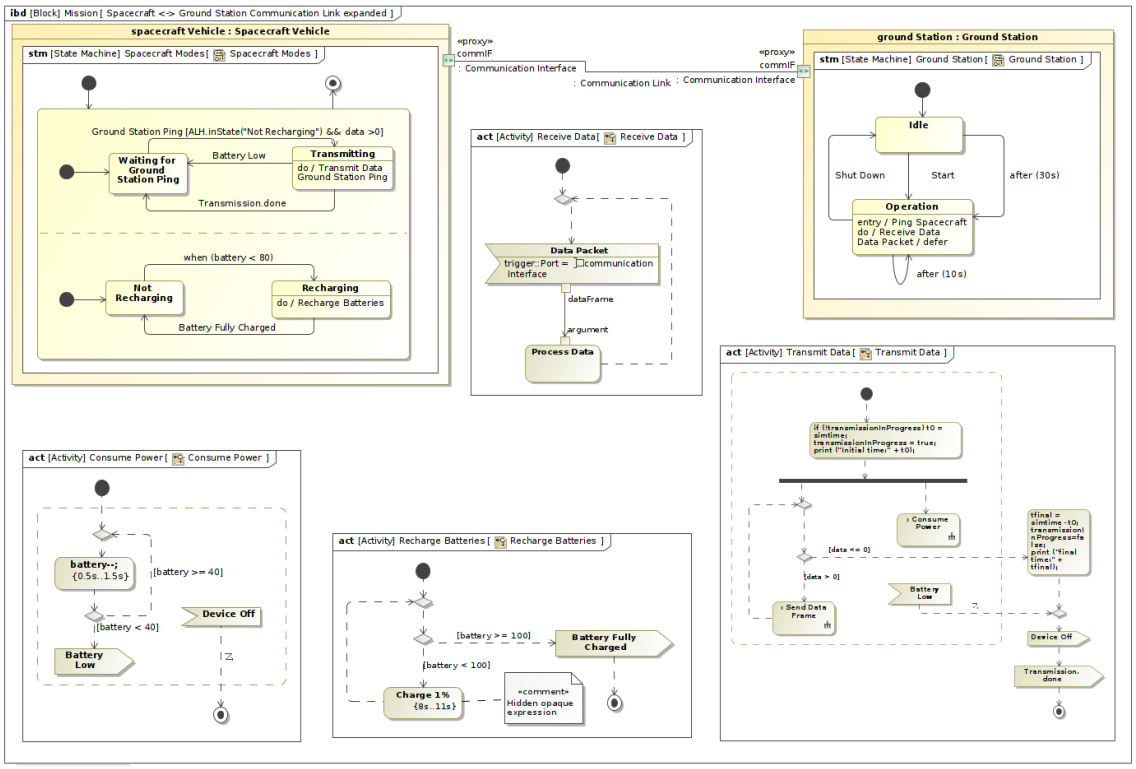

Figure 2.2: A heterogenous SysML model example.

SysML offers various formalisms to support the modeling of heterogeneous systems, such as the *requirement diagram*, the *sequence diagram* or structural diagrams like the *block definition diagram* and the *internal block diagram*. The *statechart* formalism, introduced by Harel in 1987 [18], is commonly used to model complex reactive systems, while *activity diagrams* are used to model control and data flow. System models are typically compositions of heterogeneous components that are modeled in different formalisms. See Fig. 2.2 for an example of a heterogenous system model (a larger copy of the image is included in Appendix A.1 for better readability). The system consists of two communicating statecharts, whose actions are defined in activity diagrams. Similar modeling approaches are often used in the safety critical automotive, aerospace and railway industries, which makes the verification of such models particularly important.

2.2.2 Symbolic transition systems

A *symbolic transition system* [17] offers a compact way of representing the set of *states*, *transitions* and *initial states* using first order logic (FOL) formulas. 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 the transition formula over $V \cup V'$, which describes the transition relation between the current state (V) and the next state (V'). For example if $V = \{x, y\}$, then $Tran \equiv x' = x + 1 \wedge y' = y$ describes a transition, which increases the value of x by 1, and leaves y unchanged.

- *Init* is the initial state formula, which describes the values of the variables in the initial states. For example, if $Init \equiv x = 0$ and $V = \{x, y\}$, then there are infinitely many initial states (one for each possible value of y), and in all of them the value of x is 0.

A *concrete state* $s \in S \subseteq D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$ is an interpretation that assigns a value $s(v) \in D_v$ to each variable $v \in V$ of its domain D_v. A concrete state can also be regarded as a tuple of values $(s(v_1), s(v_2), ..., s(v_n))$. For example if $V = \{x, y\}$, then $(x = 0, y = 1)$ and $(x = 1, y = 4)$ are possible states of the system. A state with a prime (s') or index ($s^{\langle i \rangle}$) assigns values to V' or $V^{\langle i \rangle}$, respectively. Given a FOL formula φ let $s \models \varphi$ denote that assigning the variables in φ with the values in s evaluates to *true*. For example $(x = 0, y = 0) \models x \geq 0$. Similarly, let $s \not\models \varphi$ denote that φ in s evaluates to *false*.

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

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

2.3 Extended symbolic transition systems

Systems engineers typically work on easy-to-use high-level formalisms like SysML. Model checking algorithms on the other hand are usually defined on low-level mathematical formalisms like the STS formalism. The aim of the *extended symbolic transition system* (XSTS) [23] formalism is to bridge this gap by introducing higher-level constructs that make the mapping of high-level formalisms easier, but are expressible using logical formulas, and thus still verifiable by SMT-based model checking algorithms.

An *extended symbolic transition system*[1] [23] is a tuple $XSTS = (V, V_C, Init, Tr, En)$, where:

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

[1] I originally defined the XSTS formalism with 3 transition sets in [23], but I am going to omit the initialization transition set (In) in this work for simplification as equivalent behaviour can be expressed with other constructs of the language.

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

In STS the transition relation is described with a single FOL formula. While this offers great flexibility, the complex behaviour of engineering models is hard to translate to such a low-level formalism. In XSTS a different approach is followed, where transitions can be described using *operations*. These are higher-level constructs that allow us to encode more information in the model that we can exploit during model checking.

2.3.1 Operations

Operations $op \in Ops$ describe the transitions between 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 not at all. The XSTS formalism [23] the following operations:

- *Basic* operations contain no inner (nested) operations. I define the following basic operations:
  - *Assignments* of the form $v := \varphi$ assign a value to a single variable. Here $v \in V$ is a variable, φ is an expression of type D_v, and $var(\varphi) \subseteq V$. The semantics of assignments are the following: $tran(v := \varphi) \equiv v' = \varphi \wedge \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$. This formula expresses that the value of v in the successor state is φ, and all other variables stay unchanged. For example, if $V = \{x, y\}$, then $tran(x := 1) \equiv x' = 1 \wedge y' = y$;
  - *Assumptions* of the form $[\varphi]$, where φ is predicate with $var(\varphi) \subseteq V$. Assumptions act a guards, they can only be executed if their condition φ evaluates to *true*. The semantics are the following: $tran([\varphi]) \equiv \varphi \wedge \bigwedge_{v \in V} v' = v$. On other words, assumptions check a condition and leave the values of all variables unchanged. For example, if $V = \{x, y\}$, then $tran([y < 0]) \equiv y < 0 \wedge x' = x \wedge y' = y$.
  - *Havocs* of the form $\mathsf{havoc}(v)$, where $v \in V$. Havocs are nondeterministic assignments. where a variable v gets assigned a nondeterministic value of its domain D_v. The semantics are $tran(\mathsf{havoc}(v)) \equiv \bigwedge_{v_i \in V\{v\}} v_i' = v_i$, which means that v can have any value, while all other variables keep their value. For example, if $V = \{x, y\}$, then $tran(\mathsf{havoc}(x)) \equiv y' = y$.
- *Composite* operations contain other operations and can be used to model complex behaviour. Note that the execution of these operations is still atomic in the sense, that they either get executed in their entirety or not at all. I define the following composite operations:
  - *Sequences* of the form $op_1, op_2, ..., op_n$, where $op_i \in Ops$ are lists of operations that are executed in order after each other. Each operation of the sequence operates on the result of the previous operation. For example, $x := 2, [y < 0]; x := x + 1$ is a sequence that contains 3 inner operations.

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

By abusing the notation, let's allow operations $op \in Ops$ to appear as FOL formulas, by replacing them with their semantics $tran(op)$. This will allow us to use cleaner notation later. 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 $star(\varphi, 2) \equiv x^{***} = x^{**} + 1$.

I define the semantics of sequences as the following: $tran(op_1, op_2, ..., op_n) \equiv \bigwedge_{v \in V} v^* = v \wedge \bigwedge_{i=1}^{n} star(op_i, i) \wedge \bigwedge_{v \in V} v' = star(v^*, n)$. This definition consists of 3 parts:

- In the first part, $\bigwedge_{v \in V} v^* = v$, we store the value of each non-starred variable $v \in V$ in its starred version $v*$;
- In the second part, $\bigwedge_{i=1}^{n} star(op_i, i)$ 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 values of the variables from the successor state of the last suboperation in the primed versions of the variables, which means that the result of the last operation will be the result of the sequence.

As an example, let's consider the sequence $x := 2, [y < 0]; x := x + 1$:

- The first part in this case will be $x^* = x \wedge y^* = y$;
- The second part will be $(x^{**} = 2 \wedge y^{**} = y^*) \wedge (y^{**} < 0 \wedge x^{***} = x^{**} \wedge y^{***} = y^{**}) \wedge (x^{****} = x^{***} + 1 \wedge y^{****} = y^{***})$;
- The third part will be $x' = x^{****} \wedge y' = y^{****}$.

All 3 parts together: $x^* = x \wedge y^* = y \wedge (x^{**} = 2 \wedge y^{**} = y^*) \wedge (y^{**} < 0 \wedge x^{***} = x^{**} \wedge y^{***} = y^{**}) \wedge (x^{****} = x^{***} + 1 \wedge y^{****} = y^{***}) \wedge x' = x^{****} \wedge y' = y^{****}$.

The semantics of nondeterministic choices is the following: $tran(\{op_1\}$ 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 achieve exclusivity among the branches: it ensures that only one branch is executed, because two branches cannot be true at the same time if the $temp$ variable is assigned different values in them.

As an example, let's consider the choice $\{x := 0\}$ or $\{[y < 0]\}$ or $\{y := 1\}$. The transition formula will consist of 3 parts, one for each branch:

- $temp = 1 \wedge x' = 0 \wedge y' = y$ for $x := 0$;

- $temp = 2 \wedge y < 0 \wedge x' = x \wedge y' = y$ for $[y < 0]$;
- $temp = 3 \wedge y' = 1 \wedge x' = x$ for $y := 1$.

All 3 parts together: $(temp = 1 \wedge x' = 0 \wedge y' = y) \vee (temp = 2 \wedge y < 0 \wedge x' = x \wedge y' = y) \vee (temp = 3 \wedge y' = 1 \wedge x' = x)$.

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.

2.3.2 State space

A *concrete data state* $c \in D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$ is an interpretation that assigns a value $c(v) \in D_v$ to each variable $v \in V$ of its domain D_v. Concrete data states can also be regarded as tuples $(c(v_1), c(v_2), ..., c(v_n))$. For example, if $V = \{x, y\}$, then $(x = 1, y = 0)$ 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* $s = (c, e)$ is a pair of a concrete data state $c \in D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$, and a flag $e \in \{\text{LT}, \text{LE}\}$, which stores whether the last executed transition was from the Tr or the En set (LT corresponding to "last was from Tr" and LE corresponding to "last was from En"). The initial states are $\{(c, e) \mid c \models Init \wedge e = \text{LT}\}$, i.e. concrete states, whose concrete data state satisfies the initial state formula, and whose e flag is LT. The intent behind introducing the e flag is to limit the number of executable transitions in the states of the system and ensure that the transition sets follow each other in the desired order along all execution paths.

The set of available transitions in any state (c, e) depends on the value of the e flag. If the value of e is LT, then transitions from the En set will be able to fire, and the value of the flag in the successor state will be LE. Analogously, if the value of e is LE in a state, then transitions from Tr will be able to fire, and the value of e will be LT in the successor state. Formally, a transition exists between two states (c, e) and (c', e'), iff $op_{En} \in En$ exists with $(e = \text{LT} \wedge e' = \text{LE} \wedge (c, c') \models op_{En})$ or if $op_{Tr} \in Tr$ exists with $(e = \text{LE} \wedge e' = \text{LT} \wedge (c, c') \models op_{Tr})$.

A concrete path $\sigma = (c_1, e_1), op_1, (c_2, e_2), op_2, ..., op_{n-1}, (c_n, e_n)$ is a finite, alternating sequence of concrete states and operations, iff $c_1 \models Init \wedge e_1 = \text{LT}$ (i.e. the first state is an initial state), $\bigwedge_{1 \leq i < n} e_i \neq e_{i+1}$ (i.e. the value of the e flag alternates between consecutive states) and $(c^{\langle 1 \rangle}, c^{\langle 2 \rangle}, ..., c^{\langle n \rangle}) \models \bigwedge_{1 \leq i < n} op^{\langle i \rangle}$ (i.e. the interpretations satisfy the semantics of the operations), where for every $1 \leq i < n$, $op_i \in En$ if i is odd, and $op_i \in Tr$ if i is even. A concrete state (c, e) is *reachable* if a concrete path $\sigma = (c_1, e_1), (c_2, e_2), ..., (c_n, e_n)$ exists, so that $c = c_n$ and $e = e_n$ for some n.

2.3.3 XSTS grammar

To enable easy parsing of XSTS models, I defined an Antlr grammar for the XSTS language. The primary goal of this language is to aid the transformation of high-level engineering models to XSTS models by defining a textual interface between modeling tools and the XSTS model checker.

Types

XSTS supports the `integer` and `boolean` types by default. There is also an option to declare custom types, which are similar to enums in programming languages. Custom

types can be declared with the following syntax, where <name> is the name of the type, and <literal> are the literals of the type:

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

For example, a type for storing colours can be declared with the following line:

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

Variable declarations

Global variables can be declared with the following syntax, where <name> is the name of the variable and <type> is the type:

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

Optionally an initial value can be given as well, which will appear in the init formula *Init* if specified:

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

Control variables can be declared with the `ctrl` keyword:

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

Some examples for variable declarations can be seen below:

```
var a : integer
var b : boolean = true
ctrl var c : Colour = RED
```

Expressions

Expressions in XSTS are defined with the following grammar:

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

Operations

Assumptions have the following syntax, where <expr> is a boolean expression:

```
assume <expr>
```

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

```
<varname> := <value>
```

Havocs have the following syntax, where <varname> is a variable:

```
havoc <varname>
```

Sequence operations have the following syntax, where <operation> are arbitrary operations:

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

Choice operations have the following syntax, where <operation> are arbitrary operations:

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

An example illustrating all operations can be seen below:

```
choice {
  havoc x
  assume y > x
  choice {
    z := 1
  } or {
    z := 0
  }
} or {
  assume x > 0
}
```

Transition sets

The inner transition set of the model can be defined with the syntax below. Each branch is interpreted as a transition, <operation> are arbitrary operations.

```
tran {
  <operation>
} or {
  <operation>
} or
. . .
or {
  <operation>
}
```

The environmental transition set is defined similarly:

```
env {
  <operation>
} or {
  <operation>
} or
. . .
or {
  <operation>
}
```

Structure of the model

The structure of an XSTS model is the following:

```
<type declarations>
<variable declarations>

<inner transition set>

<environmental transition set>
```

Below is a simple example from [23] illustrating the syntax of an XSTS model. This XSTS model was generated from the statechart in Fig. 2.4 by the Gamma Statechart Composition Framework [22].

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

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)))
}

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

Figure 2.3: A simple XSTS model generated from the statechart model in Fig. 2.4.

Note how the incoming and outgoing events are transformed to boolean variables and are handled in environmental transitions. Regions are expressed with custom types, and transitions are transformed to inner XSTS transitions.

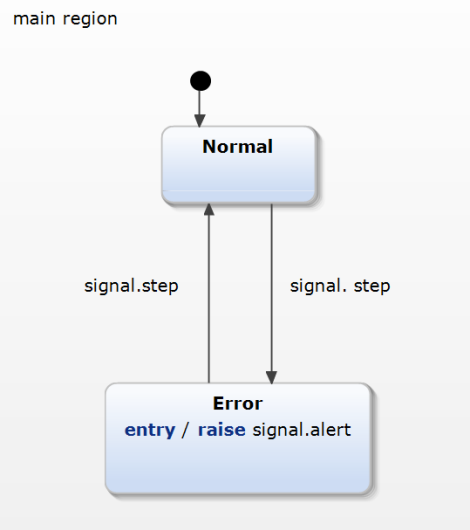

Figure 2.4: A simple statechart model.

2.4 CEGAR

State space explosion often hinders successful verification as the state space of the models is usually so large that its explicit exploration is impossible [10]. The idea behind *abstraction* [9] is to perform the model checking on a simplified model that has a smaller state space. This means that we construct an abstract state space by merging concrete states and treating them as if they were a single state. When using over-approximating abstract domain, the abstract model contains all possible behaviours of the concrete model, but can also contain additional behaviour that isn't present in the original model. This means that if no counterexamples are found in the abstract model, then the concrete model doesn't contain any either. However, all abstract counterexamples found in the abstract state space have to undergo further analysis to determine if they are present in the concrete state space as well. If the analysis finds that the counterexample was spurious, then the abstraction has to be refined accordingly. This iterative cycle of abstraction and refinement phases is called *counterexample-guided abstraction refinement* (CEGAR).

2.4.1 Abstraction

Abstraction [16] is defined based on an *abstract domain* D, a set of *precisions* Π, a *transfer function* T, and an *init function* I. An abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$, where:

- S is a (possibly infinite) 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;
- $\mathsf{expr} : S \mapsto FOL$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) with a FOL formula.

By abusing the notation, let's allow abstract states $s \in S$ to appear as FOL formulas by automatically replacing them with their meaning, $\mathsf{expr}(s)$. This will allow us to use cleaner notation later.

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

In the following, I introduce two pure abstract domains - explicit value abstraction and predicate abstraction, and a simple product domain which combines the two.

Predicate abstraction

In *predicate abstraction* [16], instead of tracking the values of the variables explicitly, we only track if certain predicates (defined by the current precision) hold in a state or not. The motivation behind this is that in many cases in order to prove a property we don't need to know the exact value of a variable, only whether some condition applies to it or not. Predicates are logical expressions that can contain variables of the model, for example $(x > 0)$, $(x < y)$, or *true*.

In *Boolean predicate abstraction* [16] an abstract state $s \in S$, is an arbitrary Boolean combination (negation, disjunction, conjunction, etc.) of FOL predicates. The top and bottom elements are $\top \equiv$ *true* and $\bot \equiv$ *false*, respectively. The partial order corresponds to implication, i.e. $s_1 \sqsubseteq s_2$ if $s_1 \Rightarrow s_2$, for $s_1, s_2 \in S$. For example, if s_1 is $(x > 0) \wedge (y \neq 2)$, and s_2 is $(x > 0)$, then $s_1 \sqsubseteq s_2$. The partial order will be used to express that an abstract state "covers" another abstract state, i.e. $s_1 \sqsubseteq s_2$ informally means that s_2 expresses all concrete states and behaviours that s_1 does. This information can be exploited during state space calculation, because if all successors of s_2 were calculated, then there is no need to calculate the successors of s_1, because all successors of s_1 appear among the successors of s_2. 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 currently tracked by the algorithm. For example, $\pi = \{(x > 0), (y \neq 2)\}$. The result of the transfer function $T(s, op, \pi)$ is the strongest Boolean combination of predicates that is entailed by the source state s and the operation op. For example, if $s = (x > y)$, $\pi = \{(x > y)\}$, then $T(s, x := x - 1, \pi) =$ *true*, and $T(s, x := x+1, \pi) = (x > y)$. This can be calculated using an SMT solver by assigning a fresh propositional variable v_i to each predicate $p_i \in \pi$ and enumerating all satisfying assignments of the formula $s \wedge op \wedge \bigwedge_{p_i \in \pi}(v_i \leftrightarrow p_i')$. For each satisfying assignment, a conjunction is formed by taking predicates with positive variables and taking the negation of predicates with negative variables. The successor state s' is the disjunction of all such conjunctions. 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*, which is calculated similarly.

In *Cartesian predicate abstraction* [16] 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 the init function are defined differently, $T(s, op, \pi)$ yields the strongest conjunction of predicates from the precision that is entailed by the source state s and the operation op, i.e. $T(s, op, \pi) = \bigwedge_{p_i \in \pi}\{p_i \mid s \wedge op \Rightarrow p_i'\} \wedge \bigwedge_{p_i \in \pi}\{\neg p_i \mid s \wedge op \Rightarrow \neg p_i'\}$. Analogously, the init function yields the strongest conjunction of predicates from the pre-

cision that is consistent with the init formula *Init*. Cartesian predicate abstraction is not as precise as Boolean predicate abstraction, but more efficient.

Explicit-value abstraction

In *explicit-value abstraction* [16], the state space is reduced by marking only a subset of the variables as "interesting", and only tracking the value of those variables. For example, if a model has two variables, x and y, then we might only track the value of x. The motivation behind this domain is that we often don't have to know the exact values of all variables to decide whether a property holds. The goal when using this domain is to find the smallest subset of variables (via transitive dependencies) that is enough to prove or refute a property.

In explicit-value abstraction, an abstract state $s \in S$ is an abstract variable assignment mapping each variable $v \in V$ to its domain extended with top and bottom values (i.e. $Dv \cup \{\top_{d_v}, \bot_{d_v}\}$). The top element $\top$ with $\top(v) = \top_{d_v}$ holds no specific value for any $v \in V$ (it represents an unknown value). The bottom element $\bot$ with $\bot(v) = \bot_{d_v}$ means no assignment is possible for any $v \in V$. For example, if $V = \{x, y, z\}$, then $(x = 0, y = 1, z = \top_{d_v})$ is an abstract state, where the value of z is unknown. If the domain of z is infinite, then this abstract state represents infinitely many concrete states. The partial order $\sqsubseteq$ is defined as follows: $s_1 \sqsubseteq s_2$ if $s_1(v) = s_2(v)$ or $s_1(v) = \bot_{d_v}$ or $s_2(v) = \top_{d_v}$ for each $v \in V$. For example, $(x = 0, y = 0) \sqsubseteq (x = 0, y = \top_{d_y})$. The expression function is defined as follows: $\mathsf{expr}(s) \equiv \mathit{true}$ if $s = \top$, $\mathsf{expr}(s) \equiv \mathit{false}$ if $s(v) = \bot_{d_v}$ for any $v \in V$, and $\mathsf{expr}(s) = \bigwedge_{v \in V, s(v) \neq \top_{d_v}} v = s(v)$. For example, if $s = (x = 0, y = \top_{d_v})$, then $\mathsf{expr}(s) = (x = 0)$.

A precision $\pi \in \Pi$ is a subset of the variables $\pi \subseteq V$ that is currently tracked by the algorithm. The transfer function is $T(s, op, \pi) = \{s' \mid s \wedge op \Rightarrow \bigwedge_{v \in \pi} v' = s'(v)\}$. Informally, s' is a successor of s with respect to the precision π and the operation op, if assigning the values in s to the non-primed, and the values in s' to the primed variables in the formula $s \wedge op$ results in a satisfying assignment. The init function is defined as $I(\pi) = \{s \mid \mathit{Init} \Rightarrow \bigwedge_{v \in \pi} v = s(v)\}$. Informally, s is an initial state, if substituting the variables in *Init* with the values assigned to them in s evaluates to *true*.

Product abstraction

It quickly becomes clear to anyone applying abstraction in practice that each abstract domain has its strengths and limitations. Product abstraction handles loops and counting with predicates difficulty, while explicit-value abstraction struggles if there is too much nondeterminism present in the model. Product abstraction aims to tackle this problem by combining the two domains.

The approaches of Beyer et al. [5, 6] combine explicit-value model checking and abstraction with algorithms that switch between the domains dynamically. Bajkai et al. present an approach in [3] that tracks all variables explicitly by default and introduces predicates whenever a variable's value can't be determined explicitly.

In a product abstraction domain with explicit-value abstraction and predicate abstraction as subdomains, the state space is the product of the state space of the explicit-value and predicate state spaces: $S = S_E \times S_P$, where S_E and S_P are the explicit-value and predicate state spaces, respectively. An abstract state $s = (s_E, s_P)$ is a pair of an explicit-value state and a predicate state, where $s_E \in S_E$ and $s_P \in S_P$. The partial order is defined

as follows: $(s_{E_1}, s_{P_1}) \sqsubseteq (s_{E_2}, s_{P_2})$ iff $s_{E_1} \sqsubseteq s_{E_2}$ and $s_{P_1} \sqsubseteq s_{P_2}$. The expression function expr yields the conjunction of the expression functions of the subdomains: $\mathsf{expr}((s_E, s_P)) \equiv \mathsf{expr}_E(s_E) \wedge \mathsf{expr}_P(s_P)$, where expr_E and expr_P are the expression functions of the explicit-value and the predicate domains, respectively. A precision $\pi = (\pi_E, \pi_P) \in \Pi_E \times \Pi_P$ is a pair of an explicit-value precision $\pi_E \in \Pi_E$ and a predicate precision $\pi_P \in \Pi_P$, where Π_E, Π_P are the precision sets of the explicit-value and the predicate domains, respectively.

The transfer function $T(s, op, \pi)$ uses the transfer functions of the subdomains $T_E(s_E, op, \pi_E)$ 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, then removes the invalid states where the semantics of the two substates contradict each other: for example if $s_E = (x = 0, y = 0)$ and $s_P = (x > y)$. This filtering is done using the so-called *strengthening-operator* $\mathsf{strengthen}$, which removes a product state $s = (s_E, s_P)$ if $\mathsf{expr}(s)$ is unsatisfiable. This can be decided with an SMT solver. Formally, $T((s_E, s_P), op, (\pi_E, \pi_P)) \equiv \mathsf{strengthen}(T_E(s_E, op, \pi_E) \times T_P(s_P, op, \pi_P))$.

The main weakness of this approach is that there is no information-exchange between the two domains: the information present in s_E can't be used by T_P, and similarly the information present in s_P can't be used by T_E. Consider the following scenario as an example: let's assume $s_E = (x = 0)$ and $s_P = (y > x)$, $\pi_E = \{x\}$ and $\pi_P = \{(y > x)\}$ and $op = [y < 0]$. When looking at the information present in both domains, we can clearly see that op can't executed, because if x is 0 and $y > x$, then $y > 0$, which contradicts the assumption $[y < 0]$. However, the transfer functions of the subdomains will only see either s_E or s_P, and will calculate states that don't get filtered by the strenghening-operator as the don't contradict each other: $(x = 0), (y > x)$ and $(x = 0), !(y > x)$. This can lead to scenarios where the analysis gets stuck in an infinite loop where the refinement will always return the same precision but the abstraction process won't exploit all the available information.

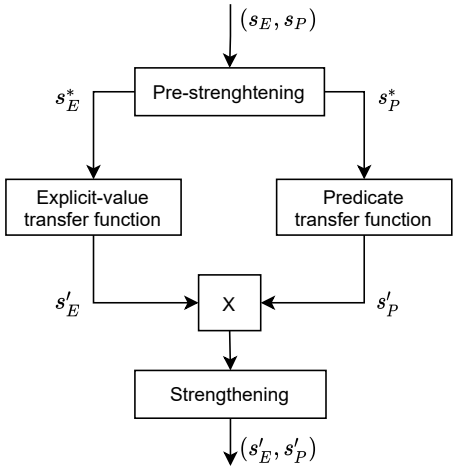

Figure 2.5: An overview of the product transfer function extended with the pre-strengthening operator.

In [23], I presented an improved product transfer function that enables information exchange in one direction: the predicate transfer function T_P can exploit the information present in the explicit state s_E. The *pre-strengthening* operator $\mathsf{ps} : S \mapsto S_E \times S_P$ takes a product state as input and returns a pair of a strengthened explicit state and strengthened predicate state, that can be used as inputs to the black-box subdomain transfer functions. I proposed the following pre-strengthening operator: $\mathsf{ps}((s_E, s_P)) \equiv$

$(s_E, \{s_P^* \mid \mathsf{expr}(s_P^*) = \mathsf{expr}_E(s_E) \wedge \mathsf{expr}_P(s_P)\})$. Informally, this operator returns the unchanged explicit state, and a predicate state that is constructed by forming the conjunction of the expressions of the two input states and interpreting it as a predicate state. For example, if $s_E = (x = 0)$ and $s_P = (y > x)$, then $\mathsf{ps}((s_e, s_P)) = ((x = 0), (x = 0) \wedge (y > x))$. The strengthened predicate state returned by the operator contains the information present in s_E as well. The transfer function extended with the pre-strengthening operator is $T_{\mathsf{ps}}((s_E, s_P), op, (\pi_E, \pi_P)) = \{T_E(s_E^*, op, \pi_E) \times T_P(s_P^*, op, \pi_P) \mid (s_E^*, s_P^*) = \mathsf{ps}(s_E, s_P)\}$. Informally, the extended transfer function feeds the outputs of the pre-strengthening operator as inputs to the black-box transfer functions and returns the Cartesian product of the results. An overview of the extended transfer function can be seen in Fig. 2.5.

The approach I presented in [23] is a static in the sense that it tracks the control variables V_C of an XSTS model explicitly, and all other information using predicates, not switching domains dynamically during execution.

Extensions for XSTS

The e flag that keeping track of the last transition set that fired is tracked explicitly in XSTS regardless of the abstract domain used. To enable explicit tracking of the e flag, I am going to introduce extensions that build upon (wrap) the previously defined constructs of abstract domains (transfer functions, init functions, etc.). 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. Abstract states $(s, e) \in S_X = S \times \{\textsc{Lt}, \textsc{Le}\}$, wrap abstract states of an abstract domain, and an explicitly encoded $e \in \{\textsc{Lt}, \textsc{Le}\}$ flag denoting which set the last executed transition belonged to. The bottom element becomes a set $\bot_X = \{(\bot, \textsc{Lt}), (\bot, \textsc{Le})\}$. The partial order between two states (s_1, e_1) and (s_2, e_2) is defined as $(s_1, e_1) \sqsubseteq (s_2, e_2)$ if $e_1 = e_2$ and $s_1 \sqsubseteq s_2$. The expression function is $\mathsf{expr}_X(s) \equiv \mathsf{expr}$, i.e. the e flag is not required in the expression, as the transfer function tracks it explicitly.

The extended transfer function selects the appropriate transition set based on the e flag, and applies the wrapped transfer function to it. Formally, the extended transfer function $T_X : S_X \times \Pi \mapsto 2^{S_X}$ is defined as $T_X((s, e), \pi) = \{(s', \textsc{Lt}) \mid op \in Tr,\ s' \in T(s, op, \pi)\}$ if $e = \textsc{Le}$ and $T_X((s, e), \pi) = \{(s', \textsc{Le}) \mid op \in En,\ s' \in T(s, op, \pi)\}$ if $e = \textsc{Lt}$. Informally, (s', e') is a successor of (s, e), if $e' \neq e$ and s' is the successor of s with respect to the wrapped transfer function, the precision, and an operation from the appropriate set.

The extended init function $I_X : \Pi \mapsto 2^{S_X}$ is the following: $I_X(\pi) = \{(s, e) \mid s \in I(\pi), e = \textsc{Lt}\}$, i.e. the states returned by the wrapped init function extended with the $\textsc{Lt}$ flag. This ensures that the first transition that fires will be from the En set.

Abstract reachability graph

The abstract state space is represented with the *abstract reachability graph*. This is a representation which ensures that each state will be explored only once.

An abstract reachability graph is a tuple $ARG = (N, E, C)$, where:

- N is the set of *nodes*, each corresponding to an abstract state in some domain D;
- $E \subseteq N \times Ops \times N$ is the set of directed *edges*, labeled with operations. An edge $(s_1, op, s_2) \in E$ is present if s_2 is a successor of s_1 with respect to op;
- $C \subseteq S \times S$ is the set of *covered-by edges*. A covered-by edge $(s_1, s_2) \in C$ 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 *expanded* if it has an outgoing covered-by edge (s, s') for some $s' \in N$. A node that violates the safety propety φ is called *unsafe*. Formally, $s \in N$ is unsafe if $s \not\models \varphi$. 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. An ARG is *complete* if there are no unmarked nodes and *unsafe* if it has at least one unsafe node.

An *abstract path* $\sigma = s_1, op_1, s_2, ..., op_{n-1}, s_n$ is an alternating sequence of abstract states and operations. An abstract path is *feasible* if a corresponding concrete path $c_1, op_1, c_2, ..., op_{n-1}, c_n$ exists, where each c_i is mapped to s_i, i.e. $c_i \models \mathsf{expr}(s_i)$. In practice, this is decided by querying an SMT solver with the formula $s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge s_2^{\langle 2 \rangle} \wedge ... \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$. If this formula is satisfiable, then the abstract path is *concretizable*, and a satisfying assignment corresponds to a concrete path.

The extended ARG for XSTS is defined as $ARG_X = (N_X, E_X, C_X)$, where:

- $N_X \subseteq S_X$ is the set of nodes, each node corresponding to an abstract state $(s, e) \in S_X$;
- $E_X \subseteq N_X \times Ops \times N_X$ is the set of directed edges, labeled with operations. An edge $(s_1, e_1, op, s_2, e_2) \in E$ is present if s_2 is a successor of s_1 with respect to op, $e_1 \neq e_2$, and op is from the set corresponding to the e_1 and e_2 flags (i.e. $op \in Tr$ if $e_1 = \text{LE}$ and $op \in En$ if $e_1 = \text{LT}$);
- $C_X \subseteq S_X \times S_X$ is the set of covered-by edges. A covered-by edge $(s_1, e_1, s_2, e_2) \in C_X$ is present if $s_1 \sqsubseteq s_2$ and $e_1 = e_2$.

Abstraction algorithm

The abstraction algorithm explores the abstract state space based on the current precision and decides if it contains any abstract counterexamples. I define the abstraction algorithm is based on the constructs I introduced above.

The abstraction algorithm receives a partially constructed abstract reachability graph $ARG(N, E, C)$, a FOL safety property φ, an abstract domain $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$, the current precision π, and a transfer function T as inputs. The result of the algorithm is whether the abstract model is *safe* or *unsafe* (i.e. whether a state that violates the safety property φ is reachable from the initial states). The algorithm also returns a complete ARG in the former case, and an ARG containing an abstract counterexample counterexample in the latter case.

In the first iteration, the ARG only contains the set of initial states returned by the init function I, and the precision is usually empty (or generated using some heuristic from the model or the safety property φ).

The algorithm (see Alg. 1) first initializes the reached set with the nodes of the ARG, and the waitlist with all unmarked nodes. The algorithm then processes the nodes of the waitlist based on some search strategy (BFS or DFS). If the current state does not satisfy the property φ, then the algorithm terminates with an unsafe result. If the state doesn't violate the property, then we check if the state is covered by an already visited state with respect to the partial order. If it's covered, then we add a covered-by edge, else we calculate the successors with the transfer function an mark the state as expanded.

Algorithm 1 Abstraction algorithm (based on [16])

Input: $ARG(N, E, C)$: partially constructed abstract reachability graph
φ: FOL safety property
$D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$: abstract domain
π: current precision
T: transfer function

Output: (*safe* or *unsafe*, ARG)

1: $\mathsf{reached} \leftarrow N$
2: $\mathsf{waitlist} \leftarrow$ unmarked nodes from N
3: **while** $\mathsf{waitlist} \neq \emptyset$ **do**
4: $\quad s \leftarrow$ remove from $\mathsf{waitlist}$
5: $\quad$ **if** $s \not\models \varphi$ **then**
6: $\quad\quad$ // s is unsafe
7: $\quad\quad$ **return** (*unsafe*,ARG)
8: $\quad$ **if** $s \sqsubseteq s'$ for some $s' \in \mathsf{reached}$ **then**
9: $\quad\quad$ // s is covered
10: $\quad\quad$ $C \leftarrow C \cup \{(s, s')\}$ // Add covered-by edge
11: $\quad$ **else**
12: $\quad\quad$ // s is expanded
13: $\quad\quad$ **for all** $s' \in T(s, \pi) \setminus \bot$ **do**
14: $\quad\quad\quad$ $\mathsf{reached} \leftarrow \mathsf{reached} \cup \{s'\}$
15: $\quad\quad\quad$ $\mathsf{waitlist} \leftarrow \mathsf{waitlist} \cup \{s'\}$
16: $\quad\quad\quad$ $N \leftarrow N \cup \{s'\}$ // Add new node
17: $\quad\quad\quad$ $E \leftarrow E \cup \{(s, s')\}$ // Add successor edge
18: **return** (*safe*,ARG)

If all reachable states were explored and none violated the safety property, then the algorithm concludes that the model is safe.

2.4.2 Refinement

Due to the over-approximating nature of the abstract domains used, abstract counterexamples found in the abstract state space are not necessarily present in the concrete state space as well. Abstract countexeramples have to be further analysed to decide if they're concretizable. If an abstract counterexample is found *spurious*, i.e. not concretizable, then the precision has to be refined: ideally introducing just enough additional information in the abstract model to avoid finding the same spurious counterexample again.

The Refinement algorithm (see details in [16]) receives an abstract-reachability graph and the current precision as arguments. I used this algorithm as black-box during my work, so I'm only giving a short overview of it here. The abstract reachability graph that the algorithm receives contains one or multiple abstract counterexamples: alternating sequences of abstract states and operations. The algorithm constructs SMT formulas from the counterexamples. If the SMT formula is satisfiable, then the counterexample is concretizable and a satisfying assignment to the formula corresponds to values that the variables could take during the execution. If a satisfying assignment is found, then the algorithm returns with a concrete counterexample formed from it. If the formula is not satisfiable, i.e. the counterexample is spurious, then the algorithm queries the SMT solver for a Craig interpolant [21], a formula that "explains" the reason for the unsatisfiability. In predicate abstraction, the interpolant is usually interpreted as a predicate, and added to the precision. In explicit-value abstraction, the variables are usually collected from the interpolant and added to the precision. The algorithm also finds the longest satisfiable prefix of the formula generated from the counterexample to prune unreachable states from the ARG. The algorithm then returns the pruned ARG and the new precision.

2.4.3 The CEGAR loop

The heart of the CEGAR algorithm is the CEGAR-loop (see Fig. 2.6, Alg. 2), a cycle of repeated invocations of the Abstraction and Refinement algorithms. The algorithm first initializes an ARG with the results of init function as nodes, and no edges. The initial precision is usually empty, or constructed with some heuristic from the model or the property.

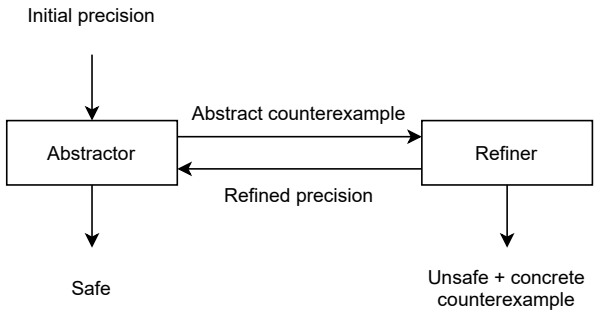

Figure 2.6: An overview of the CEGAR algorithm.

In each iteration of the loop, the Abstraction algorithm is called with the current precision to generate the corresponding abstract state space. If no counterexamples are found

during the abstraction, then the algorithm returns with a *safe* result. If a counterexample is found, then the REFINEMENT algorithm is called to analyze it. If the counterexample is concretizable, then the algortihm returns with an *unsafe* result. Otherwise, the precision is refined and a new iteration is started.

Algorithm 2 CEGAR loop (based on [16])

Input: φ: FOL safety property
$D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$: abstract domain
π_0: initial precision
T: transfer function
I: init function
Output: *safe* or *unsafe*

1: $\pi \leftarrow \pi_0$
2: $ARG \leftarrow (N \leftarrow I(\pi_0), E \leftarrow \emptyset, Cov \leftarrow \emptyset)$
3: **while** *true* **do**
4: $\mathsf{result}, ARG \leftarrow$ ABSTRACTION(ARG, φ, π, T)
5: **if** $\mathsf{result} =$ *safe* **then return** *safe*
6: **else**
7: $\mathsf{result}, \pi, ARG \leftarrow$ REFINEMENT(ARG, π)
8: **if** $\mathsf{result} =$ *unsafe* **then return** *unsafe*

2.5 Theta

Theta [26] is "a generic, modular and configurable model checking framework developed at the Critical Systems Research Group of Budapest University of Technology and Economics, aiming to support the design and evaluation of abstraction refinement-based algorithms for the reachability analysis of various formalisms. The main distinguishing characteristic of Theta is its architecture that allows the definition of input formalisms with higher level language front-ends, and the combination of various abstract domains, interpreters, and strategies for abstraction and refinement. Theta can both serve as a model checking backend, and also includes ready-to-use, stand-alone tools" [15].

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

Table 2.1: An overview of the Theta architecture.

Theta has a modular architecture, which is illustrated in Table 2.1. The *core* module contains various general building blocks for model verification, e.g. variable declarations, expressions, statements which are used in the *control flow automaton* (CFA), *symbolic transition system* (STS), *timed automaton* (XTA) formalisms. Since September 2020, the *extended symbolic transition system* (XSTS) formalism developed and maintained by me is also part of the Theta framework. Each formalism has a corresponding domain-specific language for easy parsing. The analysis backend is located in the *analysis* module and contains various constructs for the CEGAR algorithm with support for multiple abstract domains, such as *predicate abstraction* and *explicit-value abstraction*. Each formalism has

a corresponding analysis module that contains formalism-specific concretizations of the CEGAR algorithm. In order to enable the algorithms of the *analysis* module to operate on a specific formalism, a formalism-specific implementation of the general interfaces defined in the analysis module has to be provided. These implementations can also contain formalism-specific extensions and optimizations to the algorithm that can make the analysis more efficient by using additional information present in the models. The CEGAR algorithm relies on the SMT solver available through the *solver* module and its implementation in the *solver-z3* module. Each formalism has a formalism-specific command line interface in the corresponding *cli* module.

2.6 Related work

The Promela language of the SPIN model checker [19] primarily focuses on modeling of asynchronous distributed algorithms and is commonly used as an intermediate language for verification. The model checker supports LTL specifications as requirement properties and can also work as a simulator. The Promela language offers high-level constructs such as processes and message channels with synchronous and asynchronous semantics for the modeling of distributed asynchronous systems.

BoogiePL [14] is a general intermediate language for model checking of object oriented languages. BoogiePL offers constructs known from programming languages like procedures or arrays. BoogiePL is used as the input formalism of the Boogie model checker. The main distinguishing feature of BoogiePL compared to XSTS is that it has been specifically tailored to programming languages and offers constructs for this domain (such as the procedures mentioned above).

Hajdu et al. present an explicit-value transfer function in [16] that uses a similar substitution method for the calculation of successor states as the operation substitution presented in this work. They only define this substitution for the assumption and assignment operations, and do not carry out a best-effort substitution of the operations.

Beyer et al. present a static product domain in [2] that categorizes variables based on what kind of expressions they appear in in the model to automatically assign abstract domains to them.

Bajkai et al. present a dynamic product domain in [3]. Their approach works in the other direction: they start by tracking all variables explicitly and switch to tracking a variable with predicates if too many possible values are enumerated for it in the same location. Their approach uses the transfer functions of the subdomains as black-boxes, while the approach I present in this work uses a combined transfer function.

Chapter 3

Enriching the language

In this chapter, I introduce additions to the XSTS language that widen the range of high-level engineering models that can be verified using XSTS. *Local variables* introduce additional information to the model that can result in a smaller state space. *Arrays* are a well-known data structure that can make many calculations easier to express. *If-else operations* and *loop operations* are higher-level constructs[1] whose addition makes control structures known from activity diagrams and programming languages easier to express in XSTS.

3.1 Local variables

Calculations often use temporary variables: either to store partial results, or to store the values of variables at certain points in time for later use. The latter is often used in the models generated from orthogonal regions of statecharts for example. If the temporary variables are only needed during the execution of a single transition, and don't have to keep their values between transitions, then they don't have to appear in the state vector. Local variables provide a way to handle temporary variables more efficiently in XSTS, by omitting them from the state vector and thus reducing the state space of the model.

Local variables in XSTS can be declared with the following syntax, where `<name>` is the name, `<type>` is the type and `<value>` is the optional initial value of the variable:

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

After its declaration, a local variable is only added to the scope of the current block, similarly to how local variables are created on the stack in most programming languages. It's not accessible in any way outside the block, it only exists inside the transition and isn't part of the state vector even if all variables are tracked explicitly.

3.2 Arrays

Arrays are among the oldest and most important data structures that are used by almost all computer programs. They have their use in the transformation of engineering models as

[1] The if-else and loop operations are high-level in the context of the XSTS language and SMT solvers, and not in the context of high-level programming languages like C or Java of course.

well: together with loops (see Section 3.4) they can be used to describe message queues between communicating components of a composite network. Equipping the XSTS language with an array-like data structure significantly widens the range of verifiable high-level engineering models.

The array types of the XSTS language are based on associative arrays of SMT solvers. This means that they don't have a fixed size and store key-value pairs, similarly to maps known from programming languages. XSTS arrays can be indexed not only with integers, but with other types as well.

An XSTS array can be declared using the following syntax, where `<arrayname>` is the name of the array, `<keytype>` is the type of the keys, and `<valuetype>` is the type of the stored value:

```
var <arrayname> : [<keytype>] -> <valuetype>
```

An array must be initialized with an array literal, which specifies initial key-value pairs and a default value for unknown keys. An array literal has the following syntax, where `<key_*>` are literals of the key type and `<value_*>` are literals of the value type:

```
[<key_1> <- <value_1>, <key_2> <- <value_2>, . . ., default <- <value_n>]
```

Figure 3.1 shows an example of an array, which maps booleans to integers.

```
var arr : [integer] -> boolean
arr := [0 <- true, default <- false]
arr[1] := true
```

Figure 3.1: Declaring and initializing an array in XSTS.

3.3 If-else operations

The nondeterministic choice operation of the XSTS language is a powerful construct. Together with assume operations, choices can express a very diverse set of branching scenarios. There are certain patterns however, that are used frequently, but are cumbersome to express using choice and assume operations. One such pattern is the "if-else-if chain", a simple chain created from *if-else* statements - perhaps one of the most basic building blocks of procedural programming languages.

To make both the transformation of engineering models to XSTS easier, and the model checking more efficient, I added the XSTS equivalent of if-else statements to XSTS: *if-else* operations. If-else operations are composite operations that contain a condition and 2 suboperations.

Formally, an if-else operation is a composite operation $\mathsf{if}(\varphi)\ op_1\ \mathsf{else}\ op_2$, where φ is a FOL condition, and $op_1, op_2 \in Ops$ are operations. For example, $\mathsf{if}(x > 0)\ x := 0\ \mathsf{else}\ x := x+1$. If the condition φ evaluates to *true*, then op_1 is executed, else op_2 is executed. The semantics of the if-else operations are the following $tran(\mathsf{if}(\varphi)\ op_1\ \mathsf{else}\ op_2) \equiv (\varphi \Rightarrow op_1) \wedge (\neg\varphi \Rightarrow op_2)$, i.e. if φ is *true*, then op_1 should be *true*, else op_2 should be *true*. For example, if op is $\mathsf{if}(x > 0)\ x := 0\ \mathsf{else}\ x := x + 1$, then $tran(op) = (x > 0 \Rightarrow x' = 0) \wedge (x \neq 0 \Rightarrow x' = x + 1)$.

To demonstrate how the if-else operation helps make the transformation of engineering models easier, consider the example in Figure 3.2, which illustrates a simple if-else-if

```
if (cond1) {
    <operation>
} else if (cond2) {
    <operation>
} else if (cond3) {
    <operation>
} else {
    <operation>
}
```

```
choice {
    assume cond1
    <operation>
} or {
    assume !cond1 && cond2
    <operation>
} or {
    assume !cond1 && !cond2 && cond3
    <operation>
} or {
    assume !cond1 && !cond2 && !cond3
    <operation>
}
```

Figure 3.2: A simple if-else-if chain on the left, and the same control structure expressed using choice and assume operations on the right.

chain and the same control structure expressed using choice and assume operations. The `<operation>` lines are arbitrary operations. We can see that the biggest difference is how the *else* branches are expressed, which stems from semantic differences of the if-else and choice operations. In if-else operations, the else branch can only be executed if the condition is *false*, while the same has to be expressed by adding an assumption of the negated condition to the else branch for choice operations. In longer else-if chains - which appear often in XSTS models generated from statecharts for example, these assumptions of the negated conditions can quickly become lengthy expressions, which are cumbersome to write and maintain. By using the if-else operation of XSTS, these lengthy assumptions can be avoided.

The if-else operation in this form already simplifies the transformation of engineering models to XSTS models, but the model checker can't benefit from it yet: the generated SMT formulas aren't significantly simpler, than if the same was expressed using choice and assume operations. The SMTLIB standard defines a native if-then-else construct: the $\mathsf{ite}(\varphi_1, \varphi_2, \varphi_3)$ expression is semantically equivalent to the formula $(\varphi_1 \Rightarrow \varphi_2) \wedge (\neg\varphi_1 \Rightarrow \varphi_3)$. By modifying the transition formula of the if-then-else operation to use the ite expression, we can take advantage of various optimizations offered by SMT solvers for the ite expression. Formally, the modified transition function of the if-then-else operation is $tran(\mathsf{if}(\varphi)\ op_1\ \mathsf{else}\ op_2) \equiv \mathsf{ite}(\varphi, op_1, op_2)$.

3.4 Loop operations

Loops are sequences of statements that are specified only once, but can be carried out several times in succession. Loops can be expressed in XSTS by exploiting the main control loop, and using flags to keep track of whether we are before, after, or inside the loop at the moment.

Figure 3.3 shows a simple while loop on the left, and illustrates how the same control structure can be expressed in XSTS. Please note that the code snippet on the left is not valid XSTS code, only an illustration of what a while loop in the XSTS language could look like. It is easy to see, that even though expressing a loop in XSTS is possible, it is cumbersome and quickly becomes chaotic and unmanageable if there are multiple loops.

Extending the XSTS language with a dedicated loop operation could significantly simplify the way loops are expressed. However, introducing loops to the XSTS language has its theoretical constraints. Condition-based loops (e.g. the while loop) known from programming languages cannot be transformed to closed SMT formulas. Deciding the number of

```
//before the loop
while(x<5){
    x := x+1
}
//after the loop
```

```
var before : boolean = true
var after : boolean = false

tran {
    assume before
    //before the loop
    before := false
} or {
    assume !before && !after
    [x<5]
    x := x+1
} or {
    assume !before && !after
    assume !(x<5)
    after := true
} or {
    assume after
    //after the loop
}
```

Figure 3.3: A simple while loop on the left, and how it can be expressed in XSTS using the main control loop on the right.

iterations of a while loop is a model checking problem in itself. Thus, adding while loops to XSTS transitions would mean that the calculation of successor states after a transition would involve running a separate nested model checking analysis. This is called "recursive model checking" and is an active area of research. Even though techniques based on loop invariants show promising results in deciding loop termination, adding conditional loops to the XSTS language would introduce a new magnitude of complexity and several unsolved problems. For this reason, the loop operation of the XSTS language resembles the range-based for loops known from the Python and Pascal programming languages.

A *loop operation* is a composite operation for v from φ_1 to φ_2 do op, where $v \in V$ called the *loop variable* the is an integer variable, φ_1, φ_2 are integer expressions, and $op \in Ops$ is an operation. The variable v gets assigned the value φ_1 in the first iteration, then gets incremented/decremented by 1 in each iteration until it reaches the value φ_2. An additional constraint is that op cannot modify the value of the variable v or have any effect on the values φ_1, φ_2.

The semantics of the loop operation are defined differently than in the case of the other operations. Even this constrained loop can't be expressed as a closed SMT formula. An additional unrolling step is required before passing loops to the SMT solver. Unrolling involves evaluating the expressions φ_1, φ_2 to integer literals and replacing the loop operation with a sequence that repeats the operation op $|\varphi_1 - \varphi_2|$ times, while also incrementing/decrementing the variable v. Loop unrolling happens during operation substitution, which is described in more details in Section 4.1.

The syntax of loop operations in the XSTS DSL is the following, where `<varname>` is the name of the loop variable, `<value1>` is the value where the loop variable starts , `<value2>` is the value the loop variable reaches in the last iteration, and `<operation>` is an arbitrary operation:

```
for <varname> from <value1> to <value2> do <operation>
```

Chapter 4

Algorithmic optimizations

In this chapter, I present optimizations to the CEGAR algorithm that make it possible to utilize the additional information that is present in XSTS models.

4.1 Operation substitution

Defining the transition semantics of XSTS with operations instead of FOL formulas (like in the case of STS) means there is significantly more semantic information present in the model. In this section, I introduce a novel technique that exploits the additional information present in XSTS operations to simplify transitions based on the state they're firing from. This can lead to smaller and simpler SMT formulas, which can greatly improve performance, as experience shows that queries to SMT solvers are usually the bottleneck during SMT-based model checking.

A *valuation* $s \in S$ is a variable assignment mapping each variable $v \in V$ to its domain extended with top and bottom values $D_v \cup \{\top_{D_v}, \bot_{D_v}\}$. For example, $(x = 1, y = 2, z = 3)$. To simplify the notation, variables $v \in V$ whose value is not explicitly specified in a valuation s, are assumed to have an unknown value $s(v) = \top_{D_v}$. The top element $\top$ (with $\top(v) = \top_{D_v}$ for all $v \in V$) holds no specific value for any of the variables, and the bottom element $\bot$ represents an unreachable state. Let $\varphi_{/s}$ denote the expression obtained by substituting all variables in the expression φ with values from the valuation $s \in S$. For example, if φ is $(x > y)$, and s is $(x = 1)$, then $\varphi_{/s} = (1 > y)$.

Let $\mathsf{sub} : \mathit{Ops} \times S \mapsto \mathit{Ops} \times S$ denote the substitution function. For a valuation $s \in S$, and an operation $op \in \mathit{Ops}$, where $(\hat{op}, \hat{s}) = \mathsf{sub}(op, s)$, $\hat{s}$ and $\hat{op}$ denote the results of the substitution of s into op.

Assignments

For an assignment $op = v := \varphi$, and a valuation $s \in S$, the substitution $(\hat{op}, \hat{s}) = \mathsf{sub}(v := \varphi, s)$ is defined as follows. The substituted operation $\hat{op}$ is $v := \varphi_{/s}$, i.e. the variables in the expression φ are substituted with the values of s. For example, if op is $x := y + z$, and s is $(x = 0, y = 1)$, then $\hat{op}$ is $x := 1 + z$. For all $w \in V$, the resulting valuation $\hat{s}$ is defined as:

$$\hat{s}(w) = \begin{cases} s(w), & \text{if } v \neq w \\ c, & \text{if } v = w \text{ and } \varphi_{/s} \text{ evaluates to some literal } c \\ \text{unknown}, & \text{otherwise} \end{cases}$$

This means that if $\varphi_{/s}$ evaluates to some literal c, then we store that value for v in $\hat{s}$, else v has an unknown value in $\hat{s}$. All other variables' values remain the same in $\hat{s}$. For example, if op is $x := y + z$ and s is $(y = 1, z = 2)$, then $\hat{op}$ is $x := 3$, and $\hat{s}$ is $(x = 3, y = 1, z = 2)$. However if s is $(x = 0, y = 1)$ instead, then $\hat{s}$ is $(y = 1)$ as $\varphi_{/s}$ didn't evaluate to a literal.

Assumptions

If op is an assumption $[\varphi]$ and $s \in S$ is a valuation, then $(\hat{op}, \hat{s}) = \mathsf{sub}([\varphi], s)$ is defined as follows. The substituted operation $\hat{op}$ is $[\varphi_{/s}]$, i.e. the variables in φ are substituted with the values of s, and $\hat{s} = s$, i.e. all variables keep their values. For example, if op is $[x > 0]$, and s is $(x = 1, y = 1)$, then $\hat{op}$ is $[\mathit{true}]$, and $\hat{s}$ is $(x = 1, y = 1)$. The most interesting assumption substitution scenario for us will be when $\varphi_{/s}$ evaluates to *false*, as this will allow us to remove unreachable branches of choice operations. For this reason, failed assumptions will be propagated from composite operations, e.g. if any of the suboperation of a sequence fails, then the sequence will be substituted with a failing assumption.

Havocs

For a havoc operation $\mathsf{havoc}(v)$ and a valuation $s \in S$, where $v \in V$, the substitution $(\hat{op}, \hat{s}) = \mathsf{sub}(\mathsf{havoc}(v), s)$ is defined as $\hat{op} = op$ and for all $w \in V$, $\hat{s}$ is:

$$\hat{s}(w) = \begin{cases} s(w), & \text{if } v \neq w \\ \text{unknown}, & \text{if } v = w \end{cases}$$

This means that all variables keep their value, except v, which will have an unknown value in $\hat{s}$. For example, if op is $\mathsf{havoc}(x)$ and s is $(x = 0, y = 1)$, then $\hat{op}$ is $\mathsf{havoc}(x)$ and $\hat{s}$ is $(y = 1)$.

Sequences

If op is a sequence $op_1, op_2, ..., op_n$, and $s \in S$ is a valuation, then $(\hat{op}, \hat{s}) = \mathsf{sub}((op_1, op_2, ..., op_n), s)$ is defined with the algorithm in Alg. 3.

The valuation s^* stores partial results for the valuation. The algorithm first initializes s^* with s. The algorithm then substitutes all op_i suboperations of the sequence. The valuation s^* obtained from the substitution of op_i serves as an input for the substitution of op_{i+1}, and is updated with each iteration. If any of the substituted suboperations evaluate to a failed assumption $[\mathit{false}]$, then the algorithm terminates and returns $([\mathit{false}], \bot)$ as a result, because if a single assumption of a sequence fails, then the entire sequence fails. If no failing assumptions were found, then a sequence $\hat{op} = \hat{op}_1, \hat{op}_2, ..., \hat{op}_n$ is formed from the substituted suboperations, and is returned together with the valuation obtained from the last iteration. For example, if op is $(x := 2, z := x + y, [z > x])$, and $s = (x = 1, z = 3)$, then $\hat{op}$ is $(x := 2, z := 2 + y, [z > 2])$ and $\hat{s}$ is $(x = 2)$.

Algorithm 3 Substitution algorithm for sequence operations

Input: $op : op_1, op_2, ..., op_n$: sequence operation
s: valuation
Output: $(\hat{op}, \hat{s})$

$s^* \leftarrow s$
for all $i \in \{1, 2, ..., n\}$ **do**
  $(\hat{op}_i, s^*) \leftarrow \mathsf{sub}(op_i, s^*)$
  if $\hat{op}_i = [false]$ **then**
    // one of the operations is a failed assumption
    return $([false], \bot)$
$\hat{op} \leftarrow \hat{op}_1, \hat{op}_2, ..., \hat{op}_n$
$\hat{s} \leftarrow s^*$
return $(\hat{op}, \hat{s})$

If-else operations

If op is an if-else operation $\mathsf{if}(\varphi)\ op_1\ \mathsf{else}\ op_2$, and $s \in S$ is a valuation, then $(\hat{op}, \hat{s}) = \mathsf{sub}((\mathsf{if}(\varphi)\ op_1\ \mathsf{else}\ op_2), s)$ is defined as follows. First, the suboperations op_1 and op_2 are substituted, which results in $(\hat{op}_1, \hat{s}_1) = \mathsf{sub}(op_1, s)$ and $(\hat{op}_2, \hat{s}_2) = \mathsf{sub}(op_2, s)$ The substituted if-else operation $\hat{op}$ is:

$$\hat{op} = \begin{cases} \hat{op}_1, & \text{if } \varphi_{/s} \text{ evaluates to } true \\ \hat{op}_2, & \text{if } \varphi_{/s} \text{ evaluates to } false \\ \mathsf{if}(\varphi_{/s})\ \hat{op}_1\ \mathsf{else}\ \hat{op}_2, & \text{otherwise} \end{cases}$$

If $\varphi_{/s}$ evaluates to *true*, then $\hat{s} = \hat{s}_1$, if $\varphi_{/s}$ evaluates to *false*, then $\hat{s} = \hat{s}_2$, otherwise for all $v \in V$, $\hat{s}$ is defined as:

$$\hat{s}(v) = \begin{cases} \hat{s}_1(v), & \text{if } \hat{s}_1(v) = \hat{s}_2(v) \\ \text{unknown}, & \text{otherwise} \end{cases}$$

For example, if $\hat{op}$ is $(\mathsf{if}(x > 0)\ x := x + y\ \mathsf{else}\ x := x - 1)$, and $s = (x = 2, y = 1)$, then $\hat{op}$ is $x := 3$, and $\hat{s} = (x = 3, y = 1)$. However, if $s = (y = 1)$, then $\hat{op}$ is $(\mathsf{if}(x > 0)\ x := x + 1\ \mathsf{else}\ x := x - 1)$, and $\hat{s} = (y = 1)$.

Loop operations

If op is a loop operation $\mathsf{for}\ v\ \mathsf{from}\ \varphi_1\ \mathsf{to}\ \varphi_2\ \mathsf{do}\ op$, and $s \in S$ is a valuation, then $(\hat{op}, \hat{s}) = \mathsf{sub}((\mathsf{for}\ v\ \mathsf{from}\ \varphi_1\ \mathsf{to}\ \varphi_2\ \mathsf{do}\ op), s)$ is defined as follows. First, the expressions φ_1 and φ_2 are substituted, resulting in $\varphi_{1/s}$ and $\varphi_{2/s}$. If any of the substituted expressions $\varphi_{1/s}$ or $\varphi_{2/s}$ doesn't evaluate to a literal, then the analysis stops and the problem can't be decided, because the loop can't be transformed to a closed SMT formula. In the following, let's assume that $\varphi_{1/s}$ and $\varphi_{2/s}$ evaluate to integer literals c_1 and c_2, respectively.

The substitution algorithm for loop operations is defined in Alg. 4. Depending on whether c_1 or c_2 is greater, the algorithm will run a loop counting either up from c_1 to c_2 or down from c_1 to c_2, with i as the loop variable. In each iteration, the algorithm substitutes two operations, an assignment operation and the inner suboperation and stores them in $assign_i$ and $inner_i$, respectively. In each iteration the assignment

operation assigns i to v. If in any of the iterations the substituted inner operation evaluates to a failed assumption (i.e. $[false]$), then the algorithm terminates and returns $([false], \perp)$ as a result. After the loop finishes, $\hat{op}$ is constructed by forming a sequence from the substituted assignments and inner operations. Depending on whether c_1 or c_2 is greater, this means either $assign_{c_1}, inner_{c_1}, assign_{c_1+1}, inner_{c_1+1}, ..., assign_{c_2}, inner_{c_2}$ or $assign_{c_1}, inner_{c_1}, assign_{c_1-1}, inner_{c_1-1}, ..., assign_{c_2}, inner_{c_2}$. The returned valuation $\hat{s}$ will be the valuation s^* obtained from the last iteration.

For example, if op is (for z from x to $x+2$ do $y := y+z$), and s is $(z = 0, x = 3)$, then $\hat{op}$ is a sequence $(z := 3, y := y+3, z := 4, y := y+4, z := 5, y := y+5)$, and $\hat{s}$ is $(x = 3, z = 5)$.

Algorithm 4 Substitution algorithm for loop operations

Input: op : for v from φ_1 to φ_2 do $inner$: loop operation
s: valuation
Output: $(\hat{op}, \hat{s})$

if $\varphi_{1/s}$ or $\varphi_{2/s}$ doesn't evaluate to a literal **then**
  throw IllegalArgumentException
else
  $c_1 \leftarrow \varphi_{1/s}, c_2 \leftarrow \varphi_{2/s},$
$s^* \leftarrow s$
if $c_1 \leq c_2$ **then**
  for all $i \in \{c_1, c_1 + 1, ..., c_2\}$ **do**
    $(assign_i, s^*) \leftarrow \mathsf{sub}(v := i, s^*)$
    $(inner_i, s^*) \leftarrow \mathsf{sub}(inner, s^*)$
    if $inner_i = [false]$ **then**
      // one of the operations is a failed assumption
      return $([false], \perp)$
  $\hat{op} \leftarrow assign_{c_1}, inner_{c_1}, assign_{c_1+1}, inner_{c_1+1}, ..., assign_{c_2}, inner_{c_2}$
else
  for all $i \in \{c_1, c_1 - 1, ..., c_2\}$ **do**
    $(assign_i, s^*) \leftarrow \mathsf{sub}(v := i, s^*)$
    $(inner_i, s^*) \leftarrow \mathsf{sub}(inner, s^*)$
    if $inner_i = [false]$ **then**
      // one of the operations is a failed assumption
      return $([false], \perp)$
  $\hat{op} \leftarrow assign_{c_1}, inner_{c_1}, assign_{c_1-1}, inner_{c_1-1}, ..., assign_{c_2}, inner_{c_2}$
$\hat{s} \leftarrow s^*$
return $(\hat{op}, \hat{s})$

Choice operations

If op is a nondeterministic choice operation $\{op_1\}$ or $\{op_2\}$ or ... or $\{op_n\}$, and $s \in S$ is a valuation, then $(\hat{op}, \hat{s}) = \mathsf{sub}((\{op_1\} \text{ or } \{op_2\} \text{ or } ... \text{ or } \{op_n\}), s)$ is defined with the algorithm in Alg. 5.

The algorithm first carries out the substitutions for all op_i suboperations using the valuation s as input (this is because the operations of a choice aren't executed in succession). The substituted suboperations that don't evaluate to a failing assumption $[false]$, are collected in the nonfailing set, and their corresponding valuations in the valuations set. If all suboperations were found failing, then the entire choice fails, thus we re-

Algorithm 5 Substitution algorithm for choice operations

Input: $op : \{op_1\}$ or $\{op_2\}$ or ... or $\{op_n\}$: choice operation
s: valuation
Output: $(\hat{op}, \hat{s})$

```
1: nonfailing ← ∅
2: valuations ← ∅
3: for all i ∈ {1, 2, ..., n} do
4:     (ôp_i, ŝ_i) ← sub(op_i, s)
5:     if ôp_i ≠ [false] then
6:         // if the operation is not a failing assumption, add it to the nonfailing set
7:         nonfailing ← nonfailing ∪ {ôp_i}
8:         valuations ← valuations ∪ {ŝ_i}
9: if |nonfailing| = 0 then
10:    return ([false], ⊥)
11: if |nonfailing| = 1 then
12:    // nonfailing only contains ôp_i, valuations only contains ŝ_i
13:    return (ôp_i, ŝ_i)
14: ôp ← choice(nonfailing)
15: for all v ∈ V do
16:    if ŝ_i(v) = ŝ_j(v) for all ŝ_i, ŝ_j ∈ valuations then
17:        ŝ(v) ← ŝ_i(v)
18:    else
19:        ŝ(v) ← unknown
20: return (ôp, ŝ)
```

turn $([false], \bot)$. If only a single branch was found not failing, then this single suboperation and its corresponding valuation are returned. Otherwise, a choice operation is created from the operations of the nonfailing set using the choice operation constructor $\mathsf{choice} : 2^{Ops} \mapsto Ops$, which given a set of operations $\{op_1, op_2, ..., op_n\}$ returns the choice operation $\{op_1\}$ or $\{op_2\}$ or ... or $\{op_n\}$. The valuation $\hat{s}$ is calculated as follows: if a variable $v \in V$ has the same assigned value in all $\hat{s}_i \in$ valuations valuations, then it will have this same value in $\hat{s}$. Otherwise, $\hat{s}(v) =$ unknown.

For example, if op is a choice $(\{x := 1, y := 0\}$ or $\{x := 1, y := 2\}$ or $\{[x > 2]\})$, and s is $(x = 0, y = 0)$, then $\hat{op}$ is $(\{x := 1, y := 0\}$ or $\{x := 1, y := 2\})$, and $\hat{s}$ is $(x = 1)$. As we can see, the third branch is removed because of a failing assumption, and $\hat{s}(x) = 1$ because x is assigned the same value on both remaining branches.

4.2 Extended transfer functions

To be able to utilize operation substitution in the CEGAR algorithm, I define extensions that wrap the previously introduced transfer functions. This will be different for the three domains (explicit-value, predicate, product), as explicit-value states can be interpreted as valuations, while predicate states can't. The extended transfer function for explicit-value abstraction is defined as $T_{\mathsf{sub}}(s_E, op, \pi_E) \equiv \{s'_E \mid s'_E \in T_E(s_E, \hat{op}, \pi_E), (\hat{op}, \hat{s}) = \mathsf{sub}(op, s_E)\}$, i.e. the transfer function carries out the substitution on the operation using the explicit state as input valuation, then passes the substituted operation $\hat{op}$ instead of op to the transfer function. Note that the valuation $\hat{s}$ is not used. In case of predicate

abstraction, the state s_P can't be interpreted as a valuation, so $\top$ is used as input for the substitution instead: this means that the substitution can only happen based on information that is present in the operations, and not based on the state. Formally, $T_{\mathsf{sub}}(s_P, op, \pi_P) \equiv \{s'_P \mid s'_P \in T_P(s_P, \hat{op}, \pi_P), (\hat{op}, \hat{s}) = \mathsf{sub}(op, \top)\}$. In case of product abstraction, the explicit part s_E of a product state (s_E, s_P) is used in the substitution: $T_{\mathsf{sub}}((s_E, s_P), op, (\pi_E, \pi_P)) \equiv \{(s'_E, s'_P) \mid (s'_E, s'_P) \in T_P((s_E, s_P), \hat{op}, (\pi_E, \pi_P)), (\hat{op}, \hat{s}) = \mathsf{sub}(op, s_E)\}$. The consequence of this is that using loops in the predicate domain is only possible if the expressions specifying the lower and upper bounds of the loop variable can be evaluated to literals based on only the transition, i.e. the operations that appear in the transition before the loop.

Chapter 5

Product domain

In this chapter, I present optimizations to the product abstraction domain. The combined transfer function presented in Section 5.1 calculates the successor states in both subdomains simultaneously to allow for more information exchange.

5.1 Combined transfer function

The main weakness of the product abstraction domain presented in Section 2.4.1 is the lack of information-exchange between the two subdomains. The black-box transfer functions can't take advantage of the information present in the other domain. The pre-strengthening operator presented in [23] allows for information-exchange in the explicit-to-predicate direction: the predicate transfer function T_P can use the information that is encoded in the explicit state s_E as well. The predicate-to-explicit direction on the other hand has its theoretical limits: simple heuristics can be given for some edge cases (for example checking if there are predicates of the form $v = c$, where $v \in V$ is a variable and $c \in D_v$ is a literal), but a general solution is not possible, because the predicates can be arbitrarily complex FOL expressions.

In this section, I present a different approach: instead of reusing the transfer functions of the subdomains as black-boxes, I define a combined transfer function that calculates the successor states in both domains together. The combined transfer function $T((s_E, s_P), op, (\pi_E, \pi_P))$ is defined as follows. Similarly to the Boolean predicate abstraction domain, a fresh propositional variable v_i is assigned to each predicate $p_i \in \pi_P$ of the predicate precision. After this, an SMT solver is used to enumerate all satisfying assignments to the formula $(s_E \wedge s_P) \wedge op \Rightarrow (\bigwedge_{v \in \pi_E} v' = s'_E(v)) \wedge (\bigwedge_{p_i \in \pi_P} v_i \leftrightarrow p'_i)$. From each satisfying assignment, a product state (s'_E, s'_P) is formed the following way: the explicit state s'_E is constructed from the $s'_E(v)$ values, and the predicate state s'_P is constructed by forming a conjunction of the p_i predicates with positive v_i variables and the negations of the predicates with negative variables.

Using this transfer function, information exchange happens in both directions: the explicit-value domain can also benefit from the information encoded in the predicate state. There is no need for the strengthening operator either, as the transfer function by definition will only return pairs of states that don't contradict each other.

5.2 Dynamic product domain

The static product domain presented in [23] tracks the control variables V_C of an XSTS model explicitly, and all other information with predicates. In this section, I present a dynamic approach that is capable of switching between domains automatically. This differs in its direction from the algorithm presented in [3]. Their approach starts by tracking all variables explicitly, then switches a variable to the predicate domain if too many different values are enumerated for it in the same state. The approach I present here works in the other direction: by default everything except the control variables are tracked with predicates and the algorithm starts tracking further variables explicitly when certain conditions are met.

My experience shows that in most cases on safe models predicate abstraction performs better, because the CEGAR algorithm usually only has to reach the point where the guard conditions of the model appear in the precision as predicates to prove that no unsafe state is reachable. On unsafe models on the other hand, explicit-value abstraction is usually more efficient as in order prove that the model is unsafe, a precise counterexample has to be found which is usually easier to do with the variables tracked explicitly. The intuition behind my approach is to optimistically assume first that the model is safe and thus start tracking almost everything with predicates, then switch to explicit tracking when we detect signs that the model is probably unsafe.

The mechanism behind detecting that the model is unsafe and we should start tracking variables explicitly is based on the assumption that in order to prove the correctness of a model using predicate abstraction, it is enough that the guard conditions are tracked as predicates. Based on this, the conditions from all assume statements of the model are collected before the analysis and split into atomic expressions. The expressions are then brought to a canonical form, to avoid semantically equivalent but syntactically different expressions to be identified as different. For example, $(x > 5)$, $(5 < x)$ and $!(x \leq 5)$ are all transformed to the same expression, $(x > 5)$. Whenever the SMT solver returns an interpolant during refinement, the algorithm analyses whether it only contains atomic expressions that were present in the model. If a variable appears in an unknown atom, then we start tracking it explicitly.

The atom-based refinement algorithm presented above proved too strict in practice: the solver often returns formulas that only contain knowledge that is present in the model, but are formulated in a way that causes the algorithm to classify them as unknown atoms. For example, even though we are expecting $(x \leq 5)$ because it is present as a condition in the model, a solver might return $(x = 5)$ or $(x < 5)$ if they are valid interpolants with respect to the input formulas as well. To tackle this, a less strict version of the atom-based dynamic refinement algorithm is used which only starts tracking a variable explicitly if it appears in an atom with an operand that it doesn't appear together with in the model. For example, if $(x \leq 5)$ is a condition in the model, and the solver returns $(x < 5)$, $(x = 5)$, or $(x \geq 5)$, then the algorithm doesn't start tracking x explicitly. However, when x appears in $(x \leq 4)$ or $(x = 6)$, the algorithm starts tracking x explicitly.

Chapter 6

Evaluation

In this chapter, I evaluate my work. In Section 6.1, I present a case study where the XSTS language was successfully applied, which proves its practical applicability. In Section 6.2, I present the results of the experimental evaluation of the algorithmic optimizations presented in this work.

6.1 MCaaS

Model checking as a service (MCaaS) [20] is a collaboration of the NASA Jet Propulsion Laboratory[1], the IncQuery Labs Zrt.[2] and the Budapest University of Technology and Economics with the aim of creating a cloud-based, one-click, automatic model checking service for the verification of SysML models.

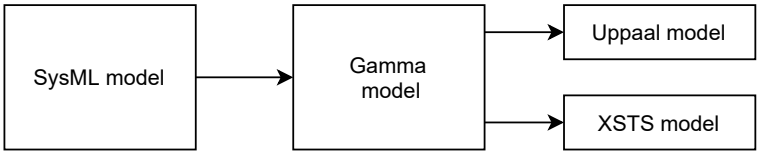

Figure 6.1: An overview of the MCaaS architecture.

Fig. 6.1 shows an overview of the MCaaS architecture. In the MCaaS workflow, users define their SysML models in well-known and familiar modeling tools, like the Cameo Systems Modeler platform, which are then transformed to an internal representation of the Gamma Statechart Composition Framework[3]. The Gamma models are then transformed to the input languages of various model checkers. Currently the timed automata language of the Uppaal model checker [4], and the XSTS language of the Theta model checker are supported.

The Thirty Meter Telescope (TMT) is an extremely large telescope that is planned to be the 2nd largest telescope in the world when constructed. With the extensions and optimizations presented in this work, parts of the open-source SysML model [12] of the TMT became verifiable: the XSTS language was successfully applied for formal verification of heterogeneous system models.

[1] https://www.jpl.nasa.gov

[2] https://incquerylabs.com

[3] The Gamma Statechart Composition Framework [22] is an open-source framework for the modeling and verification of component-based reactive systems developed at the Budapest University of Technology and Economics, available at: https://github.com/FTSRG/gamma

6.2 Experimental evaluation

To evaluate my algorithmic extensions and optimizations, I conducted a benchmarking campaign. My goal during the benchmarking was to answer the following questions:

RQ1 How does the combined product transfer function compare to the black-box transfer functions?

RQ2 Does operation substitution improve the overall performance?

RQ3 How do the dynamic product domains compare to the previous static solution?

Benchmarking environment The benchmarks were ran in the BME Cloud on virtual computers with 8 CPU cores[4] and 16384 MiB of RAM equipped with the Ubuntu 18.04 LTS Server operating system. To ensure reliable results, I used the BenchExec [7] framework, which provides proper isolation of the measured process and independence from outer noises. Each measurement was run with a timeout of 120s.

Benchmark models To increase the reliability of the results, I collected a diverse set of XSTS models, which includes simple handcrafted models as well as real-world industrial examples. The models range from smaller systems with only a few variables to large-scale examples with more than 300 variables. The models are grouped into the following sets:

- **simple**: Simple handcrafted models that cover all language constructs and features;
- **TrafficLight**: Generated from a composite statechart network with 2 traffic light components and a controller component;
- **Spacecraft**: Generated from the SysML statechart model modeling a spacecraft from [20];
- **INPE**[5]: A composite system modeling two components a communication protocol inside a nanosatellite;
- **COID**: A composite statechart network modeling components of railway safety equipment. Provided by a confidential partner of the university;
- **PIL**: A composite statechart network that models components of railway safety equipment. Models in this set are more complex than the **COID** set. Provided by a confidential partner of the university;
- **mcaas**: These models are generated from the open-source model of the Thirty Meter Telescope [12]. The models were modified by hand to remove language elements currently not supported by the mcaas framework;
- **mcaas-sliced**: These models were created by hand from the models in the **mcaas** set by splitting up the singl monolithic transition into 80 smaller transitions. The models express equivalent behaviour.

[4]The cloud provider does not provide more detailed information about the virtual machines.

[5]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

6.2.1 Benchmark results

In this section, I present the benchmarking results.

RQ1: Combined transfer function

To measure how the combined transfer function performs against the black-box transfer functions, I created 4 configurations:

- *EXPL_PRED_BOOL*: A black-box domain combining explicit-value abstraction and Boolean predicate abstraction;
- *EXPL_PRED_SPLIT*: A black-box domain combining explicit-value abstraction and Boolean predicate abstraction. The difference compared to *EXPL_PRED_BOOL* is that states are split among conjunctions;
- *EXPL_PRED_CART*: A black-box domain combining explicit-value abstraction and Cartesian predicate abstraction;
- *EXPL_PRED_COMBINED*: The combined transfer function.

All other parameters were the same for all 4 configurations. The values were all selected based on previous benchmarks:

- *Refinement = SEQ_ITP*: Sequence interpolant refinement [16];
- *Search = BFS*: Breadth-first search strategy;
- *PredSplit = WHOLE*: No predicate splitting;
- *MaxEnum = 250*: When more than 250 successors are enumerated with respect to a state and an operation, then the analysis continues with a top state instead [16];
- *PruneStrategy = LAZY*: Only unreachable states are pruned during refinement;
- *OptimizeStmts = ON*: Operation substitution is enabled;
- *AutoExpl = NEWOPERANDS*: When a variable appears in a refinement interpolant with an operand that is does not appear with in the model, the algorithm starts tracking it explicitly.

The results of the measurements are plotted on a heatmap in Fig. 6.2. The rows of the table correspond to the 4 configurations, and the columns correspond to the 8 model categories. If a cell is greener, it means that the corresponding configuration had a higher success rate (was able to verify more) models in the corresponding category within the timeout bounds. We can see that the combined domain performed the best: it was capable of verifying 66 out of the 82 models, while the second best configuration could only verify 43. There is only one category where the combined domain performed worse: in the **PIL** category it could only verify 1 model, while other configurations were able to verify 2.

Figure 6.2: Benchmark results for RQ1.

RQ2: Operation substitution

In order to measure the effectiveness of operation substitution, I constructed a diverse set of configurations. The Theta framework has more than 9 configuration parameters (domain, refinements strategy, search strategy, etc.) each with 4 options, all combinations of them can't realistically be tested. In order to get a reasonable coverage, I used the Pairwise Independent Combinatorial Testing[6] (PICT) tool of Microsoft. I bound the values values of some parameters which previous experience showed less important, and left other parameters unbound:

- *Domain*: the abstract domain used.
  - *EXPL*: Explicit-value abstraction;
  - *PRED_CART*: Cartesian predicate abstraction;
  - *PRED_BOOL*: Boolean predicate abstraction;
  - *PRED_BOOL*: Boolean predicate abstraction with states split among conjunctions;
  - *EXPL_PRED_COMBINED*: The combined transfer function.
- *Refinement*: the refinement strategy used.
  - *BW_BIN_ITP*: Backwards binary interpolation [16];
  - *SEQ_ITP*: Sequence interpolation.
- *Search = BFS*: Breadth-first search;
- *PredSplit = CONJUNCTS*: Predicates are split among conjunctions;
- *MaxEnum*: if this many explicit states are enumerated with respect to a state and operation, then the analysis continues with a top state [16]
  - 0
  - 10
  - 250

[6] https://github.com/microsoft/pict

- *InitPrec*: the inital precision
  - *EMPTY*: Nothing is tracked by default;
  - *CTRL*: Control variables are tracked by default.
- *PruneStrategy* = *LAZY*: only unreachable states are pruned during refinement;
- *OptimizeStmts*: operation substitution
  - *ON*;
  - *OFF*.
- *AutoExpl*: dynamic predicate-to-explicit switching
  - *STATIC*: no dynamic switching, static product domain;
  - *NEWATOMS*: whenever a variable appears in an unknown atom in a predicate, the algorithm starts tracking it explicitly;
  - *NEWOPERANDS*: whenever a variable appears with an unknown operand in a predicate, the algorithm starts tracking it explicitly.

Some further constraints were introduced to filter unnecessary combinations: for example when the *EXPL* domain is used, then the *AutoExpl* parameter is fixed to a value, because it has no influence on the performance of the explicit domain. I generated 32 configurations using the PICT tool.

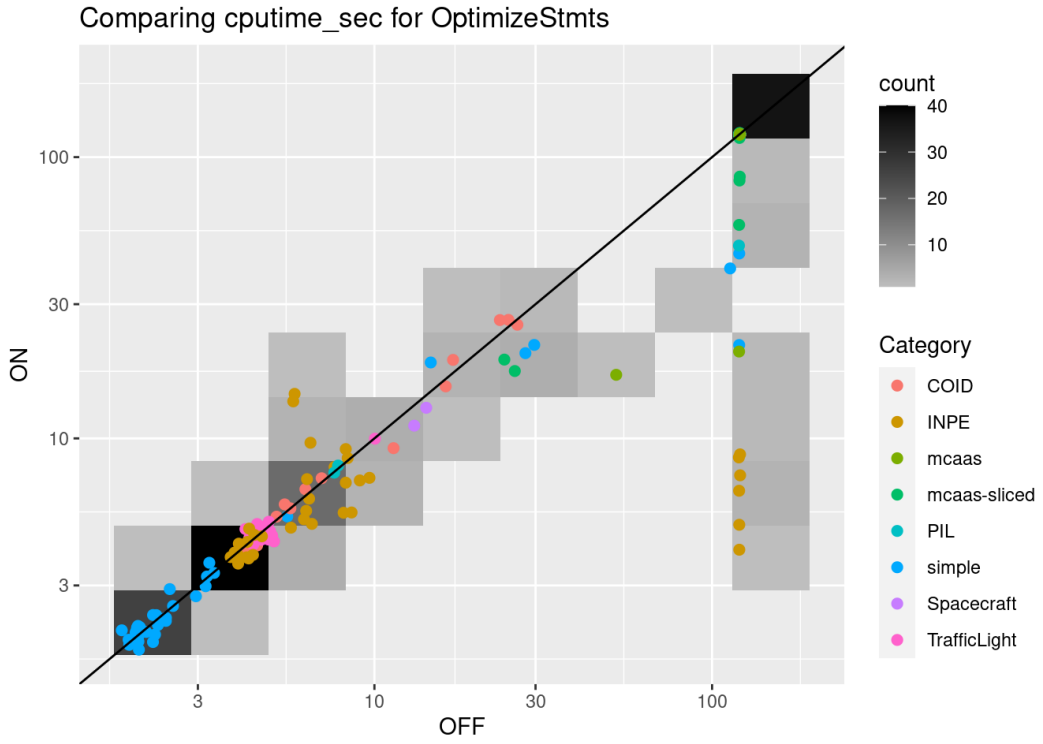

Figure 6.3: Benchmarking results for operation substitution.

The benchmarking results for operation substitution are plotted on the diagram in Fig. 6.3. Each dot represents a single run, with its x coordinate corresponding to the CPU time with operation substitution disabled, and its y coordinate to corresponding to the CPU time with operation substitution enabled. Dots that are below the line correspond

to measurements where the configurations with operation substitution performed better, while the dots above correspond to configurations where disabled operation substitution performed better. The most interesting dots are the one that are on the rightmost side of the diagram on a vertical line above each other. These are measurements where only the configuration that had operation substitution enabled could verify the models. The points that line on the diagonal correspond to measurements where both configurations performed roughly the same. We can see that in most cases either operation substitution performed better or roughly the same. There are some exceptions to this from the *INPE* set, which require further analysis. We can also see that operation substitution has a greater impact on the performance of larger models, then simpler ones.

RQ3: Dynamic product refinement

To evaluate the dynamic product refinement strategies, I defined 2 unbound parameters:

- *InitPrec*: the initial precision
  - *EMPTY*: Nothing is tracked by default;
  - *CTRL*: Control variables are tracked by default.
- *AutoExpl*: dynamic predicate-to-explicit switching
  - *STATIC*: no dynamic switching, static product domain;
  - *NEWATOMS*: whenever a variable appears in an unknown atom in a predicate, the algorithm starts tracking it explicitly;
  - *NEWOPERANDS*: whenever a variable appears with an unknown operand in a predicate, the algorithm starts tracking it explicitly.

The other parameters were bound to the following values:

- *Domain = EXPL_PRED_COMBINED*: combined product domain;
- *Refinement = SEQ_IPT*: sequence interpolation;
- *Search = BFS*: Breadth-first search;
- *PredSplit = CONJUNCTS*: Predicates are split among conjunctions;
- *PruneStrategy = LAZY*: only unreachable states are pruned during refinement;
- *OptimizeStmts = ON*: operation substitution enabled;
- *AutoExpl*: dynamic predicate-to-explicit switching

A comparison of the measurements for the earlier static product domain the dynamic refinement based on new atoms is plotted in Fig. 6.4, while a comparison of the static option and the dynamic refinement based on new operands is plotted in Fig. 6.5. In both diagrams, the CPU times measured for the static option are the x coordinates of the dots, while the measurements for the dynamic domains are the y coordinates. Dots that lie below the diagonal correspond to measurements where the dynamic domains performed better, and analogously dots above the diagonal to measurements where the static option performed better. The diagrams show that the dynamic refinement rarely brought better performance and in most cases lead to worse performance.

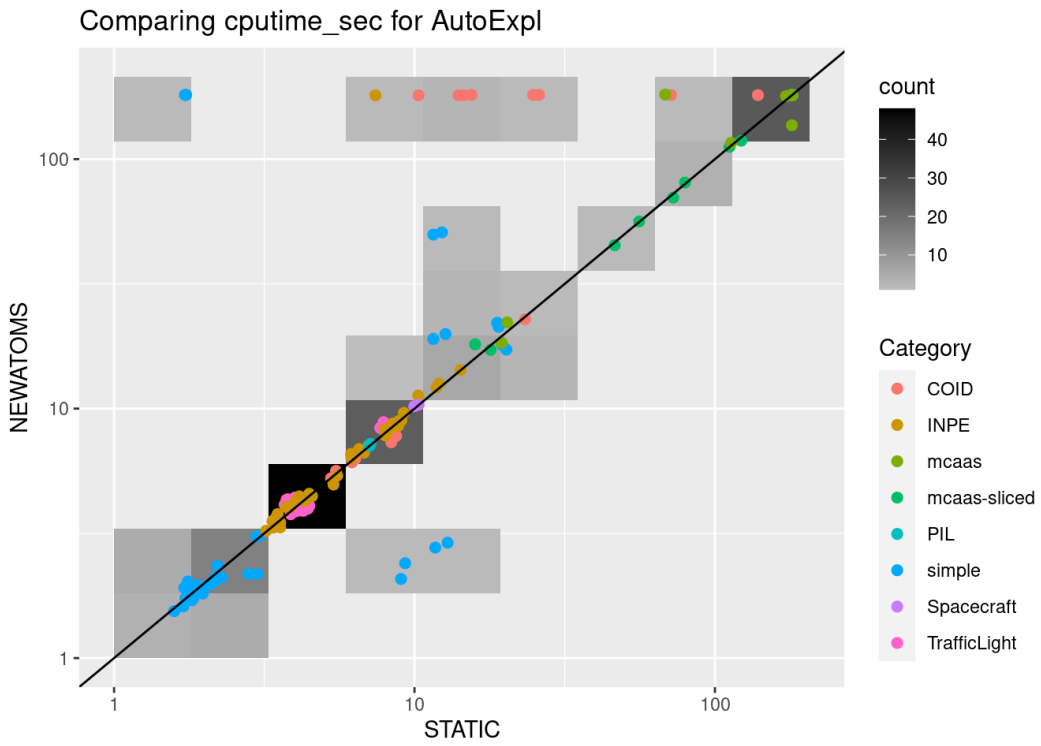

Figure 6.4: Benchmarking results for dynamic refinement based on new atoms.

Summary

The benchmarking showed promising results in the case of the combined transfer function: the configuration using it outperformed the configurations of the black-box product domains. Operation substitution showed promising results as well, the measurements indicated that in most cases it brings a performance improvement, especially in the case of larger models. The dynamic product domain refinement strategies sadly didn't perform as well as expected and require further analysis.

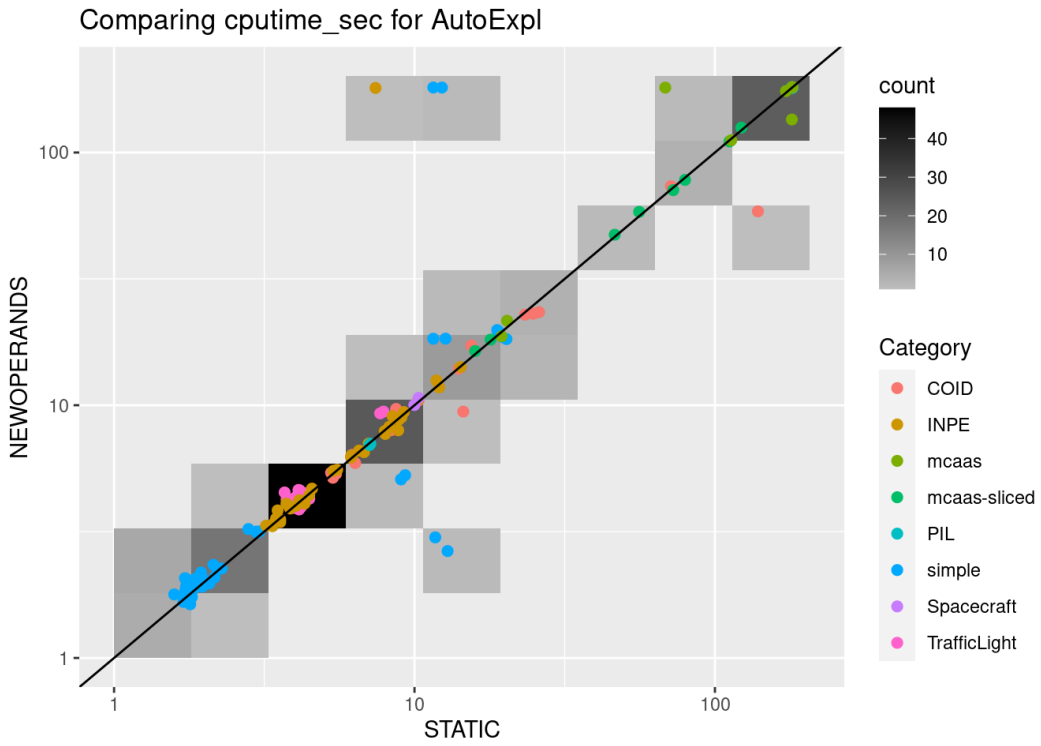

Figure 6.5: Benchmarking results for dynamic refinement based on new operands.

Chapter 7

Conclusions

In this chapter, I draw the conclusions of my work and present possible future directions. My primary goal during this work was to create extensions and optimizations that make the XSTS language and infrastructure capable of verifying large-scale industrial models. The language extensions presented in this work bring the abstraction level of the XSTS language closer to that of the high-level engineering models, while keeping the generality. The *local variables* offer a way of reducing the state space by encoding more information in the model. *Arrays* are a commonly used data structure that make the transformation of several high-level engineering formalisms easier. The *if-else* operation makes many branching scenarios easier to express in the XSTS language by introducing a construct that is known from many formalisms. The *loop* operation offers an easy-to-use construct for expressing loops but comes with strict constraints that stem from theoretical considerations.

I presented various algorithmic optimizations that make the verification of XSTS models more efficient. *Operation substitution* uses the additional information that is encoded in XSTS operations to simplify the SMT formulas generated from transitions based on the current state before passing them to the solver. The *combined transfer function* for product domains makes information exchange possible between the two subdomains of the analysis. The *dynamic refinement strategies* for product domains offer an automatic solution for detecting variables that should be tracked explicitly.

I implemented the language extensions and algorithms I presented in the open-source Theta framework. The XSTS language and model checker were successfully applied in the *MCaaS* collaboration for the verification of heterogeneous SysML system models. The open-source SysML model of the Thirty Meter Telescope (TMT) became partially verifiable in the XSTS language thanks to the extensions presented in this work. To evaluate the algorithmic extensions and optimizations presented in this work, I conducted a *benchmarking campaign* on a diverse set of models that included real-world industrial examples. The benchmarking showed promising results in case of operation substitution and the combined transfer function for product domains.

Possible future extensions of the language include introducing types for rational numbers and bitvectors, as well as introducing state invariants that have to hold in all possible states of the system. I plan on further examining the product abstraction domain. I intend to extend the scope of verifiable properties to temporal logical expressions (LTL or CTL) as well.

Bibliography

[1] Zsófia Ádam, Gyula Sallai, and Ákos Hajdu. Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In Jan Friso Groote and Kim G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 12652 of *Lecture Notes in Computer Science*, pages 435–439. Springer, 2021. DOI: 10.1007/978-3-030-72013-1_27.

[2] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain types: Abstract-domain selection based on variable usage. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing*, pages 262–278, Cham, 2013. Springer International Publishing. ISBN 978-3-319-03077-7.

[3] 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.

[4] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, page 232–243, Berlin, Heidelberg, 1996. Springer-Verlag. ISBN 354061155X.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] TMT Observatory Corporation. Thirty Meter Telescope SysML model. URL `https://github.com/Open-MBEE/TMT-SysML-Model`. Last accessed on 2021-10-27.

[13] 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`.

[14] 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.

[15] FTSRG. Theta framework GitHub repository. `https://github.com/ftsrg/theta`, 2021. Accessed: 2021-10-24.

[16] 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`.

[17] 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`.

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

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

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

[21] 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`.

[22] 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`.

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

[24] Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML™) Version 1.6. OMG Document Number formal/19-11-01 (`https://www.omg.org/spec/SysML/1.6/`), 2019.

[25] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

[26] 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. ISBN 978-0-9835678-7-5. DOI: `10.23919/FMCAD.2017.8102257`.

Appendix

A.1 Heterogeneous system model

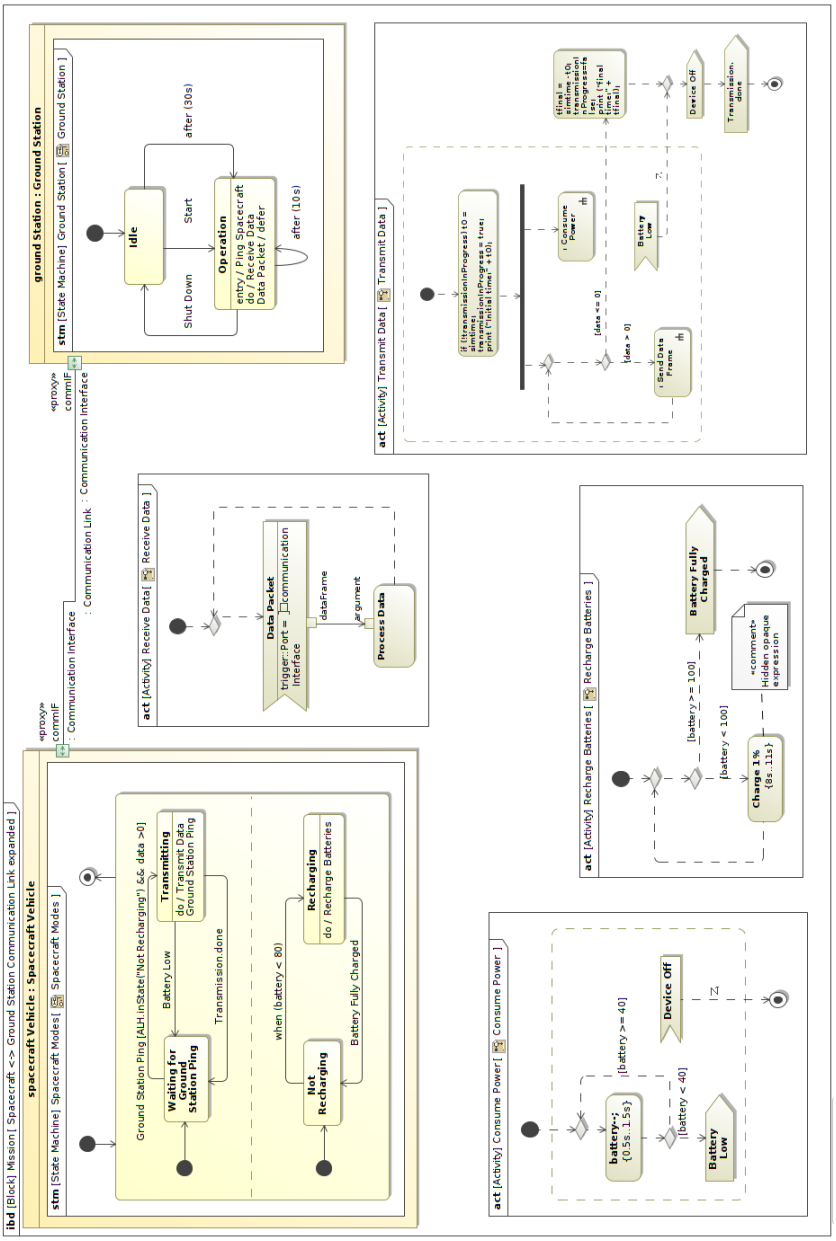

Figure A.1.1: A heterogeneous SysML model.