MŰEGYETEM 1782

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

Hierarchical Abstraction for the Verification of State-based Systems

Bachelor's Thesis

Author
Bence Czipó

Advisors
Ákos Hajdu
Tamás Tóth

December 9, 2016

Contents

HALLGATÓI NYILATKOZAT

Kivonat

Napjainkban a beágyazott rendszerek az élet minden területén egyre nagyobb teret nyernek, így helyességük ellenőrzése is egyre fontosabb, ugyanis kritikus esetben vállalatok sorsa vagy akár emberi élet is múlhat rajta. Ennek egyik fontos eszköze a formális verifikáció, melynek segítségével matematikai precizitással lehet a modellek helyességét már a tervezési fázisban vizsgálni.

A hierarchikus állapottérképek a viselkedésmodellek egyik gyakran használt eszközeként a mérnöki tervezés alapjául szolgálnak, verifikációjuk ezért kiemelt jelentőséggel bír. Gyakran azonban egy egyszerű állapottérkép ellenőrzése is nehéz feladat, ugyanis a változók számával exponenciálisan növekvő állapottér megakadályozhatja a sikeres verifikációt. Az állapottér hatékony kezelésére és bejárására az irodalomban többféle algoritmust is kidolgoztak. Ezek közül az egyik legelterjedtebb a korlátos állapotelérhetőségi analízis, amelyet leggyakrabban logikai megoldók, azaz SAT/SMT solver-ek segítségével valósítanak meg. Ehhez az állapotgépet logikai formulákkal írják le (elkódolás), majd ezeket a formulákat adják be a megoldóknak.

Gyakran azonban ezek az algoritmusok sem tudnak megbirkózni a komponensmodellekben használt változatos adattípusok és konstrukciók okozta komplex viselkedésekkel. A nagyméretű állapottér által jelentett komplexitás csökkentésére megoldást jelenthet az absztrakció alkalmazása, amely azonban elrejthet az ellenőrzés sikerességéhez elengedhetetlen részleteket. Ilyenkor finomítani kell az absztrakciót és gazdagítani a reprezentált információt. Ezen elv mentén működik az ellenpélda alapú absztrakciófinomítás (CounterExample-Guided Abstraction Refinement, CEGAR) módszere.

A gyakorlatban használt verifikációs eszközök általában nem használják ki az állapottérképekben levő hierarchiát. Dolgozatomban a célom olyan algoritmusok fejlesztése, amelyek hatékonyan tudják kezelni a hierarchikus állapottérképeket, továbbá ki tudják használni a verifikáció során a hierarchiában rejlő extra információt. Bemutatok egy általam tervezett módszert, amely lehetővé teszi komplex állapottérképek hatékony elkódolását logikai formulákká a hierarchia kihasználásával. Ezt továbbfejlesztve egy olyan absztrakciófinomításon (CEGAR) alapuló algoritmust ismertetek, amely az állapotok közötti hierarchiát felhasználja a finomítás során, és különböző logikai megoldókra építve akár komplex állapottérképek ellenőrzését is lehetővé teszi. Az elkészített algoritmusok hatékonyságát egy ipari példán demonstrálom illetve hasonlítom össze.

Abstract

Nowadays, as embedded systems take an increasingly important part in every aspect of our life, checking their correct behavior becomes more and more essential, especially in safety-critical cases, where a future of an enterprise or human lives rely on them. Formal verification is an important method, providing strong mathematical basis to check the correctness of the models in the design phase of the system's lifecycle.

Hierarchical statecharts, as a frequently used behavioral model, are one of the foundations of system design, so their verification has an increased relevance. However in many cases, even the verification of a simple statechart can be challenging, since the large state space can prevent the verification as it grows exponentially with the number of variables in the system. There are several algorithms in the literature to efficiently handle and explore the state space. One of the most common amongst them is the bounded state reachability analysis, which is often realized with logical solvers, such as SAT and SMT solvers. In order to perform the analysis, the transition relation of the statechart is transformed to logical formulas, and these formulas are fed to the solver.

However, even these algorithms may not handle the complex behavior caused by the various data types and constructions used in the component models. To reduce the complexity caused by the huge state space, a possible solution is to use abstraction, even though it can fade details that are inevitable for successful verification. In these cases, the abstraction needs to be refined and the represented details should be enriched. This concept is the so-called Counterexample-Guided Abstraction Refinement (CEGAR) approach.

Most of the verification techniques used in practice do not exploit the information underlying in the hierarchical structure of the statecharts. The aim of my work is to develop algorithms that can handle hierarchical statecharts efficiently, and furthermore, that can benefit from the underlying information encoded in the state hierarchy during verification. I present a novel approach that can be used to effectively transform complex statecharts into logical formulas, taking benefits from the hierarchy. Improving that, I introduce an algorithm based on abstraction refinement (CEGAR), that takes hierarchy information into consideration during the refinement, and makes it possible to verify complex statecharts using logical solvers. The efficiency of the previously presented algorithms is demonstrated and compared on an industrial case study.

Chapter 1

Introduction

Through the years, software evolved from a scientific environment to the industry, and as it appeared in safety-critical embedded systems, its verification became a critical requirement. Nowadays there is a strong tendency of computers taking over tasks from humans that require continuous concentration and precision, such as driving a car, or managing a railway system or the cooling of a nuclear power plant. One common attribute of the preceding examples is that one small failure in their control can lead into enormous loss in terms of people's trust, money or even human lives.

Testing the complete system with a given set of inputs and expected results might witness the presence of errors, but can not prove its faultlessness (unless tested with every possible input under every possible environmental assumption). In contrast, formal verification provides automated, mathematically precise techniques to ensure correct functionality of the system. Furthermore, as most of such techniques operate on models of the system, verification can be performed before implementing and deploying the real system.

Formal models are also the foundations of system design, and hierarchical statecharts are amongst the most widely used. They extend simple state machines with composite states, parallel regions and variables. One widespread technique for their verification is model checking, that is, the exploration of their state space, and checking it against a given requirement. Reachability analysis is an important requirement, where the purpose of verification is to check if a given erroneous state is reachable from the statechart's initial state.

A possible solution for reachability analysis is realized by transforming the transition relation of the statechart and the requirements to a logical formula in a way such that if the formula is satisfiable, then an execution of the statechart violates the requirement. The satisfiability of such formulas can be evaluated with logical solvers, mostly with SAT (boolean satisfiability) and SMT (satisfiability modulo theories) solvers.

However, in many cases, model checking statecharts can be challenging as their state space becomes unmanageably large or even infinite with the introduction of variables and parallel regions. This problem is the so-called state space explosion and it leads to high computational complexity, which can result in non-termination of the verification procedure. Several techniques have been proposed to overcome this problem, one of them is bounded model checking where a bound k is introduced that limits the maximum number of consecutive state transitions to be checked, so the given requirement is tested only against states that are reachable within k consecutive transitions from the initial state of the statechart. But as k can be chosen arbitrarily, the completeness of the checking can not be guaranteed.

An other promising way to overcome such difficulties is by applying abstraction to the statechart, that is, checking the requirement against a simplified representation of the statechart that has fewer states than the original one. There are two main types of abstractions: over- and under-approximation.

During my work, I focus on over-approximation-based abstraction techniques, which means that if the requirement stands for the abstract statechart, it also holds for the concrete one, however there might be spurious counterexamples violating the requirements that only emerge from the abstraction. Counterexample-Guided Abstraction Refinement (CEGAR) is a general approach to perform automated refinement of the abstraction to eliminate spurious counterexamples violating the requirements. The four major parts of the algorithm are the creation of an initial abstraction, verifying the abstracted system against the given requirement, searching a concrete representation of a counterexample found, and refining the abstraction if needed. CEGAR has been applied to various modeling formalisms. In my thesis, I concentrate on the application of it for the verification of statecharts. The thesis introduces different approaches for creating and refining abstractions of statecharts based on their hierarchical structure. I also introduce various methods for the verification of the abstract models.

The evaluation of my work is done by measuring and comparing the performance of the defined techniques. During the evaluation, the emergency procedure initiating PRISE logic of the Paks Nuclear Power Plant is verified with the different CEGAR implementations.

The rest of this work is structured as follows. In Chapter 2, I present the necessary background knowledge related to my work. After that, in Chapter 3, I suggest an algorithm to encode hierarchical statecharts into logical formulas and present two model checking algorithms based on the encoding, a bounded and an unbounded one. An extension of the latter to a CEGAR approach is presented in Chapter 4, while Chapter 5 holds the relevant details of the implementation. Performance of the model checker is evaluated in Chapter 6 and I sum up the conclusions of my work in Chapter 7.

Chapter 2

Background

This chapter introduces the preliminaries of this work. First, I present the basics of mathematical logic in Section 2.1, including propositional logic, first order logic and first order theories. Then, state machines and statecharts are introduced in Section 2.2, while the common practices to encode them to logical formulas are summarized in Section 2.3. Finally, Section 2.4 presents the related concepts of model checking, including bounded model checking and the CEGAR algorithm.

2.1 Mathematical Logic

In this section I present the basics of mathematical logic [5], starting with propositional logic in Section 2.1.1. Then, in Section 2.1.2 first order logic is introduced. Finally Section 2.1.3 summarizes first order theories and presents some theories and the SMT problem.

2.1.1 Propositional Logic

This section describes propositional logic (PL, also known as propositional calculus). First I present the syntax of the logic, than its semantics. Later additional concepts such as satisfiability and validity are presented and finally I introduce the SAT problem.

2.1.1.1 Syntax

The basic elements of PL are the *nullary logical connectives* $\top$ (truth) and $\bot$ (falsity), and the *propositional variables* (usually denoted by P, Q, R), together referred to as *atoms*. Every atom is a *formula*, and a new formula ψ can be constructed from formulas ψ_1, ψ_2 using *logical connectives* in the following way:

- $\psi = \neg\psi_1$ (negation),
- $\psi = \psi_1 \wedge \psi_2$ (conjunction),
- $\psi = \psi_1 \vee \psi_2$ (disjunction),
- $\psi = \psi_1 \rightarrow \psi_2$ (implication),
- $\psi = \psi_1 \leftrightarrow \psi_2$ (equivalence).

There are some other relevant definitions related to the syntax of propositional logic. A *literal* is an atom or its negation, whereas a *clause* is a disjunction of literals.

Example 2.1. *Some examples for the building blocks of proositional logic are presented below.*

- P, Q, R, $\top$, $\bot$ *are atoms.*
- P, $\neg P$, $\bot$ *are literals.*
- $P \vee Q$, $\neg P \vee R$, $P \vee \top$, P *are clauses.*
- $(P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$, $\neg\bot \leftrightarrow \top$, $\neg P$, Q *are formulas.*

2.1.1.2 Semantics

The *semantics* of a logic is the meaning assigned to the formulas defined by its syntax. In propositional logic, this meaning is a truth value, either 1 (true) or 0 (false). If each propositional variable in a logical formula is assigned a truth value, the truth value of that formula can be computed. Such assignment is called an *interpretation.*

Definition 2.1. An interpretation $\mathcal{I} : \mathcal{L}_0 \mapsto \{0, 1\}$ for the set of propositional variables $\mathcal{L}_0$ is a function that assigns a truth value to every variable in $\mathcal{L}_0$. Let $\mathcal{I}[P]$ denote the truth value of a variable $P \in \mathcal{L}_0$ under $\mathcal{I}$. ▪

As it was mentioned above, given an interpretation $\mathcal{I}$, the truth value of ψ can be evaluated. The way of calculating this value can be defined with truth tables, that express how the formula is evaluated depending on the truth value of its arguments. The truth table of logical connectives can be found in Table 2.1.

Table 2.1: The truth table of logical connectives.

P	Q	$\bot$	$\top$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \to Q$	$P \leftrightarrow Q$
0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0
1	0	0	1	0	0	1	0	0
1	1	0	1	0	1	1	1	1

Example 2.2. *Consider the formula* $\psi = (P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$*. A possible interpretation is* $\mathcal{I} = \{P \mapsto 1,\ Q \mapsto 0,\ R \mapsto 1\}$*. With this interpretation,* ψ *evaluates to true as it can be seen in the truth table of the formula in Table 2.2. This example also demonstrates a way of proving that a formula* ψ *evaluates true for every possible interpretation.*

The evaluation of a formula ψ under the interpretation $\mathcal{I}$ can be calculated recursively using such tables. However to be able to extend it for predicate logic, it is better to define semantics in a different way.

Let $\mathcal{I} \models \psi$ denote that ψ evaluates to true under $\mathcal{I}$, and $\mathcal{I} \not\models \psi$ denote that ψ evaluates to false. The truth value of propositional variables can then be defined in the following way:

$\mathcal{I} \models P \iff \mathcal{I}[P] = 1, \mathcal{I} \not\models P \iff \mathcal{I}[P] = 0.$

The connectives can be defined inductively according to the following rules:

Table 2.2: The truth table of ψ in Example 2.2.

P	Q	R	$P \vee Q$	$\neg P \vee R$	$(P \vee Q) \wedge (\neg P \vee R)$	$Q \vee R$	$(P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$
0	0	0	0	1	0	0	1
0	0	1	0	1	0	1	1
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1
1	0	1	1	1	1	1	1
1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1

- $\mathcal{I} \models \top$,
- $\mathcal{I} \not\models \bot$,
- $\mathcal{I} \models \neg\psi_1 \iff \mathcal{I} \not\models \psi_1$,
- $\mathcal{I} \models \psi_1 \wedge \psi_2 \iff \mathcal{I} \models \psi_1$ and $\mathcal{I} \models \psi_2$,
- $\mathcal{I} \models \psi_1 \vee \psi_2 \iff \mathcal{I} \models \psi_1$ or $\mathcal{I} \models \psi_2$,
- $\mathcal{I} \models \psi_1 \to \psi_2 \iff$ if $\mathcal{I} \models \psi_1$ then $\mathcal{I} \models \psi_2$,
- $\mathcal{I} \models \psi_1 \leftrightarrow \psi_2 \iff \mathcal{I} \models \psi_1 \to \psi_2$ and $\mathcal{I} \models \psi_2 \to \psi_1$.

Example 2.3. *Consider the formula ψ from Example 2.2. Its value can be deduced from the interpretation $\mathcal{I} = \{P \mapsto 1,\ Q \mapsto 0,\ R \mapsto 1\}$ in the following way:*

1. *$\mathcal{I} \models P$, $\mathcal{I} \not\models Q$, $\mathcal{I} \models R$, and $\mathcal{I} \not\models \neg P$,*
2. *$\mathcal{I} \models (P \vee Q)$ because $(\mathcal{I} \models P) \vee (\mathcal{I} \models Q)$ is true,*
3. *$\mathcal{I} \models (\neg P \vee R)$ because $(\mathcal{I} \models \neg P) \vee (\mathcal{I} \models R)$ is true,*
4. *$\mathcal{I} \models (Q \vee R)$ because $(\mathcal{I} \models Q) \vee (\mathcal{I} \models R)$ is true,*
5. *$\mathcal{I} \models (P \vee Q) \wedge (\neg P \vee R)$ because $(\mathcal{I} \models P \vee Q) \wedge (\mathcal{I} \models \neg P \vee R)$ is true, according to 2) and 3),*
6. *According to 5) $\mathcal{I} \models (P \vee Q) \wedge (\neg P \vee R)$, and according to 4) $\mathcal{I} \models (Q \vee R)$, so $\mathcal{I} \models (P \vee Q) \wedge (\neg P \vee R) \to (Q \vee R)$.*

2.1.1.3 Satisfiability and Validity

A formula ψ is *satisfiable*, if and only if an interpretation $\mathcal{I}$ exists such that $\mathcal{I} \models \psi$, and ψ is *valid* if and only if $\mathcal{I} \models \psi$ holds for every interpretation $\mathcal{I}$. The formula ψ is *unsatisfiable* iff it is not satisfiable. Satisfiability and validity are duals of each other, that is, ψ is valid iff $\neg\psi$ is unsatisfiable.

Example 2.4. *The formula $\psi_1 = P \vee Q$ is satisfiable, as for the interpretation $\mathcal{I}_1 = \{P \mapsto 1,\ Q \mapsto 1\}$, $\mathcal{I}_1 \models \psi_1$.*

The formula $\psi_2 = P \wedge \neg P$ is unsatisfiable as there are only two different interpretations: $\mathcal{I}_{2a} = \{P \mapsto 1\}$ and $\mathcal{I}_{2b} = \{P \mapsto 0\}$, and $\mathcal{I}_{2a} \not\models \psi_2$ and $\mathcal{I}_{2b} \not\models \psi_2$.

The formula $\psi_3 = (P \vee Q) \wedge (\neg P \vee R) \to Q \vee R$ is valid, as it can be seen in its truth table presented in Table 2.2.

Definition 2.2 (SAT problem). The *Boolean satisfiability problem*, often referred to as the *SAT problem* is deciding if an interpretation $\mathcal{I}$ exists for a formula ψ such that $\mathcal{I} \models \psi$.∎

The problem can be solved in exponential time, however there is no known algorithm that can decide satisfiability in polynomial time. Even so, given an interpretation $\mathcal{I}$, it can be determined in polynomial time if $\mathcal{I}$ satisfies the formula ψ, so $SAT \in \mathbf{NP}$. Cook and Levin also proved that all problems in **NP** can be reduced to SAT [9].

Although the problem is algorithmically hard to solve, it has several relevant usage in science. The ever-growing need of fast solutions for the problem pushes the research community to continuously optimize the algorithms and develop new ones. Even though the problem is still exponential in the worst case, modern solvers can solve practical problems for even large inputs (ten thousands of variables) in reasonable time [12].

2.1.2 First Order Logic

First order logic (FOL), also referred to as *predicate logic* or *predicate calculus* extends propositional logic with predicates, functions and quantifiers. Formulas in predicate logic form sentences about instances of an entity set (domain).

The structure of this section is similar to the previous one, as I first describe the syntax of FOL, then I present its semantics. Finally, satisfiability and validity are defined for FOL formulas.

2.1.2.1 Syntax

The basic elements of FOL are *terms*. A simple term can be a variable (often denoted by $x, y, z, \ldots$) or a constant symbol ($a, b, c, \ldots$). More complex terms can be constructed using function symbols ($f, g, h, \ldots$). The *arity* of a function symbol is the number of arguments it takes. A constant symbols can be interpreted as a nullary function symbol.

Example 2.5. *The following list contains examples for terms:*

- 1, *"marmot" are constant symbols (nullary function symbols),*
- x *is a variable,*
- $cos(x)$ *is the application of a unary function symbol cos to variable* x*,*
- $f(x, a)$ *is the application of a binary function symbol* f *to variable* x *and constant symbol* a*.*

Predicate symbols of FOL are the generalization of propositional variables from PL. Like function symbols, predicate symbols ($p, q, r, \ldots$) also have an arity: an n-ary predicate symbol takes n terms as arguments. A nullary predicate symbol in FOL is analogous to a propositional variable ($P, Q, R, \ldots$) in PL.

Like in PL, formulas of FOL are constructed from *atoms*. An atom can be $\top$, $\bot$ or an n-ary predicate symbol applied to n terms. A *literal* is an atom or its negation.

Example 2.6. *p is a binary predicate symbol, so $\psi = p(f(x), g(x, y))$ is an atom, a binary predicate symbol applied to two terms.*

Every atom is a *formula*, and more complex formulas are constructed by the application of logical connectives $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ to formulas, or by using *quantifiers.* In FOL, there are two quantifiers, the *existential quantifier* denoted by $\exists x.\psi[x]$, and the *universal quantifier*, denoted by $\forall x.\psi[x]$. In both cases x is the *quantified variable*, also said to be *bound.* In a formula ψ a variable is *free* if it has an occurrence that is not bound by any quantifier. A formula ψ is closed if every variable in ψ is bound.

Let ψ be a FOL formula, and $V = \{v_1, v_2, \dots v_n\}$ be the set of its variables. Let ψ_k denote the FOL formula where each variable is replaced with its k-indexed equivalent from the variable set $V_k = \{v_{1k}, v_{2k}, \dots v_{nk}\}$. For example if ψ is $x < y$ then ψ_k is $x_k < y_k$.

2.1.2.2 Semantics

Terms of FOL formulas evaluate to an instance of a specified domain, so to define the semantics for FOL formulas, the concept of interpretation defined in PL has to be extended.

Definition 2.3 (Interpretation). An interpretation $\mathcal{I}$ in FOL is a pair $(D_\mathcal{I}, \alpha_\mathcal{I})$, where $D_\mathcal{I}$ is the *domain* of $\mathcal{I}$, a nonempty set of objects, and $\alpha_\mathcal{I}$ is the assignment of $\mathcal{I}$. ▪

The assignment $\alpha_\mathcal{I}$ is constructed in the following way:

- Each variable x is mapped to a value from $D_\mathcal{I}$, usually denoted by $x_\mathcal{I}$.
- Each n-ary function symbol f is mapped to an n-ary function $f_\mathcal{I}$ that maps n elements of $D_\mathcal{I}$ to one element of $D_\mathcal{I}$, that is, $f_\mathcal{I} : D_\mathcal{I}^n \mapsto D_\mathcal{I}$. In particular, each constant symbol is assigned to an element of $D_\mathcal{I}$.
- Each n-ary predicate symbol p is mapped to an n-ary relation $p_\mathcal{I} \subseteq D_\mathcal{I}^n$.

Example 2.7. *Consider the formula $\psi = (x > 1) \wedge (y > 1) \rightarrow cos(x) + y > 1$.*

Let the domain be the set of real numbers, so $D_\mathcal{I} = \mathbb{R}$. To construct an assignment let cos and $+$ be assigned the cosine and addition function over real numbers, and assign the "greater-than" relation over $\mathbb{R}$ to the binary predicate symbol $>$. The variables x and y need to be assigned to, let them be 2 and $\sqrt{3}$ respectively. It is important to note that the constant symbol 1 needs to be assigned too, as it is a nullary function symbol. So the assignment is $\alpha_\mathcal{I} : \{cos \mapsto cos_\mathbb{R}, + \mapsto +_\mathbb{R}, > \mapsto >_\mathbb{R}, x \mapsto 2_\mathbb{R}, y \mapsto \sqrt{3}_\mathbb{R}, 1 \mapsto 1_\mathbb{R}\}$, and the interpretation is $\mathcal{I} = (\mathbb{R}, \alpha_\mathcal{I})$.

Note, that although in the preceding example all function and predicate symbols were assigned to their intuitive meaning, assigning the sine function over real numbers to the symbol cos, or the relation "less than" to the binary predicate $>$ also results in an assignment.

Like in case of PL, semantics determine if the formula ψ evaluates to true, or false under a given interpretation $\mathcal{I} = (D_\mathcal{I}, \alpha_\mathcal{I})$ (denoted by $\mathcal{I} \models \psi$ or $\mathcal{I} \not\models \psi$). The semantics is defined recursively.

Terms, that is, variables (x), constant symbols (a) and function symbols (f) get meanings based on $\alpha_\mathcal{I}$, denoted by $\alpha_\mathcal{I}[x]$, $\alpha_\mathcal{I}[a]$ and $\alpha_\mathcal{I}[f]$. Arbitrary terms can be evaluated recursively as follows:

- $\alpha_\mathcal{I}[f(t_1, t_2, \dots, t_n)] = \alpha_\mathcal{I}[f](\alpha_\mathcal{I}[t_1], \alpha_\mathcal{I}[t_2], \dots, \alpha_\mathcal{I}[t_n])$.

Then predicates can be evaluated as follows:

- $\mathcal{I} \models p(t_1, t_2, \ldots, t_n) \iff (\alpha_\mathcal{I}[t_1], \alpha_\mathcal{I}[t_2], \ldots, \alpha_\mathcal{I}[t_n]) \in \alpha_\mathcal{I}[p]$.

More complex formulas can be built using logical connectives the same way as it was defined in PL in Section 2.1.1.2.

Example 2.8. *Consider the formula* $\psi = (x > 1) \wedge (y > 1) \rightarrow cos(x) + y > 1$ *from Example 2.7, with the interpretation* $\mathcal{I} = (\mathbb{R}, \alpha_\mathcal{I} = \{cos \mapsto cos_\mathbb{R}, + \mapsto +_\mathbb{R}, > \mapsto >_\mathbb{R}, x \mapsto 2_\mathbb{R}, y \mapsto \sqrt{3}_\mathbb{R}, 1 \mapsto 1_\mathbb{R}\})$*. The truth value of* ψ *under the interpretation* $\mathcal{I}$ *can be computed in the following way:*

- $(\alpha_\mathcal{I}[x], \alpha_\mathcal{I}[1]) = (2_\mathbb{R}, 1_\mathbb{R}) \in >_\mathbb{R} = \alpha_\mathcal{I}[>]$*, thus* $\mathcal{I} \models x > 1$.
- $(\alpha_\mathcal{I}[y], \alpha_\mathcal{I}[1]) = (\sqrt{3}_\mathbb{R}, 1_\mathbb{R}) \in >_\mathbb{R} = \alpha_\mathcal{I}[>]$*, thus* $\mathcal{I} \models y > 1$.
- $(\alpha_\mathcal{I}[cos(x) + y], \alpha_\mathcal{I}[1]) = (\alpha_\mathcal{I}[cos(x)] +_\mathbb{R} \alpha_\mathcal{I}[y], 1_\mathbb{R}) = (cos_\mathbb{R}(\alpha_\mathcal{I}[x]) +_\mathbb{R} \sqrt{3}_\mathbb{R}, 1_\mathbb{R}) = (cos_\mathbb{R}(2_\mathbb{R}) +_\mathbb{R} \sqrt{3}_\mathbb{R}, 1_\mathbb{R}) \in >_\mathbb{R} = \alpha_\mathcal{I}[>]$*, thus* $\mathcal{I} \models cos(x) + y > 1$.

Applying the semantics of $\wedge$ *and* $\rightarrow$*,* $\mathcal{I} \models \psi$ *can be deduced.*

In order to define the semantics of quantifiers, the *x-variant* of an interpretation has to be defined. Given $\mathcal{I} = (D_\mathcal{I}, \alpha_\mathcal{I})$, an x-variant of $\mathcal{I}$ is an interpretation $\mathcal{I} \triangleright \{x \mapsto v\} = (D_\mathcal{I}, \alpha_{\mathcal{I}'})$ such that for every variable, function symbol and predicate symbol $y \neq x$ we have $\alpha_\mathcal{I}[y] = \alpha_{\mathcal{I}'}[y]$, and $\alpha_{\mathcal{I}'}[x] = v$. Then

- $\mathcal{I} \models \forall x.\psi \iff$ for all $v \in D_\mathcal{I}$ we have $\mathcal{I} \triangleright \{x \mapsto v\} \models \psi$,
- $\mathcal{I} \models \exists x.\psi \iff$ there exists $v \in D_\mathcal{I}$ such that $\mathcal{I} \triangleright \{x \mapsto v\} \models \psi$.

2.1.2.3 Satisfiability and Validity

The definition of satisfiability in FOL is similar to PL, a formula ψ is *satisfiable* iff there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \psi$, and *valid* iff for all interpretations $\mathcal{I}$ we have $\mathcal{I} \models \psi$. As in PL, the two concepts are the duals of each other.

Technically, satisfiability and validity can not be applied to FOL formulas with free variables. However, a non-closed formula ψ' is considered valid if $\forall * .\psi'$ is valid, so for every possible value for its free variables, the formula is valid. Using duality, the satisfiability of a non-closed formula can be deducted, ψ' is satisfiable if $\exists * .\psi'$. In the general case however, satisfiability and validity is undecidable, as it was proven by Church [6] and Turing [18].

2.1.3 First Order Theories

In case of FOL the interpretation of a formula could be literally anything, thus satisfiability and validity is undecidable. *First order theories* formalize structures like numbers or lists, in order to enable reasoning about them. For many quantifier free theories used in practice, satisfiability (thus validity) is decidable [14].

Definition 2.4 (First order theory). A first order theory $\mathcal{T}$ is a set of closed formulas, called *axioms*. ▪

An interpretation $\mathcal{I}$ is called a $\mathcal{T}$-interpretation iff $\mathcal{I} \models \psi$ for all axioms $\psi \in \mathcal{T}$. A formula is satisfiable in $\mathcal{T}$ iff it is satisfiable by a $\mathcal{T}$-interpretation. Dually, a formula is valid if it is satisfiable by all $\mathcal{T}$-interpretations.

Like the SAT problem for PL, the problem of deciding the satisfiability of a formula can be expressed for first order theories too.

Definition 2.5 (SMT problem). The *satisfiability modulo theories* problem, often referred as *SMT problem* is to decide the satisfiability of a formula in a theory $\mathcal{T}$. ▪

The algorithmic complexity of solving an SMT problem is dependent on the theory itself. There are decidable theories, such as the *theory of equality* ($\mathcal{T}_E$), the *theory of Presburger arithmetic* ($\mathcal{T}_{\mathbb{N}}$), or the *theory of integers* ($\mathcal{T}_{\mathbb{Z}}$). Some theories, like the extension of $\mathcal{T}_{\mathbb{Z}}$ with multiplication, the so-called *Peano arithmetic* ($\mathcal{T}_{PA}$), or the theory of rationals ($\mathcal{T}_{\mathbb{Q}}$) are undecidable [14]. For a decidable theory $\mathcal{T}_d$, there are SMT solver algorithms that always terminate, however for an undecidable theory $\mathcal{T}_u$, the solver may terminate, but it can also fail to decide satisfiability for ψ.

2.2 Statecharts

The language of statecharts [16] is a basic modeling formalism in system design, which offers various syntactic elements to simplify the modeling of complex systems. In Section 2.2.1 I introduce state machines, a mathematical and modeling concept from which statecharts originated. Section 2.2.2 introduces state hierarchy and statecharts, and Section 2.2.3 defines configurations and execution sequences (paths) for statecharts.

2.2.1 State Machines

Definition 2.6 (State). A *state* is a unique configuration of information about the system. ▪

Definition 2.7 (State machine). A finite state machine (finite state automaton, state machine) is a tuple $M = (S, \Sigma, Tr, s_0)$, where

- S is a finite set of states,
- Σ is the alphabet, the set of allowed symbols,
- $Tr \subseteq S \times S \times \Sigma$ is the set of transitions, with each of them connecting exactly one source state to one target state, and having an input symbol assigned,
- $s_0 \in S$ is the initial (start) state. ▪

For a transition t, the source state of the transition is denoted by $src(t)$ and the target state of the transition is denoted by $trgt(t)$. Let $sym(t) \in \Sigma$ denote the input symbol assigned to the transition.

Note that a transition does not always require an input symbol to be taken. This can be interpreted as introducing a ϵ symbol. Let the default notation be that if a transition has no symbol assigned, it is assigned ϵ. Let ϵ be in every Σ by default.

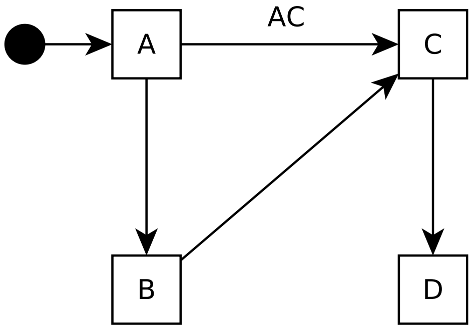


Figure 2.1: An example for state machine.

Example 2.9. *Figure 2.1 presents an example state machine. For this machine* $S = \{A, B, C, D\}$*,* $\Sigma = \{AC\}$*,* $Tr = \{(A, B), (A, C, AC), (B, C), (C, D)\}$*,* $s_0 = A$*.*

A state machine can be in exactly one state of its finite number of states, which is the so-called *active state*. A *transition* is the change of the active state.

The basic concept of state machines can be extended with *actions* (output of the machine). An action is a sequence of operations, that are usually interpreted as a sentence of a programming language. A simple operation can be either an assignment of a variable, or a generation of an event. Depending on which item of the tuple M is the output associated with, state machines can be considered as Mealy or as Moore machines.

Definition 2.8 (Mealy Machine). A *Mealy machine* is a tuple $M_{Mealy} = (S, \Sigma, Tr, s_0, Act)$ where S, Σ, s_0 are the same as for standard state machines, Act is a set of actions and $Tr \subseteq S \times S \times \Sigma \times Act$. ▪

Definition 2.9 (Moore machine). A *Moore machine* is a tuple $M_{Moore} = (S, \Sigma, Tr, O, s_0, Act)$ where S, Σ, Tr, s_0 are the same as for standard state machines, Act is a set of actions and $O : S \mapsto Act$. ▪

Informally, if a state machine's outputs are associated with the transition of the automaton, the state machine is considered a Mealy machine. In case of a Moore machine, the output is associated with the states.

Definition 2.10 (Path). For a state machine M, $\pi = (s_0, s_1, \ldots, s_n)$ is a *path* iff $s \in S$ (for $0 \leq i \leq n$), and $(s_i, s_{i+1}) \in Tr$ (for $0 \leq i < n$). ▪

The input of the state machine determines the path. Let $input_M(k)$ denote the input of the state machine after k elapsed transitions.

Example 2.10 (Path). *Consider the example state machine (M) presented in Figure 2.1. For M* $\pi_1 = (A, B, C, D)$ *and* $\pi_2 = (A, C, D)$ *are paths. Note that* π_1 *is always a path for M, while* π_2 *is a path only if* $input_M(0) = AC$*.*

2.2.2 Hierarchical Statecharts

In order to define statecharts, the concept of hierarchy has to be defined first.

Definition 2.11 (Hierarchy function). Let S be a set of states, R a set of regions and *root* an abstract object representing the top of the hierarchy. $par : S \cup R \mapsto S \cup R \cup \{root\}$ is a function that maps states to their parent region, and regions to their parent state or directly to the root of the hierarchy in a way that:

- for all $s \in S$ we have $par(s) \in R$, so every state is contained in a region,
- for all $r \in R$ it holds that $par(r) \in S \cup \{root\}$, so the parent of each region is either a state or the root object,
- there exists $r \in R$ such that $par(r) = root$, which means informally there is at least one region, that is contained directly by the root element of the hierarchy,
- for every $r \in R$ there exists a state $s \in S$ such that $par(s) = r$, so there are no empty regions. ▪

For a region $r \in R$, the state $par(r)$ is called the *parent state* of r, and for a state $s \in S$, the region $par(s)$ is called the *parent region* of s.

For convenience, lets define *chld* as the inverse of *par*. Note however, that *chld* is not an inverse in the mathematical sense as it maps a state to a set of regions and a region to a set of states. For a region $r \in R$, $chld(r) = \{s \in S \mid par(s) = r\}$, for a state $s \in S$, $chld(s) = \{r \in R \mid par(r) = s\}$, and for the *root*, $chld(root) = \{r \in R \mid par(r) = root\}$.

For a region r, the elements of $chld(r)$ are called *child-states* of r, and for a state s, the elements of $chld(s)$ are called *child-regions* of s. The member regions of $chld(root)$ are called *top-level regions.*

Let S be a set of states, R a set of regions, and *par* the hierarchy mapping between them. A state $s_{com} \in S$ is a *composite state* if $chld(s_{com}) \neq \emptyset$, and a state $s_{sim} \in S$ is a *simple state* iff s_{sim} is not composite. Informally, composite states are states that contain regions, whereas simple states are the ones that do not. Regions $r_1, r_2 \in R$ are *orthogonal* (or parallel) if $par(r_1) = par(r_2)$.

Definition 2.12 (Statechart). A statechart is a tuple $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ where the members of the tuple are the following.

- S is a finite set of states.
- R is a finite set of regions.
- $par : S \cup R \mapsto S \cup R \cup \{root\}$ is the state-hierarchy function as defined above.
- $I \subseteq S$ are the initial states such that for all $r \in R$, $|chld(r) \cap I| = 1$. Informally I contains exactly one initial state for every region.
- V is the set of variables.
- $Tr \subseteq S \times S \times EV \times G \times Act$ is the set of transitions with a trigger event, a guard and output actions assigned, where the trigger event $e \in EV$, the set of the possible events for Sc, guard is from the set of FOL formulas G that evaluate to a boolean value, and the output action is from the set of possible actions Act .
- $\mathcal{H} \subseteq R$ is the history marker, a set of regions that have history. ▪

The source and target state for a transition can be defined and denoted the same way as for a state machine, $src(t)$ denoting the source and $trgt(t)$ denoting the target state.

An event $e \in EV$ is a trigger for a transition t ($e = trig(t)$), if the transition is initiated by e. Informally, the transition can fire if and only if a trigger event is active. Since having a trigger is not required for a transition, there is a *default event* $\epsilon \in EV$ that is always considered active. A $g \in G$ is the guard of t transition ($g = grd(t)$), where g is an expression that can be evaluated to a boolean value, if $g = true$ is required for the transition to fire.

The output of a statechart is the same as the output of a Mealey machine. Act is the set of all possible output actions. Since an action is not required during a transition, there is an action $skip \in Act$ that has no effect. For a transition t, $act(t)$ denotes the output action that takes place when the transition fires.

Example 2.11. *Consider the statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ presented in Figure 2.2. Note that in the figure regions are denoted with dashed rectangles, whereas states denoted by rectangles with solid border. For this statechart*

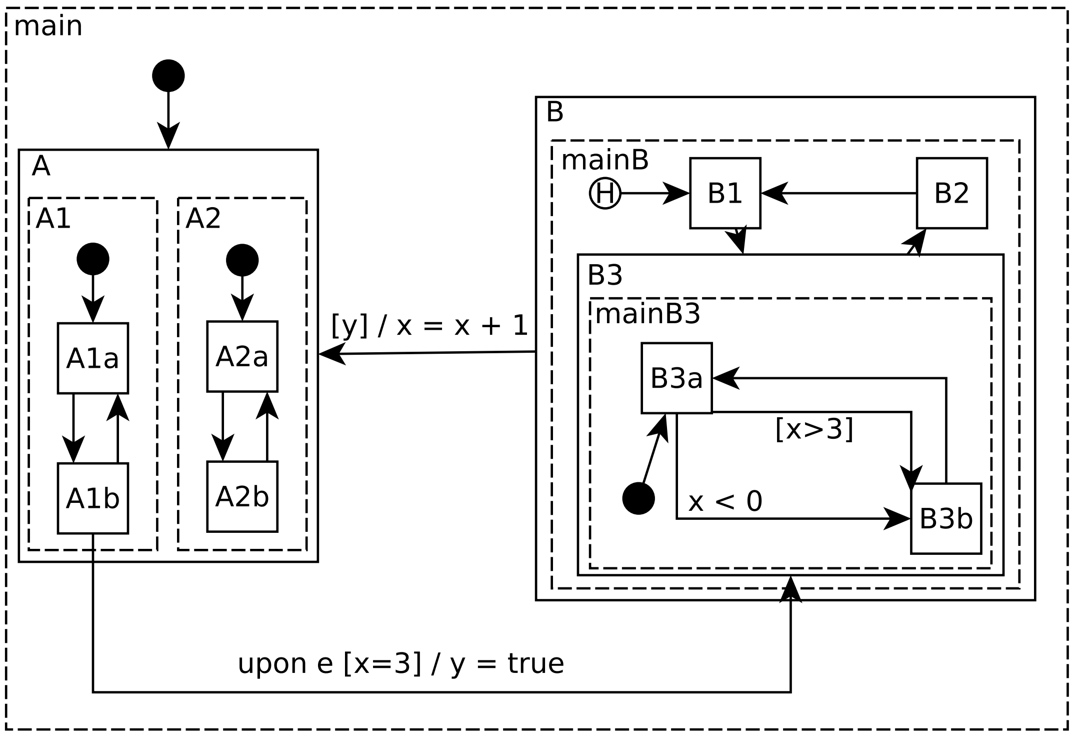


Figure 2.2: An example for statechart.

- $S = \{A, A1a, A1b, A2a, A2b, B, B1, B2, B3, B3a, B3b\}$. *The states* $A, B, B3$ *are composite states, the others are simple states,*
- $R = \{main, A1, A2, mainB, mainB3\}$,
- $par = \{main \mapsto root, A \mapsto main, A1 \mapsto A, A1a \mapsto A1, A1b \mapsto A1, \ldots\}$
- $I = \{A, A1a, A2a, B1, B3a\}$,

- $V = \{x, y\}$*, and their type can be implicitly derived from their values,* x *is an integer, and* y *is a boolean,*
- $Tr = \{(A1b, B3, e, x = 3, y \leftarrow true), (B, A, \epsilon, y = true, x \leftarrow x + 1) \ldots\}$*,*
- $\mathcal{H} = \{mainB\}$*.*

The regions $A1$ *and* $A2$ *are orthogonal regions.*

For the transition $t = (A1b, B3, e, x = 3, y \leftarrow true)$*,*

- $src(t) = A1b$*,*
- $trgt(t) = B3$*,*
- $trig(t) = e$*,*
- $grd(t) = (x = 3)$*,*
- $act(t) = \{(y \leftarrow true)\}$*.*

2.2.3 Statechart Configurations

Unlike a state machine, a statechart might have more than one active states at a time during its execution. However, there are strict rules for active states.

Formally, let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. Then $\omega \subseteq S$ is the set of active states for the statechart in a way that

1. for all $s_1, s_2 \in \omega$ we have $par(s_1) \neq par(s_2)$, so a region has at most one active state,
2. for every $r \in chld(root)$, $chld(r) \cap S \neq \emptyset$, which informally means that every top-level region has an active state,
3. for all $s \in \omega$ and $r \in chld(s)$ there exists $s' \in \omega$ such that $par(s') = r$, meaning that every region that is a child of an active state must contain an active state,
4. for all $s \in \omega$ we have $par(par(s)) = root$ or $par(par(s)) \in \omega$, so if a state is active, the parent state of its parent region is also active, unless it is in a top-level region.

Note, that the second constraint can be replaced with $root \in \omega$ if $\omega \subseteq S \cup \{root\}$.

Definition 2.13 (Statechart Configuration). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. Then $c = (\omega, \rho, \mathcal{F}, H)$ is the *configuration* of the statechart if

- ω is a valid set of active states for Sc,
- ρ is the currently active events on the input of the statechart,
- $\mathcal{F}$ is an interpretation for variables V,
- $H : \mathcal{H} \mapsto S$ is the history information, that stores the active state for every region marked with a history indicator. ▪

Example 2.12. *Consider the statechart presented in Figure 2.2. Example valid configurations for* Sc *are*

- $c_1 = (\{B1, B\}, \emptyset, \{x \mapsto 1, y \mapsto \mathit{true}\}, \{\mathit{mainB} \mapsto B1\})$. *Note that with this configuration, the history information of region mainB is not allowed to be anything else but $B1$,*
- $c_2 = (\{A, A1a, A2a\}, \{e\}, \{x \mapsto 2, y \mapsto \mathit{true}\}, \{\mathit{mainB} \mapsto B1\})$.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ let c_I denote the initial configuration of a statechart, and $C_{Sc} = \{c_1, c_2, \ldots\}$ denote all the possible configurations of Sc. Note that C_{Sc} is not necessarily a finite set.

Definition 2.14 (Transition Relation). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. $N \subseteq C_{Sc} \times C_{Sc}$ is the *transition relation* of Sc where $(c_1, c_2) \in N$ if there exists $t \in Tr$ such that t is enabled in c_1 (it is enabled by its trigger and guard), and after t fires, the configuration of Sc will be c_2. Furthermore, for a $c \in C_{Sc}$ let $N(c) = \{c' \in C_{Sc} \mid (c, c') \in N\}$. ▪

Informally, $N(c)$ is the set of the configurations that are reachable from c within a transition of Tr.

The definition of a path in a state machine, defined in Definition 2.10 can be extended to a definition for path in a statechart.

Definition 2.15 (Path). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. A sequence of configurations $\pi = (c_0, c_1, \ldots, c_n)$ is a *path* for Sc if $c_i \in C_{Sc}$ (for $0 \leq i \leq n$) and $(c_i, c_{i+1}) \in N$ (for $0 \leq i < n$) and $c_0 = c_I$. ▪

Informally, a path is a sequence of configurations with the initial configuration of the statechart as the first element, and each configuration in the path is reachable with a transition from the preceding one.

Example 2.13. *Consider the statechart Sc presented in Figure 2.2. Let the initial value of the variables be $x = 3, y = \mathit{false}$. Let the configurations c_0, c_1, c_2 be*

- $c_0 = (\{A, A1a, A2a\}, \{e\}, \{x \mapsto 3, y \mapsto \mathit{false}\}, \{\mathit{mainB} \mapsto B1\})$,
- $c_1 = (\{A, A1b, A2a\}, \{e\}, \{x \mapsto 3, y \mapsto \mathit{false}\}, \{\mathit{mainB} \mapsto B1\})$,
- $c_2 = (\{B, B3, B3a\}, \emptyset, \{x \mapsto 3, y \mapsto \mathit{true}\}, \{\mathit{mainB} \mapsto B3\})$.

In that case $\pi = (c_0, c_1, c_2)$ is a valid path for Sc.

2.3 Encoding Statecharts

In order to be able to automatically reason about the behavior of a statechart, it needs to be encoded to formulas. In this section, I present well-known techniques from the literature to encode state machines (Section 2.3.1) and primitive statecharts (Section 2.3.2) to logical formulas.

2.3.1 Encoding State Machines

Let $BV_n = \{0, 1\}^n$ be the set of bit vectors of length n. For a bit vector $bv \in BV_n$ let $bv(i)$ denote its i-th component $(0 \leq i < n)$.

For a state machine $M = (S, \Sigma, Tr, s_0)$ let $enc : S \mapsto BV_n$ be a function that assigns a distinct bit vector to every state in S.

Given a bit vector $bv \in BV_n$ and a variable set $\{v_0, v_1, \ldots v_{n-1}\}$, let $lit(bv(i)) = \{v_i$ if $bv(i) = 1$ and $\neg v_i$, if $bv(i) = 0\}$ assign a literal to each element of the bit vector.

Let $form : BV_n \mapsto FOL$ (where FOL denotes the set of first order formulas) be a function that assigns a formula to a bit vector in a way that

$$form(bv) = \bigwedge_{i=0}^{n-1} lit(bv(i)). \tag{2.1}$$

Informally this means, that a bit vector is encoded as a conjunction of variables, where for each bit, 0 is encoded as a negated and 1 is encoded as a ponated variable.

Finally, let $\psi_s : S \mapsto FOL$ be a function in a way that for a state $s \in S$

$$\psi_s(s) = form(enc(s)). \tag{2.2}$$

Given a bit vector $bv \in BV_n$ and a state s, the indexed formulas $form(bv)_k$ and $\psi_s(s)_k$ can be defined as presented in Section 2.1.2.1. The reason behind indexing this formula is to be able to reason about a sequence of bit vectors, and a sequence of states in a state machine.

There can be several different *enc* functions for a set of states S, one of the most intuitive ways are the *binary* encoding, and the 1 out of n encoding. As these two are quite analogous and the methods presented in Chapter 3 build on the binary encoding, only that one is presented.

In this case, for a set of states S, bit vectors of length $n = \lceil \log_2 |S| \rceil$ are required to assign each state in S a unique vector. This can be achieved by numbering the states starting from 0 to $|S| - 1$, and assigning a bit vector as an n long binary representation of the given number.

Example 2.14. *Consider the state machine presented in Figure 2.1, with $S = \{A, B, C, D\}$. Let them be numbered as $A \mapsto 0$, $B \mapsto 1$, $C \mapsto 2$, $D \mapsto 3$, so the assigned bit vectors are $\overline{00}$, $\overline{01}$, $\overline{10}$, $\overline{11}$ respectively. The value of function form for the four states given the variable set $\{V_0, V_1\}$ are*

- $form(\overline{00}) = \neg V_1 \wedge \neg V_0,$
- $form(\overline{01}) = \neg V_1 \wedge V_0,$
- $form(\overline{10}) = V_1 \wedge \neg V_0,$
- $form(\overline{11}) = V_1 \wedge V_0.$

Given the set of variables $\{P_{0,0}, P_{1,0}\}$ for $k = 0$ and $\{P_{0,1}, P_{1,1}\}$ for $k = 1$, the values of $\psi_s(s)_k$ for states A and B, and for $k = 0, 1$ are:

- $\psi_s(A)_0 = \neg P_{1,0} \wedge \neg P_{0,0},$
- $\psi_s(A)_1 = \neg P_{1,1} \wedge \neg P_{0,1},$
- $\psi_s(B)_0 = \neg P_{1,0} \wedge P_{0,0},$

- $\psi_s(B)_1 = \neg P_{1,1} \wedge \quad P_{0,1}$.

A transition $t \in \mathit{Tr}$ in a path $\pi = (s_0, s_1, \ldots s_n)$ occurs if there exists an index i such that $\mathit{src}(t) = s_i$ and $\mathit{trgt}(t) = s_{i+1}$ $(0 \leq i < n)$. For transition t to fire, the input symbol $\mathit{sym}(t) \in \Sigma$ is also required.

For a state machine $M = (S, \Sigma, \mathit{Tr}, s_0)$ let $\psi_t : \mathit{Tr} \mapsto \mathit{FOL}$ be a function such that for every $t \in \mathit{Tr}$,

$$\psi_t(t)_k = \psi_s(\mathit{src}(t))_k \wedge \psi_s(\mathit{trgt}(t))_{k+1} \wedge (\mathit{input}_M(k) = \mathit{sym}(t)). \tag{2.3}$$

Informally, a formula assigned to a transition is the conjunction of the formula of the source state, the target state at the next step of the execution, and the existence of the input symbol that is required for the transition to fire.

Combining Equations 2.2 and 2.3,

$$\psi_t(t)_k = \mathit{form}(\mathit{enc}(\mathit{src}(t)))_k \wedge \mathit{form}(\mathit{enc}(\mathit{trgt}(t)))_{k+1} \wedge (\mathit{input}_M(k) = \mathit{sym}(t)). \tag{2.4}$$

For a path $\pi = (s_0, s_1, \ldots, s_n)$ in the state machine M, define an interpretation $\mathcal{I}_\pi$ in a way that $\mathcal{I}_\pi \models \psi_s(s_i)_i$ for every $0 \leq i \leq n$, and $\mathcal{I}_\pi \not\models \psi_s(s_i)_j$ if $i \neq j$.

Informally, $\mathcal{I}_\pi$ is an interpretation for the variables used to encode M into formulas in a way that $\mathcal{I}_\pi$ uniquely determines π.

Note that for a path $\pi = (s_0, s_1, \ldots, s_n)$ $\mathcal{I}_\pi \models \psi_t(t)_k$ iff the k'th element of π is $\mathit{src}(t)$ and the $k+1$'th is $\mathit{trgt}(t)$.

Define the FOL formula ψ_{Tr} as $(\psi_{\mathit{Tr}})_k = \bigvee_{t \in \mathit{Tr}} \psi_t(t)_k$. Note that $(\psi_{\mathit{Tr}})_k$ evaluates to true, if after k transitions, another transition fires in M. The formula $(\psi_{\mathit{Tr}})_k$ can be referred to as the *transition relation formula of M*.

For a state machine $M = (S, \Sigma, \mathit{Tr}, s_0)$ let ψ_{M_k} be a formula such that

$$\psi_{M_k} = \left(\bigwedge_{i=0}^{k}((\psi_{\mathit{Tr}})_i\right) \wedge \psi_s(s_0)_0 = \left(\bigwedge_{i=0}^{k} \bigvee_{t \in \mathit{Tr}} \psi_t(t)_i\right) \wedge \psi_s(s_0)_0. \tag{2.5}$$

The formula ψ_{Tr} contains restrictions about the transitions that are allowed to fire. If it is *unfolded* k times, it restricts k consecutive transitions to be valid. The formula ψ_{M_k} is the conjunction of $(\psi_{\mathit{Tr}})_i$, as all the transitions are required to be valid. The conjunction of the formula $\psi_s(s_0)_0$ is required as the execution of the state machine can only start from the initial state.

It can be proven that for each possible k long path π of M, $\mathcal{I}_\pi \models \psi_{M_k}$, and for every other interpretation $\mathcal{I}'$, $\mathcal{I}' \not\models \psi_{M_k}$. The construction of formula ψ_{M_k} is also referred as *unfolding ψ_{Tr} k times*.

It can also be seen, that from the interpretation satisfying the formula ψ_{M_k}, a path π can be retained, by decoding the interpretation for each $0 \leq i \leq k$, and constructing a path from s_i's.

Example 2.15. *Consider the example state machine in Figure 2.1. For this machine $S = \{A, B, C, D\}$, $\Sigma = \{AC\}$, $\mathit{Tr} = \{(A, B), (A, C, (AC)), (B, C), (C, D)\}$, and $s_0 = A$. ψ_t for the machine is:*

- $\psi_t((A,B))_k = \psi_s(A)_k \wedge \psi_s(B)_{k+1}$
- $\psi_t((A,C))_k = \psi_s(A)_k \wedge \psi_s(C)_{k+1} \wedge (input_M(k) = AC)$
- $\psi_t((B,C))_k = \psi_s(B)_k \wedge \psi_s(C)_{k+1}$
- $\psi_t((C,D))_k = \psi_s(C)_k \wedge \psi_s(D)_{k+1}$

For this state machine $(\psi_{Tr})_k = \psi_t((A,B))_k \vee \psi_t((A,C))_k \vee \psi_t((B,C))_k \vee \psi_t((C,D))_k$ *that can be expressed as presented in Equation 2.6.*

$$\begin{aligned}&\psi_s(A)_k \wedge \psi_s(B)_{k+1} \ \vee \psi_s(A)_k \wedge \psi_s(C)_{k+1} \wedge (input_M(k) = AC) \vee \\ &\psi_s(B)_k \wedge \psi_s(C)_{k+1} \ \vee \psi_s(C)_k \wedge \psi_s(D)_{k+1}\end{aligned} \tag{2.6}$$

This unfolded twice is $\psi_{M2} = (\psi_{Tr})_0 \wedge (\psi_{Tr})_1 \wedge \psi_s(A)_0$*, which results in Equation 2.7.*

$$\begin{aligned}\psi_{M2} = &\psi_s(A)_0 \wedge \\ &(\psi_s(A)_0 \wedge \psi_s(B)_1 \ \vee \ \psi_s(A)_0 \wedge \psi_s(C)_1 \wedge (input_M(0) = AC) \vee \\ &\psi_s(B)_0 \wedge \psi_s(C)_1 \ \vee \ \psi_s(C)_0 \wedge \psi_s(D)_1) \wedge \\ &(\psi_s(A)_1 \wedge \psi_s(B)_2 \ \vee \ \psi_s(A)_1 \wedge \psi_s(C)_2 \wedge (input_M(1) = AC) \vee \\ &\psi_s(B)_1 \wedge \psi_s(C)_2 \ \vee \ \psi_s(C)_1 \wedge \psi_s(D)_2)\end{aligned} \tag{2.7}$$

Using the binary state encoding method presented before, this can be transformed to a FOL *formula. One possible interpretation satisfying this formula is* $\mathcal{I} = \{P_{0,0} = 0, P_{1,0} = 0, P_{0,1} = 1, P_{1,1} = 0, P_{0,2} = 1, P_{1,2} = 1, input_M = \{0 \mapsto AC, 1 \mapsto \epsilon, \ldots\}, \ldots\}$*. So the path* π *corresponding to this interpretation is* (A, C, D)*.*

2.3.2 Encoding Statecharts

The transformation presented above can easily be extended to a subset of statecharts that meet some additional requirements.

Definition 2.16 (Flat Statechart). Let statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a *flat statechart* if Sc does not contain hierarchy, parallel regions (so there is only one region, contained by the root of the hierarchy), formally $|R| = 1$, $|chld(root)| = 1$. ▪

From now on, lets assume that for a statechart $|V| = 0$, and $|\mathcal{H}| = 0$, so there are no vars in the statechart, and in the only region, there is no history.

The encoding of flat statecharts is really similar to the encoding of state machines presented in the previous section, as with only one region, for every $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$, $\omega = \{s\}$ and as there are no variables and history, $\mathcal{F} = \{\}$ and $|H| = 0$. For now, lets also assume that $|\rho| \leq 1$, so there is at most one active event. This can be also modeled as $\rho = \{e\}$, where e is allowed to be the default event ϵ, noting that there is no active event in the statechart. For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ with configurations C_{Sc}, let $\psi_c : C_{Sc} \mapsto FOL$ be a function that assigns a formula to every state configuration. Due to the restrictions presented above, for a configuration $c \in C_{Sc}$, $\psi_c(c)_k$ can de defined as $\psi_s(s)_k \wedge \psi_{EV}(e)_k$, where $\psi_s(s)_k$ can be defined similarly as for state machines.

However ψ_{EV} has to be defined. Events can be encoded just the same as states, let $enc : EV \mapsto BV_n$ be a function that assigns a unique bit vector for each event. Then

$\psi_{EV} : EV \mapsto FOL$ is a function such that

$$\psi_{EV}(e)_k = \mathit{form}(\mathit{enc}(e))_k \tag{2.8}$$

As for states, there are several methods to assign bit vectors to events, and the presented one is the binary. However binary encoding, like the 1 out of n allow only one active event. In order to support the cases where $|\rho| > 1$, the 1 out of n method can be extended to k out of n.

A transition of a statechart can only fire, if its guard evaluates to true, and if its trigger event is active. For a transition t, $\mathit{grd}(t)$ is a formula that evaluates to a truth value, thus the transition is enabled if it evaluates to $\top$.

For a flat statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, let $\psi_t : Tr \times \mathbb{N} \mapsto FOL$ be a function that assigns a first order logic formula to every transition of Sc. The value of ψ_t for a $t \in Tr$ is presented in Equation 2.9.

$$\psi_t(t)_k = \psi_s(\mathit{src}(t))_k \wedge \psi_s(\mathit{trgt}(t))_{k+1} \wedge \psi_{EV}(\mathit{trig}(t))_k \wedge \mathit{grd}(t)_k \tag{2.9}$$

$\mathcal{I}_\pi$ can be introduced for statecharts too, only with the extension of $\pi = (c_0, c_1, \dots, c_n)$ being a sequence of configurations for Sc.

The formula ψ_{Tr} can be defined as the disjunction of formulas ψ_t for every transition in Tr, and ψ_{Sck} can be defined just the same as for state machine, such that for every valid path $\pi = (c_0, c_1, \dots, c_n)$ in Sc, $\mathcal{I}_\pi \models \psi_{Sck}$, and for every other interpretation $\mathcal{I}' \not\models \psi_{Sck}$.

The only difference is that instead of the initial state formula $\psi_s(s_0)_0$, the initial configuration formula $\psi_c(c_I)_0$ is conjuncted to the transition conjunction as seen below.

$$\psi_{Sck} = \psi_c(c_I)_0 \wedge \left(\bigwedge_{i=0}^{k} (\psi_{Tr})_i \right) \tag{2.10}$$

The preceding equation can be extracted to

$$\psi_c(c_I)_0 \wedge \left(\bigwedge_{i=0}^{k} \bigvee_{t \in Tr} (\psi_s(\mathit{src}(t))_i \wedge \psi_s(\mathit{trgt}(t))_i \wedge \psi_{EV}(\mathit{trig}(t))_i \wedge \mathit{grd}(t)_i) \right). \tag{2.11}$$

For encoding hierarchical statecharts, the state-of-the-art solutions [3] are transforming the statecharts to the input of an other model checker [1] or *flattening* [13]. For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ flattening creates a statechart $Sc' = (S', R', par', I', V', Tr', \mathcal{H}')$ such that Sc' is flat, and there is a bijection between the elements of C_{Sc} and $C_{Sc'}$ such that the transition relation is the same for both of them. With flattening, the size of the statechart grows, and the information stored in the hierarchy is lost.

Example 2.16. *An example for flattening a statechart can be found in Figure* 2.3.

The statechart on the top is the hierarchic one, and its flat equivalent can be seen below. Each transition has one or more corresponding transitions in the flattened statechart.

This example points out that parallel regions reduce the number of states and transitions, whereas composite states reduce the number of transitions in the statechart, however they introduce more states.

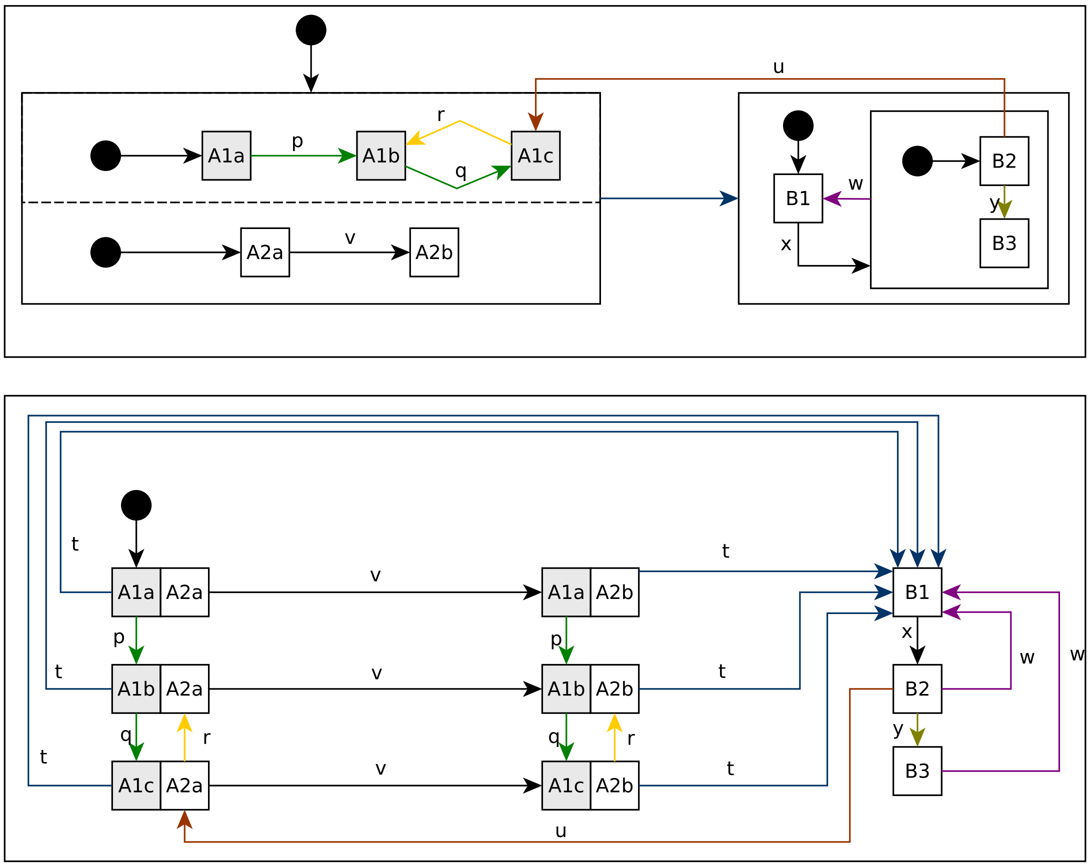


Figure 2.3: Example for flattening.

2.4 Model Checking

Model checking is the concept of automatically verifying the model of a behavioral system against a set of given requirements by systematically exploring the state space of the system. As models and requirements both vary on a wide spectrum, there are several algorithms. In this section, I present model checking techniques related to reachability properties that can be applied during the verification of statecharts.

2.4.1 Safety and Reachability

Safety and reachability are global properties of a statechart.

Definition 2.17 (Reachable configuration). For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, the configuration $c \in C_{Sc}$ is considered *reachable* if there exist a path $\pi = (c_0, c_1, \ldots, c_n)$ in Sc such that $c = c_n$ for some n. ∎

Let $C_{Sc\,r} \subseteq C_{Sc}$ denote the set of the reachable configurations for Sc.

An *error-state* or *false-state* is a state configuration for a statechart, that should not be reached during the execution in order to assure faultless functionality.

Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. For each configuration $c \in C_{Sc}$ a predicate p can be defined such that $p(c) = true$, if c is an error state, otherwise $p(c) = false$.

The predicate function p is defined by the designer of the system, as it can be different for every statechart, so it can be interpreted as an input for the verification method.

Definition 2.18 (Safety). A statechart Sc is safe for the predicate function p if for every $c_r \in C_{Scr}$ $p(c_r) = \mathit{false}$. ▪

Definition 2.19 (Reachability). In a statechart Sc with the predicate function p, reachability holds if there is a $c_r \in C_{Scr}$ such that $p(c_r) = \mathit{true}$. ▪

Note that safety and reachability are duals to each other, as the reachability of a bad state is equivalent to the unsafety of the statechart.

The existence of bad-states can be proven by providing a path to it.

Definition 2.20 (Counterexample). A path $\pi = (c_0, c_1, \ldots, c_n)$ is a counterexample for the predicate function p if $p(c_n) = \mathit{true}$. ▪

Note that for convenience, the counterexample does not only contain the error state, but it contains it as its last member.

2.4.2 State Space Exploration

The concept of *state space exploration* is the basic method for model checking. The algorithm explores all the reachable states for a system. In case of a statechart, it is equivalent to C_{Scr}, the set of reachable configurations.

It is important to note, that the state space can be unmanageably large, or even infinite as the domains of variables can be infinite too. However state space exploration still can be used to test the statechart against reachability requirement as upon finding a counterexample, the algorithm terminates.

The exploration can be done by an interpreter, starting from the initial configuration, and preforming a BFS[1], maintaining the reached configurations. If the execution reaches a configuration that is an error state, the algorithm terminates and an explored path to the configuration is returned as a counterexample. Logical solvers can also be used for finding reachable configurations from a configuration $c \in C_{Sc}$ within one transition, as described in Section 2.3.

2.4.3 Bounded Model Checking

State space exploration can handle statecharts with small state space, however as with the introduction of parallel regions and variables the state space grows exponentially, or even becomes infinite, preventing termination. *Bounded model checking* aims to overcome this problem.

Bounded Model Checking [4], abbreviated as BMC, is an iterative process of checking if a reachability requirement is violated.

Definition 2.21 (k-reachability). Let C_{Sc} be the set of configurations for a statechart $Sc = (S, R, \mathit{par}, I, V, \mathit{Tr}, \mathcal{H})$. A configuration $c \in C_{Sc}$ is *k-reachable* if a path π with length k exists to c. ▪

[1]It can also be performed by DFS, but it may fail to find the shortest counterexample.

Informally, a configuration is k-reachable, if it is reachable from the initial configuration within k transitions.

During the process, the value of k is incremented in each step, starting from 0, and the k-reachability of the given configuration is tested. If the state is k-reachable, it is also reachable, so the requirement is violated, otherwise k is incremented. The loop continues until a counterexample is found or a limit of execution (in computational resources or in the value of parameter k) is reached. For that reason, BMC can not be considered complete, as there might be false positives (reachable configurations marked as unreachable).

In practice, checking k-reachability for a configuration is often realized by logical solvers (SAT/SMT). The transition relation of the statechart is transformed into formulas, and unfolded k times (as presented in Section 2.3), and so does the reachability requirement in such way that the satisfiability of the and clause of these formulas is equivalent to the reachability of the configuration, and a satisfying interpretation gives a path as a counterexample.

2.4.4 Counterexample-Guided Abstraction Refinement

BMC might handle infinite state space, but for a large state space the solvers still have to find a value for each variable, however this is not always necessary to prove reachability or safety.

Counterexample-Guided Abstraction Refinement (CEGAR) [7] is a general approach to perform analysis in state transition systems with large or even infinite state space. The CEGAR algorithm verifies requirements in an abstract representation of the system. *Abstraction* is a mathematical approach to hide irrelevant details of a system. CEGAR uses existential abstraction, that is an over-approximation of the system, meaning that if a requirement holds in the original model, it also holds in the abstract one, however the abstraction can introduce additional behavior. If such behaviors have impact on the result of the verification, the abstraction has to be *refined*.

The algorithm contains four major steps, creating an initial abstraction for the system, verifying the abstract model against a given requirement, examining the output of the verification and refining the abstraction if needed. The flowchart of the CEGAR algorithm is presented in Figure 2.4.

The creation of the initial abstraction is based on some heuristics, however coarse abstractions are preferred as the algorithm refines it if needed. The verification can be done by one of the model checking methods, i.e. the ones presented above. If the checking does not find a counterexample, due to the existential property of the abstraction, there is no counterexample in the original model. Although if a counterexample is found, it has to be *concertized*, which means searching for a corresponding counterexample in the original model. If such concrete counterexample exists, the requirement is violated, otherwise the counterexample is called *spurious*, and the abstraction needs to be refined, with adding extra details to prevent checking methods on the abstract model to find the same counterexample again.

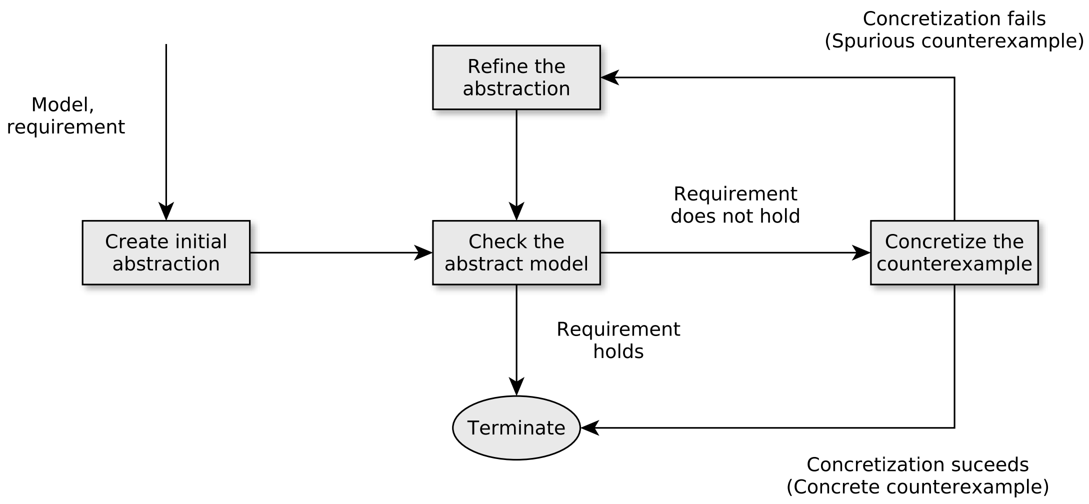


Figure 2.4: Flowchart of CEGAR.

Chapter 3

Encoding Hierarchical Statecharts

In this chapter, I present techniques that can be used to encode hierarchical statecharts. The methods presented in Section 2.3 do not take hierarchy into consideration. Furthermore, instead of making benefit of it, the algorithms become more complex as the depth of the hierarchy increases.

First, in Section 3.1, I suggest a method to assign interpretations to states that perseveres the hierarchy. In Section 3.2, I introduce an algorithm to transform statecharts with hierarchy into logical formulas, using the previously introduced numbering. In Sections 3.3 and 3.4 I demonstrate how these formulas can be used in practice to verify statecharts.

3.1 Numbering States Persevering the Hierarchy

Section 2.3 presented two possible approaches to implement the function *enc*, but those can only be applied to state machines and simple statecharts since only one active state was allowed. In case of hierarchical statecharts, all information stored in the hierarchy was lost when the statechart was flattened.

During my work, I focused on creating an encoding that transforms states to bit vectors in a way that the hierarchy information is persevered.

For this purpose, lets extend the concept of bit vectors: let a bit vector be a sequence of symbols from the set $\{0, 1, X\}$, where X is the *don't care* bit, marking that its value can be either 0 or 1. Let $BV_n = \{0, 1, X\}^n$ be the set of bit vectors of length n.

A bit vector bv of length n is *complete* if it does not have a don't care bit, formally $bv(i) \neq X$ for every $0 \leq i < n$.

The bit vectors $bv_1, bv_2 \in BV_n$ can be *combined* if they don't have any conflicting bits. Two bits are conflicting if they are different and none of them is don't care. This can be formalized as $bv_1(i) = bv_2(i)$ or $bv_1(i) = X$ or $bv_2(i) = X$ for every $0 \leq i < n$.

Let the *combination* of two combinable bit vectors bv_1, bv_2 be denoted by $comb(bv_1, bv_2)$. The combination vector's each bit is the not don't care bit at the same position in the two combined bits. If both bits are don't care, the corresponding bit in the combined bit vector is don't care too. Formally, if $comb(bv_1, bv_2) = bv_c$ for each $0 \leq i < n$ we have $bv_c(i) = bv_1(i)$ if $bv_1(i) = bv_2(i)$ or $bv_2(i) = X$, otherwise $bv_c(i) = bv_2(i)$.

If two bit vectors can not be combined, they are *conflicting*.

The bit vectors $bv_1, bv_2 \in BV_n$ are *disjunct* if $bv_1(i) = X$ or $bv_2(i) = X$ for every $0 \leq i < n$. Every disjunct bit vector pair bv_1, bv_2 can be combined.

Example 3.1. *Let bit vectors bv_1, bv_2, bv_3 be $\overline{00XX}$, $\overline{0X11}$, $\overline{XXX0}$ respectively.*

- *Bit vectors bv_1 and bv_2 can be combined, their combination is $bv_4 = \overline{0011}$. Note that bv_4 is complete.*
- *Bit vectors bv_1 and bv_3 can be combined, their combination is $\overline{00X0}$. Note that bv_1 and bv_3 are disjunct.*
- *Bit vectors bv_2 and bv_3 can not be combined as they are conflicting in their fourth bit.*

Combination can be defined inductively for $k \geq 2$ bit vectors too. With combination defined for $k-1$ bit vectors, bit vectors $bv_1, bv_2, \ldots bv_k$ are combinable if the combination $bv_{c'} = comb(bv_1, bv_2, \ldots bv_{k-1})$ exists and $bv_{c'}$ is combinable with bv_k. Then the combination of the k bit vectors is the combination of $bv_{c'}$ and bv_k, so

$$comb(bv_1, bv_2, \ldots bv_k) = comb(bv_{c'}, bv_k). \tag{3.1}$$

If k bit vectors can't be combined, they are conflicting. Note that the conflict can only be due to the conflict of two bit vectors. This results in that if in a set of bit vectors $\{bv_1, bv_2, \ldots bv_k\}$ every pair of bit vectors bv, bv' is non-conflicting (so combinable), the k bit vectors are also combinable.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, let the function $enc : S \mapsto BV_n$ be the encoding function that assigns a unique bit vector to each state in a way that for every configuration $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$, the set of bit vectors $\{enc(s_i) | s_i \in \omega\}$ is not conflicting, and their combination is complete. The enc for the set of states ω can be defined as

$$enc(\omega) = comb(\{enc(s_i) | s_i \in \omega\}). \tag{3.2}$$

During my work, as the main priority was to minimize the variables used for encoding states, I extended the binary way of encoding states to bit vectors.

Let Sc be a flat statechart, and let enc be the same function that was presented in Section 2.3.1. It is trivial that this encoding meets the requirement of the encoding function for statecharts, as there is always exactly one active state for a flat statechart.

From now on, the encoding of flat statecharts will be generalized with each subsection releasing the constraints required for the encoded statechart.

3.1.1 Parallel Regions

Parallel regions have the property that in each region there can be only one active state. Lets release the constraint of a flat statechart, by allowing more than one regions, with the restriction that each region is a top level region. Note that this is equivalent to the constraint that the statechart must not contain any composite state.

As the active states in orthogonal regions are independent to each other, the bit vectors assigned to configurations should have independent segments, each segment referring to the active state in its associated region. A *segment* is represented by a pair of integers, the offset and the length of the segment. The segments combined should make up the whole bit vector, and they should not have common bits.

For a region $r \in R$ let $bits(r)$ denote the minimum number of bits required to encode states in r. In order to assign a unique bit vector to each state in r, bit vectors of length $n \geq \lceil \log_2 |chld(r)| \rceil$ are required, so

$$bits(r) = \lceil \log_2 |chld(r)| \rceil \, . \tag{3.3}$$

This assures that each region will have enough bits to encode all the states inside the region.

Let $R = \{r_1, r_2, \ldots r_n\}$ be a set of regions with a given order. Let the function $\mathit{offs} : R \mapsto \mathbb{N}$ be a function that assigns the offset of the segment assigned to each region, and let the value of it be calculated as presented by Equation 3.4.

$$\mathit{offs}(r_i) = \sum_{j=1}^{i-1} bits(r_j) \tag{3.4}$$

So for each region $r \in R$, a segment $(\mathit{offs}(r), bits(r))$ is assigned, and the total length of the assigned bit vector will be the sum of the length of the assigned segments so $\sum_{i=1}^{n} bits(r_i)$.

This guarantees that the segments are distinct for every $r_i, r_j \in R$, since the intervals $[\mathit{offs}(r_i), \mathit{offs}(r_i) + bits(r_i))$ and $[\mathit{offs}(r_j), \mathit{offs}(r_j) + bits(r_j))$ are distinct. Each segment is inside the interval $[0, \sum_{r \in R} bits(r))$. Note that [and] denote an inclusive, whereas (and) denote an exclusive interval boundary.

For a state $s \in S$, let bv_s be a bit vector of length $bits(par(s))$, a unique bit vector in the scope of the states inside region $par(s)$ that does not contain any don't care bits, with the extra requirement that if s is the initial state of r, the bit vector assigned to it contains only 0's.

Let $enc : S \mapsto BV_n$ be a function that assigns a bit vector of length $n = \sum_{r \in R} bits(r)$ to every state $s \in S$ such that $bv(i) = bv_s(i - \mathit{offs}(r))$, if $0 \leq i < bits(r)$, otherwise $bv(i) = X$.

Example 3.2. *Consider the example statechart presented in Figure 3.1.*

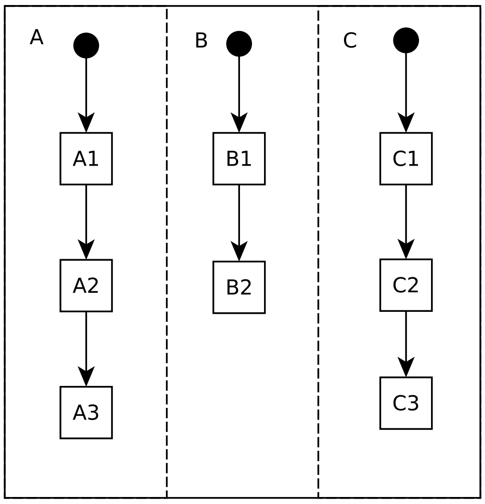


Figure 3.1: Statechart for Example 3.2.

There are three regions:

- *A, containing 3 states: $A1, A2, A3$, so $bits(A) = 2$,*
- *B, containing 2 states: $B1, B2$, so $bits(B) = 1$,*
- *C, containing 3 states: $C1, C2, C3$, so $bits(C) = 2$.*

One possible value for the offset function is $\{A \mapsto 0, B \mapsto 2, C \mapsto 3\}$. [1] *Let the unique bit vector bv_s value for states in the same regions as follows:*

- *For region A, $bv_{A1} = \overline{00}$, $bv_{A2} = \overline{01}$, $bv_{A3} = \overline{10}$*
- *For region B, $bv_{B1} = \overline{0}$, $bv_{B2} = \overline{1}$*
- *For region C, $bv_{C1} = \overline{00}$, $bv_{C2} = \overline{01}$, $bv_{C3} = \overline{10}$*

The value of enc for each state is as follows: $A1$: $\overline{00XXX}$, $A2$: $\overline{01XXX}$, $A3$: $\overline{10XXX}$, $B1$: $\overline{XX0XX}$, $B2$: $\overline{XX1XX}$, $C1$: $\overline{XXX00}$, $C2$: $\overline{XXX01}$, $C3$: $\overline{XXX10}$.

For the active states $\omega = \{A2, B1, C3\}$, the combination of the encoded bit vectors is $\overline{01110}$.

Theorem 3.1. Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a flat statechart with parallel regions. For every state configuration $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$, the set of bit vectors assigned to each active state are combinable, and their combination is complete. ▪

Proof. For each region $r \in R$ there is an active state in ω. The distinction of intervals $[\mathit{offs}(r), \mathit{offs}(r) + \mathit{bits}(r'))$ and $[\mathit{offs}(r'), \mathit{offs}(r') + \mathit{bits}(r'))$ for every region pair $r, r' \in R$ assures that the bit vectors can be combined, as there is only one state in ω for each region. As for every $s \in S$, bv contains only 0 and 1 bits, the combination of the bit vectors assigned to the states in ω will be complete. ▪

3.1.2 Hierarchically Nested States

In this subsection an other constraint is released, namely states are allowed to be composite.

Definition 3.1 (Ancestor state). For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, s is an *ancestor state* of s', if s is the parent state of s' or one of its ancestors, formally if $par(par(s')) = s$, or s is an ancestor to $par(par(s'))$. For a state s the set of ancestor states is denoted by $anc(s)$. ▪

Note that a state can have more than one ancestors, and that the *root* pseudo state is an ancestor of every state.

Definition 3.2 (Descendant states). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. For a state $s \in S$ the *descendant states* are the elements of the set $\{s_i \mid s_i \in S$ and $s \in anc(s_i)\}$. The state s' is the *descendant* of s, which is denoted by $s' \in desc(s)$ iff $s \in anc(s')$. ▪

[1] It is possible for the offset function to have other values, in this case for example $\{A \mapsto 0, C \mapsto 2, B \mapsto 4\}$ also meets the given requirements.

Informally, the descendant states of s are the states in a statechart, for which s is an ancestor.

Ancestors and descendants can be defined for regions as well. The state s is an ancestor to region r ($s \in anc(r)$) if $s = par(r)$ or $s \in anc(par(r))$, and the state s is the descendant of region r ($s \in desc(r)$), if there exists $s' \in chld(r)$ such that $s' = s$ or $s' \in anc(s)$, so if the region contains the state, or one of its ancestors.

Definition 3.3 (Depth of state). For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, let $depth : S \mapsto \mathbb{N}$ be a function that assigns the number of its ancestor states to a state. Inductively defined, $depth(root) = 0$, and for every $s \in S$ $depth(s) = depth(par(par(s))) + 1$.■

The integer $d = \max(\{depth(s) \mid s \in S\})$ is the *maximum depth* of the hierarchy. Let the i-th *level* of the hierarchy refer to the set of states $\{s \in S \mid depth(s) = i\}$.

Example 3.3. *Consider the statechart presented in Figure 3.2.*

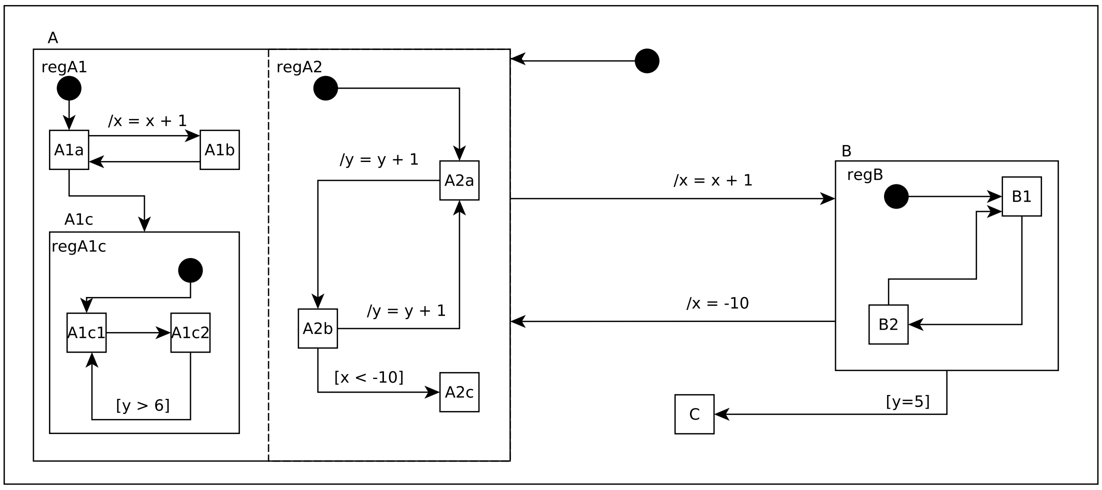


Figure 3.2: An example statechart.

The states with $depth(s) = 1$, so the stated of level 1 are A, B and C.

The states with $depth(s) = 2$, so the stated of level 2 are A1a, A1b, A1c, A2a, A2b, A2c, B1, B2.

The states with $depth(s) = 3$, so the stated of level 3 are A1c1 and A2c2.

The depth of the statechart is 3, as the deepest level is 3

For a region $r \in R$, let $bits(r)$ be the minimum number of bits required to encode the region, assuming that each contained state is simple, so $bits(r) = \log_2\lceil|chld(r)|\rceil$.

For a composite state $s_c \in S$, let $bits(s_c)$ be $\sum_{r \in chld(s_c)} bits(r)$, so the sum of the minimum bits required to encode each region and for a simple state $s_s \in S$, let $bits(s_s)$ be 0.

For the i-th level in a statechart, let $bits(i)$ denote the minimum bits required to encode that level, which is obviously the maximum of the minimum required bits for each state in that level, so formally $bits(i) = \max(\{bits(s) \mid s \in S \text{ and } depth(s) = i\})$. In the deepest level, there is no composite state (otherwise there would be another level), so $bits(d) = 0$.

The active states of the statechart can be encoded into a bit vector in a way that to each level a bit segment of fixed length is assigned that marks in which state the statechart is at that level. As for level i, $\mathit{bits}(i)$ bits are enough, let the length of the assigned segment be $\mathit{bits}(i)$.

In order to formalize it, let the function *offs* assign an offset to a level such that $\mathit{offs}(0) = 0$, and $\mathit{offs}(i) = \sum_{j=0}^{i-1} \mathit{bits}(j)$ for every $0 < i \leq d$, where d is the maximum depth of the statechart.

Eventually, all states will be encoded to a bit vector of length $n = \sum_{i=0}^{d} \mathit{bits}(i)$.

Example 3.4. *The statechart Sc presented in Example 3.3 is three levels deep, and the value of* $\mathit{bits}(i)$ *are* $2, \max(2+2, 1), 1, 0$ *respectively. For each state, a* $2 + 4 + 1 + 0 = 7$ *bit long bit vector is assigned. The offset that is assigned for the levels is listed below.*

- *For level 0,* $\mathit{offs}(0) = 0$ *is assigned.*
- *For level 1,* $\mathit{offs}(1) = \mathit{bits}(0) = 2$ *is assigned.*
- *For level 2,* $\mathit{offs}(2) = \mathit{bits}(0) + \mathit{bits}(1) = 6$ *is assigned.*
- *For level 3,* $\mathit{offs}(3) = \mathit{bits}(0) + \mathit{bits}(1) + \mathit{bits}(2) = 7$ *is assigned.*

So the bit segments of a bit vector bv assigned to each level are

- *For level 0, as* $\mathit{bits}(0) = 2$*, bits* $bv(0)$ *and* $bv(1)$*.*
- *For level 0, as* $\mathit{bits}(1) = 4$*, bits* $bv(2)$*,* $bv(3)$*,* $bv(4)$ *and* $bv(5)$*.*
- *For level 0, as* $\mathit{bits}(2) = 1$*, the bit* $bv(6)$*.*
- *For level 0, as* $\mathit{bits}(3) = 0$*, no bits are assigned. It makes sense since there are no contained states of that state.*

Not all the simple states are in the deepest level, so there might be segments whose value is ambiguous. If this state is on level i, the bits after the first $\mathit{offs}(i)$ bits are not defined, but still, the first $\mathit{offs}(i)$ bits determine the state. By convention, let the remaining ambiguous bits be 0-s. For composite states, the bits after $bv(\mathit{offs}(i))$ are also ambiguous, however unlike in the case of the simple state, their value can be anything, so let these bits be filled up with don't care bits.

This kind of encoding involves that not all bit vectors are valid. It also has the feature that a bit vector assigned to a state $s \in S$ is combinable with the bit vector assigned to every descendant state of s. Furthermore, for a state $s \in S$ at level $i = \mathit{depth}(s)$, the first $\mathit{offs}(i)$ bits are the same for s and all the descendant states of s as they are on the same state at level i.

Example 3.5. *Recall the statechart in Figure 3.2. This statechart has a composite state A on level* 1 *that is assigned the bit vector* bv_A*, amongst others, a simple children state* $A1a$ *and a composite* $A1c$*. Let the state* $A1a$ *be assigned the bit vector* bv_{A1a}*, and* $A1c$ *be assigned* bv_{A1c}*. Both vectors are* 7 *bits long, corresponding to Example 3.4*

As for all three states, regarding level 1 *the statechart is at the same states, the first two bits are the same for all three vectors. According to the convention of assigning ambiguous bits, the last* 5 *bits of* bv_A *are don't care bits, such as the last bit of* bv_{A1c}*. However the last bit of* bv_s *is* 0*.*

However, *enc* is not complete as it is not defined how to assign bit vectors to states in a level.

Definition 3.4 (Substatechart). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart and s be the root object or a composite state in it, let *sub-statechart* $Sc' = (S', R', par', I', V', Tr', \mathcal{H}')$ be a statechart such that $R' = chld(s)$, $S' = \{s_i \,|\, par(par(s_i)) = s\}$, $I' = I \cap S'$ and $par' = \{par(x) \mid x \in r' \cup s'\}$. ▪

Informally said, $Sc(s)'$ is a statechart that contains s and the root element, its regions and its child states. Note that $Sc(s)'$ is a statechart, for which encoding was defined in Section 3.1.1.

The values of $V', Tr', \mathcal{H}'$ were omitted deliberately, as variables, transitions and their labeling and history have no impact on assigning numbers to states.

Let enc_p be a function that assigns a bit vector to each state $s \in S$ such that $enc_p(s) = bv$, where bv is the value of $enc(s)$ for $Sc'(par(par(s)))$.

Informally enc_p assigns a bit vector to the state s that is assigned to s in the substatechart of the parent state of s.

Example 3.6. *Recall the statechart presented in Figure 3.2. Example 3.4 showed, that for each level, the offset assigned is* 0*,* 2*,* 6*,*7 *respectively. According to that, and the bit vector assigning conventions presented above, the bit vectors for level* 1 *regions are:*

- $enc(A) = \overline{00XXXXX}$,
- $enc(B) = \overline{01XXXXX}$,
- $enc(C) = \overline{1000000}$.

Note that C is assigned terminal zeros as it is not a composize state.

For the states in level 2*, the next* 4 *bits can be assigned. A is considered as a statechart with two parallel regions, 3-3 states in each, and B is considered as a statechart with two substates. The vectors assigned are:*

- $enc(A1a) = \overline{0000XX0}$,
- $enc(A1b) = \overline{0001XX0}$,
- $enc(A1c) = \overline{0010XXX}$,
- $enc(A2a) = \overline{00XX000}$,
- $enc(A2b) = \overline{00XX010}$,
- $enc(A2c) = \overline{00XX100}$,
- $enc(B1) = \overline{1000000}$,
- $enc(B2) = \overline{1000010}$.

Finally, the substates of $A1c$ can be assigned a bit vector.

- $enc(A1c1) = \overline{0010XX0}$.
- $enc(A1c2) = \overline{0010XX1}$.

Theorem 3.2. Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart, and $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$ of Sc a configuration of it. The set of bit vectors $\{enc(s) \mid s \in \omega\}$ is not conflicting.

Proof. To prove the non conflicting behavior of the bit vectors assigned, lets assume that there is a conflict amongst the vectors. By the *offs* function, the conflicting level i can be determined. The conflict can not be due to parallel regions, as segments in the bit vectors assigned to each region are distinct, so it can be assumed that there are two active states $s_1, s_2 \in \omega$ such that their ancestors at level i are different, but they are in the same region. However, this contradicts the first point of the definition of ω. ∎

Theorem 3.3. Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart, and $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$ of Sc a configuration of it. Let bv be the combination of a set of combinable bit vectors $\{enc(s) \mid s \in \omega\}$. The vector bv is complete.

Proof. To prove the completeness of bv, examine the possible occurrences of X bits. A don't care bit can come from the bit vector assigned to an abstract state, or to a parallel region. However by definition, for every composite state $s_c \in \omega$, there is an active state in ω from each subregion of s_c, so there is a simple state, whose assigned bit vector can not contain hierarchy related don't care bits. So if there is a don't care bit in bv it is due to parallel regions. But also by definition, if there is a composite state $s_c \in \omega$ with more than one regions, each region must have an active state, and as it was pointed out by Theorem 3.1, when every region has an active state, the combination of the bit vectors assigned is complete. ∎

3.2 Transforming the Transition Relation to Logical Formulas

The previous section released constraints about the statechart that was the object of reasoning. According to the hierarchy, every constraint has been released. However Section 2.3.2 introduced constraints about history, variables and events in statecharts. From now on, I assume that for every statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, $|\mathcal{H}| = 0$. Assume also that for every configuration $c = (\omega, \rho, \mathcal{F}, H)$ in Sc, there is at most one active event, so $|\rho| \leq 1$, and let this event be denoted by e.

This section extends the techniques presented in Section 2.3 for encoding statecharts meeting the requirements above.

3.2.1 Encoding States and Events

Given a bit vector $bv \in \{0, 1, X\}^n$ and a variable set $\{v_0, v_1, \ldots v_{n-1}\}$, let $lit(bv(i)) = \{v_i$ if $bv(i) = 1$, $lit(bv(i)) = \neg v_i$, if $bv(i) = 0$, and $lit(bv(i)) = \top$ otherwise $\}$ assign a literal for each element of the bit vector.

Informally, this function extends the *lit* for bit vectors over $\{0, 1\}^n$, with handling the don't care value as it is always true.

The function *form* can be defined similarly to bit vectors over $\{0, 1\}^n$.

Let $form : BV_n \mapsto FOL$ be a function that assigns a formula to a bit vector in a way that

$$form(bv) = \bigwedge_{i=0}^{n-1} lit(bv(i)). \tag{3.5}$$

Example 3.7. *Consider the bit vector $bv = \overline{01X1}$. Given the variable set $\{v_0, v_1, v_2, v_3\}$, the value of $form(bv)$ is $\neg v_0 \wedge v_1 \wedge \top \wedge v_3$, which can be abbreviated as $\neg v_0 \wedge v_1 \wedge v_3$ since the two formulas are equivalent.*

The value of $form(bv)_k$ for every $k \in \mathbb{N}$ is $\neg v_{0,k} \wedge v_{1,k} \wedge v_{3,k}$.

Similar to the encoding of simple statecharts, let $\psi_s : S \mapsto FOL$ be a function that assigns a formula to a state $s \in S$ such that

$$\psi_s(s) = form(enc(s)). \tag{3.6}$$

Define the function ψ_ω for a set of states ω as

$$\psi_\omega(\omega) = \bigwedge_{s \in \omega} form(enc(s)). \tag{3.7}$$

This function can be used to assign bit vectors to a set of active states. Theorem 3.4 states that the assigned bit vector will be the same as if each state was assigned a vector, and these vectors were combined.

Theorem 3.4. Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart, and let $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$ be a configuration in it. Let bv be the combination of the bit vectors in the set $BV_\omega = \{enc(s) \mid s \in \omega\}$. For every $c = (\omega, \rho, \mathcal{F}, H)$, $form(bv) = \bigwedge_{s \in \omega} \psi_s(s)$. ▪

Proof. First of all, according to Equation 3.6,

$$\bigwedge_{s \in \omega} \psi_s(s) = \bigwedge_{bv' \in BV_\omega} form(bv'). \tag{3.8}$$

By definition,

$$form(bv) = \bigwedge_{i=0}^{n-1} lit(bv(i)). \tag{3.9}$$

Also by definition,

$$\begin{aligned} \bigwedge_{bv' \in BV_\omega} form(bv') &= \bigwedge_{bv' \in BV_\omega} \left(\bigwedge_{i=0}^{n-1} lit(bv'(i)) \right) = \\ &= \bigwedge_{i=0}^{n-1} \left(\bigwedge_{bv' \in BV_\omega} lit(bv'(i)) \right). \end{aligned} \tag{3.10}$$

From the formula $\bigwedge_{bv' \in BV_\omega} lit(bv'(i))$, the members where $lit(bv'(i)) = \top$ can be excluded as for every boolean formula, $\psi \leftrightarrow (\psi \wedge \top)$. Note that if for every bv' the formula $lit(bv'(i)) = \top$, than in every bit vector, there is a don't care bit at position i, which would contradict the completeness, defined by Theorem 3.3. However if there were bit vectors $bv', bv'' \in BV_\omega$ such that $lit(bv'(i)) = \neg lit(bv''(i))$, that would mean that the i-th bit of bit vectors bv', bv'' are different, which contradicts the combinability of the assigned bit vectors stated by Theorem 3.2.

So for all bit vector $bv' \in BV_\omega$, the value of $lit(bv'(i))$ is either $\top$ or the same literal that is assigned to the combination of them. And for every formula $\psi \leftrightarrow \psi \wedge \psi$, so for every

$0 \le i < n$,

$$lit(bv(i)) = \bigwedge_{bv' \in BV_\omega} lit(bv'(i)) \tag{3.11}$$

And from that,

$$\bigwedge_{i=0}^{n-1} lit(bv(i)) = \bigwedge_{i=0}^{n-1} \left(\bigwedge_{bv' \in BV_\omega} lit(bv'(i)) \right) \tag{3.12}$$

is trivial. ▪

Regarding the preceding theorem, if ω is a complete set of active states in a statechart configuration,

$$\psi_\omega(\omega) = \bigwedge_{s \in \omega} \psi_s(s). \tag{3.13}$$

The function $enc : EV \mapsto BV_n$ is defined the same way as for simple statecharts. Note that this implies that don't care bits are not allowed in bit vectors assigned to events.

$$\psi_{EV}(e) = form(enc(e)). \tag{3.14}$$

From further on, the k-indexed form of the assigned FOL formulas will be used. The main reason behind this is to be able to reason about the sequence of state sets or events, and the index expresses the consecutiveness of them. Due to similar consideration, define $\psi_V : V \times \mathbb{N} \mapsto FOL_{const}$, where $FOL_{const} \subset FOL$ is the set of the first order logic constants.

Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart where $|\mathcal{H}| = 0$, and let $c = (\omega, \rho, \mathcal{F}, H)$ be a configuration of it where $\rho = \{e\}$. Let $\psi_c : C_{Sc} \times \mathbb{N} \mapsto FOL$ be a function that assigns a unique first order formula to every configuration in $c = (\omega, \rho, \mathcal{F}, H) \in C_{Sc}$ such that

$$\psi_c(c)_k = \left(\bigwedge_{s \in \omega} \psi_s(s)_k \right) \wedge \psi_{EV}(e)_k \wedge \left(\bigwedge_{v \in V} (\psi_V(v)_k = \mathcal{F}[v]) \right). \tag{3.15}$$

For a path $\pi = (c_0, c_1, \ldots, c_n)$, let $\mathcal{I}_\pi$ be an interpretation such that $\mathcal{I} \models \psi_c(c_i)_i$ for every $0 \le i \le n$, but for every other configuration $c \in C_{Sc}$, $\mathcal{I}_\pi \not\models \psi_c(c)_j$ if $c \ne c_j$.

3.2.2 Transforming the Transition Relation

This section presents a possible method to transform the transition relation of the statechart into logical formulas.

3.2.2.1 Source and Target States

With parallel regions and hierarchically nested states, transitions connect two, not necessarily real states of the system. This could mean that the source and the target of the transition can correspond to more than one state. For example, a transition originating from a composite state corresponds to every descendant state of the source state, and a source state in a region with parallel regions refers that for every state in the Cartesian product of the states elements the transition is allowed. Consider the method presented in the previous section for the encoding of states to bit vectors. Don't care bits exactly denoted this.

However, after the transition fires, the execution of a statechart must arrive to an explicitly given state configuration. If the explicit target state is not the only one, that becomes active after a transition, the target state configuration is not trivial. The expected behavior has to be defined for each cases of nontrivial target states.

- In the case, when the target state is a composite state, the execution continues from the initial state of the region inside the state, or from the set of initial states if the target state has more regions.
- If the containing region of the target state has orthogonal pairs, there are two cases.
  - If the source of the transition is from inside the region, the transition should not have any effect outside the region, the other parallel regions continue their execution from the state they were before.
  - If the source state is outside of the region, the other regions should start their execution from their initial state.

Compositeness and parallelity are not opposites to each other, both criterion can stand for a state. In that case, of course both rules has to be applied, as they are not contradicting each other.

Taking these into consideration, different formulas should be assigned to a state if it is regarded as a source state of a transition, than when it is regarded as the target state.

Example 3.8. *Recall the statechart presented in Figure 3.2. In this statechart, the state A1c is a composite state, so a direct transition to A1c implies a transition to its initial state A1c1.*

This state is contained in a region that has orthogonal pairs. So a transition from inside region regA1 with A1c as target state implies, that the active state in region regA2 remains unchanged. However, if the transition source is outside of the region, the statechart's execution not only enters regA1, but also regA2 in its initial state A2a.

Define a function *init* that can be used to determine the target state configuration for a transition, that has a given state as target.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, let *init* be a function that assigns a set of states ω_r to a region r such that $\omega_r = \{s | s \in I \text{ and } s \in desc(r)\}$. Informally said, ω_r is the set of initial states in r. Note that these states will be active, if the execution enters the region.

For a state $s \in S$ let $init(s)$ be $\bigcup\{init(r) \mid r \in chld(s)\}$, so the set of all the initial states that are descendant of s.

Define the function *target* that assigns a set of states to every state $s \in S$ that will be active if a transition with s as target state fires.

The value of *target* depends on the compositeness and parallelity of the state.

For a simple state s_s that is not contained by any parallel region, let $target(s_s) = \{s' \mid s' = s_s \vee s' \in anc(s_s)\}$, so the state and all of its ancestors, as these will be the only active states.

For a composite state s_c that is not contained by any parallel region, let $target(s_c) = \{s' \mid s' = s_c \vee s' \in anc(s_c)\} \cup init(s_c)$, so the state, its ancestors and the nested initial states of it. These two rules can be united as $|init(s_s)| = 0$ for a simple state s_s.

In case of a state s contained in any region that is orthogonal to another region, let the value of the function *target* be the union of the value of $\mathit{target}(s)$ if the state was not contained in a parallel region and the initial states of the regions, to which a parallel region the state is in, but the source state is not. So from each region the set of initial states, but from the region, from where the source state is added to the union.

Before formalizing this, consider that this is required because if a statecharts execution arrives to a region, it arrives to every region next to them, even though it is only derived implicitly from the semantics of statecharts, but these regions should continue from their initial state. However, if during the execution the statechart just steps in a region, the parallel regions in the same state do not change their active states, the active states in those regions remains what is was before the transition.

For a state s_p in a parallel region, let $\mathit{target}(s_p)$ be

$$\{s' \,|\, s' = s_p \vee s' \in \mathit{anc}(s_p)\} \cup \mathit{init}(s_p) \cup \{\mathit{init}(r) \,|\, r \in \mathit{chld}(s') \wedge s' \in \mathit{anc}(s_p) \wedge \neg(r \in \mathit{anc}(s_p))\} \tag{3.16}$$

Informally, the value of $\mathit{target}(s_p)$ is the set of s_p and initial states for regions that are contained in an ancestor of s_p, however they do not contain s_p.

Example 3.9. *Consider the statechart Sc presented in Figure 3.2. A1b is a simple state in a region with parallel pair in Sc. The initial state of the other parallel region is A2a, so the value of* $\mathit{target}(A1b)$ *is* $\{A1b, A2a\}$*.*

In Sc, the state A is a composite state. It has two regions, regA1 and regA2, each having a simple initial state. So the value of $\mathit{target}(A) = \{A1a, A2a\}$*.*

3.2.2.2 Guards

With the previously defined functions the source and target states of the transitions can be encoded into formulas. However in order for the transformation to be complete, the variables in the statechart have to be handled. Guards in statecharts are first order logic formulas, so in order to reason about them their k-indexed version can be used.

Example 3.10. *Consider the transition t with* $\mathit{grd}(t) = (x = 2) \vee (y + 1 < 4)$*. The variables in the guard are x and y, so* $\mathit{grd}(t, k) = (\psi_V(x, k) = 2) \vee (\psi_V(y, k) + 1 < 4)$*.*

3.2.2.3 Actions

To enable reasoning about their effect on variables, actions also have to be encoded into logical formulas. Regarding the encoding, actions can be interpreted as formulas that evaluate to true, if the action is executed. Restrict the set of allowed statements in actions to raising events and assigning variable values. Let $\mathit{act} : \mathit{Act} \mapsto \mathit{FOL}$, such that for every $a \in \mathit{Act}$, the value of $\mathit{act}(a)_k$ is $\bigwedge\limits_{\mathit{stm} \in a} \psi_{STM}(\mathit{stm})_k$, where stm is a statement in a, and ψ_{STM} assigns a formula for every statement. For a statement stm in a,

- if stm raises event e, let $\psi_{STM}(\mathit{stm})_k = \psi_{EV}(e)_{k+1}$,
- if stm assigns the value of a formula ψ to a variable v, let $\psi_{STM}(\mathit{stm})_k$ be $\psi_V(v)_{k+1} = \psi_k$.

Informally, the raising of an event causes the event at the next execution step to be active, and and assigning the variable assigns its value at the next execution step, based on its current values.

3.2.2.4 Transition relation

From now on, the functions defined in Section 2.3.2 are defined again for hierarchical statecharts.

Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. For Sc, let $\psi_t : Tr \mapsto FOL$ be a function that assigns a first order logic formula to every transition of Sc. The value of ψ_t for a $t \in Tr$ is presented in Equation 3.17.

$$\psi_t(t)_k = \psi_\omega(target(src(t)))_k \wedge \psi_s(trgt(t))_{k+1} \wedge \psi_{EV}(trig(t))_k \wedge grd(t)_k \wedge act(t)_k \quad (3.17)$$

The transition relation formula of the statechart is defined the same as for simple statecharts. The value of the formula ψ_{Tr} is

$$\psi_{Tr} = \bigvee_{t \in Tr} \psi_t(t) \quad (3.18)$$

The definition of ψ_{Sc} is the same as it was in case of a flat statechart.

$$\psi_{Sck} = \psi_c(c_I)_0 \wedge \left(\bigwedge_{i=0}^{k} (\psi_{Tr})_i \right) \quad (3.19)$$

The properties of $\mathcal{I}_\pi$ still hold, with the extensions made here.

3.3 State Space Exploration

The previous section presented encoding of statecharts to FOL formulas. This section presents the application of the encoding in the verification of them based on the basic concept of verifying statecharts, which was presented in Section 2.4.2.

The input of the algorithm is a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ with an initial configuration c_I, and a set of error states that should not be reached. The iterative algorithm of exploring the state space is summarized in Algorithm 1.

The core of the algorithm is a breadth-first search in the state space. In each step, the set of configurations Q' is explored, in which each configuration is reachable from the previously discovered configurations Q witihin one transition. The configurations in Q processed in a loop, one at each iteration, and the configuration c is only processed, if it is not already in the set of reached configuration R, and is added to R while being processed. Hence each configuration is only processed once.

A configuration c_1 is reachable from c_0 within one transition, if for the path $\pi = (c_0, c_1)$, there is an interpretation $\mathcal{I}_\pi$ satisfying the constraints $\psi_c(c) \wedge (\psi_{Tr})_0$. The interpretations can be explored with logical solvers, in case of only boolean variables, a SAT solver will do, however to handle more complex formulas, an SMT solver is required. Since a logical solver can find all the satisfying interpretations for a formula, no reachable configuration is omitted.

Algorithm 1: The many-at-once method

```
Input  : $Sc = (S, R, par, I, V, Tr, \mathcal{H})$: the verified statechart,
         $C_f$: the set of error states
Output : Path $\pi$ as a counterexample or success
$R \leftarrow \emptyset$;
$Q \leftarrow \{c_I\}$;
while $Q \neq \emptyset$ do
    $Q' \leftarrow \emptyset$ ;
    foreach $c \in Q$ do
        if $c \notin R$ then
            $R \leftarrow R \cup \{c\}$ ;
            $\psi \leftarrow \psi_c(c)_0 \wedge (\psi_{Tr})_0$;
            foreach $\pi = (c, c')$ where $\mathcal{I}_\pi \models \psi$ do
                if $c' \in C_f$ then return path to $c'$ $(c_I, \ldots, c, c')$ ;
                $Q' \leftarrow Q' \cup \{c'\}$ ;
            end
        end
    end
    $Q \leftarrow Q'$ ;
end
return success;
```

Given a configuration $c' = \pi[1]$, it is checked if it is an error configuration ($\pi[1] \in C_f$). If it is so, a path to it is returned, for example based on the information for each configuration c, from which state configuration it was reached. If not, it is added to the set of freshly discovered configurations (Q').

At the end of the iteration over the last discovered configuration, Q is assigned the set of the newly discovered configurations Q', and the loop starts again. It terminates when Q' is empty, which is equivalent to the fact that no new configurations were discovered in the previous iteration, which can only be, because all the reachable states have been explored.

Example 3.11. *Consider the statechart presented in Figure 3.2. Let the initial value for variables x and y be 0.*

For simplicity, as there are no events or history in the statechart, let the configurations be denoted by the set of the active states and variable values.

Let the set of error states C_f contain only one configuration, $\{A1a, A2a, x = 0, y = 2\}$.

The execution of the algorithm start from configuration $\{A1a, A2a, x = 0, y = 0\}$, as it is the initial configuration of the statechart. So $Q = \{\{A1a, A2a, x = 0, y = 0\}\}$.

The configurations reached in the first step are:

- $\{A1b, A2a, x = 1, y = 0\}$,
- $\{A1c1, A2a, x = 0, y = 1\}$,
- $\{A1a, A2b, x = 0, y = 1\}$,
- $\{B1, x = 1, y = 0\}$.

Each configuration is put into Q' and the exectuion continues.

The next step, from each configuration, the reachable configurations are listed.

- *From* $\{A1b, A2a, x = 1, y = 0\}$*, the reachables are:*
  - $\{A1a, A2a, x = 1, y = 0\}$,
  - $\{A1b, A2b, x = 1, y = 1\}$,
  - $\{B1, x = 2, y = 0\}$.
- *From* $\{A1c1, A2a, x = 0, y = 0\}$*, the reachables are:*
  - $\{A1c2, A2a, x = 0, y = 0\}$,
  - $\{A1c1, A2b, x = 0, y = 1\}$,
  - $\{B1, x = 1, y = 0\}$.
- *From* $\{A1a, A2b, x = 0, y = 1\}$*, the reachables are:*
  - $\{A1b, A2b, x = 1, y = 1\}$*, however that was dicovered before, so it will not be put in* Q'*,*
  - $\{A1c1, A2b, x = 0, y = 1\}$,
  - $\{A1a, A2a, x = 0, y = 2\}$.

An error configuration is found, the execution terminates, however there would be more states that are reachable from $\{A1a, A2b, x = 0, y = 1\}$*. The path* $(\{A1a, A2a, x = 0, y = 0\}, \{A1a, A2b, x = 0, y = 1\}, \{A1a, A2a, x = 0, y = 2\})$ *is returned as counterexample.*

The reachable configurations are also presented in a reachability graph in Figure 3.3. The light red configuration marks the error configuration. Note, that the reachability graph is a tree, when the dashed edges marking the already found configurations are excluded.

The algorithm however has the disadvantage, that it polls the solver all the reachable configurations, including the already reached ones. Furthermore, it is possible that from one configuration, an infinite amount of configurations are reachable, and the solver algorithm never terminates, even though all of them are error states.

This can be optimized with only getting one satisfying interpretation from the solver, and then add extra formulas to the solver, expressing that already found states are not valid solutions, such as for every configuration c in the set of already discovered configurations R, the formula $\neg\psi_c(c)_1$ is conjuncted to the unfolded transition relation.

Algorithm 2 demonstrates the pseudo code of the optimized algorithm.

Let the method presented by Algorithm 1 be called as the *many-at-once exploring* whereas the method of Algorithm 2 as the *one-at-once* method.

3.4 Bounded Reachability Checking

Section 2.4.3 presented the basics of bounded model checking. The concept can be applied in the verification of statecharts, using the formulas defined in Section 3.2. Recall that bounded model checking iteratively checks if a requirement holds for every path π of length k. In each step, k is incremented until a counterexample is found or a limit of checking is reached.

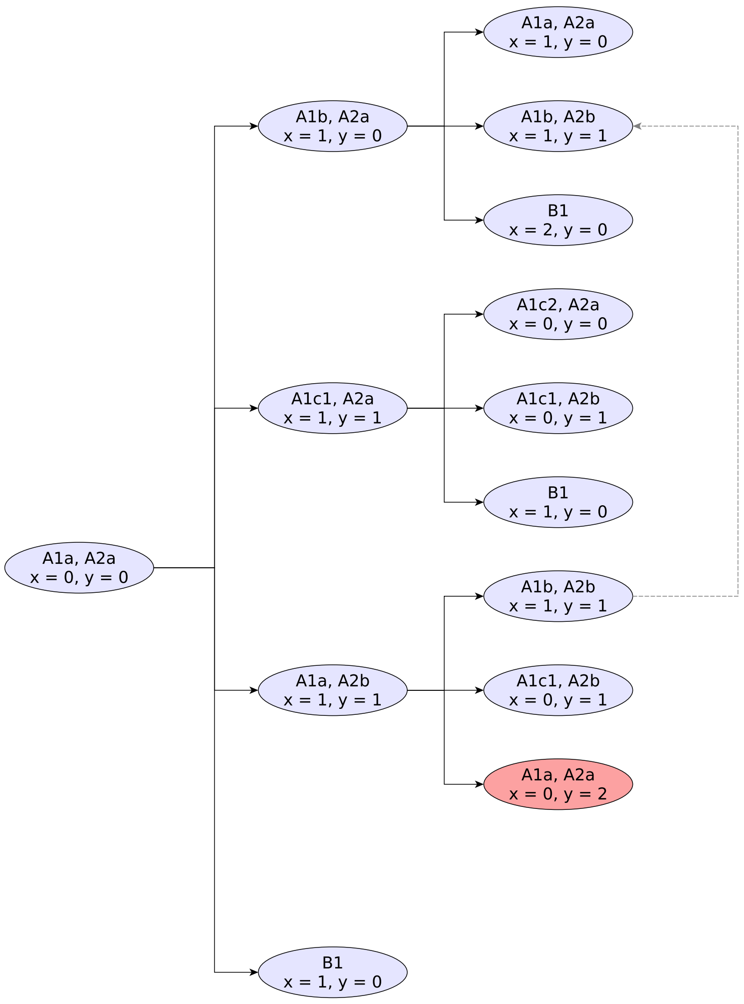


Figure 3.3: Reachability graph for Example 3.11.

The k-reachability of a state configuration can be checked with logical solvers. The formula ψ_{Sc_k} evaluates true with interpretations that represents a valid path, and the formula $\psi_c(c)_k$ evaluates to true if the path has c as its k-th configuration. So the formula $\psi_{Sc_k} \wedge \psi_c(c)_k$ evaluates to true, if there is a path π of length k with the configuration c as last element. If c is an error state, π is a k long counterexample.

The iterative process of the bounded model checking is presented in Algorithm 3.

Note, that the algorithm has a disadvantage that if there is a counterexample, which is longer than MAX, the algorithm still returns not reachable. Apart from special cases (e.g. acyclic statecharts) it can not be known, if there would be a counterexample, if the limit was higher. Formally speaking, the bounded model reachability can not prove safety, only k-safety and reachability.

Algorithm 2: The one-at-once method

```
Input  : $Sc = (S, R, par, I, V, Tr, \mathcal{H})$: the verified statechart,
         $C_f$: the set of error states
Output : Path $\pi$ as a counterexample or success
1  $R \leftarrow \{c_I\}$ ;
2  $Q \leftarrow \{c_I\}$;
3  while $Q \neq \emptyset$ do
4      $Q' \leftarrow \emptyset$ ;
5      foreach $c \in Q$ do
6          $\psi \leftarrow \psi_c(c)_0 \wedge (\psi_{Tr})_0 \wedge \left(\bigwedge_{c' \in R} \neg\psi_c(c')_1\right)$;
7          if $\exists\, \pi(c, c')$ such that $\mathcal{I}_\pi \models \psi$ then
8              if $c' \in C_f$ then return path to $c'$ $(c_I, \ldots, c, c')$ ;
9              $Q' \leftarrow Q' \cup \{c'\}$ ;
10             $R \leftarrow R \cup \{c'\}$
11         end
12     end
13     $Q \leftarrow Q'$ ;
14 end
15 return success;
```

Algorithm 3: Bounded reachability checking.

```
Input  : $Sc = (S, R, par, I, V, Tr, \mathcal{H})$: the verified statechart,
         $c_f$: an error configuration,
         $MAX$ the limit of the iterations
Output : Path $\pi$ as a counterexample or not reachable
1  $k \leftarrow 0$;
2  while $k < MAX$ do
3      $\psi = \psi_{Sck} \wedge \psi_c(c_f)_k$ ;
4      if $\psi$ is SAT then
5          return path to $c_f$
6      end
7      $k \leftarrow k + 1$
8  end
9  return not reachable;
```

Chapter 4

Applying CEGAR to Hierarchical Statecharts

The techniques presented in Chapter 3 provide sufficient functionality to check statecharts against reachability requirements. However, for statecharts with huge or even infinite state space, the efficiency (or even termination) of those algorithms is not guaranteed. Abstracting the statechart and checking the abstract model against the requirements offers a method to overcome this problem. In this chapter I propose an adaption of the Counterexample-Guided Abstraction Refinement method (Section 2.4.4) for statecharts.

In Section 4.1 I introduce the concept of abstraction for statecharts. Section 4.2 presents the CEGAR algorithm for statecharts, and the following sections presents one step of the CEGAR algorithm each. Section 4.3 presents the construction of an initial abstraction, Section 4.4 introduces model checking techniques for abstracted statecharts, Section 4.5 demonstrates the concretization of an abstract counterexample and Section 4.6 presents refinement algorithms for abstractions.

4.1 Abstraction of Statecharts

CEGAR operates on abstract system, thus in order to apply it on a statechart, a notion of abstraction has to defined for them. As CEGAR requires existential abstraction, the abstract stataechart has to be an over-approximation of the concrete statechart. This requires that if a statechart has a transition between two states that is allowed to fire under some circumstances, the abstracted statechart must also have a transition between the two corresponding states. However, the abstract statechart might have other transitions that have no corresponding pairs in the original statechart.

The top-down design of systems involves a generalization, first defining the behavior of the top level components, and later expanding the inner implementation of components. In case of statecharts, this top-down design results in hierarchy, providing an intuitive way of abstraction. During my work, I focused on creating and applying hierarchy-based abstractions for the verification of statecharts.

Definition 4.1 (State Abstraction). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. Let $\mathbf{h}_S : S \mapsto S$ be the *state-abstraction function* of Sc that assigns to each state $s \in S$ its abstracted pair, such that

- for each state $\mathbf{h}_S(s) \in anc(s)$ or $\mathbf{h}_S(s) = s$,

- for each state $s \in S$ such that $\mathbf{h}(s) \neq s$, for every $s' \in desc(s)$, $\mathbf{h}(s) = \mathbf{h}(s')$. ▪

Informally, state abstraction function maps each state to its corresponding abstraction, which can either be the state itself or one of its ancestors. Furthermore, if a state s is mapped to one of its ancestor states s', all the descendant states of s is also mapped to s'.

If $\mathbf{h}_S(s) = s$ for a state $s \in S$, the state is considered *refined*, otherwise it is regarded as *abstracted*.

Abstraction can be defined for variables too [8]. In this case abstraction means that only a subset of the variables is considered during verification. Refinement means to extend this set with additional variables.

Definition 4.2 (Variable Abstraction). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart. Separate the variable set V into two distinct sets, the set of *visible* and *invisible* variables. For a set of variables V let the value of the function $\mathbf{h}_V(V)$ be the set of the visible variables. The function $\mathbf{h}_V : V \mapsto \{\top, \bot\}$ is the *variable abstraction function* that assigns each variable of V if it is *refined* in the abstraction. ▪

Refined variables can be referred as *visible*.

The two previously defined functions can also be combined to one function, loosely speaking.

Definition 4.3 (Statechart Abstraction). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart, let $\mathbf{h}_S$ be a state abstraction function, and $\mathbf{h}_V$ be a variable abstraction function for it. The functions $\mathbf{h}_S$ and $\mathbf{h}_V$ together are referred to as the *abstraction function*, denoted by $\mathbf{h}$, or $\mathbf{h} = \{\mathbf{h}_S, \mathbf{h}_V\}$. For a state $s \in S$ let $\mathbf{h}(s)$ denote $\mathbf{h}_S(s)$ and for a variable $v \in V$ let $\mathbf{h}(v)$ denote $\mathbf{h}_V(v)$. ▪

An abstraction for a statechart intuitively implies the definition of the abstract equivalent of Sc.

Definition 4.4 (Abstract Statechart). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart, and let $\mathbf{h}$ be an abstraction function for Sc. The tuple $Sc' = (S', R', par', I', V', Tr', \mathcal{H}')$ is the *abstract statechart* of Sc corresponding to $\mathbf{h}$ if the following requirements hold.

- $S' = \{\mathbf{h}(s) \mid s \in S\}$, where S' is a mathematical set, not containing any instance more than once.
- $R' = \{r \mid \exists s \in chld(r) \text{ such that } \mathbf{h}(s) = s\}$, that is, the regions kept are those, that have at least one child state that is mapped to itself.
- $par' = \{(r, s) \mid (r, s) \in par,\ r \in R',\ s \in S' \cup \{root\}\} \cup \{(s, r) \mid (s, r) \in par,\ s \in S',\ r \in R'\}$. Informally, the hierarchy is persevered between a state and a region if both the state and the region is included in the abstract statechart.
- $I' = S' \cap I$.
- $V' = \{v \mid \mathbf{h}(v) = \top\}$.
- $Tr' = \{(\mathbf{h}(src(t)), \mathbf{h}(trgt(t)), trig(t), grd(t), Act(t)) \mid t \in Tr\}$.
- $|\mathcal{H}'| = 0$, as $|\mathcal{H}| = 0$. ▪

Let Sc' be denoted by $\mathbf{h}(Sc)$.

Example 4.1. *Recall the statechart Sc presented in Example 3.3, that is presented again in Figure 4.1*

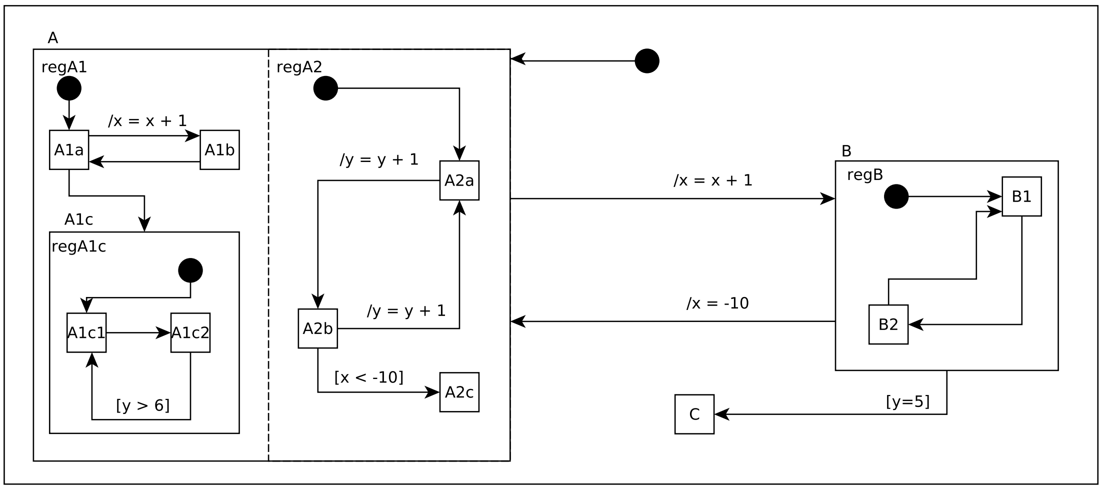


Figure 4.1: An example statechart.

Let $\mathbf{h}_S$ *be a function such that* $\mathbf{h}_S = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B1, B2 \mapsto B2, C \mapsto C\}$*, and let* $\mathbf{h}_V$ *be a function such that* $\mathbf{h}_V(\{x, y\}) = \{x\}$*.*

The abstraction $\mathbf{h} = \{\mathbf{h}_S, \mathbf{h}_V\}$ *is a valid abstractiom for Sc as all the requirements defined above are satisfied. The abstract statechart for* $\mathbf{h}$ *is presented in Figure 4.2*

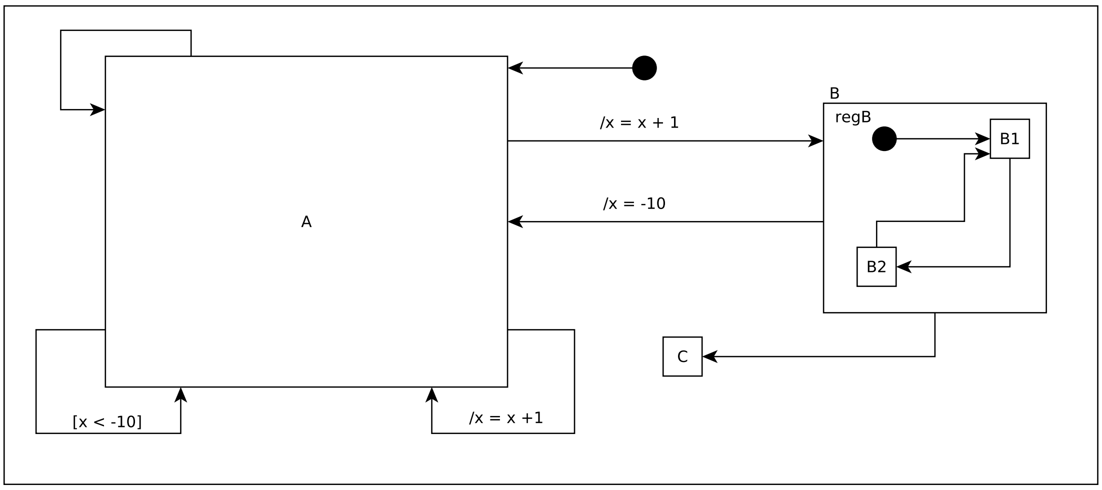


Figure 4.2: Example abstraction for the statechart presented in Figure 3.2

Note, that $s \in S' \rightarrow s \in S$ and $v \in V' \rightarrow v \in V$, that is, the states of the abstract statechart are states of the original statechart, and variables of the abstract statechart are variables of the original statechart.

For the set of states ω, abstraction can be defined as $\mathbf{h}(\omega) = \{\mathbf{h}(s) \mid s \in \omega\}$. Informally, the abstraction of a set of states is the set of the abstract states.

For a configuration $c = (\omega, \rho, \mathcal{F}, H)$ of the statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, the value of $\mathbf{h}$ can be defined as $\mathbf{h}(c) = (\mathbf{h}(\omega), \rho, \{\mathcal{F}(v) \mid v \in V \text{ and } \mathbf{h}(v) = \top\}, \{\})$, so the set of active states is abstracted, the active events are kept, the history is not allowed (note that $|H| = 0$), and in $\mathcal{F}$ only visible variables are kept. Note that $\mathbf{h}(c)$ is a configuration for the abstract statechart $\mathbf{h}(Sc)$ if c is a configuration for Sc.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, the state abstraction $\mathbf{h}'_S$ is *finer* than the abstraction $\mathbf{h}_S$, if for every state $s \in S$, $\mathbf{h}_S(s) = \mathbf{h}'_S(s)$ or $\mathbf{h}_S(s) \in anc(\mathbf{h}'_S(s))$. Informally, if for every state, $\mathbf{h}'_S$ assigns $\mathbf{h}_S(s)$ or one of its descendant.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, the variable abstraction $\mathbf{h}'_V$ is *finer* than the abstraction $\mathbf{h}_V$, if for every variable $v \in V$ we have $\mathbf{h}_V(v) \to \mathbf{h}'_V(v)$, informally, if v is visible in $\mathbf{h}_V$, it is also visible in $\mathbf{h}'_V$.

For a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$, the abstraction $\mathbf{h}' = \{\mathbf{h}'_S, \mathbf{h}'_V\}$ is finer than $\mathbf{h} = \{\mathbf{h}_S, \mathbf{h}_V\}$, if $\mathbf{h}'_S$ is finer than $\mathbf{h}_S$ and $\mathbf{h}'_V$ is finer than $\mathbf{h}_V$, and $\mathbf{h}' \neq \mathbf{h}$. If $\mathbf{h}'$ is finer than $\mathbf{h}$, then $\mathbf{h}$ is *coarser* than $\mathbf{h}'$.

Example 4.2. *Consider the statechart and the abstraction, presented in Example 4.1 and the abstraction* $\mathbf{h}$ *presented there. The abstraction* $\mathbf{h}' = \{\{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B, B2 \mapsto B, C \mapsto C\}, \{x\}\}$*, for which the abstract statechart is presented in Figure 4.3 is finer than* $\mathbf{h}$.

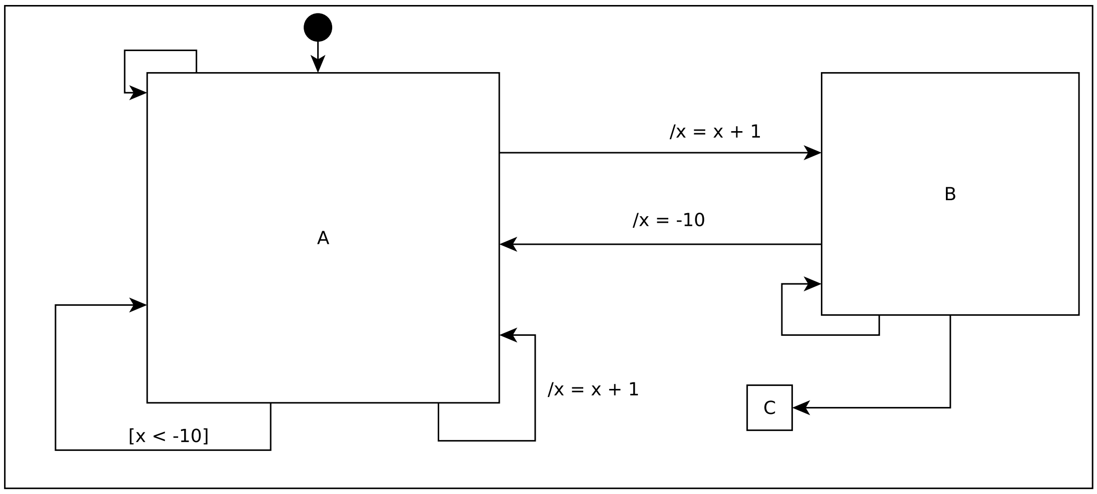


Figure 4.3: A coarser abstraction for the statechart presented in Figure 4.1.

For an abstraction function $\mathbf{h}$, an inverse can be defined, however like in the case of the hierarchy function of statecharts, this inverse is not a real mathematical inverse, as an abstract object (e.g. variable, state, configuration) can be an abstraction of more than one object of the original statechart.

Definition 4.5 (Inverse abstraction). Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart and $\mathbf{h}$ be an abstraction function for it.The *inverse abstraction function* $\mathbf{h}^{-1}$ is as follows:

- $\mathbf{h}^{-1}(s) = \{s' \mid \mathbf{h}(s') = s\}$

- $\mathbf{h}^{-1}(c) = \{c' \mid \mathbf{h}(c') = c\}$
- $\mathbf{h}^{-1}(Sc) = \{Sc' \mid \mathbf{h}(Sc') = Sc\}$ ▪

Note that there is no point defining an inverse function for the abstraction of variables, as they are mapped to truth symbols.

Applying the inverse abstraction function on an abstraction is also referred to as a concretization.

Example 4.3. *Recall the abstraction* $\mathbf{h}$ *presented in Example 4.1. The inverse of* $\mathbf{h}$ *is* $\mathbf{h}^{-1} = \{A \mapsto \{A, A1a, A1b, A1c, A1c1, A1c2, A2a, A2b, A2c\}, B \mapsto \{B\}, B1 \mapsto \{B1\}, B2 \mapsto \{B2\}, C \mapsto \{C\}\}$.

Definition 4.6 (Identity Abstraction). Let $M = (S, \Sigma, Tr, s_0)$ be a statechart. The abstraction $\mathbf{h}$ is the *identity abstraction* of the statechart if $\mathbf{h}(Sc) = Sc$. ▪

Note that there is no finer abstraction that the identity abstraction.

During my work, I constructed and examined two kind of abstractions, one that abstracts both states and variables, and one that only abstracts states. From further on, let them be referenced as *states-only abstraction* and *generic abstraction*.

4.2 CEGAR Loop for Statecharts

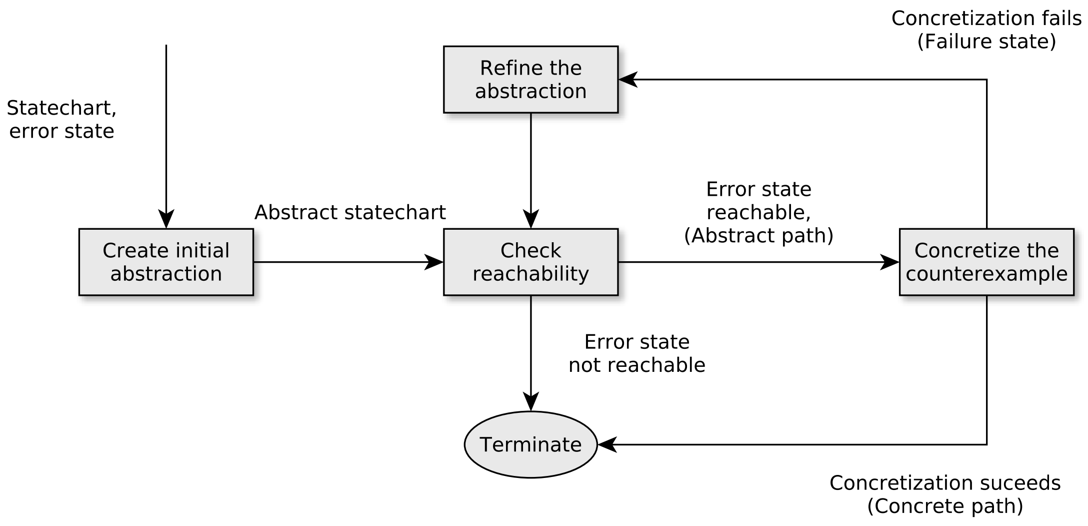
As presented in Section 2.4.4, the CEGAR loop has four major steps: creating the initial abstraction, checking the given requirement against the abstract model, and if the requirement was violated, trying to concretize the counterexample returned by the checker, and finally refining the abstraction if needed.

The CEGAR loop can be implemented for the verification of statecharts using the abstraction defined above. The input of the algorithm is a statechart $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ and a set of error states C_f.

The set of error states C_f can be given by explicitly enumerating its member configuration, however a set of configurations can be expressed by bounding the value of some active states and variables. For example in case of three varaibles a, b, c, the C_f set can be implicitly declared as the set of configurations where $a = 3$. With this declaration, for an error state, the value of variables b and c are *unbound*. Let it be denoted by $unb(C_f)$, and let the set of *bound* variables be denoted by $bound(C_f) = V \setminus unb(C_f)$.

The algorithm first creates an initial abstraction for Sc, and then the execution enters the CEGAR loop. In each step, the abstract statechart $\mathbf{h}(Sc)$ is tested against the given requirements. The checking techniques presented in Sections 3.3 and 3.4 can be used, as $\mathbf{h}(Sc)$ is a valid statechart. Then if no counterexample is found, the execution terminates, however in case of a counterexample, the algorithm tries to concertize the counterexample, which amounts to searching a path in Sc fitting the counterexample. Note, that although this check is performed on the original statechart, only a subset of its state space has to be considered because the counterexample bounds the search. If the concretization succeeds, a concrete path to a failure state is returned, and the execution terminates. However if the counterexample turns out to be spurious, the algorithm creates a finer abstraction for Sc, based on the configuration in which the concretization failed. If there is no finer abstraction, and the identity abstraction is reached, then the previous checking was done in the concrete statechart, so it can be said that there is no reachable error state. Otherwise, the loop continues.

The flowchart of the CEGAR loop specified for statecharts is presented in Figure 4.4.



Statechart,
error state
Create initial abstraction
Abstract statechart
Refine the abstraction
Check reachability
Error state reachable, (Abstract path)
Concretize the counterexample
Concretization fails (Failure state)
Error state not reachable
Terminate
Concretization suceeds (Concrete path)

Figure 4.4: Flowchart of CEGAR for statecharts.

4.3 Initial Abstraction

The initial abstraction is the first abstraction that is checked against the requirements during the loop. Coarser abstractions are preferred, as a completely refined abstraction does not omit any irrelevant detail of the statechart, and it leaves no room to the refinement algorithm. In general, the coarser the initial abstraction is, the more power the refinement algorithm has.

During my work, I constructed and examined two kind of abstractions, one that abstracts both states and variables, and one that only abstracts states. The initial abstraction for the two are inevitably different, as the states only abstraction must contain every variable as visible. Still, the main idea for the two is the same, namely creating an abstraction as coarse as possible.

In terms of state abstraction, setting every state abstracted is not rewarding, as the algorithm will refine them anyway. Setting the states in top-level regions (so the states for which $depth(s) = 1$) as refined, and every other state abstract averts the refiner doing the same.

In case of the states-only abstraction, the initial abstraction is $\mathbf{h}_0 = \{\mathbf{h}_{S0}, true\}$, where *true* assigns $\top$ to every variable, and for a state $\mathbf{h}_{S0}(s) = s$ if $depth(s) = 1$, and $\mathbf{h}_{S0}(s) = s'$ such that $depth(s') = 0$ and $s' \in anc(s)$ otherwise. In this case, the value of $\mathbf{h}_0(V) = V$.

In case of the generic abstraction, the initial abstraction is $\mathbf{h}_0 = \{\mathbf{h}_{S0}, \mathbf{h}_{V0}\}$, where $\mathbf{h}_{V0}$ assigns $\bot$ to every variable that is unbound by C_f, and $\top$ to the bound variables. In this case, the value of $\mathbf{h}_0(V) = bound(C_f)$.

Example 4.4. *Consider the statechart presented in Figure 3.2, and let an error configuration c for it be* $(\{C\}, \emptyset, x = 3, \emptyset)$.

For the states-only abstraction, the initial abstraction for the statechart is $\mathbf{h}_0 = \{\mathbf{h}_{S0}, \{x \mapsto \top, y \mapsto \top\}\}$*, where* $\mathbf{h}_{S0} = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B, B2 \mapsto B, C \mapsto C\}$.

The abstract statechart corresponding to this abstraction is presented in Figure 4.5

In case of generic abstraction, the initial abstraction is $\mathbf{h}_0 = \{\mathbf{h}_{S0}, \{x \mapsto \top, y \mapsto \bot\}\}$ *where* $\mathbf{h}_{S0}$ *is the same as the one described in case of the states-only abstraction. Note, that this is the same abstraction that was presented in Figure 4.3*

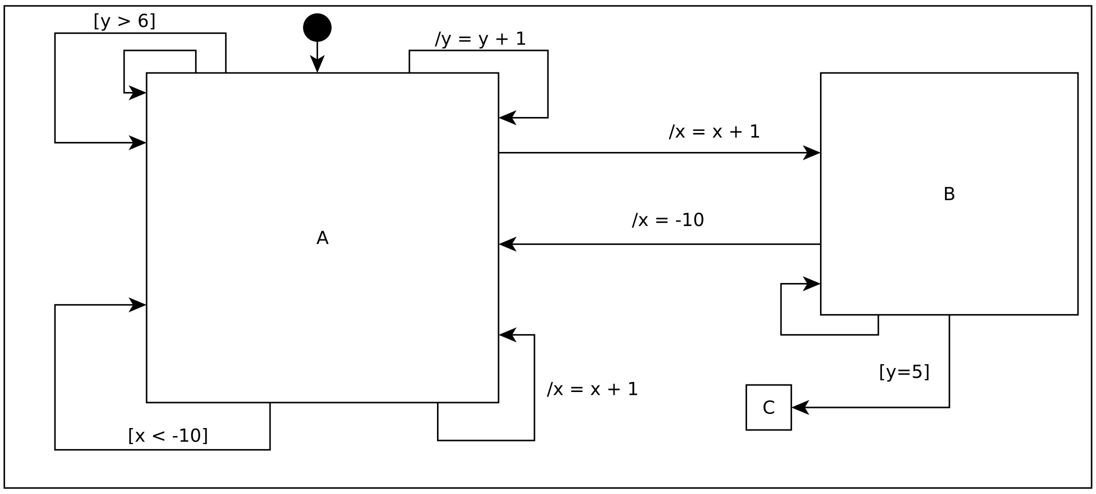


Figure 4.5: Abstract statechart for the initial states-only abstraction in Example 4.4

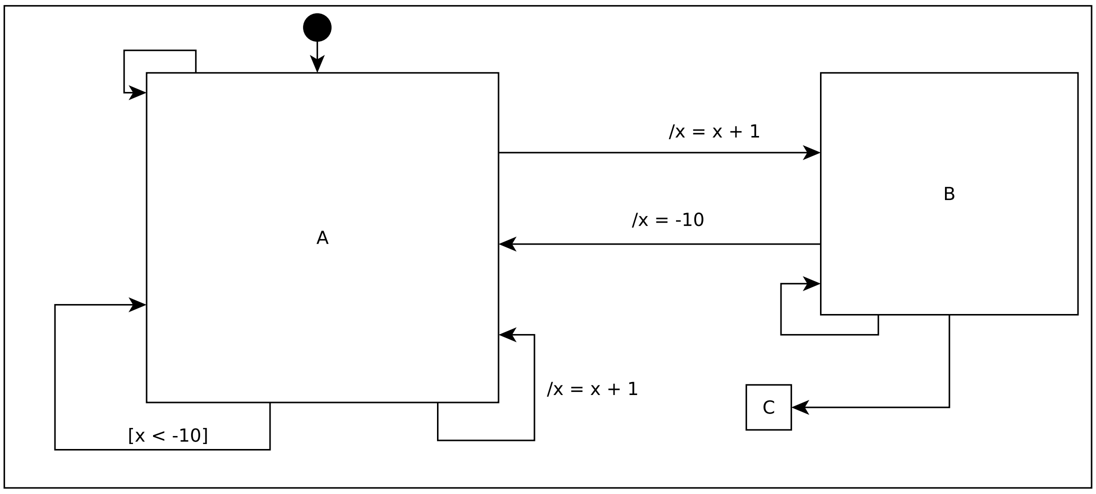


Figure 4.6: Abstract statechart for the initial generic abstraction in Example 4.4

4.4 Model Checking

For both abstractions, state space exploration can be applied. Let $Sc = (S, R, par, I, V, Tr, \mathcal{H})$ be a statechart and $\mathbf{h}$ be an abstraction to it. Recall the algorithms that were described in Section 3.3. Each of them takes a statechart Sc' and a set of error configurations C_f as input, and returns *success*, or an abstract path $\pi_{\mathbf{h}}$ to one of the error configurations as counterexample. Let the input be the abstract statechart $\mathbf{h}(Sc)$, and the set of abstract configurations $C_{f\mathbf{h}} = \{\mathbf{h}(c) \mid c \in C_f\}$. Due to the existential property of the abstraction, if in the abstract statechart $\mathbf{h}(Sc)$ a safety requirement holds, it also holds in the concrete statechart. However if it does not, a path to one of the abstract configurations of $C_{f\mathbf{h}}$ is returned.

The state configurations in the counterexample $\pi_{\mathbf{h}}$ are abstractions of state configurations in Sc, however it is not guaranteed that there is a path $\pi = (c_0, c_1, \ldots, c_n)$ in Sc such that $c_i \in \mathbf{h}^{-1}(c_{\mathbf{h}i})$ for $0 \leq i \leq n$.

Example 4.5. *Recall the statechart Sc presented in Example 4.4. Let the error configuration c be* $(\{C\}, \emptyset, \emptyset, \emptyset)$*. With the generic abstraction, the initial abstraction is* $\mathbf{h}_0 = \{\mathbf{h}_{S0}, \{x \mapsto \bot, y \mapsto \bot\}\}$*, where* $\mathbf{h}_{S0} = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B, B2 \mapsto B, C \mapsto C\}$.

For this reachability problem, with the generic abstraction, a model checker will return the following counterexample: $\pi = ((\{A\}, \emptyset, \emptyset, \emptyset), (\{B\}, \emptyset, \emptyset, \emptyset), (\{C\}, \emptyset, \emptyset, \emptyset))$.

This notation is a bit complex, however the point is, that the value of x and y are invisible for the checker, and so is the inner hierarchy of states. So the path contains abstract configurations where only top-level states are listed. For this reason, the fact that the transition to state C requires y to be 5 is neglected by the checker. However, this path will fail on the concretization.

4.5 Concretizing the Counterexample

If the model checking marks an error state reachable, and provides an abstract counterexample $\pi_{\mathbf{h}} = (c_{\mathbf{h}0}, c_{\mathbf{h}1}, \ldots, c_{\mathbf{h}n})$ for it, that is, a path to it in $Sc_{\mathbf{h}} = \mathbf{h}(Sc)$, it has to be verified if a corresponding path exists in Sc.

A convenient way to do that is to check the existence of an $0 \leq i \leq n$ long path $\pi = (c_0, c_1, \ldots, c_n)$ in Sc, such that $\mathbf{h}(c_i) = c_{\mathbf{h}i}$.

Since the length of the searched path is determined by the value of i, the algorithm of bounded model checking can be applied here, with some modification. The abstract configuration $c_{\mathbf{h}i}$ can be transformed into the encoding function ψ_c defined for Sc, as for every state, event and variable in $Sc_{\mathbf{h}}$ is also a state, event or variable in Sc. However Theorem 3.3 is not applicable here, as for an abstract configuration $c_{\mathbf{h}} = (\omega_{\mathbf{h}}, \rho_{\mathbf{h}}, \mathcal{F}_{\mathbf{h}}, \{\})$ the state set $\omega_{\mathbf{h}}$ is not necessarily a valid set of active states for Sc, and $\mathcal{F}_{\mathbf{h}}$ is also not necessarily a proper value assignment for variables in Sc.

The subject of the concretization is finding values for don't care bits in bit vectors and unassigned variables in configurations. If they have an interpretation $\mathcal{I}$ such that $\mathcal{I}_{\pi_{\mathbf{h}}} \cup \mathcal{I} = \mathcal{I}_{\pi}$, where π is a valid path for Sc and $\mathcal{I}_{\pi_{\mathbf{h}}}$ is the interpretation for the counterexample given by the model checker, the counterexample is concretizable, and the concretized counterexample is π.

Example 4.6. *Consider the result of the checking presented in Example 4.5. The checker method returned an abstract counterexample* (A, B, C)*. The bit vectors assigned to each set of active states in the conuterexample, according to Example 3.6 are*

- $enc(A) = \overline{00XXXXX}$,

- *$enc(B) = \overline{01XXXXX}$,*
- *$enc(C) = \overline{1000000}$.*

For this reason the subject of concretizing, is that is there any interpretation $\mathcal{I}_\pi$ for the don't care bits in the code of A for $i = 0$ and B for $i = 1$, and the value of x and y, such that $\mathcal{I}_\pi \models \psi_{Sc2}$.

This is checked iteratively, first searching a concrete path of 1 configuration, that is abstracted to A, and an initial state. The formula to satisfy is

$$\psi_c(c_I)_0 \wedge \psi_c((A, \emptyset, \emptyset, \emptyset))_0. \tag{4.1}$$

Such interpretation exists, when all the don't care bits are 0, and $x = 0$, $y = 0$.

The next step, a path of two concrete configurations searched, the first abstracting to $(A, \emptyset, \emptyset, \emptyset)$ and the second abstracting to $(B, \emptyset, \emptyset, \emptyset)$. This case the formula to satisfy is

$$\psi_c(c_I)_0 \wedge \psi_c((A, \emptyset, \emptyset, \emptyset))_0 \wedge (\psi_{Tr})_0 \wedge \psi_c((B, \emptyset, \emptyset, \emptyset))_1. \tag{4.2}$$

Such interpretation also exists, as one example path can be $(((\{A1a, A2a\}, \emptyset, \{x = 1, y = 0\}, \emptyset), (\{B1\}, \emptyset, \{x = 1, y = 0\}, \emptyset))$.

However the concretization will fail on the third step, as the following formula is unsatisfiable.

$$\psi_c(c_I)_0 \wedge \psi_c((A, \emptyset, \emptyset, \emptyset))_0 \wedge (\psi_{Tr})_0 \wedge \psi_c((B, \emptyset, \emptyset, \emptyset))_1 \wedge (\psi_{Tr})_1 \wedge \psi_c((C, \emptyset, \emptyset, \emptyset))_2. \tag{4.3}$$

The interpretation of the abstract counterexample assigns variables that can be added as extra constraints to the solver.

Note that not the full transition relation is required to be feeded to the solver, as with abstract configurations $c_{\mathbf{h}i}, c_{\mathbf{h}i+1}$ in the abstract path, the i-th transition can only be a transition t for which the source state is abstracted to a state in configuration $c_{\mathbf{h}i}$ and the target state to a state that is in configuration $c_{\mathbf{h}i+1}$. However this filtering of transition can have relevant impact on the performance of the solver, in order to simplify the description of the algorithm, so lets overlook this filtering.

If for the bound i a concrete path exists, the first i configurations of the path are concretizable. If the $i + 1$'th concretization fails, $c_{\mathbf{h}i}$ is called a *failure state.*

If the concretization succeeds, the concretized counterexample is returned, otherwise, a failure state is returned.

Example 4.7. *Consider the concretization in Example 4.6. The concretization failed upon trying the find the third state of the path. In that case $(B, \emptyset, \emptyset, \emptyset)$ is a failure state, and it has to be refined. (The real cause of failure is that the variable y is not in the set of visible variables, and it has nothing to do with B, however the concretization until B was successful.)*

4.6 Refinement

Should the counterexample be spurious, the abstraction needs to be refined based on the failure state. The applicable techniques differ for states-only abstraction and generic abstraction. For different statecharts, different methods can be effective, an ultimate refinement algorithm can not be created, still, there are some reasonable heuristics. During my work, I defined and investigated two refinement methods.

The method of hierarchy-only refinement presented in Section 4.6.1 can be applied to the states-only abstraction, and the hierarchy first technique, presented in Section 4.6.2, that refines variables only if the state hierarchy can not be refined, can be used for both.

4.6.1 Hierarchy-only Refinement

The hierarchy only refinement only refines state abstraction, and does not modify the variable abstraction. As generic abstraction contains invisible variables, with this method, this kind of abstraction can be refined, however the termination of refinement is not guaranteed.

Given a failure configuration $c_\mathbf{h} = (\omega_\mathbf{h}, \rho_\mathbf{h}, \mathcal{F}_\mathbf{h}, \{\})$, the algorithm modifies the abstraction $\mathbf{h}$ to $\mathbf{h}'$ the following way: for each $s \in \omega_\mathbf{h}$, the direct descendants of s, so states s' where $par(par(s')) = s$ added as refined, formally $\mathbf{h}'(s') = s'$, for every other state $\mathbf{h}'(s) = \mathbf{h}(s)$.

It can be seen that each abstraction is finer than the preceding one, and as there are finite number of states, the algorithm terminates.

Example 4.8. *Recall the previous examples. Examples 4.6 and 4.7 showed that the counterexample of the previous examples was spurious, and the failure state for it was B. Although the abstraction was created for generic abstraction, the hierarchy refinement can be presented on it.*

As the refinement method states, the direct children states of the failure state has to be refined. This case, as the original abstraction was $\mathbf{h}_S = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B, B2 \mapsto B, C \mapsto C\}$*, the refined abstraction is* $\mathbf{h}'_S = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B1, B2 \mapsto B2, C \mapsto C\}$.

The difference between $\mathbf{h}_S$ and $\mathbf{h}'_S$ is in the abstract states assigned to states $B1$ and $B2$.

Note that by this method, the variable abstraction is not refined. This results in the failure of the execution again in B, and than the refinement algorithm can not refine the abstraction any more. This example demonstrates, why is it deprecated to use this refinement method on abstractions, initially created as a generic abstraction.

4.6.2 Hierarchy-first Refinement

The hierarchy first refinement refines the hierarchy first with the method defined in Section 4.6.1 if the set of active states in the abstract failure configuration is not completely refined. However if the failure state can not be refined further, the visibility of variables modified.

It can be seen that if a configuration is a failure state, and all the active states in the configuration are completely refined, but the execution can not continue to the next abstract configuration, it is due to a guard expressions that contain variables invisible by the abstraction. For example is the variable a is invisible, and from state s_1 there is a transition to s_2 with the guard $a = 3$, but the state s_1 only appear in reachable configurations with $a = 1$, the transition can not fire. However, if a is abstracted, there is a transition from configuration s_1 to s_2, because there is a transition from $(s_1, a = 3)$ to $(s_2, a = 3)$.

So the hierarchy-first refinement refines variables appearing in guards. Two approaches are possible, the eager one makes all the contained variables visible, however the lazy approach refines them one by one.

With this refinement, the CEGAR algorithm will not get into an endless loop as by each step a variable is refined, and the only case the execution can not continue if there is no outgoing transition enabled, however in that case, there is no transition enabled in the level of abstraction.

Example 4.9. *The previous example showed the possible refinement for the failure state* $(B, \emptyset, \emptyset, \emptyset)$ *if the states in the set of active states are not completely refined, and in case of hierarchy-first refinement, this step is the same. However it was mentioned at the end of the previous example, that the concretization of the statechart will fail again, and the hierarchy only refinement method could not handle that case. The hierarchy-first refinement however refines variables if the hierarchy has been completely refined.*

The only variable that appears in triggers is y, so independent to the eagerness of the approach, the refinement will be the same. Consider the result of the refinement in Example 4.8 $\mathbf{h}' = (\mathbf{h}'_S, \mathbf{h}'_V)$, *where* $\mathbf{h}'_S = \{A \mapsto A, A1a \mapsto A, A1b \mapsto A, A1c \mapsto A, A1c1 \mapsto A, A1c2 \mapsto A, A2a \mapsto A, A2b \mapsto A, A2c \mapsto A, B \mapsto B, B1 \mapsto B1, B2 \mapsto B2, C \mapsto C\}$, *and* $\mathbf{h}'_V = \{x \mapsto \bot, y \mapsto \bot\}$ *as variables weren't refined.*

The refinement of $\mathbf{h}'$ *is* $\mathbf{h}'' = (\mathbf{h}'_S, \mathbf{h}''_V)$, *where* $\mathbf{h}''_V = \{x \mapsto \bot, y \mapsto \top\}$.

After the refinement, the execution continues with the next iteration checking the abstract model. As each refinement step refines a state or a variable, and there are finite states and variables in a statecharts, after a while, the loop terminates as the identity abstraction is reached.

Chapter 5

Implementation

During my work, I implemented the previously described algorithms in Java. This section summarizes the key features of the implementation and the external tools used.

The implementation is a part of the theta framework, which is presented in Section 5.1. The module architecture is presented by Section 5.2. Finally, Section 5.3 presents a shell that can be used to load and verify statecharts from command line.

5.1 The theta Framework

theta is a verification framework developed at the Fault Tolerant System Research Group of Budapest University of Technology and Economics. The framework provides formalisms and algorithms to describe and verify software and hardware systems. The theta project is currently under development, and does not have any public documentation yet.

The toolkit of the framework is really diverse, this section presents the relevant parts utilized by my work.

5.1.1 Expressions

The framework provides classes to represent first order logic expressions. The interface `Expr` is a common interface for expressions allowed by the syntax of FOL, each having an implementing representation. The ones used during my work are listed below.

- Boolean connectives, for example `AndExpr`.
- Functions, for example `AddExpr`.
- Predicates, for example `EqExpr`.
- To reference constants, `ConstRefExpr` is used.
- To reference variables, `VarRefExpr` is used.

theta also offers utility functions to manipulate the expressions, provided by utility classes. A non exhaustive list of utilities is

- collecting variables in expressions,

- unfolding expressions by replacing variables with the k-indexed constant version.

The framework also provides a general interface for SAT/SMT solvers, so that the underlying solver is interchangeable. The relevant functions of the `Solver` interface are:

- `add`: add a formula or a set of formulas (regarded as the conjunction of the formulas) to the solver,
- `check`: check if the formulas are satisfiable,
- `getStatus`: get the result of checking (satisfiable or unsatisfiable),
- `getModel`: if the formula is satisfiable, get the satisfying interpretation,
- `push`: push the state of the solver to a stack,
- `pop`: remove the formulas added to the solver since the last `pop`.

Currently, the only supported solver implementation is Z3, which is an open source SMT solver developed by Microsoft Research [10].

5.2 Architecture

The software that was realized follows layered architecture. The basic summary of the architecture is presented in Figure 5.1.

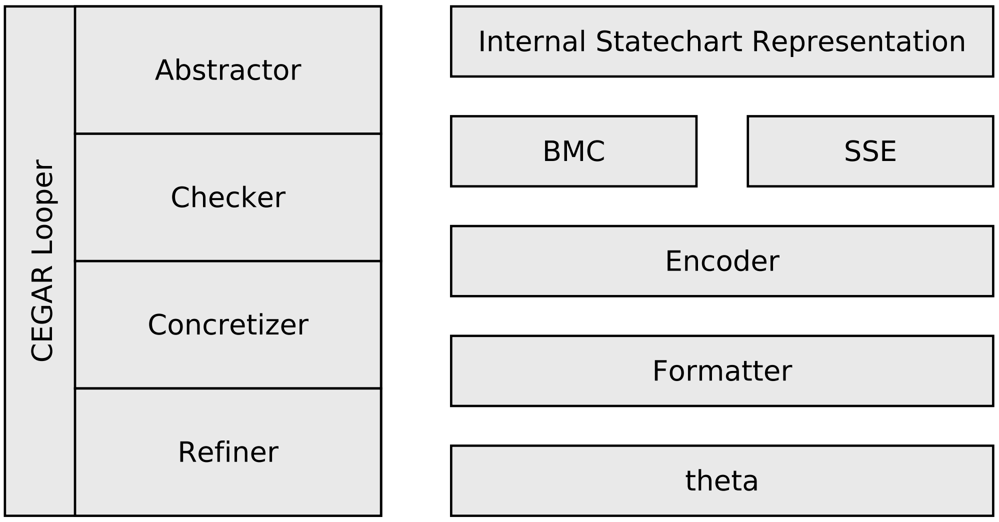


Figure 5.1: Overview of the architecture.

The topmost level is an internal statechart representation, to which all the parsed models are transformed. The representation is detailed in Section 5.2.1.

The next layer contains checkers such as BMC and SSE, where BMC is a bounded model checker and SSE is a state space explorer.

Both rely on the encoder layer, that is the implementation for the *enc* function described in Section 3.1. It assigns a bit vector to states and events in statecharts.

The encoder layer passes the bit vectors to the formatter layer, that creates logic formulas, as described in Section 3.2.

The formulas are passed on to the theta layer, that references a call to the theta framework which is presented in Section 5.1. The framework is used to search a satisfying interpretation for the formula.

The CEGAR Looper layer is an implementation of a CEGAR loop, that follows the design described in Chapter 4. It relies on all the other layers as it uses different checking methods to perform abstraction based verification, for example state space exploration used in the model checking step of the CEGAR loop, whereas BMC used during the concretization. Its detailed implementation is presented in Section 5.2.4.

The CEGAR Looper layer consists four main bricks for resemblance to the modules presented in Chapter 4, however in terms of implementation, the Abstractor and the Refiner logic are implemented by the same module, as specific refinement methods only work with a corresponding abstraction, as it was pointed out in Section 4.6.

5.2.1 Internal Statechart Representation

The theta framework defines an Xtext grammar of statecharts, and provides features for parsing EMF models from text files. EMF is a modeling framework for Eclipse that offers various features for models, for example code generation and building tools. Objects defined in the grammar have their Java object representation, however these objects serve modeling purposes, and they have unnecessary and redundant fields and methods. Furthermore, the grammar allows wilder scale of statecharts than what my algorithms currently support.

During my work, I created and implemented a package for objects to represent elements of statecharts. The implementation corresponds the formalisms introduced in Section 2.2, with the restrictions required by the algorithms presented in Chapter 3 and Chapter 4.

The only change in formalism is the parameters in statecharts that are introduced to be able to verify the same statechart with different constant values. The Xtext grammar defined by the theta framework allows constants and the value assigned to these constants are replaced to the parameter value during the parsing of a statechart.

5.2.2 SSE - State Space Explorer

The state space explorer is the implementation of the algorithms presented in Section 3.3. The two key algorithms are realized by three concrete explorers.

The common feature for them is that with each implementation, during the process of exploration a map is maintained, marking for each reached configuration which configuration it was reached from. Based on the information stored in that map, the path to each reached configuration can be retained.

The naive method, introduced by Algorithm 1 is realized by two different classes, one uses the push-pop functionality of the solvers, and adds the transition relation once, however the other computes the transition formulas over and over again. The two implementations were separated to measure the efficiency of transforming the transition relation to boolean formulas.

The optimized method, presented by Algorithm 2 is realized by feeding the transition relation to the solver, and after getting the reachable configurations one by one, adding an extra constraint to the solver, that prevents finding the same configurations again.

5.2.3 BMC - Bounded Model Checker

The project contains two bounded model checkers, one is based on the bounded model checking algorithm presented in Section 3.4, whereas the other is created for the concretization of an abstract counterexample, as presented in Section 4.5. The first can be used as a replacement of the state space explorer algorithm in the checking step of CEGAR.

5.2.4 The CEGAR Looper

The core of the CEGAR algorithm is a class `CegarLooper`. It takes an initial abstraction, a checker, a concretizer and a refiner object as parameter in its constructor, and has a method `searchCounterexample`, that takes a configuration as an input, and returns a counterexample if any found, or returns null, in case of the configuration being unreachable. The class hierarchy is sketched in Figure 5.2

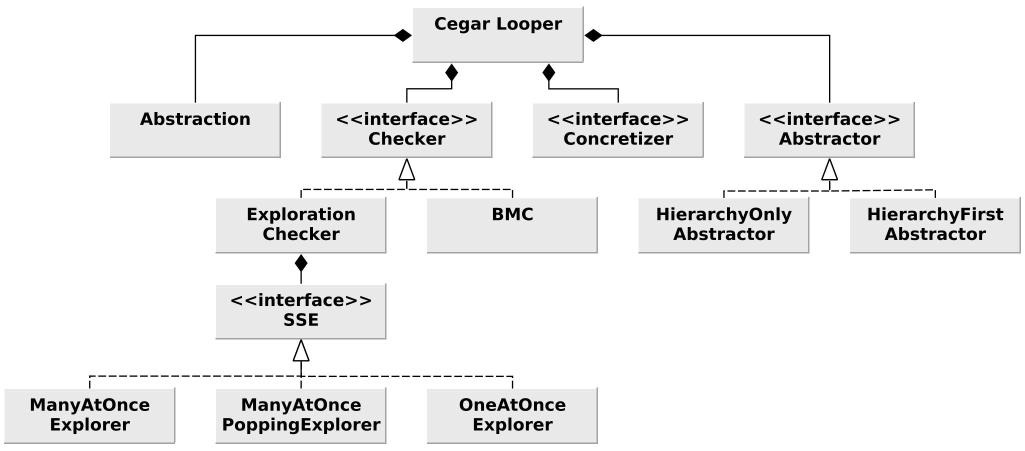


Figure 5.2: Structure of CEGAR related classes

The abstraction is represented by a class `Abstraction`, that is the Java representation of the abstraction function **h**, its `get(`s`)` method returns $\mathbf{h}(s)$ for a state, and for a variable, it has a method `isVisible` returning true if the variable is visible in **h**. Its inner implementation is realized by Maps from the Java collections.

For the objects implementing the checking logic, `Checker` is a common interface with one `check` function, taking an abstraction and a error configuration as parameter, and returning a `Path` object, a representation of paths.

The different checker implementations are listed in Table 5.1. The ID column shows the values, by which they can be referenced from the shell. The abbreviation column contains a label for each method, by which they will be referenced during the evaluation.

`Concretizer` is a common interface for concretizers, however the only existing implementation of it is currently the `BoundedConcretizer`, for which the algorithm was sketched in Section 4.5. It has a `concretize` method, that takes an abstract path, an abstraction and a error configuration as input, and returns a concretized path. If the length of the

Table 5.1: Different check modes for a CEGAR loop.

Name	ID	Abbreviation	Details
Many at once (non-popping)	0	MON	The absolute naive implementation of a state space explorer, presented by Algorithm 1
Many at once (popping)	1	MOP	A state space explorer implementation, that uses the push-pop functionality of the solver to reduce the computation time spent on constructing the transition relation formulas. This implementation is also based on Algorithm 1
Exploring one	2	OAO	The optimized implementation of a state space explorer, based on Algorithm 2.
BMC	3	BMC	With this implementation, instead of exploring the state space, the checking of the abstract model is performed with a bounded model checker. The implementation is based on Algorithm 3

concretized path is not the same as the length of the abstract path, the concretization failed, and the last configuration of the concretized path is a failure state that is passed to a refiner.

The refiners are both implementing the interface `Abstractor`. (The reason behind this convention that different abstractions have to be refined differently, thus the creator of the abstraction can not be separated from the refiner of it.) The interface has two methods, the `createInitial`, that returns an initial abstraction for a statechart, and the method `refine`, that takes an abstraction and a concretized counterexample (for which the concretization failed), and returns a refined abstraction.

There are two abstraction types, states-only abstraction and generic abstraction, the refiners for them are listed in Table 5.2. The ID column shows the values, by which they can be referenced from the shell, and Abbreviation column assigns labels to methods, by which they will be referenced in the tables of the evaluation chapter.

Table 5.2: Different abstraction and refinement modes for a CEGAR loop.

Name	ID	Abbreviation	Details
Hierarchy only refinement	10	STT	The implementation of the refinement method presented in Section 4.6.1. Can only be used with states-only abstraction.
Hierarchy first refinement	11	GEN	The implementation of the lazy approach of the refinement method presented in Section 4.6.2. Can only be used with generic abstraction.

Note that refiners can be used with other abstractions, than the ones created by them, however it is not recommended, as for example a hierarchy only refiner cannot refine the initial abstraction for generic abstraction properly.

5.2.5 Encoder

The `Encoder` encodes states and events to bit vectors. The encoding is done by mapping states to bit vectors. Bit vectors are represented with the pair of an id and a mask, where both id and mask are bit sequences. The two can be combined into a bit vector, by taking the id's value where the masks corresponding bit is 1, otherwise taking a don't care. With the help of this id representation, complex operations, like checking ascendancy or parallelity can be performed with bit operations.

The state-bit vector encoding and event-bit vector mapping is duplex, meaning that both bit vectors for objects and objects for bit vectors can be polled from the encoder. However for a bit vector, there might be more states matching, so in that case, a set of states is returned.

5.2.6 Formatter

The `Formatter` transforms the elements of a statechart to logical formulas of the `theta` framework, using the bit vector mapping implemented by the encoder. The encoding based on the algorithms presented in Section 3.2, however it lacks the support for transitions into a state of a parallel region with the source out of the region. Differently said, the execution of a parallel region always starts from its initial state.

5.3 Shell

In order to be able to change the parameters of tests without modifying the source code, I created a shell, that parses commands from the console or from a text file, and performs actions based on them.

The shell supports variables and loops. Variables can be assigned with the `var [name] = [value]` command and referenced with a $ prefix. Loops are also supported, but they are only limited to while loops. An example usage of the shell is presented in Listing 5.1.

```
load path/to/chart.statechart as sc
create conf scConf for sc
conf scConf state stateToReach
csv log/here.csv
var i = 0
while \$i < 4
    check reachable scConf \$i 11 1800000
    var i = \$i + 1
end
```

Listing 5.1: Example usage of the shell.

The main commands of the shell are detailed in Table 5.3.

The shell can be started as a Java program. If started with without any arguments, it operates as a shell parsing commands from its standard input, and printing results to the standard output. However it can also be started with positional parameters `[input_path]` and `[output_path]`. This case, the program parses commands from the content of the file at the input path, and prints its output to a file at the output path.

Table 5.3: The shell commands.

Command	**Details**
`alias [name] [path]`	Set an alias (string reference) for `path`. If `name` and `path` is left empty, all the all the currently assigned aliases are listed.
`chart ([name])`	Get the statechart with the given name. If `name` is left empty, all the loaded statecharts are listed.
`check reachable [conf_name] [checker_mod] [refinement_mod] ([timeout])`	Check if the configuration `conf_name` is reachable in the statechart it was created for. The parameters `checker_mod` and `refinement_mod` refers to different implementations, and their value is described in Table 5.1 and Table 5.2. If `timeout` is set, than if the verification does not terminate until the timeout in milliseconds elapses, the execution is aborted.
`conf(iguration) ([name])`	Get the state configuration with the given name. If `name` is left empty, all the created configurations are listed
`conf(iguration) [conf_name] bool [var_name] {0,1}`	Bound the boolean variable `var_name` with the value 0 or 1 in configuration `conf_name`.
`conf(iguration) [conf_name] int [var_name] [var_value]`	Bound the integer variable `var_name` with the value `var_value` in configuration `conf_name`.
`conf(iguration) [conf_name] state [state_name]`	Add the state with the given name to the active states of the configuration.
`create conf(iguration) [conf_name] for [statechart_name]`	Get the state configuration with the given name. If `name` is left empty, all the created configurations are listed
`csv {off|[path]}`	Sets or turns off the logging of the checking results into the csv file at `path`. If turned on, after each call of check, a new line is appended with parameters and the result of the checking.
`exit`	Exit the shell
`help`	Get help about the shell and the commands.
`load [path] (as [name])?`	Load a statechart from a `.statechart` file from the given path. If `name` is not empty, the statechart will be assigned `name` as an alias, and can be referred to with it later.
`man`	Get help about the shell and the commands.
`param add {int|bool} [name] [value]`	Add or update a parameter. Statecharts will be loaded with this parameter value.
`param remove [name]`	Remove the `name` parameter.
`set output [path]`	Redirects the output to a text file at `path`. If `path` is not specified, the output is set to the standard output.
`verbose {on|off}`	Turn on/off the detailed printing of checking results. Default value is `on`.

Chapter 6

Evaluation

As a performance test for the comparison of the different implementations, I evaluated each of them on a model of the PRISE (primary-to-secondary leaking) safety function of the Paks nuclear power plant [17] [2].

6.1 The PRISE Logic

A PRISE event is one of the most serious failures in a nuclear power plant, that occurs if there is a rupture or other leakage. The logic initiates an emergency procedure, based on the parameters of the plant. The functional block diagram of the PRISE logic is presented in Figure 6.1, for the detailed description of each input, output and the functionality of the logic, the reader is referred to [17].

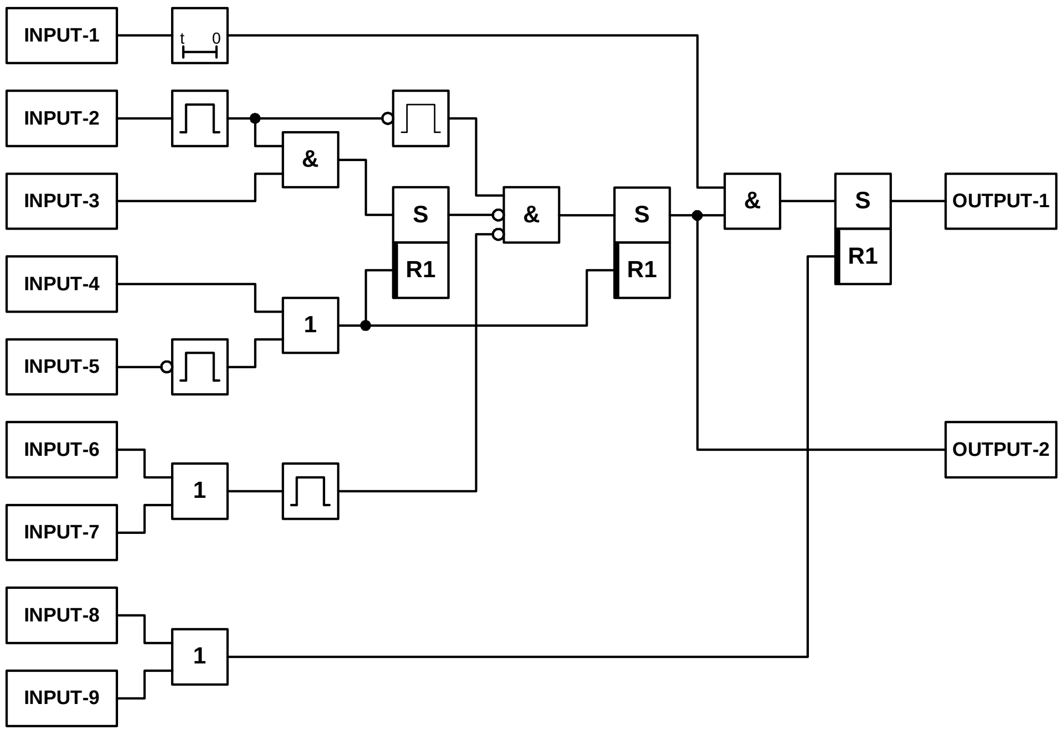


Figure 6.1: The functional block diagram of PRISE logic.

To be able to verify constraints about the logic, I transformed it to a statechart, however as the state-space became unmanageably large for some implementations of the algorithm, I reduced the model, only to the logic related to inputs 2-5, and the logic until the output of the second SR latch was verified. The values of INPUT-6 and INPUT-7 are not taken into consideration, however they serve as a transitive input for the second SR latch through an AND gate. This problem is eliminated by taking that input of the gate as high (true), since it reduces its functionality to an AND gate of 2 inputs.

The input values are the parameters of the system, and the logic performs a continuous check for the PRISE event. This continuity is simulated by an infinite loop of execution. Each functional block is dependent on the others, and their dependency determines their order in the execution loop. The time in the system is modeled by the iterations of the loop, an iteration being the time unit. The nondeterministic change of input is simulated with the possible modification of the variables at the end of each loop.

The diagram contains 3 value holders, blocks, that hold their value for a given amount of time, however, as it was mentioned before, in the statechart time is regarded as an iteration in the loop of the execution. The number of loops regarded as time instead, so the inner representation of holders is actually a counter. The maximum number of the counter value, so the number of loops while the holder holds the signal can be regarded as the parameter of the statechart. Let the limit for the counter that holds INPUT-2 be H_2, for the one that holds the output of H_2 be H_H, and the value that holds the value of INPUT-5 be H_5.

The size of the state space depends on these parameters, an increase in each increases the state space as well. However, for different reachability requirements, they have different impact on the performance. During the evaluation, the reachability of OUTPUT-2=1 (PRISE event) was tested. It turned out to be reachable, however it took several transitions, and an exploration of a rather large number of configurations.

6.2 Metrics

The metrics measured during each verification turn is summarized in the list below. In the tables summarizing the results, they are referred by their abbreviation, that is also mentioned in the list.

- Time: The time elapsed between the start of the first CEGAR iteration and the return of the result.
- Iter: The number of CEGAR iterations, denotes how many times the checker was called. The number of refinements is fewer by one, as at first, the initial abstraction is checked.
- Stt: The percentage of refined states in the abstraction.
- Var: The percentage of visible variables in the abstraction
- Confs(max): The maximum number of configurations explored during the check. It might be different than number of explored states in the last iteration, as with each refinement, the exploration starts over again. In case of the bounded model checker, this metric can not be interpreted, as it explores states indirectly by using the solver. An alternate metric could be the length of the path, however it only refers to the depth of the search, not the breadth.

- Conf(eve): The number of explored states at the last iteration of the algorithm. Similar to the previous one, this metric also can not be interpreted for the CEGAR methods using bounded model checkers.

6.3 Results

Recall, that Chapter 5 presented 4 different checker implementations and 2 different abstraction and refinement pairs were presented. For the parameters $H_2 = 2$, $H_H = 1$ and $H_5 = 1$, the results are presented in Table 6.1. For the resolution of checker ids, see Table 5.1, and the refinement ids can be found in Table 5.2. In each case, the length of the counterexample was 59, and the tests were run with a timeout of 1800s. For the bounded model checker, the maximum length of the counterexample was set to 200.

Table 6.1: The results for parameters $H_2 = 2$, $H_H = 1$, $H_5 = 1$.

Checker	Refiner	Time (s)	Iter	Stt (%)	Var (%)	Confs(max)	Confs(eve)
MON	STT	timeout	2	25.93	100	8610	8610
MOP	STT	1398.63	5	70.37	100	17036	2855
OAO	STT	1250.226	5	70.37	100	17036	2855
BMC	STT	211.499	5	70.37	100	-	-
MON	GEN	48.389	12	70.37	93.75	1484	1484
MOP	GEN	37.817	12	70.37	93.75	1484	1484
OAO	GEN	8.942	12	70.37	93.75	1484	1484
BMC	GEN	77.478	12	70.37	93.75	-	-

The table shows that the one-at-once state space explorer, which is based on the optimized state space algorithm outperforms the other two state space explorers. Amongst the implementations of Algorithm 1, the popping version performs slightly better than the non-popping one, which even fails to terminate with the hierarchy-only refiners.

In case of the refiners, the hierarchy first refinement has better results regarding every metric with every checker mode. The significant acceleration of the termination time is related to the fact that hierarchy-only abstraction does not abstract any of the variables.

However it has to be noted that with the hierarchy first refinement mode, the bounded model checker is the least effective, however with all the variables visible, it performs the best. The improvement is relative though, as it is still loosely three times slower, than with the hierarchy-first abstraction. The reason behind this is that the solver can perform significantly efficient search in the state space than the exploring algorithms.

The percentage of states refined is the same for both refinement methods, however the variable refinement varies. The hierarchy-only abstraction and refinement initially has every variable refined, however in this special case, the states first algorithm also refined $93,75\%$ of the variables, 15 out of 16 in fact.

The hierarchy-only abstraction and refinement failed for the parameters (2,1,1), and with the increase of the parameter H_2, it turned out that every checker method timed out with this refinement.

However, the hierarchy first abstraction provided promising results with parameters $H_2 = 5$, $H_H = 1$ and $H_5 = 1$, as summarized in Table 6.2. The length of the counterexample, found by the terminating methods, was 104. The verifications were run with a timeout of 1800 seconds. For the bounded model checker, the maximum length of the counterexample was set to 200.

Table 6.2: The results for parameters $H_2 = 5$, $H_H = 1$, $H_5 = 1$.

Checker	**Time (s)**	**Iter**	**Stt (%)**	**Var (%)**	**Confs(max)**	**Confs(eve)**
MON	136.712	12	70.37	93.75	3667	3667
MOP	98.787	12	70.37	93.75	3667	3667
OAO	31.06	12	70.37	93.75	3667	3667
BMC	timeout	11	70.37	81.25	-	-

The one-at-once checker still outperforms the other two state space explorers, however the bounded model checker reports time out after 11 iterations. It can be noted, that independent to the parameters, the same percentage of states and variables are refined by the refinement method.

The same examinations can be performed with parameters $H_2 = 10$, $H_H = 1$, $H_5 = 1$, but now excluding the bounded model checker, as it timed out for cases with smaller state space. This case, the length of the counterexample was 179.

Table 6.3: The results for parameters $H_2 = 10$, $H_H = 1$, $H_5 = 1$.

Checker	**Time (s)**	**Iter**	**Stt (%)**	**Var (%)**	**Confs(max)**	**Confs(eve)**
MON	421.485	12	70.37	93.75	7324	7324
MOP	210.640	12	70.37	93.75	7324	7324
OAO	112.897	12	70.37	93.75	7324	7324

In order to examine the scalability of the algorithms, checking has been run with different values of parameter H_2. The comparison of execution times for the different checker methods with generic abstraction is presented in Figure 6.2.

It can be noted, that the state space exploration based algorithms perform better. Amongst those three, the many-at-once implementation, that does not use the push-pop functionnality of the solver is the least effective, as it is remarkably slower for every parameter value than the other two, and fails to terminate for every $H_2 > 10$.

For small parameter values, the Algorithm 2-based one-at-once explorer performs better, however for bigger parameter values, the many-at-once implementation, that uses the solver's pop-push functionality performs better.

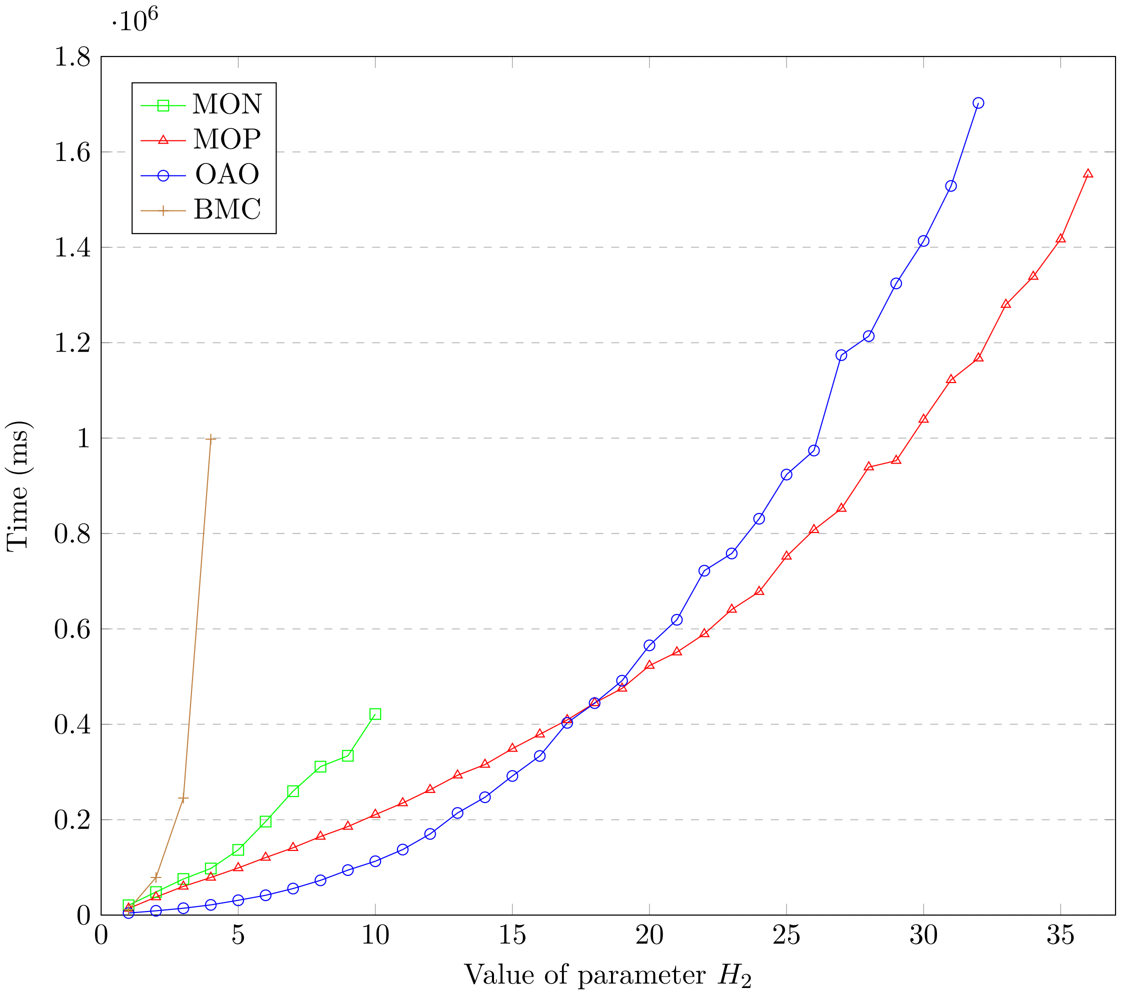
In order to examine the effect of other parameters to the performance of the methods with generic abstraction, further evaluation tests have been carried out. The tests were only ran with the one-at-once checker method. The results of the evaluation is summarized in Table 6.4.

Table 6.4: Further evaluation results.

H_2	H_H	H_5	**Time (s)**	**Iter**	**Stt (%)**	**Var (%)**	**Confs(max)**	**Confs(eve)**
10	1	1	112.897	12	70.37	93.75	7324	7324
10	2	1	119.432	12	70.37	93.75	7324	7324
20	1	1	565.461	12	70.37	93.75	14595	14595
10	1	4	1181.499	12	70.37	93.75	22152	22152
20	1	2	1443.350	12	70.37	93.75	24357	24357

It can be seen in the table, that not all parameters have impact on the number of explored configurations, however with greater state space, the algorithm terminates slower.

Figure 6.2: Comparison of execution time for different checker methods.



Chapter 7

Conclusions

In my work, I examined the possibilities for verifying hierarchical statecharts, focusing on abstraction-based methods.

From the theoretical point of view, I developed encoding methods for statecharts with hierarchically nested states to transform them into logical formulas. I also presented procedures to use the previously defined encodings for the verification of statecharts against reachability requirements. I introduced two hierarchy-based abstraction algorithms for statecharts one applying the abstraction only to the hierarchy, whereas the second approach extends the former with visibility-based abstraction of variables. I also introduced a CEGAR-based algorithm for the verification of statecharts, on top of the previous techniques. My work includes

- techniques to create initial abstractions for statecharts,
- algorithms to check abstract models against reachability requirements,
- a method to concretize the abstract result of the model checking,
- and strategies to refine the abstraction in case of spurious counterexamples.

From the practical point of view, I implemented the encoding and verification algorithm for statecharts in the `theta` framework. I also implemented CEGAR algorithms, including four model checkers for abstract models and two refinement strategies, one for each of the defined abstractions. The algorithms successfully checked a non-trivial industrial model within reasonable time.

During my work, I successfully completed all of the objectives.

- I presented hierarchical statecharts and their elements in Section 2.2.
- I developed an algorithm to transform statecharts to logical formulas in order to be able to verify them in Chapter 3.
- I created a hierarchy-based statechart abstraction and a CEGAR-based verification algorithm to support reachability analysis of statecharts in Chapter 4.
- I implemented the defined methods, as detailed in Chapter 5, and in Chapter 6 I also evaluated the implemented algorithms.

Although, the algorithms proved to be applicable for practical examples, there are several opportunities for improvement.

- The set of the supported statechart elements can be extended with the history indicator and proper handling of the active event set, allowing more that one active event to be in the event queue.
- Further abstractions can be introduced, for example predicate-based abstraction [11] of variables in the statechart.
- The refinement methods could be further improved, for example with unsat core-based variable refinement [15].
- Evaluate the performance of different algorithms on further models.

Acknowledgements

I would like to thank my advisor, Ákos Hajdu, who has been supervising this project for almost a year by now, for putting unspeakable amount of effort into this thesis. Thank you for your help and patience.

I am grateful to Tamás Tóth for his useful comments, tireless correction of my mistakes and for his help with the `theta` framework. I also would like to thank András Vörös, without his coordination this thesis would not be possible.

I would like express my acknowledgement to Gábor Szárnyas and Vince Molnár for their contribution to this thesis and their for their help as my advisor at the past semester.

Finally, I would like to say thank you to my family for their support and patience.

This work was partially supported by MTA-BME Lendület Research Group on Cyber-Physical Systems.

List of Figures

Bibliography

[1] Model checking uml state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357 – 369, 2001.

[2] Tamás Bartha, András Vörös, Attila Jámbor, and Dániel Darvas. Verification of an industrial safety function using coloured Petri nets and model checking. 2012.

[3] Purandar Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *Computing Research Repository*, cs.SE/0407, 2004.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[5] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[6] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936.

[7] Edmund 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.

[8] Edmund M Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[10] 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.

[11] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[13] Alexander Knapp, Stephan Merz, and Christopher Rauh. *Model Checking Timed UML State Machines and Collaborations*, pages 395–414. Springer, Berlin, Heidelberg, 2002.

[14] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view.* Springer, 2008.

[15] Martin Leucker, Grigory Markin, and MartinR. Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Hardware and Software: Verification and Testing*, volume 9434 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2015.

[16] Yael Meller. *Model Checking Techniques for Behavioral UML Models.* PhD thesis, Israel Institute of Technology, 1 2016.

[17] E. Németh, Tamás Bartha, Cs Fazekas, and K M Hangos. Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets. *Reliability Engineering & System Safety*, 94:942 – 953, 2009 2009.

[18] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.