Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Induction-based Verification of Synchronous and Hybrid Programs

Autorin: Xian Li

Datum der Disputation: 10. November 2017

Gutachter:
- Prof. Dr. Klaus Schneider
- Prof. Dr. André Platzer

Promotionskommission:
- Prof. Dr. Christoph Garth (*Vorsitzender*)
- Prof. Dr. Christoph Grimm
- Prof. Dr. Klaus Schneider

Dekan:
- Prof. Dr. Stefan Deßloch

D 386

Abstract

Embedded reactive systems underpin various safety-critical applications wherein they interact with other systems and the environment with limited or even no human supervision. Therefore, design errors that violate essential system specifications can lead to severe unacceptable damages. For this reason, formal verification of such systems in their physical environment is of high interest. *Synchronous programs* are typically used to represent embedded reactive systems while *hybrid systems* serve to model discrete reactive system in a continuous environment. As such, both synchronous programs and hybrid systems play important roles in the *model-based design* of embedded reactive systems. This thesis develops induction-based techniques for safety property verification of synchronous and hybrid programs. The imperative synchronous language Quartz and its hybrid systems' extensions are used to sustain the findings.

Deductive techniques for *software verification* typically use *Hoare calculus*. In this context, Verification Condition Generation (VCG) is used to apply Hoare calculus rules to a program whose statements are annotated with pre- and postconditions so that the validity of an obtained Verification Condition (VC) implies correctness of a given proof goal. Due to the abstraction of macro steps, Hoare calculus cannot directly generate VCs of synchronous programs unless it handles additional label variables or goto statements. As a first contribution, *Floyd's induction-based* approach is employed to generate VCs for synchronous and hybrid programs. Five VCG methods are introduced that use inductive assertions to decompose the overall proof goal. Given the right assertions, the procedure can automatically generate a set of VCs that can then be checked by SMT solvers or automated theorem provers. The methods are proved sound and relatively complete, provided that the underlying assertion language is expressive enough. They can be applied to any program with a state-based semantics.

Property Directed Reachability (PDR) is an efficient method for synchronous hardware circuit verification based on induction rather than fixpoint computation. Crucial steps of the PDR method consist of deciding about the reachability of Counterexamples to Induction (CTIs) and generalizing them to clauses that cover as many unreachable states as possible. The thesis demonstrates that PDR becomes more efficient for imperative synchronous programs when using the distinction between the control- and dataflow. Before calling the PDR method, it is possible to derive additional program control-flow information that can be added to the transition relation such that less CTIs will be generated. Two methods to compute additional control-flow information are presented that differ in how precisely they approximate the reachable control-flow states and, consequently, in their required runtime. After calling the PDR method, the CTI identification work is reduced to its control-flow part and to checking whether the obtained control-flow states are unreachable in the corresponding extended finite state machine of the program. If so, all states of the transition system that refer to the same program locations can be excluded, which significantly increases the performance of PDR.

Acknowledgments

First and foremost I would like to express my deepest gratitude to my Doktorvater, Prof. Dr. Klaus Schneider, who gave me the opportunity to pursue my PhD under his supervision and who led me into the fascinating world of verification. His constant encouragement, continuous guidance, and constructive ideas throughout all the stages of my PhD were a great help.

Furthermore, I am very grateful to Prof. Dr. André Platzer from Carnegie Mellon University for accepting to review this manuscript and for his comments on it. I would also like to express my gratitude to the PhD committee for helping me complete the complex organizational procedures step by step.

Additionally, I want to convey my thanks to my wonderful colleagues throughout the years I spent at the Embedded Systems Chair in Kaiserslautern. Dr. Kerstin Bauer practically tutored me through my first year of study: Without her exemplary counsel, I could definitely not have gotten familiar with my research area as fast as I did. I am also very fortunate to have learned from my predecessors Dr. Andreas Morgenstern, Dr. Mike Gemünde, Dr. Daniel Baudisch, Dr. Manuel Gesell, Dr. Alessandro Gerlinger Romero and Dr. Yu Bai before they graduated and moved forward to industry. For their support during my more recent PhD years, I would also like to thank Tripti Jain, Anoop Bhagyanath, Marc Dahlem, Martin Köhler, Omair Rafique and Maximilian Senftleben: It was a great pleasure to discuss and work with you all!

I owe my sincere gratitude to my husband Georgel without whose valuable feedback and support this thesis could not have reached its present form. Last but not least, I would like to thank my beloved parents for their confidence throughout my study years.

Kaiserslautern, November 10, 2017 *Xian Li*

Contents

List of Acronyms

AHA Affine Hybrid Automata

AIF Averest Interchange Format

CFG Control-Flow Graph

CTI Counterexample to Induction

EFSM Extended Finite State Machine

IC3 Incremental Construction of Inductive Clauses for Indubitable Correctness

LHA Linear Hybrid Automata

MILP Mix-Integer Linear Program

MINLP Mix-Integer Non-Linear Program

NLP Non-Linear Program

ODE Ordinary Differential Equation System

PDR Property Directed Reachability

SCC Strongly Connected Component

SMT Satisfiability Modulo Theories

SOS Structural Operational Semantics

VC Verification Condition

VCG Verification Condition Generation

WCET Worst-Case Execution Time

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

Embedded reactive systems are used in many safety-critical applications where they directly interact with other systems and their *physical environment* with only limited or even no supervision of human operators. Design errors that violate the given specifications can therefore lead to severe damages which are unacceptable in safety-critical applications. For this reason, formal verification of embedded reactive systems in their physical environment is of high interest.

Synchronous programs [BC85; Ber99; Sch09] can directly express the execution steps of embedded reactive systems, while a *hybrid system* [MMP92; Alu+93; Pla10; Alu11] consists of both discrete and continuous transitions that can be often understood as the result of considering a discrete reactive system in a continuous (physical) environment. Therefore, both *synchronous languages* and *hybrid systems* play important roles for embedded reactive systems in *model-based design* [NM09]: Here, a model of the system to be built is made, simulated, analyzed and optimized before actually building the system — preferably with the help of computer-aided design tools.

Safety properties are widely used to encode the system's specifications for formal analysis, which assert that observed behavior of the system always stays within some allowed set of finite behaviors [KV99]. The thesis develops induction-based techniques for safety property verification of synchronous and hybrid programs. In particular, the imperative synchronous language Quartz and its extension to hybrid systems are used to exemplify the findings.

For safety property verification, synchronous and hybrid programs are usually translated to state transition systems so that various kinds of model-checking procedures can be applied. Many sophisticated optimizations and specializations for *symbolic model checking* [Bur+90; Bur+92] have been found. However, all existing BDD-based [Bur+93; McM93], SAT/SMT-based [Aud+02; BDS02; FH07; Frä+07; EFH08; MB08; GAC12a; Cim+13; SB14; Cha+16], interpolation-based [McM03], IC3-based [Bra11], bounded model-checking [Cla+01; Bie+03] as well as abstraction and reduction techniques [Alu+00; Cla+03; ADI06; GSM07; BGS10; Tiw12] suffer from the state-space explosion problem. It is therefore desirable to also apply deductive techniques [Gup92; CW96; KG99] to safety property verification of synchronous and hybrid programs. They are usually based on automated or interactive theorem provers, like the HOL family of systems [Gor86], Isabelle [Pau94], PVS [ORS92], ACL2 [KM08], KeYmaera [PQ08; Pla10] and many others. The user interacts with these systems by setting up proof goals and applying proof rules until a proof is finally obtained. The mentioned systems are general theorem provers and support undecidable and very expressive higher order logics. These theorem provers were used for the verification of many kinds of systems, in particular, many kinds of software systems [Bon10].

Firstly, the most popular deductive techniques for *software verification* are based on *Hoare calculus* [Hoa69; Gri81; Apt81], where program statements $\mathtt{S}$ are enclosed in pre- and postconditions φ, ψ to form proof goals $\{\varphi\}\ \mathtt{S}\ \{\psi\}$ which denote that if φ holds at starting time of $\mathtt{S}$, then ψ holds at termination time of $\mathtt{S}$ (provided that $\mathtt{S}$ will terminate). The Hoare calculus then provides for every statement of the considered programing language a decomposition rule to reduce a proof goal for that statement to proof goals using only its sub-statements. In particular, loops are thereby reduced by Hoare's famous invariant rule: Given a loop statement $\mathbf{while}(\sigma)\ \mathtt{S}$ with its invariant $\mathcal{I}$, one can deduce from a proof of $\{\sigma \wedge \mathcal{I}\}\ \mathtt{S}\ \{\mathcal{I}\}$ that also $\{\mathcal{I}\}\ \mathbf{while}(\sigma)\ \mathtt{S}\ \{\neg\sigma \wedge \mathcal{I}\}$ holds. Since invariants, as well as pre- and postconditions usually have to be provided by the user, the approach is in general an interactive one.

Verification Condition Generation (VCG) [Mat+06] aggressively applies all rules of the Hoare calculus to a program whose statements are annotated with pre- and postconditions so that the validity of the obtained formulas implies the correctness of the given proof goal $\{\varphi\}\ \mathtt{S}\ \{\psi\}$. In principle, it is thereby sufficient to only provide the right loop invariants. VCG can work very fast, since they just make a linear pass over the program and the obtained formulas can then be checked one after the other or in parallel. This way, the overall verification problem is split into two independent parts: First, generating the VCs, and second proving them — this can be done by different tools and, if desired, in parallel.

VCG is not limited to Hoare calculus: Indeed, early work done by Floyd [Flo67] used *inductive assertions* instead of loop invariants to prove assertions of loops, i.e., cycles in the flow graphs of programs. Induction-based approaches without VCG have already been applied to the verification of hybrid systems: A similar idea has been presented by *barrier certificates* [PJ04] which ask for a function that maps safe and unsafe states to non-negative and negative real numbers, respectively, such that the

transitions do not make the function's value negative. The method is therefore in the spirit of Lyapunov functions for proving the stability of ordinary differential equations. An improved induction approach has been presented by *Platzer's differential induction* [Pla08; PC08; PC09] that does not require to solve the differential equations and uses the *directional derivative* instead.

On one hand, many systems built with synchronous languages have already been formally verified [BKS03; DBCB04]. However, these results were all obtained by model-checking and are therefore limited by the state-space explosion. It is therefore desirable to also apply deductive techniques like VCG to synchronous programs. On the other hand, there has been considerable progress in the area of Satisfiability Modulo Theories (SMT) solvers that can deal with arithmetic formulas like MathSAT [Aud+02; Cim+13], CVC [BDS02; Det+14], ICS [Men+04], iSAT [SKB13; SB14], HySAT [FH05; Frä+07; FH07; EFH08], ABSOLVER [BPT07], BACH [Bu+08], Z3 [MB08], MetiTarski [Pau12], HybridSAL [Tiw12], and on special techniques for non-linear arithmetics [GAC12a; GAC12b]. These tools are already used for the verification of (linear) hybrid systems and typically employ *bounded-model checking* where safety properties are unrolled for some finite number of discrete transitions. However, there is no VCG procedure that generates proof goals of correctness statements about hybrid programs before [LS15c], mainly because of the lack of modeling languages for hybrid systems with structured programming constructs.

Secondly, Property Directed Reachability (PDR) [Bra11; Bra12a; Bra12b; EMB11; GR16; HBS12; HBS13; SB11] is currently considered to be the most efficient verification method for safety properties. The core algorithm has been introduced in [Bra11] for hardware model checking and has been implemented in a tool called Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3). In essence, given a symbolic representation of a state transition system $\mathcal{K}$ and a state property Φ, the algorithm tries to prove that Φ holds on all reachable states of $\mathcal{K}$ by means of induction. It first checks whether Φ holds on all initial states (induction base), and then checks whether Φ holds on all successor states of those states that satisfy Φ (induction step). However, the latter may fail even though Φ holds on all reachable states since there may exist unreachable states satisfying Φ that have successor states that do not satisfy Φ. Such states are so called Counterexample to Induction (CTI)s and have to be incrementally learned and excluded by the PDR method.

PDR is very efficient since, in the best case, it may just use a SAT or SMT solver to prove the induction base and induction step. It doesn't need to compute fixpoints as *symbolic model checking* does, nor to unroll the transition relation as required in *bounded model checking* nor to construct Craig interpolants as required by *interpolation-based model checking*. Instead, it maintains a sequence of predicates $\Psi_0, \ldots, \Psi_k$ that include the state sets $\mathcal{X}_0, \ldots, \mathcal{X}_k$ that are reachable in no more than $0, \ldots, k$ steps, respectively. If the induction proof fails, PDR either increments this sequence with a new predicate Ψ_{k+1} or improves the approximations of the predicates Ψ_i by removing CTIs. The latter is done by first *checking the unreachability of the CTI* and then *adding a most general inductive clause* to the predicates to remove the CTI together with as many further

CTIs as possible within one step. These steps are repeatedly applied until either finally a reachable counterexample is found or one of the predicates Ψ_i becomes inductive[1]. In the worst case, PDR may have to compute the state sets that are also computed by a forward fixpoint computation of the reachable states. Usually, the construction of predicates $\Psi_0, \ldots, \Psi_k$ until the induction proof works, is typically better than the fixpoint iteration — both the number of predicates k can be less than the number of iteration steps in the fixpoint computation and also less work is usually done within one iteration step.

PDR has meanwhile been incorporated in many model checkers like nuXmv[2], ABC [BM10], IIMV[3], PdTRAV[4], and Kind2[5], and many detailed optimizations have been added: In particular, [EMB11] suggested a couple of modifications to the original PDR method to improve its performance and to simplify its implementation. Also, an algorithm using SAT solvers instead of ternary simulation was proposed in [Cho+11] to determine most general clauses from the CTIs, a lazy abstraction-refinement technique has been combined with PDR for large industrial hardware designs in [VGS12], and the clause generalization was improved to explore states farther than the counterexamples in [HBS13]. Finally, [GR16] implemented different variants of PDR for hardware model checking in nuXmv, and conducted a systematic evaluation using the benchmarks of the latest hardware model checking competition.

The original PDR method was introduced for hardware model checking, and therefore operates directly on propositional formulas $\mathcal{I}$ and $\mathcal{T}$ that encode the initial states and the transition relation of a considered hardware circuit with finitely many states. It is however clear that PDR can handle also *infinite state systems* [HB12; CG12] by replacing SAT solvers with SMT solvers, so that PDR can be also applied to *software model checking*.

Besides the use of non-boolean data types, another distinction between hardware and software model checking is the consideration of the programs' syntax, i.e., the *control-flow of the programs*. One of the first attempts in that direction was presented with the TREE-IC3 method in [CG12] which unrolls the Control-Flow Graph (CFG) to an abstract reachability tree. An adaptation of the TREE-IC3 algorithm is used in [RKL14] to synthesize controllers for discrete-event systems modeled by Extended Finite State Machines (EFSMs). Also, [WK14] presented a software verification algorithm based on the refinement of loop invariants using a generalization of PDR to the theory of quantifier-free formulas over bitvectors [WK13]. In [LNN15], the relative inductive reasoning [BM07] is performed over regions that are defined by symbolic representations with respect to certain locations of a control-flow automaton.

Hence, PDR has already been optimized in many ways. In particular, it has been integrated with *SMT solvers to deal with higher data types* and it has been enriched with

[1] This must finally happen since (1) those predicates Ψ_i that cover the reachable states $\mathcal{S}_{\mathsf{reach}}$ will converge to $\mathcal{S}_{\mathsf{reach}}$ due to improving the approximations and (2) $\mathcal{S}_{\mathsf{reach}}$ is an inductive set.

[2] `https://nuxmv.fbk.eu`

[3] `http://ecee.colorado.edu/~bradleya`

[4] `http://fmgroup.polito.it`

[5] `http://kind2-mc.github.io/kind2`

the use of *loop invariants and control-flow graphs for software model checking.* However, these refinements are not just useful for software model checking since hardware circuits are usually synthesized from more abstract high-level languages like synchronous programs or synchronous subsets of hardware description languages. In these programs, the *synchronous model of computation* reflects the executions of hardware circuits: At each tick of the clock, new input values are read, internal states are updated, and outputs are immediately computed. Similar to traditional imperative programming languages, imperative synchronous programming languages like Esterel [BC85; Ber99] and Quartz [Sch09] also have statements like conditionals, sequences, various kinds of loops, etc. Thus, one can also derive control-flow information from these kinds of hardware descriptions that is only implicitly available in the generated synchronous circuits.

To the best of our knowledge, there is not much directly related work: One of the first attempts to check synchronous programs by the PDR method is presented in [Cim+14]. However, the main topic of [Cim+14] was integrating PDR with predicate abstraction — this was demonstrated with synchronous Lustre programs [Hal+91]. The latter is however a dataflow synchronous language without a control-flow and is also the input language of the Kind2 model checker [Cha14; Cha+16] that focuses on invariant generation and compositional reasoning. Closer to the work presented in this paper is [LNN15], where additional control-flow information similar to ours has been added for sequential programs. However, the method proposed in [LNN15] is limited to sequential programs, concurrent synchronous programs were not considered there at all.

1.2 Contribution

The content of the thesis is divided into four parts. The first part is about the state-of-art. The second part presents induction-based VCG methods tailored to synchronous programs and hybrid programs. The third part shows that the PDR method benefits from imperative synchronous programs by effectively using the distinction between the control- and dataflow. The last part demonstrates the induction-based VCG methods with several synchronous and hybrid Quartz programs.

Induction-based VCG Procedure Development

The first problem to be solved would be to define a Hoare calculus for synchronous and hybrid programs. After many attempts [GS12; Ges14], we found that this is not directly possible. The main reason is that the control-flow graphs of synchronous programs are usually irreducible graphs which cannot be translated to sequential programs without goto statements or without additional variables [AM71]. Note that this problem is not just caused by the concurrency of these programs, indeed, there are extensions of the Hoare calculus to concurrent programs [OG76a; OG76b; Lam80; Sti88], which however only consider interleaved concurrency instead of synchronous concurrency: The problems that generate irreducible graphs are caused by the abstraction to macro steps done by synchronous programs, which means that several (micro) execution steps are

combined into one reaction step that corresponds with a transition in the state transition diagram. Therefore, this thesis employs *Floyd's induction-based* approach and shows how it can be used to generate VCs for synchronous and hybrid programs.

- The proposed VCG methods consist of two steps, where the first step consists of computing for a synchronous or hybrid Quartz program its Extended Finite State Machine according to the operational semantics of the language (see Figure 2.9 as an example). This EFSM has one node for every discrete control-flow state of the program, and every node is labeled with a set of guarded actions that encode the dataflow of that node, i.e., assignments to both discrete and continuous variables. Transitions between nodes are labeled with trigger conditions, and every transition corresponds with one macro step of the synchronous reactive program. In the second step, the user has to provide *inductive assertions (invariants)* for each safety property and each component of the generated Extended Finite State Machine (EFSM). The components can be either paths along several control-flow states, or single transitions between two control-flow states, or paths between different Strongly Connected Components (SCCs), or single transitions among control-flow states related to the same loop statement in the program, etc. The VCG methods then apply the induction rules and generate proof goals that correspond with induction steps and bases.

  Five VCG methods are presented in Chapter 3. Given the right assertions, the VCG methods can automatically generate a set of VCs that can then be checked by means of SMT solvers or automated theorem provers. All five methods are proved sound and can be applied to any kind of programs with a state-based semantics. The relative completeness for the methods are proved, provided that the underlying assertion language is expressive enough.

Control-flow Guided PDR Optimizations

For formal verification, imperative synchronous programs are usually translated to equivalent state transition systems that are symbolically represented by means of formulas $\mathcal{I}$ and $\mathcal{T}$ to encode the initial states and the transition relation. The second problem to be solved would be to apply PDR to this kind of state transition systems for imperative synchronous programs. As we will show in Chapter 4, PDR can become more efficient by effectively using the distinction between the control- and dataflow, which is the second contribution of the thesis.

- Before calling the PDR method, it is beneficial to enhance the usual transition relation $\mathcal{T}$ by additional control-flow invariants that contain information about the reachable control-flow states. For example, an invariant states that no state is reachable where the control would be active in both if-statement and sequential substatements. By the semantics of the programs, this can also be proven with the original formulas, and in particular, it can be verified by PDR or any other

suitable verification method as well. However, this requires the computation of reachable states that is actually what PDR wants to avoid as much as possible.

Two methods to compute additional control-flow invariants will be presented that differ in how precisely they approximate the reachable control-flow states and also in the runtime required for their computation. The first method considers the abstract control-flow transition system for any imperative synchronous program $\mathcal{P}$, and compute its reachable states with a symbolic state space traversal. This way, we obtain the reachable states $\mathsf{ReachCF}(\mathcal{P})$ that abstracts from the dataflow and just considers the control-flow. The second method that computes a control-flow invariant $\mathsf{InvarCF}(\mathcal{P})$ is straightforward and even in linear-time in the size of the program. Actually, this control-flow invariant $\mathsf{InvarCF}(\mathcal{P})$ was already computed by the compiler of the Averest[6] framework.

Both $\mathsf{ReachCF}(\mathcal{P})$ and $\mathsf{InvarCF}(\mathcal{P})$ over-approximate the reachable control-flow states with $\mathsf{ReachCF}(\mathcal{P})$ being more precise than $\mathsf{InvarCF}(\mathcal{P})$. Applying PDR to the enhanced transition relation $\mathcal{T}' := \mathcal{T} \wedge \mathsf{ReachCF}(\mathcal{P})$ or $\mathcal{T}'' := \mathcal{T} \wedge \mathsf{InvarCF}(\mathcal{P})$ some transitions of unreachable states are removed. The added control-flow information is not needed for synthesis and is therefore not explicitly encoded in the generated systems, but it can be easily derived from the original imperative synchronous programs and used for verification. As will be shown later, there are safety properties that become inductive only with the enhanced transition relation, so that PDR can prove them in one step, while otherwise, PDR would have to apply arbitrarily[7] many incremental steps.

- The PDR method can be further optimized by the control-flow of the imperative synchronous program under scrutiny. The control-flow information helps PDR reason about the unreachability of counterexamples and then generalize them to all states with the same control-flow locations.

  As we will explain later, the transition relation $\mathcal{T}$ of such an imperative synchronous program $\mathcal{P}$ can be derived as a conjunction $\mathcal{T} = \mathcal{T}^{\mathsf{cf}} \wedge \mathcal{T}^{\mathsf{df}}$ of transition relations over the same set of states, one for the control-flow $\mathcal{T}^{\mathsf{cf}}$ and another one for the dataflow $\mathcal{T}^{\mathsf{df}}$, respectively. The reachability of a state s' from a state s in $\mathcal{T}$ is equivalent to the reachability in both $\mathcal{T}^{\mathsf{cf}}$ and $\mathcal{T}^{\mathsf{df}}$. Hence, if s' is not reachable from s in $\mathcal{T}^{\mathsf{cf}}$, we can already conclude its unreachability in $\mathcal{T}$ and thus can declare it as a CTI without considering the full transition relation $\mathcal{T}$. The advantage is that the state transition system defined by the control-flow $\mathcal{T}^{\mathsf{cf}}$ is much simpler to deal with (even though it has even more reachable states) since we can compute a usually small quotient in terms of EFSM for it. We may also use traditional model checking approaches for that purpose since $\mathcal{T}^{\mathsf{cf}}$ can be usually represented efficiently by means of BDDs. This way, we can add a first, less expensive test for checking the unreachability of a counterexample in the corresponding EFSM.

[6] `http://www.averest.org`

[7] Arbitrary means here that based on parameters, we can increase the number of additionally required steps beyond every bound.

If that test should fail, we use the traditional PDR reachability checks that will generate further reachability queries that then (again) first ask for reachability by EFSM in every step.

After determining a counterexample is unreachable in the corresponding EFSM, we can reduce it to its control-flow variables and hence, obtain a generalized clause that will exclude all states of the transition system that refer to the same control-flow states (but with different values of the data variables). Thus, we can avoid expensive clause generalizations that are required in PDR to narrow the over-approximations of the clause sets $\Psi_0, \ldots, \Psi_k$. The clauses generated this way may not be relatively inductive, but by inspection of the control-flow transition relation $\mathcal{T}^{\mathsf{cf}}$, we can directly decide about their unreachability which is sufficient for excluding these states. Also, the clauses may not be minimal, but since they are quickly generated, it is usually a good compromise. Alternatively, standard methods could be applied — at a cost — to minimize them.

1.3 Thesis Structure

The thesis is organized as follows:

- Chapter 2 describes some essential related work: The imperative synchronous language Quartz and it extension to hybrid systems, techniques and tools for safety property verification and software verification.
- Chapter 3 presents the five induction-based VCG methods for synchronous and hybrid programs. Given the right assertions, the proposed methods automatically generate a set of VCs that can then be checked by means of SMT solvers or automated theorem provers. All methods are proved sound and relatively complete.
- Chapter 4 optimizes the PDR method for safety verification of imperative synchronous programs. Firstly, two methods that compute control-flow invariants to over-approximate the reachable control-flow states are presented. Afterwards, an improvement is described that helps PDR decide the unreachability of counterexamples, and then generalizes them to refine the over-approximations.
- Chapter 5 provides five synchronous and hybrid Quartz programs to show the feasibility of the proposed five VCG methods.
- Chapter 6 concludes with a summary of the thesis and discusses future work.

Chapter 2

Preliminaries

Contents

This chapter first describes some essentials for modeling embedded reactive systems in their physical environments: The synchronous model of computation, hybrid systems, the imperative synchronous language Quartz, and its extension to hybrid systems. The Averest system — with which symbolic representations of Quartz programs can be automatically derived as foundations for their formal analysis — is afterwards briefly introduced. We are just interested in *safety property verification*, i.e., we want to prove

that some property Φ holds on all reachable states of a Quartz program $\mathcal{P}$ by its symbolic representations. Therefore, the underlying satisfaction problem generated is discussed in detail, and typical model checking and deductive verification techniques and tools are categorized in accordance with the type of satisfaction problems that they could solve. In the end, we explain the main ideas behind the Verification Condition Generation (VCG) and Property Directed Reachability (PDR).

2.1 Modeling of Synchronous Systems

Synchronous languages [BB91; Hal93] such as Esterel [Ber00], Lustre [Hal+91] and Quartz [Sch09] have been developed for the design of reactive systems. These languages implement the *synchronous model of computation* that divides the execution of synchronous systems into a discrete sequence of reaction steps which are also called macro steps. A macro step consists of finitely micro steps whose maximal number is known at compile time. Macro steps correspond to reaction steps of reactive systems, and micro steps correspond to atomic actions like assignments of the program that implement these reactions. A macro step does not consume time in the abstraction while from one macro step to the next one a logical unit of time passes. Variables of a synchronous program are *synchronously updated* between macro steps so that the execution of the micro steps within a macro step is done in the same variable environment of their macro step. This synchronous update is important for avoiding data races, and therefore to ensure determinism.

Synchronous systems abstract the communication and computation delays, and reflect the ideal model of embedded reactive systems: At each reaction step, new input values are read, internal states are updated, and outputs are instantaneously computed. The synchronous model of computation not only leads to a convenient design model for embedded reactive systems that allows deterministic and efficient hardware and software syntheses as well as a simplified estimation of worst-case reaction times. It is also the key to a compositional formal semantics which is a prerequisite for formal verification and provably-correct synthesis procedures.

2.1.1 The Imperative Synchronous Language Quartz

Quartz is an imperative synchronous language that is derived from the Esterel language. The language offers many data types like booleans, bit-vectors, signed and unsigned integers that may be bounded or unbounded, real numbers, as well as compound data types like arrays and tuples. Modules are declared with an interface that determines inputs and outputs, and a body statement that may use additional local variables. In the following, we list some of the possible statements used in our later examples. A complete definition of the language can be found in [Sch09]. Provided that `S`, $\texttt{S}_1$ and $\texttt{S}_2$ are program statements, `w` is a control-flow variable, `x` is a variable, σ is a boolean expression, and α is a type, then the following are program statements as well (parts given in square brackets are optional):

- $\mathtt{x} = \tau$ and **next**(x) $= \tau$ (assignments)
- **assume**(φ), **assert**(φ) (assumptions and assertions)
- w:**pause** (start/end of macro step)
- $\mathtt{S}_1; \mathtt{S}_2$ (sequences)
- $\mathtt{S}_1 \parallel \mathtt{S}_2$ (synchronous concurrency)
- **if**(σ) $\mathtt{S}_1$ **else** $\mathtt{S}_2$ (conditional)
- **while**(σ) S (while-loops)

The **pause** statement defines a control-flow variable w — this boolean variable is true iff the control-flow is currently at w:**pause**. Since all other statements are executed in zero time, the control-flow only rests at these control-flow positions in the Quartz program, and thus the possible (discrete) control-flow states are the subsets of these control-flow variables.

There are two variants of discrete assignments that both evaluate the right-hand side τ in the current macro step: Immediate assignment $\mathtt{x} = \tau$ transfers the value of τ to the left-hand side x directly, while delayed assignment **next**(x) $= \tau$ assigns the value in the next macro step. If the value of variable x is not determined by assignments of the current or previous macro step, a default value is used according to the declaration of the variable. To this end, declarations of variables consist of a *storage class in addition to their types*. There are two storage classes, namely **mem** and **event** that choose the previous value (**mem** variables) or a default value (**event** variables) in case no assignment determines the value of a variable.

In addition to the statements known from other imperative languages (conditionals, sequences and loops), Quartz offers synchronous concurrency $\mathtt{S}_1 \parallel \mathtt{S}_2$ and sophisticated preemption and suspension statements (not shown in the above list), as well as many other statements for the comfortable description of embedded reactive systems. There is also the possibility to call once implemented modules and to store modules in packages to support their re-use in the form of software libraries.

The execution of module M1 in Figure 2.1 is explained with the help of its symbolic simulation graph depicted in Figure 2.2. Circles and rectangles represent control-flow states and discrete assignments, respectively, and only active control-flow variables are displayed in control-flow states — for example, control-flow variables w0 and w1 are

```
1  module M1(real x){
2      // variable declaration: x is a memorized inout real variable
3      while(true) {
4          w0: pause;
5              // x is assigned to 1.0
6              x = 1.0;
7              // x will be assigned to 0.0 in the next macro step
8              next(x) = x-1.0;
9          w1: pause;
10     }
11 }
```

Figure 2.1: Synchronous Quartz Module M1

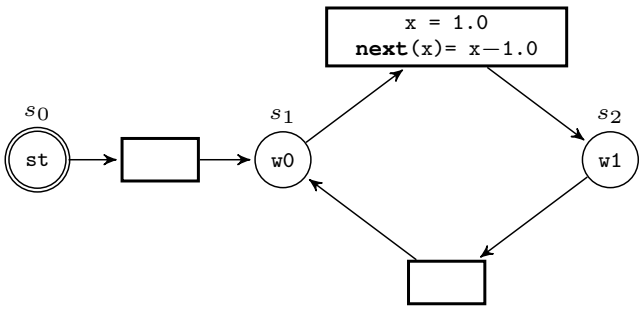

Figure 2.2: Symbolic Simulation Graph of Module `M1`

false while `st` is true in the initial control-flow state s_0 (marked with double circles). No immediate or delayed assignments will be executed in the first macro step, therefore, `x` is set to default, i.e., `0.0`. Afterwards, control-flow moves to s_1, where only the control-flow variable `w0` holds, and then starts the second macro step: `1.0` is assigned to `x` due to the immediate assignment in line 6 in Figure 2.1. After that control-flow moves to s_2 for the third macro step, in which the value of `x` is equal to `0.0` for the delayed assignment of the previous macro step in line 8 of Figure 2.1. Later, the control-flow moves to s_1, since line 4-9 in Figure 2.1 is enclosed by a while-loop statement. The program will never terminate, which means once the control-flow leaves the initial control-flow state s_0, it will move between s_1 and s_2: In the even numbers of macro steps, the value of `x` is `1.0`, whereas in the odd ones, `x` is assigned to `0.0`.

2.2 Modeling of Hybrid Systems

The behavior of a *hybrid system* [Alu+93; MMP92; Pla10; Alu11] consists of both discrete and continuous transitions which are often the result of considering a discrete reactive system in a continuous (physical) environment. Within a discrete state, *differential equations* determine the values of the variables until some *release condition* becomes true to change the discrete state. A discrete transition performs then a sequence of assignments to the variables and determines a new set of differential equations and a new release condition.

Hybrid automata [Alu+93; Alu+95; Hen96] are often used as the underlying semantic model for hybrid systems and is supported by many languages and tools [Car+06; DLSV12]. Linear Hybrid Automata (LHA) and Affine Hybrid Automata (AHA) are special hybrid automata, where LHAs are restricted to linear dynamics for continuous state variables, while affine dynamics are allowed for AHAs. While this semantic model is widely accepted, it is less clear how to define *comfortable modeling languages* that lend themselves well for both the discrete and the continuous facets of hybrid systems, and that offer a realistic kind of *concurrency* which can be used for *compositional reasoning* [AH97; Fre05]. The semantics of these languages is often difficult to describe and it has to deal with special phenomena like unwanted *Zeno behaviors* [Ame+06; ZLA06].

Hybrid programs are based on precise and readable modeling languages, so that the hybrid systems could be encoded with data types and programming statements. Typical examples are KeYmaera [Pla10], HyDI [CMT11], HybridSAL [Tiw12] and the extension of Quartz to hybrid systems [Bau12].

2.2.1 The Extension of Quartz to Hybrid Systems

While time in synchronous languages is given in the abstract form of macro steps, hybrid systems require the consideration of physical time. In order to combine these inherently different concepts of time, the computational model of a macro step is endowed with continuous evolution that takes place between the immediate and delayed assignments of the macro step. During the physical time consuming continuous evolution, variables of the new storage class **hybrid** change their values according to the new *flow assignments* x $\leftarrow \tau$ or **drv**(x) $\leftarrow \tau$ (that equate variable x or its derivation on time **drv**(x) with the expression τ). All this kind of **hybrid** variables are reals and, for convenience, we call those Quartz programs with hybrid variables *hybrid Quartz programs* in this thesis. An example can be seen in Figure 2.3 below.

```
1  module M2(){
2      // x is local hybrid real variable
3      hybrid real x;
4      // in the discrete variable environment, x is assigned to 0.5
5      x = 0.5;
6      while(true) {
7          w0,w1:flow{
8              // continuous transition starts from 1.0 in the continuous
9              // variable environment
10             x <- 1.0;
11             // x increases the value with derivative x' = 1.0 in the
12             // continuous variable environment
13             drv(x) <- 1.0;
14         // continuous transition will terminate until the value of x
15         // in the continuous environment is greater or equal than 2.0
16         }until(cont(x) >= 2.0);
17         // x will be assigned to 0.0 in the next macro step in the
18         // discrete variable environment
19         next(x) = 0.0;
20         w2: pause;
21     }
22 }
```

Figure 2.3: Hybrid Quartz Module M2

The additional program statements of hybrid Quarz programs are

- **flow**{S_1;...;S_n}**until**(σ) (flow statements)
- x <- τ (continuous assignments)
- **drv**(x) <- τ (derivative assignments)

The flow statement **flow**{S_1;...;S_n}**until**(σ) can replace a **pause** statement and will then extend the discrete transition by a continuous evolution where the continuous variables behave according to the continuous or derivative assignments by S_1;...;S_n.

In contrast to the discrete transitions, continuous evolution requires physical time and terminates as soon as the condition σ becomes true.

To distinguish between the discrete value determined by the discrete immediate or delayed assignments and the continuously changing value during the continuous (time) evolution, a new operator **cont**(x) is introduced: x always refers to the discrete value of a variable determined by the immediate or delayed assignments, whereas **cont**(x) refers to the (changing) value during the continuous evolution, which can be mapped to some continuous variable environment. For **hybrid** variables, **cont**(x) starts with the discrete value x as the initial value for the continuous evolution, if there is no continuous assignment for the variable. For **mem** and **event** variables, the discrete value x and the continuous value **cont**(x) always coincide as these variables do not change during continuous evolution.

Figure 2.4 shows the symbolic simulation graph of hybrid Quartz module M2 from Figure 2.3, where the continuous and derivative assignments are enclosed by diamonds. Module M2 starts from control-flow state s_0 where the immediate assignment x=0.5 that updates the discrete variable environment is directly executed in the first macro step. The continuous evolution starts with **cont**(x)=1.0, the derivation of x is then 1.0 and the continuous evolution terminates as soon as the release condition **cont**(x)>=2.0 holds — afterwards the control-flow moves to s_1. The macro step starting from s_1 contains neither an immediate assignment nor continuous evolution, but a delayed assignment will assign 0.0 to x in the next macro step from control-flow state s_3. For this program, the control-flow will never move to control-flow state s_2. The transitions that will not take place are marked with dashed lines in Figure 2.4.

Consider also the case that the same flow statement runs in parallel with some other flow statement and a macro step terminates before that release condition holds as shown in module M3 from Figure 2.5. The control-flow then can move to a control-flow

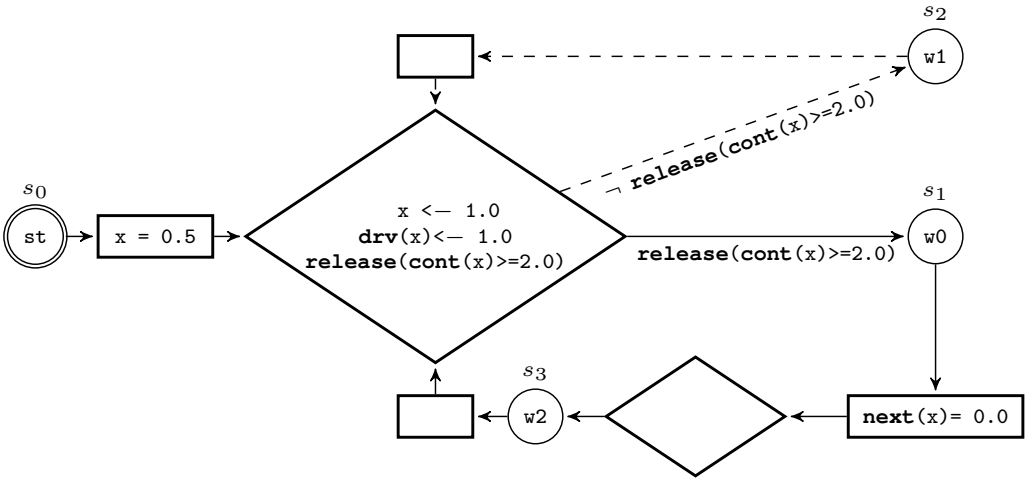

Figure 2.4: Symbolic Simulation Graph of Module M2

```
1  module M3(){
2      // x,y are local hybrid real variable
3      hybrid real x,y;
4      {// this branch will terminate later
5          w0,w1:flow{
6              // x increases the value with derivative x' = 1.0 in the
7              // continuous variable environment
8              drv(x) <- 1.0;
9          // continuous transition will terminate until the value of x
10         // in the continuous environment is greater or equal than 2.0
11         }until(cont(x) >= 2.0);
12     }
13     ||
14     {// this branch will terminate first
15         l0,l1:flow{
16             // y increases the value with derivative y' = 1.0 in the
17             // continuous variable environment
18             drv(y) <- 1.0;
19         // continuous transition will terminate until the value of y
20         // in the continuous environment is greater or equal than 1.0
21         }until(cont(y) >= 1.0);
22     }
23 }
```

Figure 2.5: Hybrid Quartz Module `M3`

state where `w1` is active. For ease of notation, '⊗' in Figure 2.6 stands for those unreachable control-flow states by infeasible transitions. Hybrid variables `x` and `y` increase their values starting from the same initial value `0.0` and by the same derivative `1.0` in the continuous environment. The first macro step ends when the release condition **`cont`**`(y)>=1.0` holds. Meanwhile, the other release condition **`cont`**`(x)>=2.0` is evaluated to false. Therefore, the control-flow moves to the control-flow state s_1, where `w1` holds. From there the program proceeds with a new continuous evolution: **`cont`**`(x)` evolves until the release condition **`cont`**`(x)>=2.0` holds.

2.3 The Averest System

The *Averest* toolset for HW/SW co-design and verification has been developed at the Embedded Systems Chair of the University of Kaiserslautern as a long-term project based on the imperative synchronous Quartz language. The *Averest* system is publicly available at `www.averest.org` for research and teaching purposes. Currently, it offers a constantly evolving infrastructure containing tools for compilation, analysis, as well as for hardware/software synthesis and different formal verification techniques. The design flow used in Averest is shown in Figure 2.7.

In short:

- Embedded reactive systems with or without their physical environments are described by the imperative synchronous language Quartz with its extension to hybrid systems.
- The compiler translates synchronous or hybrid Quartz programs into files in the

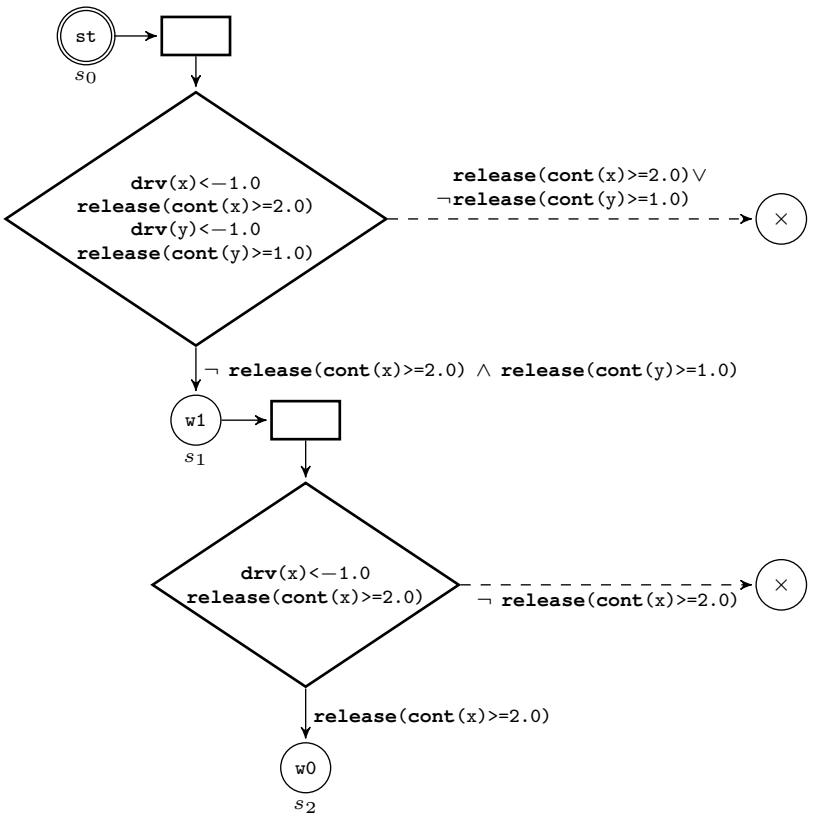

Figure 2.6: Symbolic Simulation Graph of Module `M3`

Averest Interchange Format (AIF), which is given in the form of guarded actions [Sch09; Bau12]. The formal correctness of the translation — which runs in quadratic time over a program $\mathcal{P}$ and will generate also at most quadratically many guarded actions in terms of the size of a program $\mathcal{P}$ — is proven in [Sch00; Sch01; Sch02; SBS06].

- Several different transformation procedures are provided to modify AIF-files for special needs. For example, partition of compound data types like tuples and arrays to scalar types, reduction to Boolean types for hardware synthesis, aggregation of all guarded actions on one variable into a single guarded action (so that equations are generated), dead code elimination, generation of an Extended Finite State Machine (EFSM), etc.
- Several tools for hardware and software synthesis are provided. For example, there are code generators for software synthesis (producing C, Java or SystemC) or hardware synthesis (producing VHDL and Verilog files).
- Formal analysis of imperative synchronous and hybrid Quartz programs can be performed as well. A simulator named Trace and an interactive theorem prover named AIFProver [GS13a] are integrated in the Averest system. Moreover, in-

terfaces [BS11; Bau12; GS13b; LBS13; LS15a] with openModelica[1], the symbolic model checker nuSMV[2], the interactive theorem prover SAL [de +04], KeYmaera [Pla10], and the algebraic computation tool Bonmin [Bon+08] are available for simulation and verification.

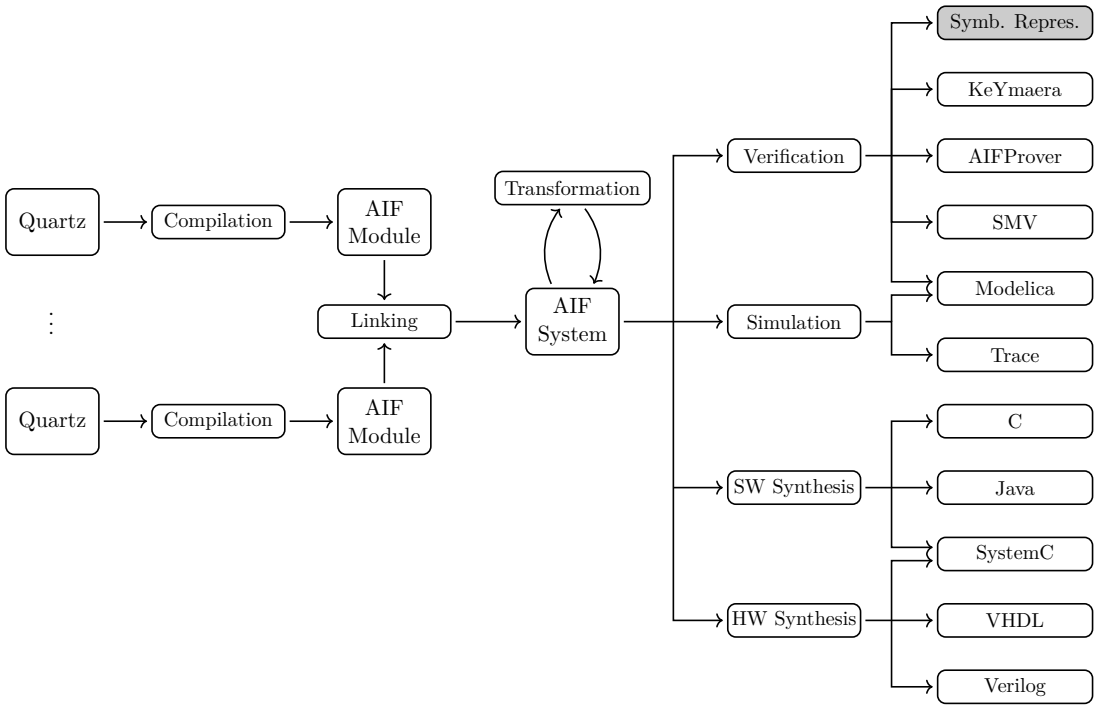

Figure 2.7: The Averest System

The thesis provides additional methods for safety property verification of synchronous and hybrid systems [LBS13; LS14; LS15a; LS15b; LS15c; Rop+16; LS16a; LS16b]. In particular, the imperative synchronous language Quartz and its extension to hybrid systems are used to exemplify the findings. As a preparation for that, the symbolic representations of Quartz programs are the basis for developing induction-based VCG methods and exploring control-flow guided PDR.

2.4 Symbolic Representations of Quartz Programs

The Averest system provides algorithms that translate a Quartz program to a set of guarded actions $\mathcal{G}$ (see [Sch09; Bau12]). Guarded actions are pairs (γ, α) that consist of a trigger condition γ and an action α, and express that α is executed whenever γ holds. Actions are thereby immediate $\mathtt{x} = \tau$ and delayed assignments $\mathbf{next}(\mathtt{x}) = \tau$ (for discrete

[1] `www.openModelica.org`
[2] `http://nusmv.fbk.eu/`

transitions), flow assignments `x` $\leftarrow \tau$ and **`drv(x)`** $\leftarrow \tau$ (for continuous evolutions), assumptions **`assume`**(φ), assertions **`assert`**(φ) and release conditions **`release`**(σ), where γ, τ, φ and σ are program expressions. For each pause statement `w` : **`pause`** with control-flow variable `w`, there is also a guarded action (γ, **`next`**`(w)= true`) such that γ holds whenever the control-flow moves to program location `w` in the next macro step. In this section, we construct EFSMs and state transition systems with those guarded actions as symbolic representations of Quartz programs.

2.4.1 EFSMs

The semantics of synchronous Quartz programs has been formally defined (similar to the semantics of Esterel [Ber99]) by means of Structural Operational Semantics (SOS) rules [Plo81] as given in [Sch09]. Like synchronous Quartz programs, also the extension part to hybrid systems has a precise formal semantics that defines unique behaviors for given input traces [Bau12]. The operational semantics of synchronous and hybrid Quartz programs is defined by two sets of SOS rules: SOS reaction rules define computation of the outputs within a macro step, and SOS transition rules determine the movement of the control-flow from the current to the next macro step. Applying SOS rules, we can directly generate EFSMs for synchronous and hybrid Quartz programs, e.g., Figure 2.9 is the EFSM of module `M4` in Figure 2.8.

The EFSM of a Quartz program has one state for every *reachable* control-flow state of the program, and every state is labeled with a set of guarded actions that encode the dataflow of that state, which means EFSMs show the control-flow information explicitly and keep the dataflow symbolic. Such EFSM has finitely many nodes and transitions, but describes an infinite transition system that has discrete and continuous evolutions occurring in pairs, which is frequently used as a convenient formalism to describe programs with potentially infinite data types. Additionally, EFSMs are useful for deductive verification in that one can decompose the proof goal with respect to the

```
1  module M4(int x,y ){
2    // two loops run in parallel
3    while(!(x > 5)) { // the first while-loop increases the value of x until 5
4      w1:pause;
5      next(x) = x+1;
6    }
7    ||
8    while(!(y > 5)) { // the second while-loop increases the value of y until 5
9      w2:pause;
10     next(y) = y+1;
11   }
12   while(true) { // the third while-loop swaps the values of x and y
13     w3:pause;
14     next(x) = y; next(y) = x;
15   }
16 }
```

Figure 2.8: Synchronous Quartz Module `M4`

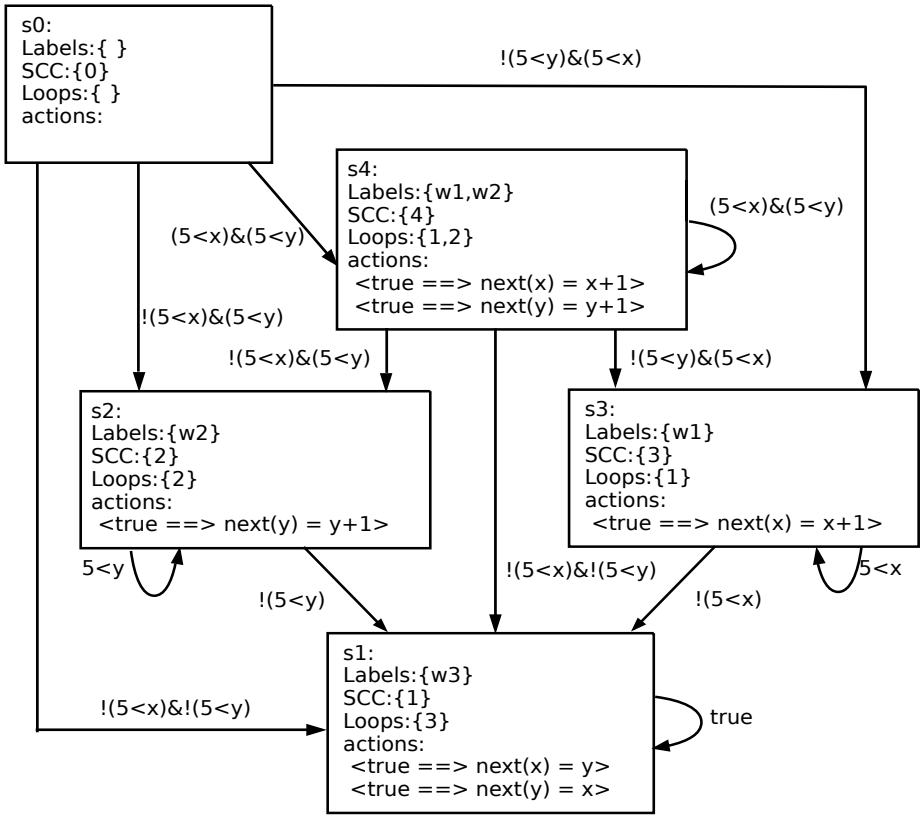

Figure 2.9: EFSM of Module `M4`

reachable control-flow states. As we will show in Chapter 3, five induction-based VCG methods are presented based on the structure of Extended Finite State Machines.

2.4.1.1 The EFSM Structure

Every node s_i in an EFSM is labeled with a set of guarded actions, namely $\mathcal{G}(s_i)$. We can divide $\mathcal{G}(s_i)$ to $\mathcal{G}_d(s_i)$ and $\mathcal{G}_c(s_i)$ for guarded actions that encode discrete transitions and continuous evolutions, respectively. For convenience, we call $\mathcal{G}_d(s_i)$ discrete guarded actions in node s_i, and $\mathcal{G}_c(s_i)$ continuous guarded actions in node s_i. For any pair of nodes (s_i, s_{i+1}), there is moreover a path condition $\varphi(s_i, s_{i+1})$ that must hold to activate the transition from s_i to s_{i+1}. Note that every transition corresponds to one macro step of the program, and EFSMs have infeasible paths.

We call an Strongly Connected Component (SCC) a *trivial* SCC if it contains only a single node without a self-loop transition, while all the other SCCs are called *nontrivial*. By construction of an EFSM, the root node is always a trivial SCC. Recall that the EFSM of a Quartz program has one node for every reachable control-flow state of the program that can be encoded by the control-flow variables. Additional, some control-

flow variables are declared inside loop statements of the Quartz program, so we can also identify the nodes by the loop statements in the program. To sum up, each node s_i in the EFSM of a Quartz program carries the following information:

- $\mathsf{Labels}(s_i)$: Control-flow variables that hold in node s_i.
- $\mathsf{SCC}(s_i)$: SCC index of node s_i.
- $\mathsf{Loops}(s_i)$: Indices of the corresponding loop statements.
- $\mathcal{G}(s_i)$: Guarded actions in node s_i, i.e., $\mathcal{G}_d(s_i) \cup \mathcal{G}_c(s_i)$.

Take node s_4 in Figure 2.9 as an example: It represents the control-flow state where control-flow variables `w0` and `w1` are active; it belongs to SCC $\mathcal{C}_4$ in the EFSM; and its loop indices are $\mathcal{L}_1$ and $\mathcal{L}_2$ which represents the case that neither of the two while-loops run in parallel in module `M4` have terminated yet. $\mathcal{G}_c(s_4)$ is an empty set since the program contains no hybrid variables.

Generally speaking, nodes in an EFSM can be categorized to the following sets:

- $\mathcal{S}_{\mathcal{C}_i}$: Nodes in SCC $\mathcal{C}_i$.
- $\mathcal{S}_{\mathcal{L}_i}$: Nodes whose loop indices contain $\mathcal{L}_i$.
- $\mathcal{S}_{\mathcal{C}_t}$: Nodes in trivial SCCs.
- $\mathcal{S}_{\mathcal{L}_\emptyset}$: Nodes whose loop indices is empty.

Consider the EFSM of module `M4` again: We have $s_4 \in \mathcal{S}_{\mathcal{C}_4}$, $s_4 \in \mathcal{S}_{\mathcal{L}_1}$, $s_4 \in \mathcal{S}_{\mathcal{L}_2}$ and $\mathcal{S}_{\mathcal{L}_\emptyset} = \mathcal{S}_{\mathcal{C}_t} = \{s_0\}$.

2.4.1.2 The EFSM Semantics

EFSM nodes encode program states that are variable environments $\xi : \mathcal{V} \to \mathsf{Val}$ mapping variables to values. To describe the continuous states along continuous evolutions, we consider however variable environments $\chi : \mathcal{V} \to (\mathbb{R} \to \mathsf{Val})$ that map variables to functions over time. A run π through an EFSM is described by an infinite sequence of triples (ξ_i, χ_i, t_i) for $i \in \mathbb{N}$, where

- ξ_i is the discrete variable environment of the i-th macro step (mapping variables to values in the corresponding state).
- χ_i is the continuous variable environment of the i-th macro step (mapping continuous variables `x` and a real time $t \in \mathbb{R}$ to a value $(\chi_i(\mathtt{x}))(t)$).
- t_i is the duration of the continuous evolution of the i-th macro step.

Since discrete states are associated with variable environments $\xi : \mathcal{V} \to \mathsf{Val}$, we define

- $[\![\mathtt{x}]\!]_\xi := \xi(\mathtt{x})$ for variables `x`

For continuous states, we use the following definition with a variable environment $\chi : \mathcal{V} \to (\mathbb{R} \to \mathsf{Val})$ and some $t \in \mathbb{R}$:

- $[\![\mathtt{x}]\!]_{\chi,t} := (\chi(\mathtt{x}), t)$ for variables `x`
- $[\![\mathsf{drv}(\mathtt{x})]\!]_{\chi,t} := ((\chi(\mathtt{x}))', t)$ for variables `x` where $(\chi(\mathtt{x}))'$ is the derivation of $\chi(\mathtt{x})$

To evaluate a transition relation with transitions (s_i, s_{i+1}), we have to consider three variable environments: (1) ξ_i which refers to the starting node s_i, (2) χ_i which describes the continuous evolution in s_i, and (3) ξ_{i+1} which refers to the target node s_{i+1}. We then define $[\![(s_i, s_{i+1})]\!]_{\xi_i,\chi,\xi_{i+1}}$ accordingly, so that whenever it holds, the triple (ξ_i, χ, ξ_{i+1}) describes a discrete transition followed by the cumulated continuous evolution symbolically represented in (s_i, s_{i+1}). To that end, χ_i must map the variables to the solution

of the corresponding Ordinary Differential Equation System (ODE) imposed by the continuous evolution.

Notice that, each node s_i can have multiple continuous evolutions, for the reason that each node can be entered at different (multiple) physical times by the iteration statements. At each physical time t, the program can reach two different states, since each variable has a discrete value $[\![\mathtt{x}]\!]_{\xi_i}$ and a continuous value $[\![\mathtt{x}]\!]_{\chi_i,t}$, where generally $[\![\mathtt{x}]\!]_{\xi_i} \neq [\![\mathtt{x}]\!]_{\chi_i,t}$. For convenience, we call state $[\![\mathtt{x}]\!]_{\xi_i}$ the discrete state at physical time t in node s_i, and $[\![\mathtt{x}]\!]_{\chi_i,t}$ the continuous state at physical time t in node s_i. If t is a terminal physical time of a continuous evolution in node s_i, which is also some starting physical time of a continuous evolution in node s_j, then each variable can have at most three different values at t by the delayed assignment, i.e., $[\![\mathtt{x}]\!]_{\xi_i} \neq [\![\mathtt{x}]\!]_{\chi_i,t} \neq [\![\mathtt{x}]\!]_{\xi_j}$. To distinguish state $[\![\mathtt{x}]\!]_{\xi_j}$ from the other two, we call it the discrete state at physical time t in node s_j. Obviously, each node represents multiple discrete and continuous states at different physical time. In sum, each program state encoded in a node of the corresponding EFSM is identified by the following three aspects:

- Which node the state belongs to.
- The state is a discrete or continuous state.
- What is the physical time of the state.

2.4.2 Symbolic Transition Systems

EFSMs are a symbolic representation for Quartz programs that decompose the entire guarded actions to nodes. The other way to construct a symbolic representation in form of guarded actions is by evaluating the changes of each program variable.

2.4.2.1 Abbreviations for Transition Relations

In every macro step, we have to evaluate all guards γ and fire all actions α whose guards are enabled. For ease of notation, assumptions and assertions are not considered in this section. If $\mathtt{x}$ is a variable of some Quartz program, then the guarded actions for the variable $\mathtt{x}$ are given by:

$$\begin{array}{llll} (\alpha_1, \mathtt{x} = \tau_1) & , \ldots , & (\alpha_p, \mathtt{x} = \tau_p) \\ (\beta_1, \mathbf{next}(\mathtt{x}) = \pi_1) & , \ldots , & (\beta_q, \mathbf{next}(\mathtt{x}) = \pi_q) \\ (\gamma_1, \mathtt{x} \mathrel{<\!\!-} \tau_1') & , \ldots , & (\gamma_r, \mathtt{x} \mathrel{<\!\!-} \tau_r') \\ (\delta_1, \mathbf{drv}(\mathtt{x}) \mathrel{<\!\!-} \pi_1') & , \ldots , & (\delta_s, \mathbf{drv}(\mathtt{x}) \mathrel{<\!\!-} \pi_s') \end{array}$$

If $\mathtt{x}$ is not a hybrid variable, then neither continuous nor derivative assignments exist for $\mathtt{x}$. Furthermore, the program has the set of guarded release conditions for flow statements:

$$(\varepsilon_1, \mathbf{release}(\sigma_1)) \quad , \ldots , \quad (\varepsilon_k, \mathbf{release}(\sigma_k))$$

Figure 2.10 shows then the definition of the initial states predicate $\mathsf{Init}_{\mathtt{x}}$ and the transition relation $\mathsf{Trans}_{\mathtt{x}}$ for variable $\mathtt{x}$, where we use abbreviations $\Gamma_1 := \bigvee_{i=1}^{p} \alpha_i$, $\Delta_1 := \bigvee_{i=1}^{q} \beta_i$, $\Gamma_2 := \bigvee_{i=1}^{r} \gamma_i$, and $\Delta_2 := \bigvee_{i=1}^{s} \delta_i$. The initial states predicate $\mathsf{Init}_{\mathtt{x}}$ can be explained as follows:

$$
\mathsf{Init}_{\mathtt{x}} \quad := \quad \left(\begin{array}{ll} & \bigwedge_{i=1}^{p} (\alpha_i \rightarrow \mathtt{x} = \tau_i) \\ \wedge & (\neg\Gamma_1 \rightarrow \mathtt{x} = \mathsf{Default}(x)) \\ \wedge & \bigwedge_{i=1}^{r} (\gamma_i \rightarrow \mathbf{cont}(\mathtt{x}) = \tau_i') \\ \wedge & (\neg\Gamma_2 \rightarrow \mathbf{cont}(\mathtt{x}) = \mathtt{x}) \end{array} \right)
$$

$$
\mathsf{Trans}_{\mathtt{x}} \quad := \quad \left(\begin{array}{ll} & \bigwedge_{i=1}^{p} (\alpha_i \rightarrow \mathtt{x} = \tau_i) \\ \wedge & \bigwedge_{i=1}^{q} (\beta_i \rightarrow \mathbf{next}(\mathtt{x}) = \pi_i) \\ \wedge & (\mathbf{next}(\neg\Gamma_1) \wedge \neg\Delta_1 \rightarrow \mathbf{next}(\mathtt{x}) = \mathsf{Abs}(\mathtt{x})) \\ \wedge & cont_{\forall}(\bigwedge_{i=1}^{r} (\gamma_i \rightarrow \mathbf{cont}(\mathtt{x}) = \tau_i')) \\ \wedge & cont_{\forall}(\bigwedge_{i=1}^{s} (\delta_i \rightarrow \mathbf{drv}(\mathtt{x}) = \pi_i')) \\ \wedge & cont_{\forall}(\neg\Delta_2 \rightarrow \mathbf{drv}(\mathtt{x}) = 0.0) \\ \wedge & cont_{\forall}(\bigwedge_{i=1}^{k} (\varepsilon_i \rightarrow \neg\sigma_i)) \end{array} \right)
$$

$$
\mathsf{Abs}(\mathtt{x}) \quad := \quad \begin{cases} \mathtt{x}, & \text{if } \mathtt{x} \text{ is a memorized variable} \\ \mathbf{cont}(\mathtt{x}), & \text{if } \mathtt{x} \text{ is a hybrid variable} \\ \mathsf{Default}(\mathtt{x}), & \text{otherwise} \end{cases}
$$

Figure 2.10: Transition Relation for Variable x

- The initial value of `x` can only be determined by its immediate actions. Hence, if one of the guards α_i of the immediate actions holds, the corresponding immediate assignment defines the value of `x`. If none of the guards α_i should hold, i.e., $\Gamma_1 := \bigvee_{i=1}^{p} \alpha_i$ is false, the initial value of `x` is determined by its default value (which is determined by the semantics, e.g., false for boolean variables and 0 for numeric ones).
- if `x` is a hybrid variable, then the initial value of **`cont(x)`** can only be determined by its continuous assignments. Therefore, if one of the guards γ_i of the continues assignments holds, the corresponding continuous assignment defines the value of **`cont(x)`**. If none of the guards α_i should hold, i.e., $\Gamma_2 := \bigvee_{i=1}^{r} \gamma_i$ is false, the initial value of **`cont(x)`** is equal to `x`.

The transition relation $\mathsf{Trans}_{\mathtt{x}}$ can be explained similarly:

- The immediate assignments have to be respected for the current point of time, i.e., whenever a guard α_i of the immediate actions holds, the corresponding immediate assignment defines the current value of `x`. If one of the guards β_i of the delayed assignments holds at the current point of time, the next value of `x` is determined by the corresponding delayed assignment. Finally, if the next value of `x` is not determined by an action, i.e., neither $\Gamma_1 := \bigvee_{i=1}^{p} \alpha_i$ holds at next point of time nor does $\Delta_1 := \bigvee_{i=1}^{q} \beta_i$ hold at the current point of time, then the next value of `x` is determined by the reaction to absence. Depending on whether it is an event, a memorized or hybrid variable, it may be reset to a default value, or store its previous value, i.e., either `x` or **`cont(x)`**.
- For hybrid variable `x`, we use predicate $cont_{\forall}(\phi)$, which demands that ϕ holds at all points in time during a continuous evolution. The continuous assignments of `x` have to be respected for the current point of time, i.e., whenever a guard γ_i of the continuous assignment holds, the corresponding continuous assignment defines the initial value of **`cont(x)`** for a continuous evolution. If one of the guards δ_i of the derivative assignments holds, then the value of **`cont(x)`** evolves accordingly. Otherwise, i.e., $\Delta_2 := \bigvee_{i=1}^{s} \delta_i$ does not hold, the derivation of **`cont(x)`** is 0.0 which means **`cont(x)`** keeps unchanged. In addition, the release conditions need to be considered in order to determine the end of a continuous evolution. To ease of notation, we use **`drv(x)`** instead of **`drv(cont(x))`**.

Notice that the abbreviations given in Figure 2.10 can be literally used to define input files for symbolic model checkers.

2.4.2.2 State Transition Systems

Imperative synchronous Quartz programs have a clearly distinct control- and dataflow. If $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{T})$ is the transition system of a Quartz program $\mathcal{P}$, we can then explicitly differentiate its control- and dataflow by defining two other transition systems $\mathcal{K}^{\mathsf{cf}} = (\mathcal{V}, \mathcal{I}^{\mathsf{cf}}, \mathcal{T}^{\mathsf{cf}})$ and $\mathcal{K}^{\mathsf{df}} = (\mathcal{V}, \mathcal{I}^{\mathsf{df}}, \mathcal{T}^{\mathsf{df}})$ as follows:

- $\mathcal{I}^{\mathsf{cf}} := \bigwedge_{\mathtt{w} \in \mathcal{V}^{\mathsf{cf}}} \mathsf{Init}_{\mathtt{w}}$ and $\mathcal{T}^{\mathsf{cf}} := \bigwedge_{\mathtt{w} \in \mathcal{V}^{\mathsf{cf}}} \mathsf{Trans}_{\mathtt{w}}$
- $\mathcal{I}^{\mathsf{df}} := \bigwedge_{\mathtt{x} \in \mathcal{V}^{\mathsf{df}}} \mathsf{Init}_{\mathtt{x}}$ and $\mathcal{T}^{\mathsf{df}} := \bigwedge_{\mathtt{x} \in \mathcal{V}^{\mathsf{df}}} \mathsf{Trans}_{\mathtt{x}}$

- $\mathcal{I} := \mathcal{I}^{\mathsf{cf}} \wedge \mathcal{I}^{\mathsf{df}}$ and $\mathcal{T} := \mathcal{T}^{\mathsf{cf}} \wedge \mathcal{T}^{\mathsf{df}}$

where $\mathcal{V}^{\mathsf{cf}}$ denotes all names of **pause** locations (i.e., control-flow variables) and $\mathcal{V}^{\mathsf{df}}$ denotes all dataflow (hybrid and non-hybrid) variables.

Note that the transition systems $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{T})$, $\mathcal{K}^{\mathsf{cf}} = (\mathcal{V}, \mathcal{I}^{\mathsf{cf}}, \mathcal{T}^{\mathsf{cf}})$ and $\mathcal{K}^{\mathsf{df}} = (\mathcal{V}, \mathcal{I}^{\mathsf{df}}, \mathcal{T}^{\mathsf{df}})$, are defined over the same states which are subsets of $\mathcal{V} := \mathcal{V}^{\mathsf{cf}} \cup \mathcal{V}^{\mathsf{df}}$, but have different transitions and initial states. For example, we can define $\mathcal{K}_{\mathtt{M1}} = (\mathcal{V}_{\mathtt{M1}}, \mathcal{I}_{\mathtt{M1}}, \mathcal{T}_{\mathtt{M1}})$, i.e., the state transition system of Quartz module `M1`, as follows:

- $\mathcal{V}_{\mathtt{M1}} := \underbrace{\{\mathtt{run}, \mathtt{w0}, \mathtt{w1}\}}_{\mathcal{V}^{\mathsf{cf}}} \cup \underbrace{\{\mathtt{x}\}}_{\mathcal{V}^{\mathsf{df}}}$, $\mathcal{I}_{\mathtt{M1}} := \mathcal{I}^{\mathsf{cf}}_{\mathtt{M1}} \wedge \mathcal{I}^{\mathsf{df}}_{\mathtt{M1}}$ and $\mathcal{T}_{\mathtt{M1}} := \mathcal{T}^{\mathsf{cf}}_{\mathtt{M1}} \wedge \mathcal{T}^{\mathsf{df}}_{\mathtt{M1}}$
- $\mathcal{I}^{\mathsf{cf}}_{\mathtt{M1}} := \neg\mathtt{run} \wedge \neg\mathtt{w0} \wedge \neg\mathtt{w1}$
- $\mathcal{T}^{\mathsf{cf}}_{\mathtt{M1}} := \begin{pmatrix} & (\mathbf{next}(\mathtt{run}) \leftrightarrow \mathsf{true}) \\ \wedge & (\mathbf{next}(\mathtt{w0}) \leftrightarrow (\neg\mathtt{run} \vee \mathtt{w1})) \\ \wedge & (\mathbf{next}(\mathtt{w1}) \leftrightarrow \mathtt{w0}) \end{pmatrix}$
- $\mathcal{I}^{\mathsf{df}}_{\mathtt{M1}} := (\mathtt{x} = 0)$
- $\mathcal{T}^{\mathsf{df}}_{\mathtt{M1}} := ((\mathtt{x} = 1.0) \leftrightarrow \mathtt{w0}) \wedge ((\mathbf{next}(\mathtt{x}) = \mathtt{x} - 1.0) \leftrightarrow \mathtt{w0})$

For hybrid Quartz module `M2`, $\mathcal{K}_{\mathtt{M2}} = (\mathcal{V}_{\mathtt{M2}}, \mathcal{I}_{\mathtt{M2}}, \mathcal{T}_{\mathtt{M2}})$ is the following:

- $\mathcal{V}_{\mathtt{M2}} := \underbrace{\{\mathtt{run}, \mathtt{w0}, \mathtt{w1}, \mathtt{w2}\}}_{\mathcal{V}^{\mathsf{cf}}} \cup \underbrace{\{\mathtt{x}\}}_{\mathcal{V}^{\mathsf{df}}}$, $\mathcal{I}_{\mathtt{M2}} := \mathcal{I}^{\mathsf{cf}}_{\mathtt{M2}} \wedge \mathcal{I}^{\mathsf{df}}_{\mathtt{M2}}$ and $\mathcal{T}_{\mathtt{M2}} := \mathcal{T}^{\mathsf{cf}}_{\mathtt{M2}} \wedge \mathcal{T}^{\mathsf{df}}_{\mathtt{M2}}$
- $\mathcal{I}^{\mathsf{cf}}_{\mathtt{M2}} := \neg\mathtt{run} \wedge \neg\mathtt{w0} \wedge \neg\mathtt{w1} \wedge \neg\mathtt{w2}$
- $\mathcal{T}^{\mathsf{cf}}_{\mathtt{M2}} := \begin{pmatrix} & (\mathbf{next}(\mathtt{run}) \leftrightarrow \mathsf{true}) \\ \wedge & (\mathbf{next}(\mathtt{w0}) \leftrightarrow (\neg\mathtt{run} \wedge (\mathbf{cont}(\mathtt{x}) \geq 2.0))) \\ \wedge & (\mathbf{next}(\mathtt{w1}) \leftrightarrow ((\neg\mathtt{run} \vee \mathtt{w1} \vee \mathtt{w2}) \wedge \neg(\mathbf{cont}(\mathtt{x}) \geq 2.0))) \\ \wedge & (\mathbf{next}(\mathtt{w2}) \leftrightarrow \mathtt{w0}) \end{pmatrix}$
- $\mathcal{I}^{\mathsf{df}}_{\mathtt{M2}} := (\mathtt{x} = 0.5)$
- $\mathcal{T}^{\mathsf{df}}_{\mathtt{M2}} := \begin{pmatrix} & ((\mathbf{drv}(\mathtt{x}) = 1.0) \leftrightarrow (\neg\mathtt{run} \vee \mathtt{w1} \vee \mathtt{w2})) \\ \wedge & ((\mathbf{cont}(\mathtt{x}) = 1.0) \leftrightarrow (\neg\mathtt{run} \vee \mathtt{w1} \vee \mathtt{w2})) \\ \wedge & ((\mathbf{next}(\mathtt{x}) = 0.0) \leftrightarrow \mathtt{w0}) \\ \wedge & ((\neg\mathtt{run} \vee \mathtt{w1} \vee \mathtt{w2}) \rightarrow \neg(\mathbf{cont}(\mathtt{x}) \geq 2.0)) \end{pmatrix}$

In general, the control-flow information is not needed for synthesis and is therefore not explicitly encoded in the generated systems, but it can be easily derived from the original imperative synchronous programs and used for verification. As we will show in Chapter 4, the PDR method benefits from the control-flow invariants that contain information about the reachable control-flow states.

2.5 Safety Property Verification

The synchronous and hybrid systems discussed in this thesis are specified by symbolic representations supporting standard data types (real, integer and boolean) and non-linear dynamics. Safety property verification of these formal models calls for an effective method to reason about boolean combinations of propositional logic atoms and atoms of non-linear arithmetic theories over integers and reals with quantifiers. In the following, the underlying satisfiability problem is defined formally.

2.5.1 The Satisfiability Problem

2.5.1.1 Syntax

We assume a finite set of real, integer and boolean variables $\mathcal{V} = \mathcal{V}_{\mathbb{R}} \cup \mathcal{V}_{\mathbb{Z}} \cup \mathcal{V}_{\mathbb{B}}$, and define boolean expressions by the grammar:

$$e_b \;:=\; x \in \mathcal{V}_{\mathbb{B}} \mid \neg e_b \mid e_b \wedge e_b \mid e_b \vee e_b \tag{2.1}$$

and, respectively, numerical expressions by the grammar:

$$e \;:=\; x \in \mathcal{V}_{\mathbb{R}} \cup \mathcal{V}_{\mathbb{Z}} \mid e + e \mid e - e \mid e \cdot e \mid e/e \mid uop_1(e) \mid uop_2(e)$$

where uop_1 are transcendental unary operation symbols, like sin, cos, etc., while uop_2 are exponential and logarithmic operation symbols, like log, exp etc..

Given numerical expressions e and e', $e \odot e'$ defines a *constraint* for any $\odot \in \{\leq, =\}$. The other relational operators can be obtained by using negation and conjunction.

$$c \;:=\; e \odot e' \mid \neg c \mid c \wedge c \tag{2.2}$$

Let $X_{\mathbb{B}} \subseteq \mathcal{V}_{\mathbb{B}}$ be the set of quantified boolean variables, the following formula defines a *boolean expression with quantifiers.*

$$(e_b)_Q \;:=\; (\exists X_{\mathbb{B}}).\ e_b \mid (\forall X_{\mathbb{B}}).\ e_b. \tag{2.3}$$

Similarly, given the quantifier sets $X_{\mathbb{R}} \subseteq \mathcal{V}_{\mathbb{R}}$ and $X_{\mathbb{Z}} \subseteq \mathcal{V}_{\mathbb{Z}}$, a *constraint with quantifier* is defined as follows.

$$c_Q \;:=\; (\exists X_{\mathbb{R}}, X_{\mathbb{Z}}).\ c \mid (\forall X_{\mathbb{R}}, X_{\mathbb{Z}}).\ c \tag{2.4}$$

The satisfiability problem is then syntactically defined as the boolean combination of finitely many sub-formulas:

$$c_b := c_{b_0} \circledcirc c_{b_1} \circledcirc \cdots \circledcirc c_{b_{n-1}} \circledcirc c_{b_n} \tag{2.5}$$

where $\circledcirc \in \{\wedge, \vee\}$, $n \in \mathbb{N}$, and each sub-formula c_{b_i} with $0 \leq i \leq n$ is either a constraint with quantifier or a boolean expression with quantifier.

Constraints and boolean expressions without quantifiers are no longer considered by formula (2.5). It is reasonable to do that since the constraints and boolean expressions without quantifiers could be represented by those with quantifiers.

Throughout the rest of this section, we assume that the constraints on variables that appear in (2.1-2.5) are given, denoted as c_v. To have some idea on how the formulas look like, we assume that $\mathcal{V}_{\mathbb{R}} = \{r_1, r_2\}$, $\mathcal{V}_{\mathbb{Z}} = \{z_1, z_2\}$, $\mathcal{V}_{\mathbb{B}} = \{b_1, b_2\}$ and list some examples in Table 2.1.

Table 2.1: Example Formulas

Category	Formula
e_b	$E_0 : b_1 \wedge b_2$
c	$E_1 : z_1 \leq z_2$ $E_2 : z_1 + r_1 \leq z_2 + r_2$ $E_3 : r_1 \leq r_2 * r_1 + r_2/r_1$ $E_4 : r_1 \leq r_2 + z_1 * (r_1 - z_2)$ $E_5 : z_1/r_1 \leq z_2 * cos(r_2)$ $E_6 : r_1 \leq log(r_1)/cos(r_2)$
$(e_b)_Q$	$E_7 : (\forall \mathcal{V}_{\mathbb{B}}).\ b_1 \wedge b_2$
c_Q	$E_8 : (\exists \mathcal{V}_{\mathbb{R}}, \mathcal{V}_{\mathbb{Z}}).\ E_5$
c_b	$E_9 : E_7 \vee E_8$ $E_{10} : ((\exists \{r_1\}).\ r_1 \geq 1) \wedge ((\forall \{r_1\}).\ r_1 \leq 1)$
c_v	$E_{11} : (-2.0 \leq r_1 * r_2 \leq 10.0) \wedge (z_1 \leq z_2 \leq 5)$ $E_{12} : 0 \leq r_1$

2.5.1.2 Semantics

The semantics of a formula (2.1) or a formula (2.2) is defined by the interpretation $[\![\cdot]\!]_v$, which evaluates the inside $\cdot$ to *True* or *False* using the variable valuation function v.

Definition 1 (Variable Valuation)**.** *A variable valuation is a function $v : \mathcal{V} \to \mathbb{R} \cup \mathbb{Z} \cup \mathbb{B}$ that assigns to each variable a real, integer or boolean value. If e is a numerical expression then $e[v]$ replaces occurrences of its variables by their v-image. We use similar $e_b[v]$ and $c[v]$ notation in the case of boolean expression e_b and constraint c, respectively.*

Definition 2 (Satisfaction relation)**.** *For formula f, which is either a boolean expression e_b or constraint c, a variable valuation v satisfies formula f, denoted $v \models f$, if $[\![f]\!]_v$ evaluates to* True*.*

For boolean expressions $x \in \mathcal{V}_{\mathbb{B}}$, e_b, e_{b_1} and e_{b_2}, we have:

- $[\![x]\!]_v := x[v]$, *for* $x \in \mathcal{V}_{\mathbb{B}}$
- $[\![\neg e_b]\!]_v := \begin{cases} \text{True}, & \text{if } [\![e_b]\!]_v = \text{False} \\ \text{False}, & \text{if } [\![e_b]\!]_v = \text{True} \end{cases}$
- $[\![e_{b_1} \wedge e_{b_2}]\!]_v := [\![e_{b_1}]\!]_v \wedge [\![e_{b_2}]\!]_v$
- $[\![e_{b_1} \vee e_{b_2}]\!]_v := [\![e_{b_1}]\!]_v \vee [\![e_{b_2}]\!]_v$

Similarly, for constraints $e_1 \odot e_2$, c, c_1 and c_2, where both e_1 and e_2 are numerical expressions, we have:

- $[\![e_1 \odot e_2]\!]_v := e_1[v] \odot e_2[v]$
- $[\![\neg c]\!]_v := \begin{cases} \text{True}, & \textit{if } [\![c]\!]_v = \text{False} \\ \text{False}, & \textit{if } [\![c]\!]_v = \text{True} \end{cases}$
- $[\![c_1 \wedge c_2]\!]_v := [\![c_1]\!]_v \wedge [\![c_2]\!]_v$

Take formulas E_2 and E_{11} in Table 2.1 as an example. If some variable valuation v_1 maps r_1, r_2, z_1 and z_2 to 0.0, 1.0, 1 and 2, respectively, then $v_1 \models E_2$ and $v_1 \models E_{11}$.

Based on the above relation, the semantics of formulas (2.3)–(2.4) is defined by the interpretation $[\![\cdot]\!]$, which evaluates the inside formula $\cdot$ to *True* or *False*. Thus, we have the following:

- $[\![(\exists X_{\mathbb{B}}).\ e_b]\!] := \begin{cases} \textit{True}, & \text{if } \exists\ v : \mathcal{V} \to \mathbb{B}.\ v \models e_b \\ \textit{False}, & \text{Otherwise} \end{cases}$
- $[\![(\forall X_{\mathbb{B}}).\ e_b]\!] := \begin{cases} \textit{False}, & \text{if } \exists\ v : \mathcal{V} \to \mathbb{B}.\ v \models \neg e_b \\ \textit{True}, & \text{Otherwise} \end{cases}$
- $[\![(\exists X_{\mathbb{R}}, X_{\mathbb{Z}}).\ c]\!] := \begin{cases} \textit{True}, & \text{if } \exists\ v : \mathcal{V} \to \mathbb{R} \cup \mathbb{Z}.\ v \models c \\ \textit{False}, & \text{Otherwise} \end{cases}$
- $[\![(\forall X_{\mathbb{R}}, X_{\mathbb{Z}}).\ c]\!] := \begin{cases} \textit{False}, & \text{if } \exists\ v : \mathcal{V} \to \mathbb{R} \cup \mathbb{Z}.\ v \models \neg c \\ \textit{True}, & \text{Otherwise} \end{cases}$

For convenience of analysis and explanation, formula (2.5) could be reorganized as the following disjunctive form:

$$C_b := \bigvee_{i \in \mathbb{N}} ((C_Q)_i \wedge (E_Q)_i) \tag{2.6}$$

where each $(C_Q)_i$ is a conjunction of finitely many constraints with quantifiers, and each $(E_Q)_i$ is a conjunction of finitely many boolean expressions with quantifiers.

We use formula (2.6) to display how the interpretation $[\![\cdot]\!]$ works for conjunction and disjunction of formulas. Concretely, since

$$[\![\bigvee_{i \in \mathbb{N}} ((C_Q)_i \wedge (E_Q)_i)]\!] := \bigvee_{i \in \mathbb{N}} ([\![(C_Q)_i]\!] \wedge [\![(E_Q)_i]\!])$$

formula (2.6) is interpreted by $[\![\cdot]\!]$ as follows:

$$[\![C_b]\!] := \begin{cases} \textit{True}, & \text{if } \exists\ i \in \mathbb{N}\ .\ [\![(C_Q)_i]\!] \wedge [\![(E_Q)_i]\!] = \textit{True} \\ \textit{False}, & \text{Otherwise} \end{cases}$$

Consider the formula E_{10} in Table 2.1. Even though $r_1 = 2$ is a candidate valuation that satisfies $r_1 \geq 1$, it violates $r_1 \leq 1$, therefore, $[\![E_{10}]\!]$ is *False*.

2.5.2 Decidability and Tools

We discuss decidability of the satisfiability problem specified by formula (2.6). The relations between subclass problems are illustrated, so that we could classify the available techniques and tools to different categories corresponding to the type of satisfiability problems they could solve.

2.5.2.1 The Logics Perspective

As shown in Figure 2.11, the outmost yellow circle includes the following three subset problems in the category of first-order logic that are decidable, and that admit quantifier-elimination.

- $(\mathbb{N}, +, \leq, 0, 1)$: Presburger arithmetic [Pre30] for natural numbers with addition, e.g., formula E_1 in Table 2.1.
- $(\mathbb{R}, +, \leq, \mathbb{Z})$: First-order logic with real and integer addition, e.g., formula E_2 in Table 2.1.
- $(\mathbb{R}, +, \cdot, 0, 1, <)$: First-order theory of real-closed fields [Tar36], e.g., formula E_3 in Table 2.1.

If we add a predicate for the integers to form the structure ($\mathbb{R}$, $+$, $\cdot$, $\mathbb{Z}$, 0, 1, $<$), then the theory is no longer decidable. Formula E_4 is a typical example for this logic. For the same reason, the theory of ($\mathbb{R}$, $\mathbb{Z}$, $+$, $\cdot$, $uop_1(x)$, $uop_2(x)$, 0, 1, $<$) — where uop_1, uop_2 stand for the real-valued special function symbols introduced in Section 2.5.1.1 — is not decidable either. Formulas E_5 and E_8 are examples for this logic. Also, its fragment structure ($\mathbb{R}$, $+$, $\cdot$, $uop_1(x)$, $uop_2(x)$, 0, 1, $<$) is not decidable: E_6 is an example for this subclass logic.

The satisfiability problem defined by formula (2.6) consists of boolean combinations

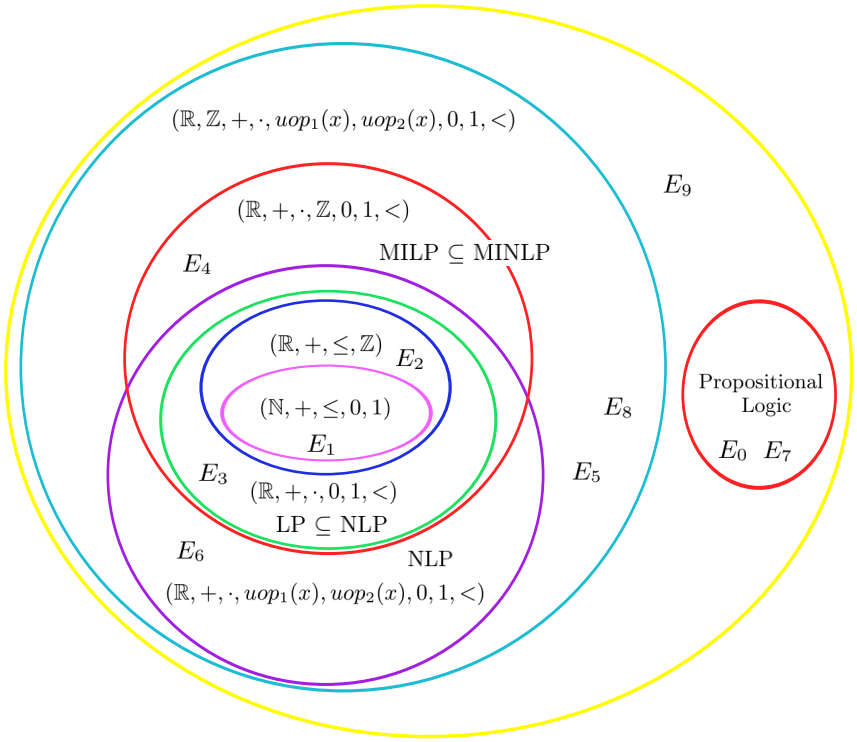

Figure 2.11: The Relations of Logics

of propositional logic atoms and atoms of non-linear arithmetic theories over integers and reals with quantifiers, but quantifier alternations are not allowed. Therefore, the satisfiability problem described by formula (2.6) is undecidable and strictly included in the outmost yellow circle.

Most of the SMT Solvers and theorem provers support decidable theories. However, when modeling and analyzing systems, the increasing use of non-linear arithmetic over reals and integers motivates an extension of SMT problems to non-linear theories, which correspond to subsets of the unsatisfiable satisfiability problem defined by formula (2.6).

2.5.2.2 The Constraint Problem Perspective

From the view of constraint problems [BHZ06], a standard form formula that describes a Mix-Integer Non-Linear Program (MINLP) problem, denoted as ${c_Q}'$, is defined as follows:

$$ {c_Q}' \;:=\; (\exists X_{\mathbb{R}}, X_{\mathbb{Z}}).\; c' \quad \text{where} \quad c' \;:=\; e \odot e' \mid c' \wedge c' \tag{2.7} $$

Compared to formula (2.4), $\forall$-quantifiers are not allowed. Notice also that c' restricts formula (2.2) to only $\odot$ and $\wedge$ operators. In this way, ${c_Q}'$ represents an instance of a MINLP problem.

If we get rid of the boolean variables and the $\forall$-quantifier, then the remaining subset problem defined by formula (2.6) can be reorganized as follows:

$$ {C_b}' := \bigvee_{i \in \mathbb{N}} {c_Q}_i' \tag{2.8} $$

where each ${c_Q}_i'$ is defined by formula (2.7).

Solving the satisfiability problem for the above formula (2.8) amounts to checking whether there exists a solution for a set of MINLP problems, where each MINLP problem corresponds to a sub formula ${c_Q}_i'$. The solution satisfies at least one MINLP problem in the set. Therefore, we need effective techniques to check the existence of this solution for the set of MINLP problems generated by ${C_b}'$.

The MINLP problem is one of the most general modeling paradigms in optimization and includes both Non-Linear Program (NLP) and Mix-Integer Linear Program (MILP) as subproblems. Linear constraint problems are a subset of NLP problems, while, MILP problems are a subset of MINLP problems.

Relations between constraint satisfaction problems and first-order logics are illustrated in Figure 2.11. Linear constraint problems can be described by the first-order theory of real-closed fields (the region encircled by green). NLP problems fall into both the green and purple encircled regions that stand for the first-order theory of real-closed fields including real-valued special function symbols. The theory of the structure ($\mathbb{R}$, $+$, $\cdot$, $\mathbb{Z}$, 0, 1, $<$) (the region encircled by red) is strong enough to specify the MILP problems. Both the theory of the structure ($\mathbb{R}$, $+$, $\cdot$, $\mathbb{Z}$, 0, 1, $<$) and the theory of the structure ($\mathbb{R}$, $\mathbb{Z}$, $+$, $\cdot$, $uop_1(x)$, $uop_2(x)$, 0, 1, $<$) (the region encircled by blue) can yield MINLP problems, which, as mentioned before, can be specified by formula (2.4).

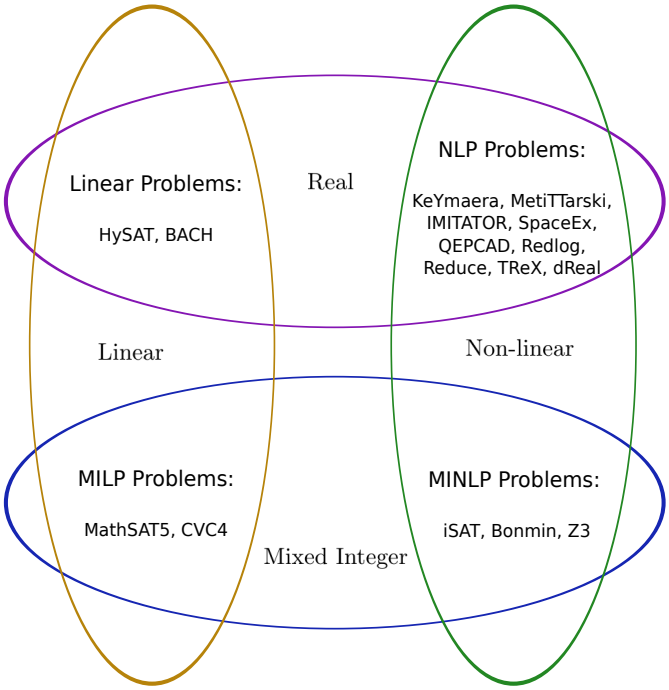

Figure 2.12: The Relations of Constraint Problems

When classifying the available tools and techniques for synchronous program and hybrid system analysis, according to the constraint problems they could solve, as shown in Figure 2.12, we have the following results:

- Linear constraint problems:
  HySAT [FH07] and BACH [Bu+08] are designed for linear arithmetic over reals.
- MILP problems:
  MathSAT5 [Cim+13], CVC4 [BDS02; Det+14] and Z3 [MB08] contain decision procedures for mixed integer linear arithmetic.
- NLP problems:
  KeYmaera and MetiTarski [Pau12] accept formulas over reals that are non-linear. Originally based on the inverse method [And+09], IMITATOR [And09] is a tool for efficient synthesis for Timed Automata [AD94]. In [FK11], the methodology has been extended for LHAs and AHAs. SpaceEx [Fre+11] facilitates quite a lot of algorithms related to reachability and safety verification in the theory of reals. iSAT [EFH08] could solve large boolean combinations of non-linear arithmetic constraints involving transcendental functions. Z3 also posseses partial support for non-linear arithmetic: It contains quantifier elimination procedures for linear arithmetic over the reals and integers.
- MINLP problems: iSAT [EFH08] could solve large boolean combinations of non-linear arithmetic constraints involving transcendental functions. According to our

experiments, Z3 also presents partial support of mixed-integer linear arithmetic. Some algebraic computation tools developed in different application areas attracted our attention as well:

- NLP problems: QEPCAD [Bro03] can produce quantifier-free equivalent formulas for Tarski formulas. Reduce [Hea84] can handle integer and real arithmetic. Its package Redlog [DS96] has some quantifier elimination procedures to solve parameterized non-linear real arithmetic problems. Both of them are widely used as external decision procedures for hybrid system verification tools, like KeYmaera. TReX [ABS01], a tool for automatic analysis of automata-based models, relies on Reduce to solve the constraint problems in the theory of non-linear arithmetic over reals.
- MINLP problems: The survey [Bel+13] introduces a range of approaches to tackle MINLP problems. Bonmin [Bon+08] is one of the open source tools mentioned in it. Bonmin contains a combination of techniques that lends itself to be a good candidate for convex MINLP problems.

In general, tools for MINLP problems try to find a solution for a single MINLP problem each time. However, they cannot always find a solution. This might be due to the limitations of the tool itself or due to the fact that satisfiability of MINLP problems is undecidable. No solution returned does not mean no solution exists.

Solving the satisfiability problem requires to map the formula to a logical value (*True* or *False*). It is beyond the ability of the tools to solve the class of formulas defined by formula (2.8). This is the case since assigning a logical value is strongly related to whether a tool can find a solution for each individual MINLP problem or not. Taking $C = c_{Q_1} \vee c_{Q_2}$ as an example, if no solution is found for the MINLP problems that correspond to c_{Q_1} and c_{Q_2} respectively, then it is unclear what logical value should $[\![C]\!]_v$ be assigned to. Moreover, SAT/SMT solvers and the algebraic computation tools are developed with different application backgrounds. It is still unclear which tool performs best to solve the satisfiability problem that we consider.

Based on the above analysis, we conclude that currently available tools could solve a certain subclass of the undecidable satisfiability problem, but they all have their own limitations. They could still be used as components in the context of other frameworks, in accordance with the different synthesis requirements to solve different logical formulas or mathematical problems. In Section 5, we use iSAT and Z3 to check the VCs in various input formats, and compare their performance by the execution time and the number of solved VCs.

2.6 Verification Condition Generation

As the system descriptions are structured programs, it is natural to integrate software verification methods with the already established model checking techniques. It is well-known that axiomatic semantics as given by the Hoare calculus are a basic foundation for the formal verification of software.

2.6.1 Hoare Calculus based VCG

Hoare calculus provides proof rules for each statement of the considered program languages: A decomposition rule reduces a proof goal for a statement to proof goals using only its sub-statements. Figure 2.13 shows Hoares axioms for a simple sequential programming language. The verification of algorithms can even be done in a parameterized way that abstracts from the size of data structures (like array sizes) as well as from the data types themselves (by considering polymorphic types). While systems with complex dataflow are usually difficult to handle using model checking, they can often be easily verified by means of interactive verification using the Hoare calculus.

Verification Condition Generations (VCGs) are used to aggressively apply all rules of the Hoare calculus to a program whose statements are annotated with pre- and postconditions so that the validity of the obtained VCs implies the correctness of the given proof goal $\{\varphi\}\ \mathtt{S}\ \{\psi\}$, which states that if φ holds at starting time of the program statement S and if S will terminate, then ψ holds at termination time of S. In principle, it is sufficient to only provide the right loop invariants, since intermediate assertions can be computed automatically by *weakest preconditions*. VCGs can work very fast, since they just make a linear pass over the program text, and the obtained formulas can then be checked one after the other. This way, the overall verification problem is split into two independent parts: First, generating the VCs, and second proving them (which can be done by different tools and also in parallel).

As already explained in Section 2.5.2.2, there has been considerable progress on SMT solvers that can deal with arithmetic formulas, especially non-linear arithmetics. Such tools that are already used for the verification of (linear) hybrid systems, and mainly employ *bounded-model checking* where safety properties are unrolled for some finite number of discrete transitions. However, it is quite difficult to adapt Hoare Calculus to imperative synchronous programs, and there is no VCG procedure to generate proof

nothing: $$\frac{}{\{\Phi\}\ \text{nothing}\ \{\Phi\}}$$

assign: $$\frac{}{\{[\Phi]^{\tau}_{x}\}\ \mathtt{x}{=}\tau\ \{\Phi\}}$$

sequence: $$\frac{\{\Phi_1\}\ \mathtt{S}_1\ \{\Phi_2\} \qquad \{\Phi_2\}\ \mathtt{S}_2\ \{\Phi_3\}}{\{\Phi_1\}\ \mathtt{S}_1;\mathtt{S}_2\ \{\Phi_3\}}$$

conditional: $$\frac{\{\sigma\wedge\Phi\}\ \mathtt{S}_1\ \{\Psi\} \qquad \{\neg\sigma\wedge\Phi\}\ \mathtt{S}_2\ \{\Psi\}}{\{\Phi\}\ \text{if}\ (\sigma)\mathtt{S}_1\ \text{else}\ \mathtt{S}_2\ \{\Psi\}}$$

loop: $$\frac{\{\sigma\wedge\Phi\}\ \mathtt{S}\ \{\Phi\}}{\{\Phi\}\ \text{while}(\sigma)\ \mathtt{S}\ \{\neg\sigma\wedge\Phi\}}$$

weaken: $$\frac{\models\Phi_1\rightarrow\Phi_2 \qquad \{\Phi_2\}\ \mathtt{S}\ \{\Phi_3\} \qquad \models\Phi_3\rightarrow\Phi_4}{\{\Phi_1\}\ \mathtt{S}\ \{\Phi_4\}}$$

Figure 2.13: Hoare Calculus for a sequential Programming Language

goals of correctness statements for hybrid programs before [LS15c].

2.6.2 Difficulties of Adapting Hoare Calculus

Compilers for sequential programs usually translate the programs into an internal representation for code optimization and code generation. In most cases, compilers use control-flow graphs whose nodes are basic blocks that contain instructions without branches. It is well-known that sequential programs with structured statements like sequences, if-then-else statements, while-loops etc. yield special control-flow graphs that are called reducible flow graphs [BT79; Elg76; AM71]. These reducible flow graphs have many special properties like having dominators [LT79] that are exploited in dataflow analysis of compilers.

A control-flow graph is reducible if a sequence of transformations collapse it into a single node. To this end, only two transformations are required: The first transformation T_1 removes self-loops from nodes in the graph, and the second transformation T_2 merges a node with only one incoming edge with its parent node (that inherits then the outgoing transitions of its child). One can use these collapsing operations also to translate a flowgraph to a sequential structured program.

Synchronous programs can yield EFSMs that are irreducible. For example, module `Irreducible` in Figure 2.14 yields the EFSM shown in Figure 2.15. It copies input `i` to the output `o` with a delay of one unit of time. Since only immediate assignments are used, it has to remember the value of the previous input in its control-flow states so that `w1` is entered iff `i` holds, and otherwise `w2` is entered. Depending on the initial input, the EFSM then has a transition from the initial state to either `w1` or `w2`, and we have transitions also between `w1` or `w2`. The EFSM therefore contains the (only) typical irreducible graph.

This is remarkable since it is known that for sequential programs, only the use of 'unstructured' statements like goto statements can lead to irreducible control-flow graphs. In case of synchronous programs, this is however the result of the abstraction to macro steps where all micro step actions and all control-flow decisions are evaluated in an EFSM node, and then transitions are made depending on the results. Hence, an EFSM node may have any number of successor states, while basic blocks in a control-flow graph can only have one or two successor states.

The abstraction to macro steps makes the decompositional style of Hoare calculus

```
1 module Irreducible(bool ?i, bool !o) {
2   o = false;
3   while(true) {
4     while(i) {w1: pause; o = true;}
5     while(!i) {w2: pause; o = false;}
6   }
7 }
```

Figure 2.14: Module `Irreducible`

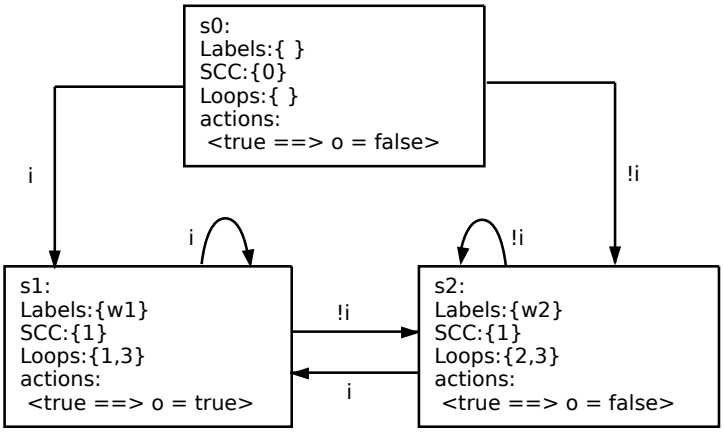

Figure 2.15: EFSM of Module `Irreducible`

impossible for synchronous programs. Note that the transformations T_1 and T_2 correspond to the introduction of loops and conditional sequences, respectively, and therefore can be used to translate the graph to a structured sequential program, so that this way Hoare calculus can be applied. Since this is not possible for synchronous programs, it shows that Hoare calculus cannot deal with synchronous programs.

It is also well-known that one can rewrite sequential programs so that their control-flow graphs become reducible, e.g., the EFSM in Figure 2.17 of module `Reducible` in Figure 2.16. However, this requires in general an exponential blow-up of the control-flow graph [All70], and is therefore not practical. Also, [GS12] have proved that this is possible for only a restricted class of synchronous programs.

Besides the argument of irreducible EFSMs, another argument for the non-existence of a Hoare calculus is that macro steps are composed of micro steps that are collected from different places of the program (in case parallel statements are active). This way, Hoare calculus can no longer be applied in a decompositional manner reducing to the next step to sub-statements only.

```
1  module Reducible(bool ?i, bool !o) {
2    o = false;
3    if(!i) {
4      while(!i) {w0: pause; o = false;}
5    }
6    while(true) {
7      while(i) {w1: pause; o = true;}
8      while(!i) {w2: pause; o = false;}
9    }
10 }
```

Figure 2.16: Module `Reducible`

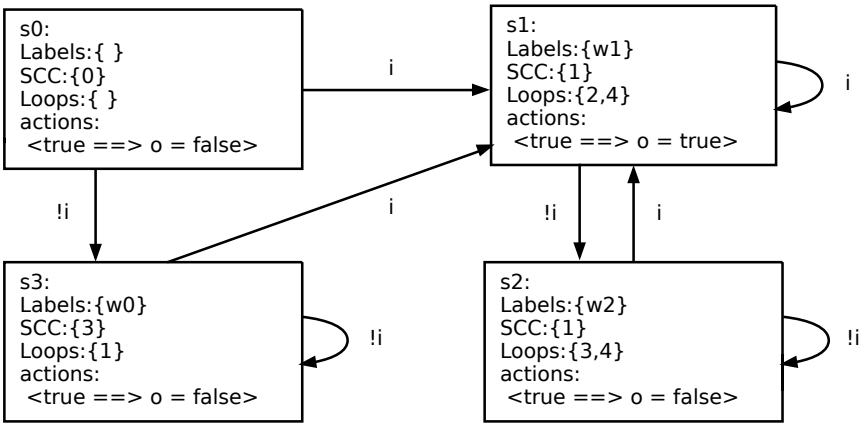

Figure 2.17: EFSM of Module `Reducible`

2.7 PDR in a Nutshell

In this section, we restrict our consideration to boolean programs and give a brief introduction to Property Directed Reachability (PDR) as far as required for this thesis.

2.7.1 Symbolic Model Checking

To start with, we assume that a state transition system $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{T})$ is symbolically represented by means of a finite set of boolean variables $\mathcal{V}$, and propositional formulas $\mathcal{I}$ and $\mathcal{T}$ for its initial states and its transitions, respectively. A state $s \subseteq \mathcal{V}$ of $\mathcal{K}$ is a subset of $\mathcal{V}$ such that those variables hold in the state that belong to s while others are false. As usual, every propositional formula φ over the variables $\mathcal{V}$ is associated with a set of states $[\![\varphi]\!]_{\mathcal{K}} \subseteq 2^{\mathcal{V}}$ of the transition system which are those states that satisfy φ if the states are viewed as variable assignments. Analogously, every propositional formula over the variables $\mathcal{V}$ and a related set $\mathcal{V}'$ denotes a set of transitions so that the assignments to variables $\mathcal{V}$ and $\mathcal{V}'$ correspond with the current and next state, respectively.

To reason about temporal relationships of states, we define the *existential/universal predecessor/successor states* of a state set Q as follows:

- $\mathsf{pre}_{\exists}(Q_2) := \{s_1 \in \mathcal{S}_1 \mid \exists s_2.(s_1, s_2) \in \mathcal{T} \wedge s_2 \in Q_2\}$
- $\mathsf{pre}_{\forall}(Q_2) := \{s_1 \in \mathcal{S}_1 \mid \forall s_2.(s_1, s_2) \in \mathcal{T} \rightarrow s_2 \in Q_2\}$
- $\mathsf{suc}_{\exists}(Q_1) := \{s_2 \in \mathcal{S}_2 \mid \exists s_1.(s_1, s_2) \in \mathcal{T} \wedge s_1 \in Q_1\}$
- $\mathsf{suc}_{\forall}(Q_1) := \{s_2 \in \mathcal{S}_2 \mid \forall s_1.(s_1, s_2) \in \mathcal{T} \rightarrow s_1 \in Q_1\}$

While the above definitions are given in terms of sets of states, their counterparts in the syntax of formulas are the following *modal operators taken from μ-calculus* [Sch03]:

- $[\![\Diamond\varphi]\!]_{\mathcal{K}} := \mathsf{pre}_\exists([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\Box\varphi]\!]_{\mathcal{K}} := \mathsf{pre}_\forall([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\overleftarrow{\Diamond}\varphi]\!]_{\mathcal{K}} := \mathsf{suc}_\exists([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\overleftarrow{\Box}\varphi]\!]_{\mathcal{K}} := \mathsf{suc}_\forall([\![\varphi]\!]_{\mathcal{K}})$

Propositional logic with the above modal operators is however not powerful enough to reason about interesting temporal properties. In the μ-calculus [Sch03], one adds *fixpoint formulas* $\mu x.\varphi$ and $\nu x.\varphi$ that denote the least and greatest sets of states $\mathtt{x}$ such that $x = \varphi$ holds. Fixpoint formulas can be evaluated by means of the Tarski-Knaster iteration [Sch03]. For example, the set of *reachable states* $\mathcal{S}_{\mathsf{reach}}$ can be computed as the set of states satisfying $\Psi_{\mathsf{reach}} := \mu x.\mathcal{I} \vee \overleftarrow{\Diamond} x$ which leads to the fixpoint iteration $\mathcal{X}'_0 := \{\}$ and $\mathcal{X}'_{i+1} := [\![\mathcal{I}]\!]_{\mathcal{K}} \cup \mathsf{suc}_\exists(\mathcal{X}'_i)$ or the equivalent one starting with $\mathcal{X}_0 := [\![\mathcal{I}]\!]_{\mathcal{K}}$ and iterating $\mathcal{X}_{i+1} := \mathcal{X}_i \cup \mathsf{suc}_\exists(\mathcal{X}_i)$ (as used in the following).

Temporal logics like CTL, LTL and CTL* can all be expressed in the μ-calculus [Sch03]. For this paper, we are just interested in *safety properties*, i.e., we want to prove that some property Φ holds on all reachable states. This can be expressed in the temporal logic CTL as AGΦ where A quantifies over all paths leaving a state and G quantifies over all points of time on that path. It is known that AGΦ is equivalent to the μ-calculus formula $\nu x.\ \Phi \wedge \Box x$, i.e., the greatest set of states $\mathtt{x}$ such that $x = \Phi \wedge \Box x$ holds. A fixpoint-based model checker would therefore compute this set of states according to the Tarski-Knaster iteration starting with $Q_0 := \mathcal{S}$ (the set of all states) and iterating $Q_{i+1} := [\![\Phi]\!]_{\mathcal{K}} \cap \mathsf{pre}_\forall(Q_i)$.

Instead of computing all states that satisfy a safety property AGΦ by fixpoint iteration, and checking afterwards whether all initial states are included, we can also prove them by means of one of the following induction rules:

$$\frac{\mathcal{I} \to \Phi \quad \Psi_{\mathsf{reach}} \wedge \Phi \to \Box\Phi}{\Psi_{\mathsf{reach}} \to \Phi} \qquad \frac{\mathcal{I} \to \Phi \quad \Phi \to \Box\Phi}{\Psi_{\mathsf{reach}} \to \Phi}$$

The rule on the right-hand side obviously follows from the one on the left-hand side, since its second subgoal is stronger (validity of $\Phi \to \Box\Phi$ implies validity of $\Psi_{\mathsf{reach}} \wedge \Phi \to \Box\Phi$). To prove the correctness of the rule on the left-hand side, one can prove (by induction on i) that $\mathcal{X}_i \subseteq [\![\Phi]\!]_{\mathcal{K}}$ holds where $\mathcal{X}_i$ are the sets of states that can be reached in no more than $i-1$ steps, i.e., those that occur in the fixpoint iteration $\mathcal{X}_0 := \{\}$ and $\mathcal{X}_{i+1} := [\![\mathcal{I}]\!]_{\mathcal{K}} \cup \mathsf{suc}_\exists(\mathcal{X}_i)$ to compute the reachable states $\mathcal{S}_{\mathsf{reach}}$.

In these kinds of proofs[3], as well as those of the correctness of PDR, the following lemmata are very useful and hold for every set of states Q:

- $\mathsf{suc}_\exists(\mathsf{pre}_\forall(Q)) \subseteq Q$ and $Q \subseteq \mathsf{pre}_\forall(\mathsf{suc}_\exists(Q))$

[3]For example, the correctness of the first induction rule is proved as follows: The induction step is trivial since $\mathcal{X}_0 := \{\}$ holds. For the induction step, we have the induction hypothesis IH $\mathcal{X}_i \subseteq [\![\Phi]\!]_{\mathcal{K}}$ and we have to prove that $\mathcal{X}_{i+1} \subseteq [\![\Phi]\!]_{\mathcal{K}}$ holds: By the second subogal, we have (1) $\mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}} \subseteq \mathsf{pre}_\forall([\![\Phi]\!]_{\mathcal{K}})$. By monotonicity, we get (2) $\mathsf{suc}_\exists(\mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}}) \subseteq \mathsf{suc}_\exists(\mathsf{pre}_\forall([\![\Phi]\!]_{\mathcal{K}}))$ from (1). From (2), we now get (3) $\mathsf{suc}_\exists(\mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}}) \subseteq [\![\Phi]\!]_{\mathcal{K}}$ and from the IH, we get (4) $\mathcal{S}_{\mathsf{reach}} \cap \mathcal{X}_i \subseteq \mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}}$, i.e., (4') $\mathcal{X}_i \subseteq \mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}}$. Further, by monotonicity, we have (5) $\mathsf{suc}_\exists(\mathcal{X}_i) \subseteq \mathsf{suc}_\exists(\mathcal{S}_{\mathsf{reach}} \cap [\![\Phi]\!]_{\mathcal{K}})$ and by transitivity of (3) and (5), we get (6) $\mathsf{suc}_\exists(\mathcal{X}_i) \subseteq [\![\Phi]\!]_{\mathcal{K}}$. By the first subgoal, we have (0) $[\![\mathcal{I}]\!]_{\mathcal{K}} \subseteq [\![\Phi]\!]_{\mathcal{K}}$, and therefore $\mathcal{X}_{i+1} = [\![\mathcal{I}]\!]_{\mathcal{K}} \cup \mathsf{suc}_\exists(\mathcal{X}_i) \subseteq [\![\Phi]\!]_{\mathcal{K}}$ holds by (0) and (6).

- $Q \subseteq \mathsf{pre}_\forall(Q)$ holds iff $\mathsf{suc}_\exists(Q) \subseteq Q$ (induction steps)
- $\mathsf{suc}_\exists([\![\mathsf{AG}\Phi]\!]_\mathcal{K}) \subseteq [\![\mathsf{AG}\Phi]\!]_\mathcal{K} \subseteq [\![\Phi]\!]_\mathcal{K} \cap \mathsf{pre}_\forall([\![\Phi]\!]_\mathcal{K})$
- If $\mathsf{AG}\Phi$ — i.e., $\nu x.\, \Phi \wedge \Box x$ — holds on all initial states, then it holds on all reachable states (and is therefore a safety property).

2.7.2 Incremental Induction by PDR

Property Directed Reachability (PDR) tries to prove that a property Φ holds on all reachable states $\mathcal{S}_\mathsf{reach}$ by means of induction. The induction rules of the previous section may however not be strong enough, since even though Φ may hold on all reachable states, the induction step may fail due to *counterexamples to induction* (CTIs): Φ may also hold on an unreachable state that has a successor state where Φ does not hold. For this reason, PDR checks the unreachability of state sets to identify counterexamples as CTIs, and generalizes them to larger state sets.

A property φ where $\varphi \rightarrow \Box\varphi$ is valid[4] is called *inductive*, which means that no transition starting in states satisfying φ will leave this state set, i.e., φ contains all its successor states. In particular, Ψ_reach is inductive, and the goal of PDR is to generate an inductive property Ψ_i that implies a considered property Φ.

PDR is used to check whether a property Φ holds on at least all reachable states of a transition system $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{R})$. To this end, PDR first makes the following checks

- $\mathcal{I} \rightarrow \Phi$: If not valid, we have a counterexample of length 0, since the counterexample describes a state that satisfies $\mathcal{I}$ but not Φ.
- $\mathcal{I} \rightarrow \Box\Phi$: If not valid, we have a counterexample of length 1, since the counterexample describes a state that satisfies $\mathcal{I}$ but not $\Box\Phi$, thus there is an initial state with a successor state that violates Φ.

If one of the two is not valid, we have a counterexample for Φ, otherwise, there are no counterexamples of length 0 or 1. PDR continues then by checking the following:

- $\mathcal{I} \rightarrow \Box\mathcal{I}$: If valid, we have a proof, since then only the initial states are reachable states, and we already know that they all satisfy Φ.
- $\Phi \rightarrow \Box\Phi$: If valid, we have a proof since then also the induction step for Φ has been proved this way.

Otherwise, there are reachable states other than the initial states, and there are states that satisfy Φ, but at least one of their successor states violates Φ. Thus, PDR starts then its real work. To this end, it sets up the following initial sequence of predicates[5]:

[4] Note that checking an implication $\psi \rightarrow \Box\varphi$ can be done by a SAT solver by just checking $\psi \wedge \mathcal{R} \rightarrow \varphi'$ where φ' is obtained from φ by replacing every occurrence of a variable $x \in \mathcal{V}$ by its corresponding one $x' \in \mathcal{V}$.

[5] For the sake of simplicity, we ignore here that PDR actually works on clause sets, which is however important for its implementation.

- $\Psi_0 := \mathcal{I}$
- $\Psi_1 := \Phi$

This way, we have an initial sequence of predicates Ψ_0 and Ψ_1. In general, PDR maintains an incrementally increasing sequence $\Psi_0, \ldots, \Psi_k$ with the following properties:

- $\mathcal{I} = \Psi_0$
- $\Psi_i \to \Psi_{i+1} \wedge \Box\Psi_{i+1}$ for $i = 0, \ldots, k-1$
- $\Psi_k \to \Phi$, i.e., also $\Psi_i \to \Phi$ for $i = 0, \ldots, k$
- no Ψ_i is inductive, in particular, we have $\Psi_i \neq \Psi_{i+1}$ for $i = 0, \ldots, k-1$

One can easily prove that by these invariants, we have the following situation where $\mathcal{X}_i$ denotes the states that can be reached from the initial states in no more than i transition steps, and $\mathcal{S}_{\mathsf{reach}}$ are the reachable states:

$$\begin{array}{ccccccccccc} [\![\mathcal{I}]\!]_{\mathcal{K}} & = & \mathcal{X}_0 & \subseteq & \mathcal{X}_1 & \subseteq & \ldots & \subseteq & \mathcal{X}_k & \subseteq & \mathcal{S}_{\mathsf{reach}} \\ \| & & \| & & \cap & & & & \cap & & \\ [\![\mathcal{I}]\!]_{\mathcal{K}} & = & [\![\Psi_0]\!]_{\mathcal{K}} & \subseteq & [\![\Psi_1]\!]_{\mathcal{K}} & \subseteq & \ldots & \subseteq & [\![\Psi_k]\!]_{\mathcal{K}} & \subseteq & [\![\Phi]\!]_{\mathcal{K}} \end{array}$$

Having an approximation $\Psi_0, \ldots, \Psi_k$ of $\mathcal{X}_0, \ldots, \mathcal{X}_k$, the main algorithm of PDR then checks whether $\Psi_k \to \Box\Phi$ is valid, i.e., whether there are some states in Ψ_k that have successors violating Φ. Depending on this, one of the following is done:

- *Case 1: $\Psi_k \to \Box\Phi$ is valid.* In this case, we have not yet a counterexample, but also the induction proof was not yet successful. Thus, we have to incrementally extend the sequence of predicates (note that with a sufficiently large k, we must finally have $\mathcal{S}_{\mathsf{reach}} = \mathcal{X}_k \subseteq [\![\Psi_k]\!]_{\mathcal{K}} \subseteq [\![\Phi]\!]_{\mathcal{K}}$ so that a proof will be finally found if we can make Ψ_k close enough to $\mathcal{S}_{\mathsf{reach}}$ so that it must become inductive). To this end, we could define $\Psi_{k+1} := \Phi$ which would satisfy all PDR invariants, but PDR narrows Φ first by propagating clauses from Ψ_k to Φ. Hence, it defines $\Psi_{k+1} := \mathsf{narrow}(\Phi, \Psi_k)$ where the latter is defined as follows: $\mathsf{narrow}(\Phi, \Psi_k)$ is obtained by checking for all clauses c_i of $\Psi_k := \bigwedge_{i=0}^{m} c_i$ whether $\Psi_k \to \Box c_i$ is valid. Assume this is the case for the clauses $c_0, \ldots, c_n$ with $n \leq m$, then we define $\Psi_{k+1} := \mathsf{narrow}(\Phi, \Psi_k) := \Phi \wedge \bigwedge_{i=0}^{n} c_i$ to increase the chance of making it inductive. All PDR invariants are maintained this way, and with increasing k, we proceed towards a proof or a counterexample.
- *Case 2: $\Psi_k \to \Box\Phi$ is not valid.* Then, the SAT solver generates a counterexample that satisfies $\Psi_k \wedge \neg\Box\Phi$, i.e., a (partial) assignment to some of the variables $\mathcal{V}$, which is a cube $\mathcal{C}_k$, i.e., a conjunction of literals of these variables. PDR now has to check *whether one of these states is reachable from the initial states within k steps* as explained in Section 4.3.1 below. If one of them is reachable within k steps, it is a real counterexample since the property Φ then does not hold on

all reachable states. Otherwise, the states in $\mathcal{C}_k$ are all unreachable, thus form a CTI that can now be removed from all predicates Ψ_i. To that end, PDR first *generalizes the cube* $\mathcal{C}_k$ to another cube $\mathcal{C}'_k$ as explained in Section 2.7.4 below so that as many unreachable states as possible are removed by adding the clause $\neg\mathcal{C}'_k$ to all $\Psi_0, \ldots, \Psi_k$.

PDR therefore terminates with either a counterexample in case 2, or with a proof as soon as one Ψ_i satisfies $\Psi_i \Leftrightarrow \Psi_{i+1}$, i.e., became inductive in case 1 or in case 2. As explained above in these two cases, two important procedures are therefore (1) *checking the unreachability of a cube* $\mathcal{C}_k$ *within* k *steps* and (2) the *generalization of a cube* $\mathcal{C}_k$ *that has been identified as a CTI*. These procedures work as explained in more detail in the following two sections.

2.7.3 Checking Unreachability of Cubes

To prove the unreachability of (all states of) a cube $\mathcal{C}_k$ within k steps, we would essentially have to check whether $\Psi_{k-1} \rightarrow \Box\neg\mathcal{C}_k$ is valid. If it is valid, then no state of $\mathcal{C}_k$ can be reached within k steps. Otherwise, there are states in Ψ_{k-1} that have successor states in $\mathcal{C}_k$ (but these could be unreachable). Bradley observed that it suffices to check the validity of $\Psi_{k-1} \wedge \neg\mathcal{C}_k \rightarrow \Box\neg\mathcal{C}_k$ and $\mathcal{I} \rightarrow \neg\mathcal{C}_k$ instead: If both are valid, we conclude that no state in $\mathcal{C}_k$ is reachable within k steps[6], and we can therefore remove all of these states from the predicates $\Psi_0, \ldots, \Psi_k$. On the other hand, if $\mathcal{I} \rightarrow \neg\mathcal{C}_k$ should not be valid, we see that some states of $\mathcal{C}_k$ are initial states and are therefore reachable.

Finally, if $\Psi_{k-1} \wedge \neg\mathcal{C}_k \rightarrow \Box\neg\mathcal{C}_k$ should not be valid, then there is another counterexample, thus another cube $\mathcal{C}_{k-1}$, which contains states that satisfy $\Psi_{k-1} \wedge \neg\mathcal{C}_k$ and that have at least one successor state in $\mathcal{C}_k$. Therefore, we have to recursively check the reachability of $\mathcal{C}_{k-1}$ in $k-1$ steps in the same way. Finally, if we get a counterexample $\mathcal{C}_0$ from $\Psi_0 \wedge \neg\mathcal{C}_1 \rightarrow \Box\neg\mathcal{C}_1$, then the sequence $\mathcal{C}_0, \ldots, \mathcal{C}_k$ contains a path in the transition system reaching a state in $\mathcal{C}_k$. Otherwise, we conclude that no state in $\mathcal{C}_k$ is reachable in k steps and can remove it from the approximations $\Psi_0, \ldots, \Psi_k$ by adding the clause $\neg\mathcal{C}_k$ to these approximations, or better by adding a generalized clause $\neg\mathcal{C}'_k$ as explained in the next section.

2.7.4 Generalization of CTIs

Once a cube $\mathcal{C}_k$ has been proved unreachable within k steps, i.e., finally $\Psi_{k-1} \wedge \neg\mathcal{C}_k \rightarrow \Box\neg\mathcal{C}_k$ is valid, we may add its clause $\neg\mathcal{C}_k$ to all approximations $\Psi_0, \ldots, \Psi_k$. However,

[6] Assume $\mathcal{X}_i$ are the states that are reachable within no more than i steps, i.e., $\mathcal{X}_0 := \mathcal{I}$ with $\mathcal{I} := [\![\mathcal{I}]\!]_\mathcal{K}$ and $\mathcal{X}_{i+1} := \mathcal{X}_i \cup \mathsf{suc}_\exists(\mathcal{X}_i)$. Given that $\Psi_{k-1} \wedge \neg\mathcal{C}_k \rightarrow \Box\neg\mathcal{C}_k$ and $\mathcal{I} \rightarrow \neg\mathcal{C}_k$ hold, we can prove by induction that $\mathcal{X}_i \cap \mathcal{C}_k = \{\}$ holds for $i \leq k$: The induction base $\mathcal{X}_0 \cap \mathcal{C}_k = \{\}$ is equivalent to the validity of $\mathcal{I} \rightarrow \neg\mathcal{C}_k$ that we assumed. For proving $\mathcal{X}_{i+1} \cap \mathcal{C}_k = \{\}$, we have to prove by the definition $\mathcal{X}_{i+1} := \mathcal{X}_i \cup \mathsf{suc}_\exists(\mathcal{X}_i)$ that $\mathcal{X}_i \cap \mathcal{C}_k = \{\}$ and $\mathsf{suc}_\exists(\mathcal{X}_i) \cap \mathcal{C}_k = \{\}$ holds. The former follows by induction hypothesis, and the latter is derived as follows: By induction hypothesis, we have $\mathcal{X}_i \cap \mathcal{C}_k = \{\}$, so that $\mathcal{X}_i \rightarrow \Psi_i \wedge \neg\mathcal{C}_k$ is valid ($\mathcal{X}_i \rightarrow \Psi_i$ holds by construction of Ψ_i). Since $\Psi_i \rightarrow \Psi_{k-1}$ holds for $i < k$, also $\mathcal{X}_i \rightarrow \Psi_{k-1} \wedge \neg\mathcal{C}_k$ is valid, and thus, we finally conclude with $\Psi_{k-1} \wedge \neg\mathcal{C}_k \rightarrow \Box\neg\mathcal{C}_k$ that $\mathcal{X}_i \rightarrow \Box\neg\mathcal{C}_k$ holds, which finally implies $\mathsf{suc}_\exists(\mathcal{X}_i) \cap \mathcal{C}_k = \{\}$.

PDR tries to generalize the clause $\neg\mathcal{C}_k$ before adding it by removing single literals from the clause as long as it remains unreachable. This way as many unreachable states as possible are removed. In essence, PDR searches a subclause $\neg\mathcal{C}'_k$ of $\neg\mathcal{C}_k$ such that $\Psi_{k-1} \wedge \neg\mathcal{C}'_k \rightarrow \Box\neg\mathcal{C}'_k$ and $\mathcal{I} \rightarrow \mathcal{C}'_k$ still remain valid (hence a larger set of unreachable states).

To this end, PDR constructs the subclause lattice $\mathcal{L}_{\mathcal{C}_k} := (2^{\mathcal{C}_k}, \sqsubseteq_{\Psi_k})$ whose elements are the subclauses of $\neg\mathcal{C}_k$ that are ordered by the subclause relative relation $\sqsubseteq_{\Psi_k}$ defined as follows: Two subclauses $c_1, c_2 \in 2^{\mathcal{C}_k}$ satisfy the relation $c_1 \sqsubseteq_{\Psi_k} c_2$ iff $c_1 \subseteq c_2$ and $c_1 \wedge \Psi_k \rightarrow \Box c_2$ holds. For example, assume that the SAT solver returns $\mathcal{C}_k := \mathtt{p}_1 \wedge \neg\mathtt{p}_2$ as a CTI over $\mathcal{V} = \{\mathtt{p}_1, \mathtt{p}_2\}$. We have $\neg\mathcal{C}_k = \{\neg\mathtt{p}_1, \mathtt{p}_2\}$, and the subclause lattice is defined by $\mathcal{L}_{\mathcal{C}_k} = (\{\{\}, \{\neg\mathtt{p}_1\}, \{\mathtt{p}_2\}, \{\neg\mathtt{p}_1, \mathtt{p}_2\}\}, \sqsubseteq_{\Psi_k})$. Starting from the top element $\neg\mathcal{C}_k$ of $2^{\mathcal{C}_k}$, which is inductive relatively to Ψ_k as proved by the above reachability analysis, all the subclauses of $\neg\mathcal{C}_k$ that are inductive relatively to Ψ_k can be computed. Among these, PDR chooses one that does not contain any strict subclause that is inductive relatively to Ψ_k as a minimal inductive subclause. Instead of strengthening each $\Psi_0, \ldots, \Psi_k$ with $\neg\mathcal{C}_k$, these sets are then updated by $\Psi_i \wedge c$, so that $\mathcal{C}_k$ and many other unreachable states are removed from $\Psi_0, \ldots, \Psi_k$.

Chapter 3

Verification Condition Generation Using Inductive Assertions

Contents

This chapter uses *Floyd's induction-based* approach to generate verification conditions for synchronous and hybrid programs. The proposed VCG methods consist of two steps, where the first step consists of computing for a synchronous or hybrid Quartz program its EFSM. In the second step, the user has to provide *inductive assertions (invariants)*

for each safety property and each component of the generated EFSM. The components can be either paths along several control-flow states or single transitions between two control-flow states, or paths between different SCCs or single transitions among control-flow states related to the same loop statement in the program, etc. The VCG methods apply then the induction rules and generate this way proof goals that correspond with induction steps and bases:

- the Transition-based method sets up proof goals over single EFSM transitions.
- the SCC-Path method performs induction over nontrivial SCCs: Induction bases and steps are set up for paths to nontrivial SCCs. A better optimization is SCC-Trans where the induction is performed for both trivial and nontrivial SCCs, so that the induction bases are constructed by single transitions instead of paths.
- Loop-based methods set up induction proofs according to the loop statements in the program. Depending on whether the induction bases are set up by paths or single transitions, two variants, i.e., Loop-Path and Loop-Trans, are provided.

3.1 SafeTrans and SafePath Predicates

Most of the verification conditions generated by the proposed VCG methods are in the form of either SafeTrans or SafePath predicate.

3.1.1 Abbreviations for Predicates

Assume that a Quartz program $\mathcal{P}$ has m control-flow variables, i.e., $\mathcal{V}^{\mathsf{cf}} := \{\mathtt{w}_0, \ldots, \mathtt{w}_{m-1}\}$. Its EFSM Σ has nodes $s_0, \ldots, s_n$ — among which s_0 is the initial node — and transitions $\mathsf{Trans}(\Sigma)$. Each EFSM node s_i is labeled with a unique set of control-flow variables $\mathsf{Labels}(s_i) \subseteq \mathcal{V}^{\mathsf{cf}}$ and can be identified by the following *minterm*:

$$\sigma(s_i) := \bigwedge_{i=0}^{m-1} \mathtt{w}'_j \quad \text{where } \mathtt{w}'_j := \begin{cases} \mathtt{w}_j, & \text{iff } \mathtt{w}_j \in \mathsf{Labels}(s_i) \\ \neg \mathtt{w}_j, & \text{otherwise} \end{cases}$$

The set of guarded actions $\mathcal{G}(s_i)$ for each node s_i can be reorganized as follows:

$$\mathcal{D}_d(s_i) \;:=\; \bigwedge_{(\gamma,\alpha)\in\mathcal{G}_d(s_i)} (\gamma \to \alpha) \qquad\qquad \mathcal{D}_c(s_i) \;:=\; \bigwedge_{(\gamma,\alpha)\in\mathcal{G}_c(s_i)} (\gamma \to \alpha)$$

where $\mathcal{D}_d(s_i)$ and $\mathcal{D}_c(s_i)$ determine the discrete environment and continuous evolutions in node s_i, respectively. Thus, we have $\mathcal{D}(s_i)$ for node s_i:

$$\mathcal{D}(s_i) \;:=\; \sigma(s_i) \;\wedge\; \mathcal{D}_d(s_i) \;\wedge\; \mathit{cont}_\forall(\mathcal{D}_c(s_i))$$

where the predicate $\mathit{cont}_\forall$ demands that $\mathcal{D}_c(s_i)$ must hold during the continuous phase in the node s_i. For every transition $(s_i, s_j) \in \mathsf{Trans}(\Sigma)$, $\mathcal{D}(s_i)$ and $\mathcal{D}(s_j)$ will be used to determine the variable environment of the current and next nodes, respectively.

Assume that the dataflow is defined over a set of variables $\vec{x} = (x_1, \ldots, x_n)$, and for every variable there is exactly one enabled assignment, i.e., there is per-variable exactly one immediate, delayed or continuous assignment in every transition (compilers can add *default actions* when needed), so that the discrete environment coincides with initial values of continuous evolutions for each node. ψ_i is a continuous invariant for node s_i, if and only the formula $cont_\forall(\psi_i)$ is valid. Because of the delayed assignments, $\mathcal{D}_d(s_i)$ has occurrences of the variable x and also of $\mathbf{next}(x)$, while $\mathcal{D}_c(s_i)$, $\varphi(s_i, s_{i+1})$, and $\psi(s_i)$ have only occurrences of the variable x. Finally, we write

$$[\mathcal{D}_d(s_i)]^{\vec{x}_i,\vec{x}_{i+1}}_{\vec{x},\mathbf{next}(\vec{x})},\ [\mathcal{D}_c(s_i)]^{\vec{x}_i}_{\vec{x}},\ [\varphi(s_i, s_{i+1})]^{\vec{x}_i}_{\vec{x}},\ \text{and}\ [\psi_i]^{\vec{x}_i}_{\vec{x}}$$

for the formulas that are obtained by replacing all occurrences of variables $\vec{x}$ and $\mathbf{next}(\vec{x})$ by new variables $\vec{x}_i$ and $\vec{x}_{i+1}$ in the formulas $\mathcal{D}_d(s_i)$, $\mathcal{D}_c(s_i)$, $\varphi(s_i, s_{i+1})$, and ψ_i, respectively.

3.1.2 The SafeTrans Predicate

For every transition $(s_i, s_j) \in \mathsf{Trans}(\Sigma)$ starting from the discrete environment of node s_i and ending in discrete environment of node s_j, we define the following abbreviation using (for simplicity) fresh variables $\vec{x}_0$, $\vec{x}_1$ and $\vec{x}_2$ instead of $\vec{x}_i$, $\vec{x}_{i+1}$ and $\vec{x}_{i+2}$:

$$\begin{aligned}
&\mathsf{SafeTrans}((s_i, s_j), \alpha_i, \psi_i, \alpha_j) \ :=\\
&\Big(\ [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \ \wedge\ cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \ \wedge\ [\mathcal{D}_d(s_j)]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \ \wedge\\
&\quad [\alpha_i]^{\vec{x}_0}_{\vec{x}} \ \wedge\ \sigma(s_i) \ \wedge\ [\varphi(s_i, s_j)]^{\vec{x}_0}_{\vec{x}} \ \Big)\\
&\rightarrow \ \Big(\ cont_\forall\big([\psi_i]^{\vec{x}_0}_{\vec{x}} \big) \ \wedge\ [\alpha_j]^{\vec{x}_1}_{\vec{x}} \ \wedge\ \sigma(s_j) \ \Big)
\end{aligned}$$

The SafeTrans predicate formalizes the following two statements:

- The variables $\vec{x}_0, \vec{x}_1$ have values that are consistent with all the transitions that start from the discrete environment in s_i to the discrete environment in s_j, where (1) the control-flow information $\sigma(s_i)$ and $\sigma(s_j)$, (2) all discrete, continuous and derivative assignments of node s_i as well as the discrete assignments of node s_j are respected and (3) transition condition $\varphi(s_i, s_j)$ holds too.
- If assertion α_i holds for the discrete environment of node s_i, then ψ_i holds in the continuous environment of node s_i, and assertion α_j holds in the discrete environment of node s_j. For such α_i and ψ_i, it is not difficult to prove that $\alpha_i \rightarrow \psi_i$ holds always, since (as required) the discrete environment coincides with initial values of continuous evolutions for that node.

The SafeTrans predicate is comparable with a Hoare calculus triple $\{\alpha_i\}\ \mathtt{S}\ \{\alpha_j\}$ which intuitively says that the post-condition α_j must hold after executing the program $\mathtt{S}$ that satisfies the pre-condition α_i. It is also comparable with dynamic logic expressions $\alpha \rightarrow [\mathtt{S}]\,\gamma$, but additionally expresses *properties that must hold during the execution.*

3.1.3 The SafePath Predicate

For every path $s_0, \ldots, s_n$ of the EFSM starting from the discrete environment of node s_0 and ending in discrete environment of node s_n, we define the following abbreviation:

$$\begin{aligned}
&\mathsf{SafePath}\big((s_0,\ldots,s_n),(\psi_0,\ldots,\psi_{n-1}),\alpha,\beta,\gamma\big) \;:=\\
&\left(\left(\bigwedge_{i=0}^{n} [\mathcal{D}_d(s_i)]_{\vec{x},\mathbf{next}(\vec{x})}^{\vec{x}_i,\vec{x}_{i+1}}\right) \;\wedge\; \left(\bigwedge_{i=0}^{n-1} cont_{\forall}\big([\mathcal{D}_c(s_i)]_{\vec{x}}^{\vec{x}_i}\big)\right) \;\wedge\right.\\
&\qquad \left.[\alpha]_{\vec{x}}^{\vec{x}_0} \;\wedge\; \sigma(s_0) \;\wedge\; \left(\bigwedge_{i=0}^{n-1} [\varphi(s_i,s_{i+1})]_{\vec{x}}^{\vec{x}_i}\right)\right)\\
&\rightarrow \left(\left(\bigwedge_{i=1}^{n-1} [\beta]_{\vec{x}}^{\vec{x}_i}\right) \;\wedge\; \left(\bigwedge_{i=0}^{n-1} cont_{\forall}\big([\psi_i]_{\vec{x}}^{\vec{x}_i}\big)\right) \;\wedge\; [\gamma]_{\vec{x}}^{\vec{x}_n} \;\wedge\; \sigma(s_n)\right)
\end{aligned}$$

The SafePath predicate formalizes the following two statements:

- The variables $\vec{x}_0, \ldots, \vec{x}_n$ have values that are possible on a path from the discrete environment in s_0 to the discrete environment in s_n, where (1) the control-flow information of the head and tail nodes along that path, (2) all discrete assignments of each node, and (3) continuous assignments of nodes except for the tail one are respected and (4) all transition conditions $\varphi(s_i, s_{i+1})$ hold along that path.

- If assertion α holds in the discrete environment of head node s_i, then (1) for each node s_i with $i \neq n$, formula ψ_i holds during the corresponding continuous evolution, and (2) formula β holds in all discrete environments excepting the head and tail nodes, and (3) formula γ holds in the discrete environment of tail node s_{i+1}. Similar to the SafeTrans predicate, $\alpha \rightarrow \psi_0$ is required to hold. Moreover, $\beta \rightarrow \psi_i$ must hold so that the SafePath predicate ensures that the discrete environment coincides with initial values of continuous evolutions for that node.

3.1.4 Comparison between SafeTrans and SafePath

Compared with SafeTrans, SafePath focuses on the pre/post-conditions of paths, which may consist of more than one transition. In the SafePath predicate, β is an invariant of all discrete environments of nodes $s_1, \ldots, s_{n-1}$, while SafeTrans predicate allows to given each node an individual invariant for its discrete environment. The SafePath predicate defined in Section 3.1.3 can be modified, so that it can have the same effect as the SafeTrans predicate in Section 3.1.2. However, this change may add additional work to the user since it needs more invariants.

Notice that it is not difficult to prove the following equivalences:

- $\mathsf{SafeTrans}\big((s_i,s_j),\alpha_i,\psi_i,\alpha_j\big) \;\Leftrightarrow\; \mathsf{SafePath}\big((s_i,s_j),(\psi_i),\alpha_i,\mathsf{true},\alpha_j\big)$

- $\mathsf{SafePath}\big((s_0,\ldots,s_n),(\psi_0,\ldots,\psi_{n-1}),\alpha,\beta,\gamma\big) \;\Leftrightarrow$

$$\mathsf{SafeTrans}\big((s_0,s_1),\alpha,\psi_0,\beta\big)$$

$$\wedge \quad \left(\bigwedge_{i=1}^{n-2} \mathsf{SafeTrans}\big((s_i,s_{i+1}),\beta,\psi_i,\beta\big)\right)$$

$$\wedge \quad \mathsf{SafeTrans}\big((s_{n-1},s_n),\beta,\psi_{n-1},\gamma\big)$$

Both the SafeTrans and SafePath predicates can be translated to the SMT-LIB standard language [BFT15], and then checked by SMT solvers. According to the above equivalence, instead of checking a complicated formula converted from a SafePath predicate, we can prove the validity of several relatively simpler formulas converted from SafeTrans predicates. This splitting work may shorten the execution time. However, it is not always the case and it differs from tool to tool as will be shown in Chapter 5. Furthermore, for the programs that have no hybrid variables, we can always use *true* as the continuous invariant.

3.2 VCG using Control-flow Assertions

As explained in Section 2.4.1, each EFSM node represents a reachable control-flow state of the Quartz program that is encoded by the control-flow variables. A control-flow assertion α_i restricts the discrete environment of node s_i. The Transition-based method requires the user to provide for each EFSM node s_i a control-flow assertion α_i and a continuous invariant Ψ_i, by annotating the program code. Ψ_i is expected to hold for the continuous evolutions in node s_i.

3.2.1 The Transition-based Method

To ensure the validity of a safety property Φ, we have to prove TB_1-TB_4 in Theorem 1. It may yield several VCs, most of which are SafeTrans-formulas.

Theorem 1 (Correctness of the Transition-based Method)**.** *Given any Quartz program $\mathcal{P}$, its EFSM Σ with nodes $s_0,\ldots,s_n$ — where s_0 is initial — and transitions $\mathsf{Trans}(\Sigma)$, control-flow assertions $\alpha_0,\ldots,\alpha_n$ and continuous invariants $\Psi_0,\ldots,\Psi_n$ for each node, respectively, then a property Φ holds in all EFSM nodes if the following holds:*

$$\begin{array}{ll}
\mathsf{TB}_1: & \mathcal{D}(s_0) \to \alpha_0 \\
\mathsf{TB}_2: & \alpha_i \to \Phi \quad \textit{for all} \quad \alpha_0,\ldots,\alpha_n \\
\mathsf{TB}_3: & \Psi_i \to \Phi \quad \textit{for all} \quad \Psi_0,\ldots,\Psi_n \\
\mathsf{TB}_4: & \mathsf{SafeTrans}\big((s_k,s_l),\alpha_k,\Psi_k,\alpha_l\big) \quad \textit{for all} \quad (s_k,s_l) \in \mathsf{Trans}(\Sigma)
\end{array}$$

Proof. By TB_1, the discrete environment of node s_0 satisfies the control-flow assertion α_0. If α_k holds on the discrete environment encoded in node s_k, then TB_4 ensures

that Ψ_k holds on the continuous environment of node s_k, and α_l holds on the discrete environment in any successor node s_l of s_k. Hence, by TB_1 and TB_4, it follows that the discrete and continuous environments of node s_i satisfy its control-flow assertion α_i and continuous invariant Ψ_i, respectively. Finally, by TB_2 and TB_3, all discrete and continuous environments encoded in the EFSM satisfy Φ. ■

3.3 VCG using SCC Assertions

The main idea behind the SCC-Path and SCC-Trans methods is to set up induction proofs on the SCCs of the EFSM. A continuous invariant for each node s_i of the EFSM is still required, however, instead of providing each node a control-flow assertion, SCC-Path and SCC-Trans methods require an SCC assertion for each SCC of the EFSM. The SCC assertion restricts the discrete environments of the nodes belongs to that SCC.

3.3.1 The SCC-Path Method

To ensure the validity of a safety property Φ, we have to prove SP_1-SP_5 in Theorem 2. It may generate several VCs, most of which are SafePath-formulas.

Theorem 2 (Correctness of the SCC-Path Method)**.** *Assume an EFSM Σ of a Quartz program $\mathcal{P}$ has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node. Nodes belong to sets $\mathcal{S}_{\mathcal{C}_0}, \ldots, \mathcal{S}_{\mathcal{C}_m}$ for nontrivial SCCs and to $\mathcal{S}_{\mathcal{C}_t}$ for nodes in trivial SCCs, respectively. $\mathsf{pathsTo}(\mathcal{C}_i)$ are paths entering nontrivial SCC $\mathcal{S}_{\mathcal{C}_i}$ and $\mathsf{inside}(\mathcal{C}_i)$ are transitions moving inside $\mathcal{S}_{\mathcal{C}_i}$. Given SCC assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_t$, for each nontrivial SCC $\mathcal{C}_0, \ldots, \mathcal{C}_m$ and for $\mathcal{C}_t$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, then a property Φ holds in all SCCs if the following holds:*

SP_1 : $\mathcal{D}(s_0) \rightarrow \mathcal{I}_t$

SP_2 : $\mathcal{I}_i \rightarrow \Phi$ *for all* $\mathcal{I}_0, \ldots, \mathcal{I}_m$

SP_3 : $\mathcal{I}_t \rightarrow \Phi$

SP_4 : $\Psi_i \rightarrow \Phi$ *for all* $\Psi_0, \ldots, \Psi_n$

SP_5 : $\mathsf{SafePath}\big((s_k, \ldots, s_l), (\Psi_k, \ldots, \Psi_{l-1}), \mathcal{I}_{s_k}, \mathcal{I}_t, \mathcal{I}_l\big)$ *with* $\mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_t, & \text{if } s_k = s_0 \\ \mathcal{I}_i, & \text{otherwise} \end{cases}$
for each nontrivial SCC $\mathcal{C}_j$
and all $(s_k, \ldots, s_l) \in \mathsf{pathsTo}(\mathcal{C}_j)$ *where* $s_l \in \mathcal{S}_{\mathcal{C}_j}$ *and* $s_k \in \mathcal{S}_{\mathcal{C}_i} \cup \{s_0\}$

SP_6 : $\mathsf{SafePath}\big((s_k, s_l), (\Psi_k), \mathcal{I}_j, true, \mathcal{I}_j\big)$
for each nontrivial SCC $\mathcal{C}_j$ *and all* $(s_k, s_l) \in \mathsf{inside}(\mathcal{C}_j)$

Proof. By construction of the EFSM, the root node is always a trivial SCC. By SP_1 and SP_3, Φ holds on the discrete environment of the root node. By SP_6, the nontrivial assertion $\mathcal{I}_j$ holds on all discrete environments inside nontrivial SCC $\mathcal{C}_j$ and continuous invariants are valid for the corresponding nodes inside nontrivial SCC $\mathcal{C}_j$ (induction step) provided that it holds on the discrete environment when entering the nontrivial SCC $\mathcal{C}_j$

(induction base). The latter is ensured by SP_5, which describes that the entered discrete states satisfy the assertion of their SCC. By SP_5 and SP_6, all discrete environments of nontrivial SCCs satisfy the assertions of their SCCs, and the continuous environments in the nodes belong to nontrivial SCCs that satisfy the continuous invariants of their nodes. By SP_1 and SP_5, all the discrete state of trivial SCCs satisfy $\mathcal{I}_t$, while the continuous environments of trivial SCCs satisfy the continuous invariants of their nodes. Thus, by SP_2, SP_3, and SP_4, all discrete and continuous environments encoded in the EFSM satisfy Φ. ■

3.3.2 The SCC-Trans Method

The SCC-Path method sets up induction proofs over nontrivial SCCs. To prove the induction bases, we have to consider paths that may contain more than one transition. In the worst case, to prove the induction bases the same transition that forms paths leading to different nontrivial SCCs would be checked exponentially many times (in the end) by the SAT/SMT solvers. Therefore, we propose the SCC-Trans method as an improvement. To ensure the validity of a safety property Φ, we have to prove ST_1-ST_5 in Theorem 3. It may generate several VCs, most of which are SafeTrans-formulas.

Theorem 3 (Correctness of the SCC-Trans Method)**.** *Assume an EFSM Σ of a Quartz program $\mathcal{P}$ has nodes $s_0, \ldots, s_n$, such that s_0 is the initial node. Nodes belong to sets $\mathcal{S}_{\mathcal{C}_0}, \ldots, \mathcal{S}_{\mathcal{C}_m}$ for nontrivial SCCs and to $\mathcal{S}_{\mathcal{C}_t}$ for nodes in trivial SCCs, respectively. $\mathsf{transTo}(\mathcal{C}_i)$, $\mathsf{inside}(\mathcal{C}_i)$ and $\mathsf{transTo}(\mathcal{C}_t)$, are transitions entering nontrivial SCC $\mathcal{S}_{\mathcal{C}_i}$, moving inside $\mathcal{S}_{\mathcal{C}_i}$, and reaching the nodes in $\mathcal{S}_{\mathcal{C}_t}$, respectively. Given SCC assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_t$ for each nontrivial SCC $\mathcal{C}_0, \ldots, \mathcal{C}_m$ and for $\mathcal{C}_t$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, then a property Φ holds in all SCCs if the following holds:*

$$
\begin{aligned}
&\mathsf{ST}_1: \quad \mathcal{D}(s_0) \rightarrow \mathcal{I}_t \\
&\mathsf{ST}_2: \quad \mathcal{I}_i \rightarrow \Phi \quad \textit{for all} \quad \mathcal{I}_0, \ldots, \mathcal{I}_m \\
&\mathsf{ST}_3: \quad \mathcal{I}_t \rightarrow \Phi \\
&\mathsf{ST}_4: \quad \Psi_i \rightarrow \Phi \quad \textit{for all} \quad \Psi_0, \ldots, \Psi_n \\
&\mathsf{ST}_5: \quad \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_{s_k}, \Psi_k, \mathcal{I}_t\big) \textit{ with } \mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_t, & \textit{if } s_k \in \mathcal{S}_{\mathcal{C}_t} \\ \mathcal{I}_i, & \textit{otherwise} \end{cases} \\
&\qquad \textit{for all } (s_k, s_l) \in \mathsf{transTo}(\mathcal{C}_t) \textit{ where } s_k \in \mathcal{S}_{\mathcal{C}_i} \cup \mathcal{S}_{\mathcal{C}_t} \textit{ and } s_l \in \mathcal{S}_{\mathcal{C}_t} \\
&\mathsf{ST}_6: \quad \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_{s_k}, \Psi_k, \mathcal{I}_j\big) \textit{ with } \mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_t, & \textit{if } s_k \in \mathcal{S}_{\mathcal{C}_t} \\ \mathcal{I}_i, & \textit{otherwise} \end{cases} \\
&\qquad \textit{for each nontrivial SCC } \mathcal{C}_j \\
&\qquad \textit{and all } (s_k, s_l) \in \mathsf{transTo}(\mathcal{C}_j) \textit{ where } s_k \in \mathcal{S}_{\mathcal{C}_i} \cup \mathcal{S}_{\mathcal{C}_t} \textit{ and } s_l \in \mathcal{S}_{\mathcal{C}_j} \\
&\mathsf{ST}_7: \quad \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j\big) \\
&\qquad \textit{for each nontrivial SCC } \mathcal{C}_j \textit{ and all } (s_k, s_l) \in \mathsf{inside}(\mathcal{C}_j)
\end{aligned}
$$

Proof. By ST_1 and ST_3, Φ holds on the discrete environment of the root node. By ST_7, the nontrivial SCC assertion $\mathcal{I}_j$ holds on all discrete environments inside $\mathcal{C}_j$ and continuous invariants are valid for the corresponding nodes inside $\mathcal{C}_j$ (induction step) provided that it holds on the discrete environment when entering $\mathcal{C}_j$ (induction base). The latter is ensured by ST_6, which describes that the entered discrete states satisfy the assertion of their SCC. By ST_1, ST_6 and ST_7, all discrete environments in nontrivial SCCs satisfy the assertions of their SCCs, and the continuous environments belongs to nontrivial SCCs satisfy the continuous invariants of their nodes. By ST_1 and ST_5, all discrete environments of trivial SCCs satisfy $\mathcal{I}_t$, while the continuous environments of trivial SCCs satisfy the continuous invariants of their nodes. Thus, by ST_2, ST_3, and ST_4, all discrete and continuous environments encoded in the EFSM satisfy Φ. ■

3.4 VCG using Loop Assertions

SCC-Trans and SCC-Path require assertions for discrete environments of SCCs. However, the number of SCCs in the EFSM can grow exponentially with the size of its Quartz program. As a consequence, in the worst case we may have to set up exponentially many induction proofs, while proving assertions for each loop statement in the Quartz program only requires a linear number of these induction proofs.

Continuous invariants of EFSM nodes are still needed for the Loop-Trans and the Loop-Path methods. Additionally, the loop-based methods require an assertion for each loop statement $\mathcal{L}_i$ in the Quartz program, by annotating the program code. A loop assertion $\mathcal{I}_i$ restricts the discrete environments of the nodes whose control-flow variables are contained inside $\mathcal{L}_i$. Moreover, the user has to provide an assertion $\mathcal{I}_\emptyset$ for nodes in $\mathcal{S}_{\mathcal{L}_\emptyset}$. If node s_i refers to more than one loop statement, i.e., $\mathsf{Loops}(s_i)$ contains more than one element, then the discrete environment of node s_i is restricted by the conjunction of all related loop assertions.

3.4.1 The Loop-Path Method

To prove the validity of a safety property Φ, we can prove LP_1-LP_6 in Theorem 4 by the Loop-Path method. It may generate several VCs, most of which are SafePath-formulas.

Theorem 4 (Correctness of the Loop-Path Method)**.** *For any Quartz program $\mathcal{P}$ with loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$, assume that its EFSM Σ has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, the nodes belong to sets $\mathcal{S}_{\mathcal{L}_0}, \ldots, \mathcal{S}_{\mathcal{L}_m}$ and $\mathcal{S}_{\mathcal{L}_\emptyset}$, $\mathsf{pathsTo}(\mathcal{L}_i)$ are paths entering $\mathcal{S}_{\mathcal{L}_i}$, and $\mathsf{inside}(\mathcal{L}_i)$ are transitions moving inside $\mathcal{S}_{\mathcal{L}_i}$. Given assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_\emptyset$ for loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$ and $\mathcal{L}_\emptyset$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, then a property Φ holds in all EFSM nodes if the following holds:*

LP_1 : $\mathcal{D}(s_0) \rightarrow \mathcal{I}_\emptyset$
LP_2 : $\mathcal{I}_i \rightarrow \Phi$ *for all* $\mathcal{I}_0, \ldots, \mathcal{I}_m$
LP_3 : $\mathcal{I}_\emptyset \rightarrow \Phi$
LP_4 : $\Psi_i \rightarrow \Phi$ *for all* $\Psi_0, \ldots, \Psi_n$

LP_5 : $\mathsf{SafePath}\big((s_k,\ldots,s_l),(\Psi_k,\ldots,\Psi_{l-1}),\mathcal{I}_{s_k},\mathcal{I}_\emptyset,\mathcal{I}_j\big)$
with $\mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_\emptyset, & \text{if } s_k \in \mathcal{S}_{\mathcal{L}_\emptyset} \\ \bigwedge\limits_{\mathcal{L}_i \in \mathsf{Loops}(s_k)} \mathcal{I}_i, & \text{otherwise} \end{cases}$
for each loop statement $\mathcal{L}_j$ *and all* $(s_k,s_l) \in \mathsf{pathsTo}(\mathcal{L}_j)$ *where* $s_l \in \mathcal{S}_{\mathcal{L}_j}$

LP_6 : $\mathsf{SafePath}\big((s_k,s_l),(\Psi_k),\mathcal{I}_j,true,\mathcal{I}_j\big)$
for each loop statement $\mathcal{L}_j$ *and all* $(s_k,s_l) \in \mathsf{inside}(\mathcal{L}_j)$

Proof. By construction of the EFSM, the root node always belongs to $\mathcal{S}_{\mathcal{L}_\emptyset}$. By LP_1 and LP_3, Φ holds on the discrete environment of the root node. By LP_6, the assertion $\mathcal{I}_j$ holds on all discrete environments inside $\mathcal{S}_{\mathcal{L}_j}$ and continuous invariants are valid for the corresponding nodes inside $\mathcal{S}_{\mathcal{L}_j}$ (induction step) provided that it holds on the discrete environment entering $\mathcal{S}_{\mathcal{L}_j}$ (induction base). The latter is ensured by LP_5 which describes that the entered discrete states satisfy the related loop assertion(s). By LP_5 and LP_6, all discrete environments related to loop statements satisfy their loop assertions, while continuous environments in the nodes in $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy the continuous invariants of their nodes. By LP_1 and LP_5, the discrete environments in $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy $\mathcal{I}_\emptyset$, while continuous environments in the nodes belonging to $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy the continuous invariants of their nodes. Hence, by LP_2, LP_3, and LP_4, all discrete and continuous environments encoded in the EFSM satisfy Φ. ■

3.4.2 The Loop-Trans Method

To prove the validity of a safety property Φ, we can prove LT_1-LT_7 in Theorem 5 by the Loop-Trans method. It may generate several VCs, most of which are SafeTrans-formulas.

Theorem 5 (Correctness of the Loop-Trans Method)**.** *For any Quartz program* $\mathcal{P}$ *with loop statements* $\mathcal{L}_0,\ldots,\mathcal{L}_m$*, assume that its EFSM* Σ *has nodes* $s_0,\ldots,s_n$ *such that* s_0 *is the initial node, the nodes belong to sets* $\mathcal{S}_{\mathcal{L}_0},\ldots,\mathcal{S}_{\mathcal{L}_m}$ *and* $\mathcal{S}_{\mathcal{L}_\emptyset}$*, and* $\mathsf{pathsTo}(\mathcal{L}_i)$*,* $\mathsf{inside}(\mathcal{L}_i)$ *and* $\mathsf{transTo}(\mathcal{L}_\emptyset)$ *are transitions entering* $\mathcal{S}_{\mathcal{L}_i}$*, moving inside* $\mathcal{S}_{\mathcal{L}_i}$*, and reaching* $\mathcal{S}_{\mathcal{L}_\emptyset}$*, respectively. Given assertions* $\mathcal{I}_0,\ldots,\mathcal{I}_m$ *and* $\mathcal{I}_\emptyset$*, for loop statements* $\mathcal{L}_0,\ldots,\mathcal{L}_m$ *and* $\mathcal{L}_\emptyset$*, respectively, and continuous invariants* $\Psi_0,\ldots,\Psi_n$ *for each node* $s_0,\ldots,s_n$*, respectively, then a property* Φ *holds in all EFSM nodes if the following holds:*

LT_1 : $\mathcal{D}(s_0) \to \mathcal{I}_\emptyset$
LT_2 : $\mathcal{I}_i \to \Phi$ *for all* $\mathcal{I}_0,\ldots,\mathcal{I}_m$
LT_3 : $\mathcal{I}_\emptyset \to \Phi$
LT_4 : $\Psi_i \to \Phi$ *for all* $\Psi_0,\ldots,\Psi_n$
LT_5 : $\mathsf{SafeTrans}\big((s_k,s_l),\mathcal{I}_{s_k},\Psi_k,\mathcal{I}_\emptyset\big)$ *with* $\mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_\emptyset, & \text{if } s_k \in \mathcal{S}_{\mathcal{L}_\emptyset} \\ \bigwedge\limits_{\mathcal{L}_i \in \mathsf{Loops}(s_k)} \mathcal{I}_i, & \text{otherwise} \end{cases}$
for all $(s_k,s_l) \in \mathsf{transTo}(\mathcal{L}_\emptyset)$

$$\mathsf{LT}_6: \quad \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_{s_k}, \Psi_k, \mathcal{I}_j\big) \text{ with } \mathcal{I}_{s_k} := \begin{cases} \mathcal{I}_\emptyset, & \text{if } s_k \in \mathcal{S}_{\mathcal{L}_\emptyset} \\ \bigwedge\limits_{\mathcal{L}_i \in \mathsf{Loops}(s_k)} \mathcal{I}_i, & \text{otherwise} \end{cases}$$
for each loop statement $\mathcal{L}_j$ *and all* $(s_k, s_l) \in \mathsf{transTo}(\mathcal{L}_j)$ *where* $s_l \in \mathcal{S}_{\mathcal{L}_j}$

$$\mathsf{LT}_7: \quad \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j\big)$$
for each loop statement $\mathcal{L}_j$ *and all* $(s_k, s_l) \in \mathsf{inside}(\mathcal{L}_j)$

Proof. By construction of the EFSM, the root node always belongs to $\mathcal{S}_{\mathcal{L}_\emptyset}$. By LT_1 and LT_3, Φ holds on the discrete environment of the root node. By LT_7, the assertion $\mathcal{I}_j$ holds on all discrete environments inside $\mathcal{S}_{\mathcal{L}_j}$ and continuous invariants are valid for the corresponding nodes inside $\mathcal{S}_{\mathcal{L}_j}$ (induction step) provided that it holds on the discrete environment entering $\mathcal{S}_{\mathcal{L}_j}$ (induction base). The latter is ensured by LT_6 which describes that the entering discrete states satisfy the related loop assertions. By LT_6 and LT_7, all discrete environments related to loop statements satisfy their loop assertions, while continuous environments in the nodes not in $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy the continuous invariants of their nodes. By LT_1 and LT_5, the discrete environments in $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy $\mathcal{I}_\emptyset$, while continuous environments in the nodes belonging to $\mathcal{S}_{\mathcal{L}_\emptyset}$ satisfy the continuous invariants of their nodes. Hence, by LT_2, LT_3, and LT_4, all discrete and continuous environments encoded in the EFSM satisfy Φ. ■

3.5 Relative Completeness of the VCG Methods

In this section, we prove that the five proposed VCG methods are relatively complete: Whether they can be proved complete depends on the underlying assertion language, hence, we can only prove completeness relative to that.

Proposition 1. *For the proofs, we assume that a formula* $\Phi_\mathcal{R}$ *exists that encodes the reachable states of the EFSM. Thus, this formula will have the following properties* $R1$*,* $R2$*,* $R3$*,* $R4$ *and* $R5$*:*

R1*:* $cont_\forall(\Phi_\mathcal{R})$ *is* valid *for all EFSM nodes.*

R2*: The reachable states encoded by the root node* s_0 *implies* $\Phi_\mathcal{R}$*:*

$$\Big(\sigma(s_0) \ \wedge \ \mathcal{D}_d(s_0) \ \wedge \ cont_\forall(\mathcal{D}_c(s_0))\Big) \ \rightarrow \ \Phi_\mathcal{R}$$

R3*:* $\Phi_\mathcal{R} \wedge \sigma(s_i)$ *encodes all the reachable states in node* s_i*. From any reachable states encoded by* s_i*, with the corresponding guarded actions and transition* (s_i, s_j)*, the discrete states in node* s_j *will be reached, i.e., we demand the following:*

$$\Big(\ [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\boldsymbol{next}(\vec{x})} \ \wedge \ cont_\forall\big(\, [\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \, \big) \ \wedge \ [\mathcal{D}_d(s_j)]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\boldsymbol{next}(\vec{x})} \ \wedge$$

$$[\Phi_\mathcal{R}]^{\vec{x}_0}_{\vec{x}} \ \wedge \ \sigma(s_i) \ \wedge \ [\varphi(s_i, s_j)]^{\vec{x}_0}_{\vec{x}} \ \Big) \ \rightarrow \ \Big(\ [\Phi_\mathcal{R}]^{\vec{x}_1}_{\vec{x}} \ \wedge \ \sigma(s_j) \ \Big)$$

R4: *$\Phi_{\mathcal{R}}$ is the strongest inductive safety property, from which any other safety property Φ can be derived:*

$$\Phi_{\mathcal{R}} \;\rightarrow\; \Phi$$

R5: *$\Phi_{\mathcal{R}}$ implies the control-flow information, all the transitions could only take place among the nodes:*

$$\Phi_{\mathcal{R}} \;\rightarrow\; \bigvee_{i=0}^{n} \sigma(s_i)$$

3.5.1 Relative Completeness of Transition-based

Theorem 6 (Relative Completeness of the Transition-based Method)**.** *Given any Quartz program $\mathcal{P}$ and its EFSM Σ with transitions* $\mathsf{Trans}(\Sigma)$ *and nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, and any valid safety property Φ that holds in all reachable states of Σ, then there exist suitable control-flow assertions $\alpha_0, \ldots, \alpha_n$ and continuous invariants $\Psi_0, \ldots, \Psi_n$ for nodes $s_0, \ldots, s_n$, respectively, to prove Φ as shown in Theorem 1.*

Proof. Assuming the existence of $\Phi_{\mathcal{R}}$, we define $\alpha_i := \Phi_{\mathcal{R}} \wedge \sigma(s_i)$ and $\Psi_i := \Phi_{\mathcal{R}}$ for every node s_i, then all generated subgoals are valid.

- TB_1 is valid since the second formula of the following equivalence is valid by R2 of Proposition 1.

$$\mathcal{D}(s_0) \rightarrow \alpha_0 \;\Leftrightarrow\; \Big(\sigma(s_0) \;\wedge\; \mathcal{D}_d(s_0) \;\wedge\; \mathit{cont}_{\forall}(\mathcal{D}_c(s_0))\Big) \;\rightarrow\; \Big(\Phi_{\mathcal{R}} \;\wedge\; \sigma(s_0)\Big)$$

- TB_2 is valid since the second formula of the following equivalence is valid by R4 of Proposition 1.

$$\alpha_i \rightarrow \Phi \;\Leftrightarrow\; \Big((\Phi_{\mathcal{R}} \;\wedge\; \sigma(s_i)) \rightarrow \Phi\Big)$$

- TB_3 is valid since the second formula of the following equivalence is valid by R4 of Proposition 1.

$$\Psi_i \rightarrow \Phi \;\Leftrightarrow\; \Phi_{\mathcal{R}} \rightarrow \Phi$$

- TB_4 is valid since the following equivalences are valid by R1, R3 of Proposition 1.

$$\mathsf{SafeTrans}\big((s_k, s_l), \alpha_k, \Psi_k, \alpha_l\big)$$

$$\Leftrightarrow\; \Big(\; [\mathcal{D}_d(s_k)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; \mathit{cont}_{\forall}\big(\,[\mathcal{D}_c(s_k)]^{\vec{x}_0}_{\vec{x}}\,\big) \;\wedge\; [\mathcal{D}_d(s_l)]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge$$

$$[\alpha_k]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \sigma(s_k) \;\wedge\; [\varphi(s_k, s_l)]^{\vec{x}_0}_{\vec{x}} \;\Big)$$

$$\rightarrow\; \Big(\; \mathit{cont}_{\forall}([\psi_k]^{\vec{x}_0}_{\vec{x}}) \;\wedge\; [\alpha_l]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \sigma(s_l) \;\Big)$$

$$\Leftrightarrow\; \Big(\; [\mathcal{D}_d(s_k)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; \mathit{cont}_{\forall}\big(\,[\mathcal{D}_c(s_k)]^{\vec{x}_0}_{\vec{x}}\,\big) \;\wedge\; [\mathcal{D}_d(s_l)]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})}$$

$$[\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \sigma(s_k) \;\wedge\; [\varphi(s_k, s_l)]^{\vec{x}_0}_{\vec{x}} \;\Big) \;\rightarrow\; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \sigma(s_l) \;\Big)$$

■

3.5.2 Relative Completeness of SCC-Path

Theorem 7 (Relative Completeness of the SCC-Path Method)**.** *Given any Quartz program $\mathcal{P}$ and its EFSM Σ that has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, assume the nodes belong to sets $\mathcal{S}_{\mathcal{C}_0}, \ldots, \mathcal{S}_{\mathcal{C}_m}$ for nontrivial SCCs and to $\mathcal{S}_{\mathcal{C}_t}$ for trivial SCCs, respectively, that* $\mathsf{pathsTo}(\mathcal{C}_i)$ *are paths entering nontrivial SCC $\mathcal{S}_{\mathcal{C}_i}$ and that* $\mathsf{inside}(\mathcal{C}_i)$ *are transitions moving inside $\mathcal{S}_{\mathcal{C}_i}$. Given any valid safety property Φ that holds in all reachable states of Σ, there exist suitable SCC assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_t$, for each nontrivial SCC $\mathcal{C}_0, \ldots, \mathcal{C}_m$ and $\mathcal{C}_t$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, to prove Φ as shown in Theorem 2.*

Proof. Assuming the existence of $\Phi_{\mathcal{R}}$, we define $\mathcal{I}_i := \Phi_{\mathcal{R}} \ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_i}} \sigma(s_r)$ for every nontrivial SCC $\mathcal{C}_i$, $\mathcal{I}_t := \Phi_{\mathcal{R}} \ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r)$ for nodes in trivial SCCs, and $\Psi_i := \Phi_{\mathcal{R}}$ for every node s_i. Then all generated subgoals are valid:

- SP_1 is valid since the last formula of the following equivalence is valid by R2 of Proposition 1.

$$\mathcal{D}(s_0) \rightarrow \mathcal{I}_t \ \Leftrightarrow \ \Big(\sigma(s_0) \ \wedge \ \mathcal{D}_d(s_0) \ \wedge \ \mathit{cont}_{\forall}(\mathcal{D}_c(s_0))\Big) \ \rightarrow \ \Big(\Phi_{\mathcal{R}} \ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r)\Big)$$

- SP_2 is valid since the second formula of the following equivalence is valid by R4 and R5 of Proposition 1.

$$\mathcal{I}_i \rightarrow \Phi \ \Leftrightarrow \ \Big(\Phi_{\mathcal{R}} \ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_i}} \sigma(s_r)\Big) \ \rightarrow \ \Phi$$

- SP_3 is valid since the second formula of the following equivalence is valid by R4 and R5 of Proposition 1.

$$\mathcal{I}_t \rightarrow \Phi \ \Leftrightarrow \ \Big(\Phi_{\mathcal{R}} \ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r)\Big) \ \rightarrow \ \Phi$$

- SP_4 is valid since the second formula of the following equivalence is valid by R4 of Proposition 1.

$$\Psi_i \rightarrow \Phi \ \Leftrightarrow \ \Phi_{\mathcal{R}} \rightarrow \Phi$$

- By the relation of SafePath and SafeTrans predicates, we can rewrite SP_5 as follows:

$$\mathsf{SafePath}\big((s_k, \ldots, s_l), (\Psi_k, \ldots, \Psi_{l-1}), \mathcal{I}_{s_k}, \mathcal{I}_t, \mathcal{I}_l\big)$$

$$\Leftrightarrow \quad \left(\begin{array}{l} \mathsf{SafeTrans}\big((s_k, s_{k+1}), \mathcal{I}_{s_k}, \Psi_k, \mathcal{I}_t\big) \\ \wedge \quad \left(\bigwedge_{i=k+1}^{l-2} \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_t, \Psi_i, \mathcal{I}_t\big) \right) \\ \wedge \quad \mathsf{SafeTrans}\big((s_{l-1}, s_l), \mathcal{I}_t, \Psi_{l-1}, \mathcal{I}_l\big) \end{array} \right)$$

Therefore, we only have to prove the validity of the following four general cases:

– SP_{51}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_i, \Psi_i, \mathcal{I}_t\big)$
– SP_{52}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_t, \Psi_i, \mathcal{I}_t\big)$
– SP_{53}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_t, \Psi_i, \mathcal{I}_{i+1}\big)$
– SP_{54}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_i, \Psi_i, \mathcal{I}_{i+1}\big)$

which can be done by equivalences with R1, R3 and R5 of Proposition 1:

$$
\begin{aligned}
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_i, \Psi_i, \mathcal{I}_t\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \qquad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_i}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \qquad \rightarrow \; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r) \; \Big) \\
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_t, \Psi_i, \mathcal{I}_t\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \qquad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \qquad \rightarrow \; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r) \; \Big) \\
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_t, \Psi_i, \mathcal{I}_{i+1}\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \qquad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \qquad \rightarrow \; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_{i+1}}} \sigma(s_r) \; \Big) \\
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_i, \Psi_i, \mathcal{I}_{i+1}\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \qquad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_i}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \qquad \rightarrow \; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_{i+1}}} \sigma(s_r) \; \Big)
\end{aligned}
$$

- Notice that SP_6 can be rewritten as the following $\mathsf{SafeTrans}$-formula SP_7:

$$
\begin{aligned}
\mathsf{SP}_7 : \quad & \mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j\big) \\
& \text{for each nontrivial SCC } \mathcal{C}_j \text{ and all } (s_k, s_l) \in \mathsf{inside}(\mathcal{C}_j)
\end{aligned}
$$

and SP_7 is valid since the second formula of the following equivalence is valid by R1, R3 and R5 of Proposition 1:

$$
\begin{aligned}
& \mathsf{SafeTrans}((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_k)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big(\, [\mathcal{D}_c(s_k)]^{\vec{x}_0}_{\vec{x}} \, \big) \;\wedge\; [\mathcal{D}_d(s_l)]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \quad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}} \;\wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_j}} \sigma(s_r) \;\wedge\; [\varphi(s_k, s_l)]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \rightarrow \; \Big(\; [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}} \;\wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_j}} \sigma(s_r) \; \Big)
\end{aligned}
$$

■

3.5.3 Relative Completeness of SCC-Trans

Theorem 8 (Relative Completeness of the SCC-Trans Method)**.** *Given any Quartz program $\mathcal{P}$ and its EFSM Σ that has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, assume the nodes belong to sets $\mathcal{S}_{\mathcal{C}_0}, \ldots, \mathcal{S}_{\mathcal{C}_m}$ for nontrivial SCCs and to $\mathcal{S}_{\mathcal{C}_t}$ for trivial SCCs, respectively, and that $\mathsf{transTo}(\mathcal{C}_i)$, $\mathsf{inside}(\mathcal{C}_i)$ and $\mathsf{transTo}(\mathcal{C}_t)$, are transitions entering nontrivial SCC $\mathcal{S}_{\mathcal{C}_i}$, moving inside $\mathcal{S}_{\mathcal{C}_i}$, and reaching the nodes in $\mathcal{S}_{\mathcal{C}_t}$, respectively. Given any valid safety property Φ that holds in all reachable states of Σ, there exist suitable SCC assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_t$, for each nontrivial SCC $\mathcal{C}_0, \ldots, \mathcal{C}_m$ and $\mathcal{C}_t$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, to prove Φ as shown in Theorem 3.*

Proof. Assuming the existence of $\Phi_{\mathcal{R}}$, we define $\mathcal{I}_i := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_i}} \sigma(s_r)$ for every nontrivial SCC $\mathcal{C}_i$, $\mathcal{I}_t := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{C}_t}} \sigma(s_r)$ for the nodes in $\mathcal{C}_t$, and $\Psi_i := \Phi_{\mathcal{R}}$ for every node s_i, then all generated subgoals are valid:

- ST_1-ST_4 are valid following the same arguments as for SP_1-SP_4.
- ST_5-ST_7 are valid since the proof of SP_5-SP_6 has covered all the possible general cases, i.e., SP_{51}, SP_{52}, SP_{53}, SP_{54} and SP_7.

■

3.5.4 Relative Completeness of Loop-Path

Theorem 9 (Relative Completeness of the Loop-Path Method)**.** *Given any Quartz program $\mathcal{P}$ with loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$, assume that its EFSM Σ has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, nodes belongs to sets $\mathcal{S}_{\mathcal{L}_0}, \ldots, \mathcal{S}_{\mathcal{L}_m}$ and $\mathcal{S}_{\mathcal{L}_\emptyset}$ accordingly, $\mathsf{pathsTo}(\mathcal{L}_i)$ are paths entering $\mathcal{S}_{\mathcal{L}_i}$, and $\mathsf{inside}(\mathcal{L}_i)$ are transitions moving inside $\mathcal{S}_{\mathcal{L}_i}$. Given any valid safety property Φ that holds in all reachable states of Σ, there exist suitable assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_\emptyset$, for loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$ and $\mathcal{L}_\emptyset$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, to prove Φ as shown in Theorem 4.*

Proof. Assuming the existence of $\Phi_{\mathcal{R}}$, we define $\mathcal{I}_i := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_i}} \sigma(s_r)$ for every set of nodes $\mathcal{S}_{\mathcal{L}_i}$, and $\mathcal{I}_\emptyset := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r)$ for $\mathcal{S}_{\mathcal{L}_\emptyset}$, and $\Psi_i := \Phi_{\mathcal{R}}$ for every node s_i, then all generated subgoals are valid.

- LP_1 is valid since the last formula of the following equivalences is valid by R2 of Proposition 1.

$$\mathcal{D}(s_0) \to \mathcal{I}_\emptyset \;\Leftrightarrow\; \Big(\sigma(s_0) \;\wedge\; \mathcal{D}_d(s_0) \;\wedge\; \mathit{cont}_\forall(\mathcal{D}_c(s_0))\Big) \;\to\; \Big(\Phi_{\mathcal{R}} \;\wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r)\Big)$$

- LP_2 is valid since the second formula of the following equivalence is valid by R4 and R5 of Proposition 1.

$$\mathcal{I}_i \to \Phi \;\Leftrightarrow\; \Big(\Phi_{\mathcal{R}} \;\wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_i}} \sigma(s_r)\Big) \;\to\; \Phi$$

- LP_3 is valid since the second formula of the following equivalence is valid by R4 and R5 of Proposition 1.

$$\mathcal{I}_\emptyset \to \Phi \;\Leftrightarrow\; \Big(\Phi_{\mathcal{R}} \;\wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r)\Big) \;\to\; \Phi$$

- LP_4 is valid since the second formula of the following equivalence is valid by R4 of Proposition 1.

$$\Psi_i \to \Phi \;\Leftrightarrow\; \Phi_{\mathcal{R}} \to \Phi$$

- By the relation of $\mathsf{SafePath}$ and $\mathsf{SafeTrans}$ predicates, we can rewrite LP_5 as follows:

$$\mathsf{SafePath}\big((s_k, \ldots, s_l), (\Psi_k, \ldots, \Psi_{l-1}), \mathcal{I}_{s_k}, \mathcal{I}_\emptyset, \mathcal{I}_l\big)$$

$$\Leftrightarrow \left(\begin{array}{l} \mathsf{SafeTrans}\big((s_k, s_{k+1}), \mathcal{I}_{s_k}, \Psi_k, \mathcal{I}_\emptyset\big) \\ \wedge \left(\bigwedge_{i=k+1}^{l-2} \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_\emptyset, \Psi_i, \mathcal{I}_\emptyset\big)\right) \\ \wedge \mathsf{SafeTrans}\big((s_{l-1}, s_l), \mathcal{I}_\emptyset, \Psi_{l-1}, \mathcal{I}_l\big) \end{array}\right)$$

  Therefore, we have to prove the validity of the following four general cases:
  - LP_{51}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \mathcal{I}_j, \Psi_i, \mathcal{I}_\emptyset\big)$
  - LP_{52}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_\emptyset, \Psi_i, \mathcal{I}_\emptyset\big)$
  - LP_{53}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_\emptyset, \Psi_i, \mathcal{I}_{i+1}\big)$
  - LP_{54}: $\mathsf{SafeTrans}\big((s_i, s_{i+1}), \bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \mathcal{I}_j, \Psi_i, \mathcal{I}_{i+1}\big)$

which can be done by the following equivalences with R1, R3 and R5 of Proposition 1:

$$
\begin{aligned}
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \mathcal{I}_j, \Psi_i, \mathcal{I}_\emptyset\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \quad [\Phi_\mathcal{R}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \Big(\bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_j}} \sigma(s_r) \Big) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \quad \rightarrow \; \Big(\; [\Phi_\mathcal{R}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r) \; \Big)
\end{aligned}
$$

$$
\begin{aligned}
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_\emptyset, \Psi_i, \mathcal{I}_\emptyset\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \quad [\Phi_\mathcal{R}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \quad \rightarrow \; \Big(\; [\Phi_\mathcal{R}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r) \; \Big)
\end{aligned}
$$

$$
\begin{aligned}
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \mathcal{I}_\emptyset, \Psi_i, \mathcal{I}_{i+1}\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \quad [\Phi_\mathcal{R}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \quad \rightarrow \; \Big(\; [\Phi_\mathcal{R}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_{i+1}}} \sigma(s_r) \; \Big)
\end{aligned}
$$

$$
\begin{aligned}
* \quad & \mathsf{SafeTrans}\big((s_i, s_{i+1}), \bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \mathcal{I}_j, \Psi_i, \mathcal{I}_{i+1}\big) \\
\Leftrightarrow \quad & \Big(\; [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})} \;\wedge\; cont_\forall\big([\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}} \big) \;\wedge\; [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
& \quad [\Phi_\mathcal{R}]^{\vec{x}_0}_{\vec{x}} \;\wedge\; \Big(\bigwedge_{\mathcal{L}_j \in \mathsf{Loops}(s_i)} \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_j}} \sigma(s_r) \Big) \;\wedge\; [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}} \; \Big) \\
& \quad \rightarrow \; \Big(\; [\Phi_\mathcal{R}]^{\vec{x}_1}_{\vec{x}} \;\wedge\; \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_{i+1}}} \sigma(s_r) \; \Big)
\end{aligned}
$$

- Notice that LP_6 can be rewritten as the following $\mathsf{SafeTrans}$-formula LP_7:

LP_7 : $\mathsf{SafeTrans}\big((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j\big)$
for each loop statement $\mathcal{L}_j$ and all $(s_k, s_l) \in \mathsf{inside}(\mathcal{L}_j)$

and LP_7 is valid since the second formula of the following equivalence is valid by R1, R3 and R5 of Proposition 1:

$$\begin{aligned}
&\mathsf{SafeTrans}((s_k, s_l), \mathcal{I}_j, \Psi_k, \mathcal{I}_j) \\
\Leftrightarrow\ &\Big(\ [\mathcal{D}_d(s_i)]^{\vec{x}_0,\vec{x}_1}_{\vec{x},\mathbf{next}(\vec{x})}\ \wedge\ cont_\forall(\,[\mathcal{D}_c(s_i)]^{\vec{x}_0}_{\vec{x}}\,)\ \wedge\ [\mathcal{D}_d(s_{i+1})]^{\vec{x}_1,\vec{x}_2}_{\vec{x},\mathbf{next}(\vec{x})} \\
&\quad [\Phi_{\mathcal{R}}]^{\vec{x}_0}_{\vec{x}}\ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_j}} \sigma(s_r)\ \wedge\ [\varphi(s_i, s_{i+1})]^{\vec{x}_0}_{\vec{x}}\ \Big) \\
&\quad \rightarrow\ \Big(\ [\Phi_{\mathcal{R}}]^{\vec{x}_1}_{\vec{x}}\ \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_j}} \sigma(s_r)\ \Big)
\end{aligned}$$

■

3.5.5 Relative Completeness of Loop-Trans

Theorem 10 (Relative Completeness of the Loop-Trans Method)**.** *Given any Quartz program $\mathcal{P}$ with loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$, assume that its EFSM Σ has nodes $s_0, \ldots, s_n$ such that s_0 is the initial node, nodes belong to sets $\mathcal{S}_{\mathcal{L}_0}, \ldots, \mathcal{S}_{\mathcal{L}_m}$ and $\mathcal{S}_{\mathcal{L}_\emptyset}$ accordingly, $\mathsf{pathsTo}(\mathcal{L}_i)$, $\mathsf{inside}(\mathcal{L}_i)$ and $\mathsf{transTo}(\mathcal{L}_\emptyset)$, are transitions entering $\mathcal{S}_{\mathcal{L}_i}$, moving inside $\mathcal{S}_{\mathcal{L}_i}$, and reaching $\mathcal{S}_{\mathcal{L}_\emptyset}$, respectively. Given any valid safety property Φ that holds in all reachable states of Σ, there exist suitable assertions $\mathcal{I}_0, \ldots, \mathcal{I}_m$ and $\mathcal{I}_\emptyset$, for loop statements $\mathcal{L}_0, \ldots, \mathcal{L}_m$ and $\mathcal{L}_\emptyset$, respectively, and continuous invariants $\Psi_0, \ldots, \Psi_n$ for each node $s_0, \ldots, s_n$, respectively, to prove Φ as shown in Theorem 4.*

Proof. Assuming the existence of $\Phi_{\mathcal{R}}$, we define $\mathcal{I}_i := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_i}} \sigma(s_r)$ for every set of nodes $\mathcal{S}_{\mathcal{L}_i}$, and $\mathcal{I}_\emptyset := \Phi_{\mathcal{R}} \wedge \bigvee_{s_r \in \mathcal{S}_{\mathcal{L}_\emptyset}} \sigma(s_r)$ for $\mathcal{S}_{\mathcal{L}_\emptyset}$, and $\Psi_i := \Phi_{\mathcal{R}}$ for every node s_i, then all generated subgoals are valid.

- LT_1-LT_4 are valid following the same arguments as for LP_1-LP_4.
- LT_5-LT_7 are valid since the proof of LP_5-LP_6 has covered all the possible general cases, i.e., LP_{51}, LP_{52}, LP_{53}, LP_{54} and LP_7.

■

Chapter 4

Control-flow Guided Property Directed Reachability Optimizations

Contents

Property Directed Reachability (PDR) tries to prove a safety property holds on all reachable states by means of induction. Even being valid, safety properties may not be provable by induction due to Counterexamples to Induction (CTIs) that result from the over-approximation of reachable states. The main idea of PDR is to incrementally learn from those CTIs and exclude them together with many more unreachable states from consideration in each propagation step. In the worst case, PDR may have to compute the state sets that are also computed by a forward fixpoint computation of the reachable states. In the best case, it may just check one induction base and step.

Imperative synchronous languages have a clear distinction between the control- and dataflow. The control-flow information is not needed for synthesis and is therefore not explicitly encoded in the generated systems, but it can be derived from the original synchronous programs and used for verification. In this section, PDR is optimized by effectively using this control-flow information:

- Before calling the PDR method, it is beneficial to enhance the original transition relation with additional control-flow information that contains invariants about the reachable control-flow states.
- After calling the PDR method, the control-flow helps PDR not only decide about the unreachability of states derived from counterexamples, but also generate more general clauses to refine the over-approximations.

The PDR method has already been implemented and optimized in many ways. Different implementations yield different proofs with different CTIs as intermediate steps. Without loss of generality, we ignore the higher data types and use some extreme cases, i.e., Quartz programs with only boolean variables, to demonstrate the control-flow guided PDR method. All control-flow variables are boolean, and the optimizations inspired by the control-flow have no impact on the dataflow. Therefore, we can apply this control-flow guided method to any state-based languages with clear distinction of control- and dataflow. However, this is beyond the scope of this thesis.

4.1 The Synchronous Product of Transition Systems

Given any synchronous Quartz program $\mathcal{P}$, we have the following lemma that describes the relations of its state transition systems.

Lemma 1 (State Transition Systems of a Synchronous Quartz Program)**.** *For states s, s' of the state transition systems $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{T})$, $\mathcal{K}^{\mathsf{cf}} = (\mathcal{V}, \mathcal{I}^{\mathsf{cf}}, \mathcal{T}^{\mathsf{cf}})$ and $\mathcal{K}^{\mathsf{df}} = (\mathcal{V}, \mathcal{I}^{\mathsf{df}}, \mathcal{T}^{\mathsf{df}})$ the following holds:*

- *There is a transition $s \to s'$ in $\mathcal{K}$ iff this transition exists both in $\mathcal{K}^{\mathsf{cf}}$ and $\mathcal{K}^{\mathsf{df}}$.*
- *State s' can be reached from state s, i.e., $s \to^* s'$ in $\mathcal{K}$ iff it can be reached from state s both in $\mathcal{K}^{\mathsf{cf}}$ and $\mathcal{K}^{\mathsf{df}}$.*
- *If $s \to^* s'$ in $\mathcal{K}^{\mathsf{cf}}$, then we also have $s \to^* s''$ in $\mathcal{K}^{\mathsf{cf}}$ for every state s'' with $s' \cap \mathcal{V}^{\mathsf{cf}} = s'' \cap \mathcal{V}^{\mathsf{cf}}$.*

The proof of the first proposition is straightforward since $\mathcal{K}$ is the synchronous product of the transition systems $\mathcal{K}^{\mathsf{cf}}$ and $\mathcal{K}^{\mathsf{df}}$, i.e., $\mathcal{K} = \mathcal{K}^{\mathsf{cf}} \times \mathcal{K}^{\mathsf{df}}$. The second one is proved by induction using the first proposition. For the third proposition, note that $\mathcal{T}^{\mathsf{cf}}$ has been constructed by guarded actions that do not constrain the next values of the dataflow variables — only the values of the control-flow variables in state s' are determined by the values of the variables in state s; the dataflow variables are completely unconstrained.

4.2 Transition Relation Modification

Counterexamples to Induction (CTIs) are unreachable states that satisfy Φ and that also have successor states violating Φ, therefore, using a better approximation of the

reachable states is alway welcomed by the PDR method. Predicates $\mathsf{ReachCF}(\mathcal{P})$ and $\mathsf{InvarCF}(\mathcal{P})$ are encoded with only control-flow variables that approximate the reachable control-flow states of a synchronous Quartz program $\mathcal{P}$. They contain useful information about the unreachability of states that PDR otherwise has to prove first. We can avoid this proof overhead by simply adding either $\mathsf{ReachCF}(\mathcal{P})$ or $\mathsf{InvarCF}(\mathcal{P})$ to the original transition relation $\mathcal{T}$ and thus using $\mathcal{T}' := \mathcal{T} \wedge \mathsf{ReachCF}(\mathcal{P})$ or $\mathcal{T}'' := \mathcal{T} \wedge \mathsf{InvarCF}(\mathcal{P})$ for PDR instead. As will be shown in Section 4.2.3, many safety properties even become inductive only with respect to the enhanced transition relations.

4.2.1 Control-flow Invariant $\mathsf{ReachCF}(\mathcal{P})$ by Fixpoint Computation

The first method to compute the control-flow invariant considers $\mathcal{K}^{\mathsf{abs}}$, which is an abstraction of the control-flow transition system $\mathcal{K}^{\mathsf{cf}}$, and computes its reachable states with a symbolic state space traversal.

Definition 3 (Abstract Control-flow Transition System $\mathcal{K}^{\mathsf{abs}}$)**.** Given an Imperative synchronous program $\mathcal{P}$ with control-flow variables $\mathcal{V}^{\mathsf{cf}} = \{\mathtt{w}_0, \ldots, \mathtt{w}_{m-1}\}$ and dataflow variables $\mathcal{V}^{\mathsf{df}} = \{x_1, \ldots, x_n\}$, and its control-flow transition system $\mathcal{K}^{\mathsf{cf}} = (\mathcal{V}^{\mathsf{cf}} \cup \mathcal{V}^{\mathsf{df}}, \mathcal{I}^{\mathsf{cf}}, \mathcal{T}^{\mathsf{cf}})$, the abstract control-flow transition system $\mathcal{K}^{\mathsf{abs}}$ is defined over $\mathcal{V}^{\mathsf{cf}}$ with the following initial states and transition relation:

- $\mathcal{I}^{\mathsf{abs}} := \exists x_1, \ldots, x_n.\ \mathcal{I}^{\mathsf{cf}}$
- $\mathcal{T}^{\mathsf{abs}} := \exists x_1, \ldots, x_n . \exists x'_1, \ldots, x'_n.\ \mathcal{T}^{\mathsf{cf}}$

The reachable states $\mathsf{ReachCF}(\mathcal{P})$ of $\mathcal{K}^{\mathsf{abs}}$ are defined by the iteration $\mathcal{X}_0^{\mathsf{abs}} := \mathcal{I}^{\mathsf{abs}}$ and $\mathcal{X}_{i+1}^{\mathsf{abs}} := \mathcal{X}_i^{\mathsf{abs}} \cup \mathsf{suc}_{\exists}(\mathcal{X}_i^{\mathsf{abs}})$.

The control-flow transition system $\mathcal{K}^{\mathsf{abs}}$ abstracts the dataflow and just considers the control-flow of a synchronous Quartz program $\mathcal{P}$. It has only finitely many states, and usually also not that many states. These are the corresponding EFSM states that are also often considered for code generation. The above abstraction makes nondeterministic choices on all control-flow expressions that occur in $\mathcal{I}^{\mathsf{cf}}$ and $\mathcal{T}^{\mathsf{cf}}$ so that no variables other than the control-flow variables occur in $\mathcal{I}^{\mathsf{abs}}$ and $\mathcal{T}^{\mathsf{abs}}$. In practice, we can quickly compute the reachable states $\mathsf{ReachCF}(\mathcal{P})$ of $\mathcal{K}^{\mathsf{abs}}$ by means of symbolic model checking using BDDs. And $\mathsf{ReachCF}(\mathcal{P})$ is an over-approximation of the reachable control states of the program $\mathcal{P}$ by the following theorem.

Theorem 11 ($\mathsf{ReachCF}(\mathcal{P})$)**.** $\mathcal{S}_{\mathsf{reach}}$ represents all reachable states of a synchronous Quartz program $\mathcal{P}$. $\mathcal{S}_{\mathsf{reach}}$ implies $\mathsf{ReachCF}(\mathcal{P})$, so that the projection of $\mathcal{S}_{\mathsf{reach}}$ to the control-flow variables $\mathcal{V}^{\mathsf{cf}}$, namely $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$, implies $\mathsf{ReachCF}(\mathcal{P})$ as well.

It has to be remarked here that one core idea of PDR is to avoid the computation of the reachable states by means of a fixpoint computation. It may therefore seem to be counterintuitive that we are using a fixpoint computation at this stage. However, since the fixpoint computation is just restricted to the control-flow, typical BDD-based

approaches can quickly compute $\mathsf{ReachCF}(\mathcal{P})$. The challenges for the reachable state space computation are rather in the dataflow, and we therefore employ PDR for the control-flow only.

4.2.2 Compiler Generated Control-flow Invariant $\mathsf{InvarCF}(\mathcal{P})$

The definition of certain control-flow predicates for the compilation of guarded actions were introduced already in [Sch01]. One of these predicates is $\mathsf{insd}(\mathtt{S})$, which holds whenever the control-flow is active in $\mathtt{S}$, and it is just the disjunction of the control-flow variables contained in $\mathtt{S}$. Using $\mathsf{insd}(\mathtt{S})$ we can recursively compute the control-flow predicate $\mathsf{InvarCF}(\mathcal{P})$ for any Quartz program $\mathcal{P}$ by the following theorem.

Theorem 12 ($\mathsf{InvarCF}(\mathcal{P})$)**.** For every synchronous Quartz program $\mathcal{P}$, all reachable states satisfy the predicate $\mathsf{InvarCF}(\mathcal{P})$ as defined by the following rules:

- $\mathsf{InvarCF}(\mathbf{nothing}) := \mathsf{true}$
- $\mathsf{InvarCF}(\mathtt{x} = \tau) := \mathsf{true}$
- $\mathsf{InvarCF}(\mathbf{next}(\mathtt{x}) = \tau) := \mathsf{true}$
- $\mathsf{InvarCF}(\mathtt{w}\texttt{:}\ \mathbf{pause}) := \mathsf{true}$
- $\mathsf{InvarCF}(\{\alpha\ \mathtt{x};\ \mathtt{S}\}) := \mathsf{InvarCF}(\mathtt{S})$
- $\mathsf{InvarCF}(\mathtt{S}_1; \mathtt{S}_2) := \mathsf{InvarCF}(\mathtt{S}_1) \wedge \mathsf{InvarCF}(\mathtt{S}_2) \wedge \neg(\mathsf{insd}(\mathtt{S}_1) \wedge \mathsf{insd}(\mathtt{S}_2))$
- $\mathsf{InvarCF}(\mathtt{S}_1 \parallel \mathtt{S}_2) := \mathsf{InvarCF}(\mathtt{S}_1) \wedge \mathsf{InvarCF}(\mathtt{S}_2)$
- $\mathsf{InvarCF}(\mathbf{if}(\sigma)\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2) := \mathsf{InvarCF}(\mathtt{S}_1) \wedge \mathsf{InvarCF}(\mathtt{S}_2) \wedge \neg(\mathsf{insd}(\mathtt{S}_1) \wedge \mathsf{insd}(\mathtt{S}_2))$
- $\mathsf{InvarCF}(\mathbf{while}(\sigma)\ \mathtt{S}) := \mathsf{InvarCF}(\mathtt{S})$
- $\mathsf{InvarCF}(\mathbf{do}\ \mathtt{S}\ \mathbf{while}(\sigma)) := \mathsf{InvarCF}(\mathtt{S})$
- $\mathsf{InvarCF}(\mathtt{[...]}\ \mathbf{abort}\ \mathtt{S}\ \mathbf{when}(\sigma)) := \mathsf{InvarCF}(\mathtt{S})$
- $\mathsf{InvarCF}(\mathtt{[...]}\ \mathbf{suspend}\ \mathtt{S}\ \mathbf{when}(\sigma)) := \mathsf{InvarCF}(\mathtt{S})$

Intuitively, $\mathsf{InvarCF}(\mathcal{P})$ states that the control-flow reaches can never be active at both substatements $\mathtt{S}_1$ and $\mathtt{S}_2$ of sequences and conditional statements. Actually, $\mathsf{InvarCF}(\mathcal{P})$ has already been generated in [Sch01] (see Lemma 2 in that paper), where it has been required to prove the correctness of the generated set of guarded actions with respect to the Structural Operational Semantics of the language. The validity of this predicate on all reachable states has been formally proved there, and it can also be proved by means of induction on the state transition system that is symbolically encoded by the formulas defined in Section 2.4.

$\mathsf{InvarCF}(\mathcal{P})$ is an over-approximation of the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ due to two facts. First, it does not take care of the infeasibility of control-flow conditions σ that

occur in conditionals, loops, abortion and suspension statements. Instead, $\mathsf{InvarCF}(\mathcal{P})$ considers both σ and its negation $\neg\sigma$ to be satisfiable, so that all substatements can be activated. If such a condition should however be unsatisfiable, some of the contained control-flow variables will not be reachable. Note that this can also be due to nested statements like nested conditionals where the combination of their conditions may become unsatisfiable.

Heuristics may be applied to check for infeasible path conditions as it is classically done in Worst-Case Execution Time (WCET) analysis, and it can be done today much better with the help of SMT solvers. In practice, however, it is usually never the case that such infeasible paths occur. We therefore do not consider this problem but focus on another one that is simpler to handle, but more relevant in practice.

Another reason why $\mathsf{InvarCF}(\mathcal{P})$ is an over-approximation of the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ is that it just states the disjointness of conditional and sequential substatements, but it does not consider how the control-flow variables contained in the substatements of parallel statements are related to each other.

We can significantly improve the approximation of $\mathsf{InvarCF}(\mathcal{P})$ by computing the reachable states of the state transition system in the way that projects its initial states and transition relation to the $\mathcal{V}^{\mathsf{cf}}$ labels only, which yields $\mathsf{ReachCF}(\mathcal{P})$ in Definition 3. We can even add a bit more information that is equally simple to get: In particular, note that all control-flow variables are false at the initial point of time, and this is also encoded in $\mathcal{I}^{\mathsf{cf}}$. Moreover, the single guarded action (**`true`**, **`next(run)=true`**) of the control-flow variable `run` will hold at every point of time other than the initial point of time. Therefore, any other control-flow variable holds implies `run`.

To sum it up, the computation of $\mathsf{InvarCF}(\mathcal{P})$ is straightforward: It can be done in linear time with respect to the size of `S`. However, we always have $\mathsf{ReachCF}(\mathcal{P}) \rightarrow \mathsf{InvarCF}(\mathcal{P})$, since $\mathsf{ReachCF}(\mathcal{P})$ is a better approximation of $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ than $\mathsf{InvarCF}(\mathcal{P})$. For example, $\mathsf{ReachCF}(\texttt{CfPar})$ is more accurate than $\mathsf{InvarCF}(\texttt{CfPar})$ as will be shown in Section 4.2.3.3.

4.2.3 Examples

In this section, we consider three generic Quartz modules `CfSeq`, `CfIte`, and `CfPar` to illustrate our definitions and their effects on Property Directed Reachability. All three programs are particular cases that contain only boolean variables.

4.2.3.1 Example 1: Synchronous Quartz Program `CfSeq`

Module `CfSeq` shown in Figure 4.1 is just a sequence of `N` **`pause`** statements. We compute its state transition system $\mathcal{K}_{\texttt{CfSeq}} = (\mathcal{V}_{\texttt{CfSeq}}, \mathcal{I}_{\texttt{CfSeq}}, \mathcal{T}_{\texttt{CfSeq}})$, control-flow transition system $\mathcal{K}^{\mathsf{cf}}_{\texttt{CfSeq}} = (\mathcal{V}^{\mathsf{cf}}_{\texttt{CfSeq}}, \mathcal{I}^{\mathsf{cf}}_{\texttt{CfSeq}}, \mathcal{T}^{\mathsf{cf}}_{\texttt{CfSeq}})$, together with its abstract control-flow transition system $\mathcal{K}^{\mathsf{abs}}_{\texttt{CfSeq}} = (\mathcal{V}^{\mathsf{cf}}_{\texttt{CfSeq}}, \mathcal{I}^{\mathsf{abs}}_{\texttt{CfSeq}}, \mathcal{T}^{\mathsf{abs}}_{\texttt{CfSeq}})$:

- $\mathcal{V}_{\texttt{CfSeq}} := \{\texttt{run}, \texttt{p<0>}, \ldots, \texttt{p<N-1>}\}$ $\qquad\qquad$ $\mathcal{V}_{\texttt{CfSeq}} = \mathcal{V}^{\mathsf{cf}}_{\texttt{CfSeq}} = \mathcal{V}^{\mathsf{abs}}_{\texttt{CfSeq}}$

```
1  macro N = ?;
2  module CfSeq(){
3    for (j=0..N−1) p: pause;
4  }
```

Figure 4.1: Synchronous Quartz Module `CfSeq`

- $\mathcal{I}_{\texttt{CfSeq}} := \neg\texttt{run} \wedge \bigwedge_{i=0}^{\texttt{N}-1} \neg\texttt{p}_i$ $\qquad\qquad \mathcal{I}_{\texttt{CfSeq}} = \mathcal{I}^{\mathsf{cf}}_{\texttt{CfSeq}} = \mathcal{I}^{\mathsf{abs}}_{\texttt{CfSeq}}$
- $\mathcal{T}_{\texttt{CfSeq}} := \begin{pmatrix} & (\textbf{next}(\texttt{run}) \leftrightarrow \texttt{true}) \\ \wedge & (\textbf{next}(\texttt{p<0>}) \leftrightarrow \neg\texttt{run}) \\ \wedge & (\textbf{next}(\texttt{p<1>}) \leftrightarrow \texttt{p<0>}) \\ \wedge & \vdots \\ \wedge & (\textbf{next}(\texttt{p<N−1>}) \leftrightarrow \texttt{p<N−2>}) \end{pmatrix}$ $\qquad\qquad \mathcal{T}_{\texttt{CfSeq}} = \mathcal{T}^{\mathsf{cf}}_{\texttt{CfSeq}} = \mathcal{T}^{\mathsf{abs}}_{\texttt{CfSeq}}$

Notice that their initial states and transition relations are the same, since there are no assignments to local or output variables.

The control-flow invariant $\mathsf{InvarCF}(\texttt{CfSeq})$ encodes the correct reachable states, and thus $\mathsf{ReachCF}(\texttt{CfSeq})$ cannot do better and is equivalent to $\mathsf{InvarCF}(\texttt{CfSeq})$ in this example. Hence, $\mathsf{ReachCF}(S)$ and even $\mathsf{InvarCF}(S)$ precisely compute the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$.

- $\mathsf{ReachCF}(\texttt{CfSeq}) := \left(\neg\texttt{run} \rightarrow \bigwedge_{i=0}^{\texttt{N}-1} \neg\texttt{p}_i\right) \wedge \left(\bigwedge_{\texttt{i}=0}^{\texttt{N}-2} \bigwedge_{\texttt{j}=\texttt{i}+1}^{\texttt{N}-1} \neg(\texttt{p<i>} \wedge \texttt{p<j>})\right)$
- $\mathsf{InvarCF}(\texttt{CfSeq}) = \mathsf{ReachCF}(\texttt{CfSeq})$

Figure 4.2 shows the state transition diagram of $\mathcal{K}_{\texttt{CfSeq}}$ for module `CfSeq` with $\texttt{N} := 2$ where the reachable states are in the yellow region and the initial state is drawn with double lines. For the three state variables, i.e., `run`, `p<0>` and `p<1>`, there are eight possible states, but only four of them are reachable. Thus, there are many CTIs for possible safety properties. For example, to prove that $\neg(\texttt{p<0>} \wedge \texttt{p<1>})$ holds on all reachable states, we color all the safe nodes with green in Figure 4.2, and the unsafe ones with orange. By the PDR method, we will have a CTI at state s_2 since it satisfies this property, but its successor state s_7 does not. PDR will therefore not be able to

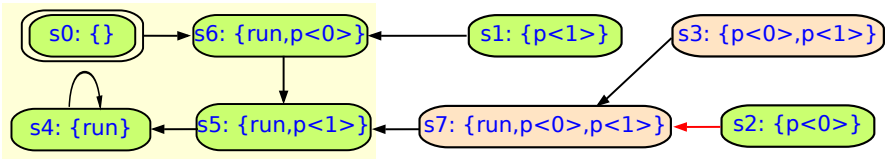

Figure 4.2: State Transition Diagram of $\mathcal{K}_{\texttt{CfSeq}}$ for Module `CfSeq` with $\texttt{N} := 2$

prove this property within one step. However, if we add the control-flow invariant InvarCF(CfSeq) or ReachCF(CfSeq) to the transition relation, then only the reachable states will have outgoing transitions. Thus, no CTIs are possible, and PDR will be able to prove every valid safety property by just checking the induction base and induction step. Note that in case of general N, there are $\mathtt{N}+2$ reachable states, but $2^{\mathtt{N}}$ states exist in the state transition system. The additional proof overhead of PDR grows with N.

4.2.3.2 Example 2: Synchronous Quartz Program CfIte

Module CfIte in Figure 4.3 is a conditional statement whose two substatements are both sequences of N **pause** statements. The decision depends on an input variable i.

```
1 macro N = ?;
2 module CfIte(mem bool i){
3   if (i) {for (j=0..N-1) p: pause;}
4   else {for (j=0..N-1) q: pause;}
5 }
```

Figure 4.3: Synchronous Quartz Module CfIte

Here are the state transition system $\mathcal{K}_{\mathtt{CfIte}} =: (\mathcal{V}_{\mathtt{CfIte}}, \mathcal{I}_{\mathtt{CfIte}}, \mathcal{T}_{\mathtt{CfIte}})$ and the control-flow transition system $\mathcal{K}^{\mathsf{cf}}_{\mathtt{CfIte}} = (\mathcal{V}^{\mathsf{cf}}_{\mathtt{CfIte}}, \mathcal{I}^{\mathsf{cf}}_{\mathtt{CfIte}}, \mathcal{T}^{\mathsf{cf}}_{\mathtt{CfIte}})$:

- $\mathcal{V}^{\mathsf{cf}}_{\mathtt{CfIte}} := \{\mathtt{run}, \mathtt{p<0>}, \ldots, \mathtt{p<N-1>}, \mathtt{q<0>}, \ldots, \mathtt{q<N-1>}\}$ $\qquad \mathcal{V}_{\mathtt{CfIte}} := \{\mathtt{i}\} \cup \mathcal{V}^{\mathsf{cf}}_{\mathtt{CfIte}}$
- $\mathcal{I}_{\mathtt{CfIte}} := \neg\mathtt{run} \wedge \bigwedge_{\mathtt{i}=0}^{\mathtt{N}-1} \neg(\mathtt{p}_i \vee \mathtt{q}_i)$ $\qquad \mathcal{I}_{\mathtt{CfIte}} = \mathcal{I}^{\mathsf{cf}}_{\mathtt{CfIte}}$
- $\mathcal{T}_{\mathtt{CfIte}} := \begin{pmatrix} & (\mathbf{next}(\mathtt{run}) \leftrightarrow \mathsf{true}) \\ \wedge & (\mathbf{next}(\mathtt{p<0>}) \leftrightarrow \mathtt{i} \wedge \neg\mathtt{run}) \\ \wedge & (\mathbf{next}(\mathtt{p<1>}) \leftrightarrow \mathtt{p<0>}) \\ \wedge & \vdots \\ \wedge & (\mathbf{next}(\mathtt{p<N-1>}) \leftrightarrow \mathtt{p<N-2>}) \\ \wedge & (\mathbf{next}(\mathtt{q<0>}) \leftrightarrow \neg\mathtt{i} \wedge \neg\mathtt{run}) \\ \wedge & (\mathbf{next}(\mathtt{q<1>}) \leftrightarrow \mathtt{q<0>}) \\ \wedge & \vdots \\ \wedge & (\mathbf{next}(\mathtt{q<N-1>}) \leftrightarrow \mathtt{q<N-2>}) \end{pmatrix}$ $\qquad \mathcal{T}_{\mathtt{CfIte}} = \mathcal{T}^{\mathsf{cf}}_{\mathtt{CfIte}}$

Notice that their initial states and transition relations are the same, since there are again no assignments to local or output variables.

In contrast to module CfSeq, we have however an input variable, so that the abstract control-flow system $\mathcal{K}^{\mathsf{abs}}_{\mathtt{CfIte}}$ has another symbolic representation that is obtained by existential quantification over the input variables, which is just i in this case: The decision whether to branch from the initial state to the control-flow state p<0> or q<0> depends on the input variable i, but is made nondeterministically in $\mathcal{T}^{\mathsf{abs}}_{\mathtt{CfIte}}$. Obviously,

this does not change the reachable control-flow states, so that $\mathsf{ReachCF}(\texttt{CfIte})$ will precisely compute the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ of module `CfIte`.

- $\mathcal{V}^{\mathsf{abs}}_{\texttt{CfIte}} = \mathcal{V}^{\mathsf{cf}}_{\texttt{CfIte}}$
- $\mathcal{I}^{\mathsf{abs}}_{\texttt{CfIte}} := \neg\texttt{run} \wedge \bigwedge_{\texttt{i}=0}^{\texttt{N}-1} \neg(\texttt{p}_i \wedge \texttt{q}_i)$
- $\mathcal{T}^{\mathsf{abs}}_{\texttt{CfIte}} := \left(\begin{array}{ll} & (\textbf{next}(\texttt{run}) \leftrightarrow \mathsf{true}) \\ \wedge & \Big((\textbf{next}(\texttt{p<0>}) \leftrightarrow \mathsf{false}) \wedge (\textbf{next}(\texttt{q<0>}) \leftrightarrow \neg\texttt{run}) \vee \\ & \quad (\textbf{next}(\texttt{p<0>}) \leftrightarrow \neg\texttt{run}) \wedge (\textbf{next}(\texttt{q<0>}) \leftrightarrow \mathsf{false})\Big) \\ \wedge & (\textbf{next}(\texttt{p<1>}) \leftrightarrow \texttt{p<0>}) \\ \wedge & \quad\vdots \\ \wedge & (\textbf{next}(\texttt{p<N-1>}) \leftrightarrow \texttt{p<N-2>}) \\ \wedge & (\textbf{next}(\texttt{q<1>}) \leftrightarrow \texttt{q<0>}) \\ \wedge & \quad\vdots \\ \wedge & (\textbf{next}(\texttt{q<N-1>}) \leftrightarrow \texttt{q<N-2>}) \end{array}\right)$

Meanwhile, the control-flow invariant $\mathsf{InvarCF}(\texttt{CfIte})$ just encodes that if one branch of the conditional statement is active, none of the labels of the other branch can be active, and since these are again sequences as module `CfSeq`, it furthermore states that only one of the control-flow variables can be active. Again, $\mathsf{InvarCF}(\texttt{CfIte})$ precisely computes the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ like $\mathsf{ReachCF}(\texttt{CfSeq})$.

- $\mathsf{ReachCF}(\texttt{CfIte}) := \left(\begin{array}{ll} & \left(\neg\texttt{run} \rightarrow \bigwedge_{\texttt{i}=0}^{\texttt{N}-1} \neg(\texttt{p}_i \vee \texttt{q<i>})\right) \\ \wedge & \neg\left(\bigvee_{\texttt{i}=0}^{\texttt{N}-1} \texttt{p<i>} \wedge \bigvee_{\texttt{i}=0}^{\texttt{N}-1} \texttt{q<i>}\right) \\ \wedge & \left(\bigwedge_{\texttt{i}=0}^{\texttt{N}-2} \bigwedge_{\texttt{j}=\texttt{i}+1}^{\texttt{N}-1} \neg(\texttt{p<i>} \wedge \texttt{p<j>})\right) \\ \wedge & \left(\bigwedge_{\texttt{i}=0}^{\texttt{N}-2} \bigwedge_{\texttt{j}=\texttt{i}+1}^{\texttt{N}-1} \neg(\texttt{q<i>} \wedge \texttt{q<j>})\right) \end{array}\right)$
- $\mathsf{InvarCF}(\texttt{CfIte}) = \mathsf{ReachCF}(\texttt{CfIte})$

Figure 4.4 shows the state transition diagram of $\mathcal{K}_{\texttt{CfIte}}$ for module `CfIte` with $\texttt{N} := 2$ where again the initial state is drawn with double lines. For the five state variables, i.e., `run`, `p<0>`, `p<1>`, `q<0>` and `q<1>`, there are $2^5 = 32$ possible states, but only six of them are reachable, as shown in the yellow region in Figure 4.4. Again, there are many

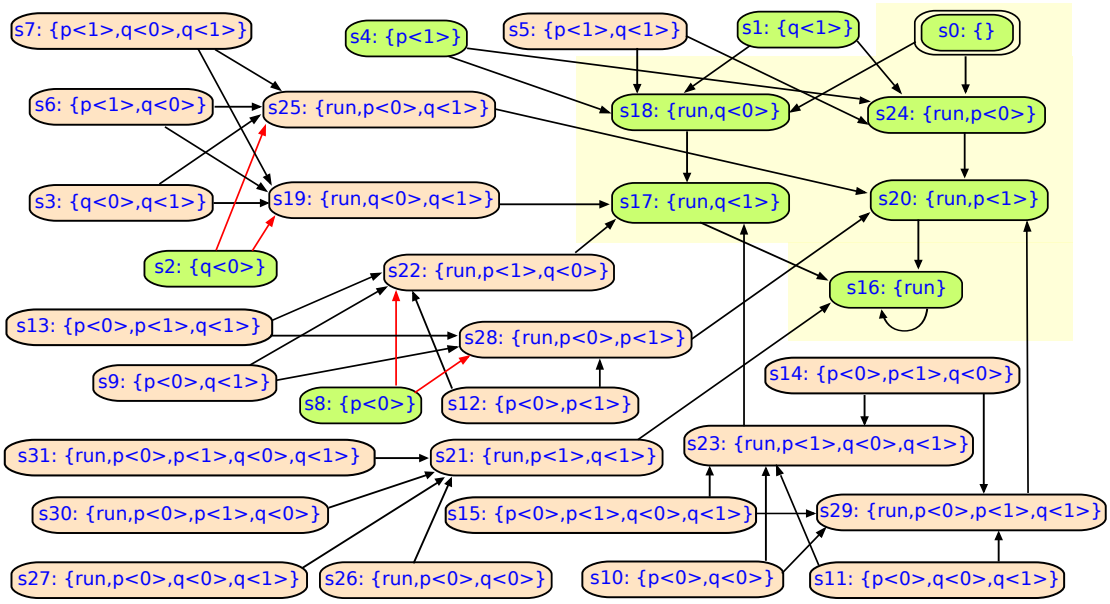

Figure 4.4: State Transition Diagram of $\mathcal{K}_{\texttt{CfIte}}$ for Module `CfIte` with `N` := 2

CTIs for possible safety properties that may trouble PDR. For example, to prove that $\neg(\texttt{p<0>} \wedge \texttt{p<1>}) \wedge \neg(\texttt{q<0>} \wedge \texttt{q<1>})\neg((\texttt{p<0>} \vee \texttt{p<1>}) \wedge (\texttt{q<0>} \vee \texttt{q<1>}))$ holds on all reachable states, we mark all the safe nodes with green and the unsafe ones with orange. We will have CTIs at states s_2 and s_8 since they satisfy this property, but their successor states s_{19}, s_{22}, s_{25} and s_{28} do not. The same as in the previous example, the addition of the control-flow invariant InvarCF(`CfIte`) or ReachCF(`CfIte`) to the transition relation will only allow the reachable states to have outgoing transitions. Thus, no CTIs are possible anymore, and PDR will be able to prove every valid safety property by just checking the induction base and induction step.

4.2.3.3 Example 3: Synchronous Quartz Program `CfPar`

Module `CfPar` shown in Figure 4.5 is a parallel statement whose two substatements are both sequences of `N` **`pause`** statements.

```
1  macro N = ?;
2  module CfPar(){
3    { for (j=0..N-1) p: pause; }
4    ||
5    { for (j=0..N-1) q: pause; }
6  }
```

Figure 4.5: Synchronous Quartz Module `CfPar`

When it starts, both **`pause`** statements `p<0>` and `q<0>` are entered by the control-flow,

then p<1> and q<1>, and so on. We compute its state transition system $\mathcal{K}_{\texttt{CfPar}} = (\mathcal{V}_{\texttt{CfPar}}, \mathcal{I}_{\texttt{CfPar}}, \mathcal{T}_{\texttt{CfPar}})$, control-flow transition system $\mathcal{K}^{\mathsf{cf}}_{\texttt{CfPar}} = (\mathcal{V}^{\mathsf{cf}}_{\texttt{CfPar}}, \mathcal{I}^{\mathsf{cf}}_{\texttt{CfPar}}, \mathcal{T}^{\mathsf{cf}}_{\texttt{CfPar}})$, and also abstraction control-flow transition system $\mathcal{K}^{\mathsf{abs}}_{\texttt{CfPar}} = (\mathcal{V}^{\mathsf{cf}}_{\texttt{CfPar}}, \mathcal{I}^{\mathsf{abs}}_{\texttt{CfPar}}, \mathcal{T}^{\mathsf{abs}}_{\texttt{CfPar}})$:

- $\mathcal{V}^{\mathsf{cf}}_{\texttt{CfPar}} := \{\texttt{run}, \texttt{p<0>}, \ldots, \texttt{p<N-1>}, \texttt{q<0>}, \ldots, \texttt{q<N-1>}\}$ $\quad \mathcal{V}_{\texttt{CfPar}} = \mathcal{V}^{\mathsf{cf}}_{\texttt{CfPar}} = \mathcal{V}^{\mathsf{abs}}_{\texttt{CfPar}}$
- $\mathcal{I}_{\texttt{CfPar}} := \neg\texttt{run} \wedge \bigwedge_{\texttt{i}=0}^{\texttt{N}-1} \neg(\texttt{p}_i \vee \texttt{q}_i)$ $\quad \mathcal{I}_{\texttt{CfPar}} = \mathcal{I}^{\mathsf{cf}}_{\texttt{CfPar}} = \mathcal{I}^{\mathsf{abs}}_{\texttt{CfPar}}$
- $\mathcal{T}_{\texttt{CfPar}} := \begin{pmatrix} & (\textbf{next}(\texttt{run}) \leftrightarrow \texttt{true}) \\ \wedge & (\textbf{next}(\texttt{p<0>}) \leftrightarrow \neg\texttt{run}) \\ \wedge & (\textbf{next}(\texttt{p<1>}) \leftrightarrow \texttt{p<0>}) \\ \wedge & \vdots \\ \wedge & (\textbf{next}(\texttt{p<N-1>}) \leftrightarrow \texttt{p<N-2>}) \\ \wedge & (\textbf{next}(\texttt{q<0>}) \leftrightarrow \neg\texttt{run}) \\ \wedge & (\textbf{next}(\texttt{q<1>}) \leftrightarrow \texttt{q<0>}) \\ \wedge & \vdots \\ \wedge & (\textbf{next}(\texttt{q<N-1>}) \leftrightarrow \texttt{q<N-2>}) \end{pmatrix}$ $\quad \mathcal{T}_{\texttt{CfPar}} = \mathcal{T}^{\mathsf{cf}}_{\texttt{CfPar}} = \mathcal{T}^{\mathsf{abs}}_{\texttt{CfPar}}$

The control-flow invariant $\mathsf{InvarCF}(\texttt{CfPar})$ just encodes that at most one of the control-flow variables p<0>, ..., p<N−1> and also that at most one of the control-flow variables q<0>, ..., q<N−1> is valid. However, it does not relate the two threads to each other. Hence, this predicate allows all combinations of pairs p<i>, q<j> to be active at the same time which is however not possible in any reachable state. $\mathsf{InvarCF}(\texttt{CfPar})$ is therefore a coarse abstraction. Meanwhile, the computation of the reachable states with $\mathcal{I}^{\mathsf{abs}}_{\texttt{CfPar}}$ and $\mathcal{T}^{\mathsf{abs}}_{\texttt{CfPar}}$ yields $\mathsf{ReachCF}(\texttt{CfPar})$, which is the precise set of reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ of module CfPar, and can therefore give PDR the exact information from the beginning.

- $\mathsf{ReachCF}(\texttt{CfPar}) := \begin{pmatrix} & \left(\neg\texttt{run} \wedge \bigwedge_{\texttt{i}=0}^{\texttt{N}-1} \neg(\texttt{p<i>} \vee \texttt{q<i>})\right) \\ \vee & \left(\texttt{run} \wedge \texttt{p<0>} \wedge \texttt{q<0>} \wedge \bigwedge_{\texttt{i}=1,\texttt{i}\neq 1}^{\texttt{N}} \neg(\texttt{p<j>} \vee \texttt{q<j>})\right) \\ \vee & \left(\texttt{run} \wedge \texttt{p<1>} \wedge \texttt{q<1>} \wedge \bigwedge_{\texttt{i}=1,\texttt{i}\neq 2}^{\texttt{N}} \neg(\texttt{p<j>} \vee \texttt{q<j>})\right) \\ \vee & \vdots \\ \vee & \left(\texttt{run} \wedge \texttt{p<N-1>} \wedge \texttt{q<N-1>} \wedge \bigwedge_{\texttt{i}=1,\texttt{i}\neq \texttt{N}}^{\texttt{N}-1} \neg(\texttt{p<j>} \vee \texttt{q<j>})\right) \end{pmatrix}$

$$\bullet\ \mathsf{InvarCF}(\texttt{CfPar}) := \left(\begin{array}{ll} & \left(\neg \texttt{run} \rightarrow \bigwedge_{\texttt{i}=0}^{\texttt{N}-1} \neg(\texttt{p<i>} \vee \texttt{q<i>}) \right) \\ \wedge & \left(\bigwedge_{\texttt{i}=0}^{\texttt{N}-2} \bigwedge_{\texttt{j}=\texttt{i}+1}^{\texttt{N}-1} \neg(\texttt{p<i>} \wedge \texttt{p<j>}) \right) \\ \wedge & \left(\bigwedge_{\texttt{i}=0}^{\texttt{N}-2} \bigwedge_{\texttt{j}=\texttt{i}+1}^{\texttt{N}-1} \neg(\texttt{q<i>} \wedge \texttt{q<j>}) \right) \end{array} \right)$$

The state transition diagram of $\mathcal{K}_{\texttt{CfPar}}$ for module `CfPar` with `N` := 2 is shown in Figure 4.6, where again the reachable states are in the yellow region and the initial state is drawn with double lines. For the five state variables, i.e., `run`, `p<0>`, `p<1>`, `q<0>` and `q<1>`, there are $2^5 = 32$ possible states, but only four of them are reachable so that PDR may have to deal with CTIs. In contrast to the previous examples, the addition of $\mathsf{InvarCF}(\texttt{CfPar})$ will not remove all transitions from unreachable states. Of course, $\mathsf{InvarCF}(\texttt{CfPar})$ holds in the four reachable states, but also in additional six states: s_{17}, s_{18}, s_{20}, s_{22}, s_{24}, s_{25}. It can therefore help PDR with some, but not with all safety properties. For example, given a safety property where all the safe nodes are marked green whereas the unsafe ones are marked with orange in Figure 4.6, to prove it holds on all reachable states with $\mathsf{InvarCF}(\texttt{CfPar})$, we will still have CTIs at states s_{18} and s_{24} since they satisfy this property, but their successor states s_{17} and s_{20} do not. On the other hand, $\mathsf{ReachCF}(\texttt{CfPar})$ computes in this example precisely the reachable control-flow states $\mathcal{S}^{\mathsf{cf}}_{\mathsf{reach}}$ of module `CfPar` so that PDR can prove any safety property directly without even starting the incrementation procedure.

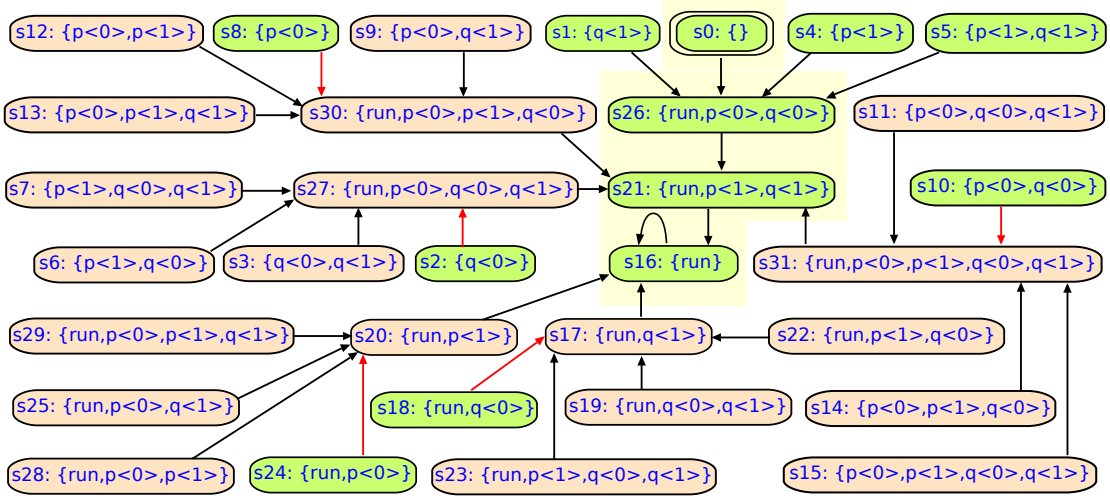

Figure 4.6: State Transition Diagram of $\mathcal{K}_{\texttt{CfPar}}$ for Module `CfPar` with `N` := 2

4.3 CTI Identification and Generalization

In this section, we describe our method that can automatically generate unreachable clauses for synchronous Quartz programs based on the analysis of their control-flow transition systems. As we will show, the unreachability of some CTIs in $\mathcal{K}$ can be proved by unreachability in the EFSM. Dropping the dataflow variables in the cube will moreover generate an unreachable cube whose clause can be used to narrow the reachable state approximations maintained by PDR.

4.3.1 Unreachability Checking by EFSMs

As explained in Section 4.1, the transition relation $\mathcal{T}$ of such a synchronous Quartz program can be derived as a conjunction $\mathcal{T} := \mathcal{T}^{\mathsf{cf}} \wedge \mathcal{T}^{\mathsf{df}}$ of transition relations over the same set of variables, one for the control-flow $\mathcal{T}^{\mathsf{cf}}$ and another one for the dataflow $\mathcal{T}^{\mathsf{df}}$, respectively. The reachability of a state s' from a state s in $\mathcal{T}$ is equivalent to the reachability in both $\mathcal{T}^{\mathsf{cf}}$ and $\mathcal{T}^{\mathsf{df}}$. Hence, if s' is not reachable from s in $\mathcal{T}^{\mathsf{cf}}$, we can already conclude its unreachability in $\mathcal{T}$ and thus can declare it as a CTI without considering the full transition relation $\mathcal{T}$.

The advantage is that the control-flow transition system $\mathcal{K}^{\mathsf{cf}}$ is much simpler to deal with (even though it has even more reachable states) since we can compute a usually small quotient (in terms of the EFSM) for it. We may also use traditional model checking approaches for that purpose since $\mathcal{T}^{\mathsf{cf}}$ can be usually represented efficiently by means of BDDs. This way, we can add a first, less expensive test for checking the unreachability of a state in the corresponding EFSM. If that test should fail, we use the traditional PDR reachability checks that will generate further reachability queries that then (again) first ask for reachability by EFSM in every step.

Our extension of PDR to the CTI identification is based on the following lemma that defines the reachability in EFSMs:

Lemma 2 (Unreachability Checking by EFSMs:)**.** *If a node s with a control-flow label $\mathsf{Labels}(s) \subseteq \mathcal{V}^{\mathsf{cf}}$ is not reachable in the EFSM from the initial node, then neither this state nor any other state $s' \subseteq \mathcal{V}^{\mathsf{cf}} \cup \mathcal{V}^{\mathsf{df}}$ with $s = s' \cap \mathcal{V}^{\mathsf{cf}}$ is reachable in $\mathcal{K}^{\mathsf{cf}}$, and thus also none of these states is reachable in $\mathcal{K}$.*

Note that the transition conditions have been ignored for defining the above reachability in the EFSM so that we do not have to consider infeasible paths.

4.3.2 Control-flow Guided Clause Generation

After determining a state is unreachable in $\mathcal{K}^{\mathsf{cf}}$ through the EFSM, we can reduce it to its control-flow variables, and obtain this way a generalized clause that will exclude all states of the transition system that refer to the same program locations (but with different values of the data variables). In this way, we can avoid expensive clause generalizations that are required in PDR to narrow the over-approximations of the clause sets $\Psi_0, \ldots, \Psi_k$.

The clauses generated this way may not be relative inductive, but by inspection of the control-flow transition relation $\mathcal{T}^{\mathsf{cf}}$, we can directly decide about their unreachability which is sufficient for excluding certain states. Also, the clauses may not be minimal, but since they are quickly generated this way, it is usually a good compromise. Alternatively, we could apply the usual methods to minimize them. Our extension of the clause generation is based on the following theorem:

Theorem 13 (Control-flow Guided Clause Generation)**.** *Let $\mathcal{P}$ be a synchronous Quartz program with the transition systems $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{T})$, $\mathcal{K}^{\mathsf{cf}} = (\mathcal{V}, \mathcal{I}^{\mathsf{cf}}, \mathcal{T}^{\mathsf{cf}})$, and $\mathcal{K}^{\mathsf{df}} = (\mathcal{V}, \mathcal{I}^{\mathsf{df}}, \mathcal{T}^{\mathsf{df}})$ as introduced in the previous section and let $\mathcal{C}$ be a cube over variables $\mathcal{V}$.*

- Reachability of Cubes: *If no state $s \subseteq \mathcal{V}$ that satisfies $\mathcal{C}$ is reachable in $\mathcal{K}^{\mathsf{cf}}$, then none of these states is reachable in $\mathcal{K}$.*
- Generalization of Clauses: *If no state of $\mathcal{C}$ is reachable in $\mathcal{K}^{\mathsf{cf}}$, then also no state of $\mathcal{C}' := \mathcal{C}_{|\mathcal{V}^{\mathsf{cf}}}$ is reachable neither in $\mathcal{K}^{\mathsf{cf}}$ nor in $\mathcal{K}$ where $\mathcal{C}' := \mathcal{C}_{|\mathcal{V}^{\mathsf{cf}}}$ denotes the restriction of $\mathcal{C}$ to the control-flow variables $\mathcal{V}^{\mathsf{cf}}$.*

Proof.

- The first proposition follows almost directly from Lemma 1, since unreachability in $\mathcal{K}^{\mathsf{cf}}$ implies unreachability in $\mathcal{K}$.
- The cube $\mathcal{C}'$ contains only control-flow literals. One of its states is reachable in $\mathcal{K}^{\mathsf{cf}}$ iff the corresponding state endowed with the dataflow literals in $\mathcal{C}$ will be reachable in $\mathcal{K}^{\mathsf{cf}}$, since reachability in $\mathcal{K}^{\mathsf{cf}}$ does not depend on the dataflow literals. Thus, if no state of $\mathcal{C}$ is reachable in $\mathcal{K}^{\mathsf{cf}}$ then and only then, no state of $\mathcal{C}'$ is reachable in $\mathcal{K}^{\mathsf{cf}}$, and thus also not reachable in $\mathcal{K}$.

■

The subclause $\mathcal{C}' := \mathcal{C}_{|\mathcal{V}^{\mathsf{cf}}}$ obtained from omitting the dataflow literals in cube $\mathcal{C}$ can therefore be used to narrow the reachable state approximations of PDR if no state of $\mathcal{C}$ is reachable in $\mathcal{K}^{\mathsf{cf}}$. Finally, checking unreachability of a cube $\mathcal{C}$ in $\mathcal{K}^{\mathsf{cf}}$ can be approximated by checking whether the node in the EFSM that corresponds to $\mathcal{C}' := \mathcal{C}_{|\mathcal{V}^{\mathsf{cf}}}$ is reachable in the EFSM.

As shown in Figure 4.7, the Averest compiler computes for a given *Quartz* program an equivalent set of guarded actions $\mathcal{G}_{\mathtt{P}}$. Different transformation procedures are provided to adapt the guarded actions for special needs. The symbolic transition system $\mathcal{K}$ is the input for the PDR method. In the blocking phase, if the CTI can not be mapped to any of the reachable nodes of the EFSM, then an unreachable subclause will be derived by restricting the cube clause to its control-flow literals, otherwise the normal reachability checks of PDR will be applied.

4.3.3 Example

To demonstrate the control-flow guided clause generalization method, we use module `ITELoop` as shown in Figure 4.8: `i` is a local array with `N` boolean variables. The initial

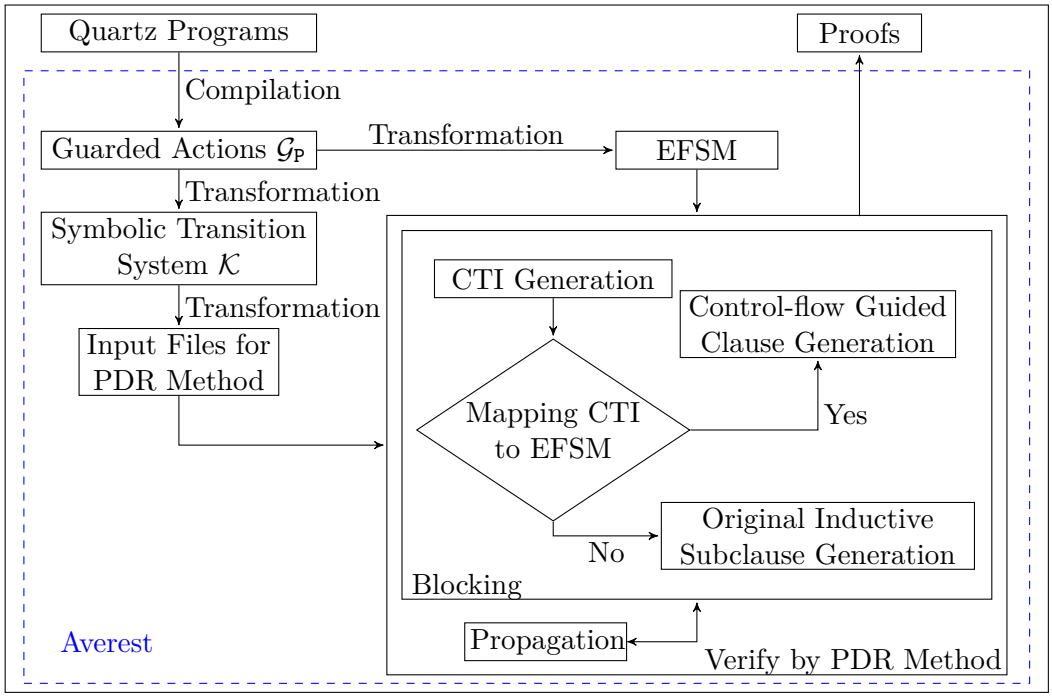

Figure 4.7: Control-flow Guided Clause Generation

value of `i[0]` is **`true`**, the other array elements are **`false`** (default initialization). The body statement of the module `ITELoop` is a conditional statement: if ¬`i[0]` holds, then the loop statement will be immediately started. In the second macro step inside the loop statement `i[0]` is assigned to **`false`**. It is not difficult to see that `i[0]` always holds in module `ITELoop` since the control-flow will never enter the loop statement.

```
1  macro N=?;
2  module ITELoop() {
3    [N]bool i; i[0] = true;
4    if (!i[0]) {
5      loop{
6        p1: pause;
7        i[0] = false;
8        p2: pause;
9      }
10   }
11 }
```

Figure 4.8: Fig: Synchronous Quartz Module `ITELoop`

Consider the instance of module `ITELoop` with `N` := 1, and its EFSM in Figure 4.9. The symbolic representations of the transition systems $\mathcal{K}_{\mathsf{ITELoop}} = (\mathcal{V}_{\mathtt{ITELoop}}, \mathcal{I}_{\mathtt{ITELoop}}, \mathcal{T}_{\mathtt{ITELoop}})$, $\mathcal{K}^{\mathsf{df}}_{\mathsf{ITELoop}} = (\mathcal{V}_{\mathtt{ITELoop}}, \mathcal{I}^{\mathsf{df}}_{\mathtt{ITELoop}}, \mathcal{T}^{\mathsf{df}}_{\mathtt{ITELoop}})$, and $\mathcal{K}^{\mathsf{cf}}_{\mathsf{ITELoop}} = (\mathcal{V}_{\mathtt{ITELoop}}, \mathcal{I}^{\mathsf{cf}}_{\mathtt{ITELoop}}, \mathcal{T}^{\mathsf{cf}}_{\mathtt{ITELoop}})$ are

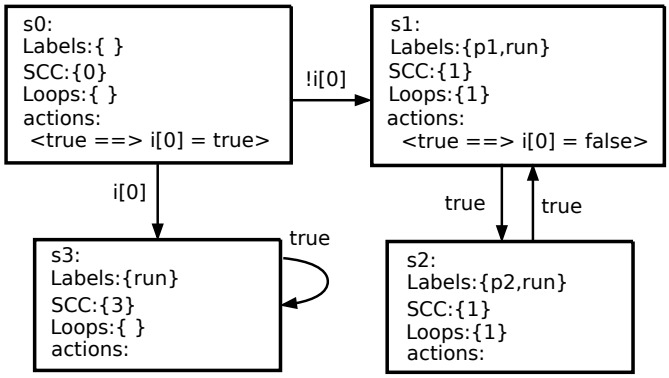

Figure 4.9: EFSM of Module `ITELoop` with `N` := 1

defined as follows:

- $\mathcal{V}^{\mathsf{cf}}_{\mathtt{ITELoop}} := \{\mathtt{p}_1, \mathtt{p}_2, \mathtt{run}\}$
- $\mathcal{V}^{\mathsf{df}}_{\mathtt{ITELoop}} := \{\mathtt{i[0]}\}$
- $\mathcal{I}^{\mathsf{cf}}_{\mathtt{ITELoop}} := \neg(\mathtt{p}_1 \vee \mathtt{p}_2 \vee \mathtt{run})$
- $\mathcal{T}^{\mathsf{cf}}_{\mathtt{ITELoop}} := \begin{pmatrix} \mathbf{next}(\mathtt{run}) \\ \wedge \ \left(\mathbf{next}(\mathtt{p1}) \leftrightarrow (\neg \mathtt{p2} \rightarrow (\neg \mathtt{run} \wedge \neg \mathtt{i[0]}))\right) \\ \wedge \ (\mathbf{next}(\mathtt{p2}) \leftrightarrow \mathtt{p1}) \end{pmatrix}$
- $\mathcal{I}^{\mathsf{df}}_{\mathtt{ITELoop}} := \mathtt{i[0]}$
- $\mathcal{T}^{\mathsf{df}}_{\mathtt{ITELoop}} := \begin{pmatrix} (\mathtt{p1} \rightarrow \neg \mathtt{i[0]}) \\ \wedge \ \left(\neg \mathbf{next}(\mathtt{p1}) \rightarrow (\mathbf{next}(\mathtt{i[0]}) \leftrightarrow \mathtt{i[0]})\right) \end{pmatrix}$
- $\mathcal{V}_{\mathtt{ITELoop}} := \mathcal{V}^{\mathsf{cf}}_{\mathtt{ITELoop}} \cup \mathcal{V}^{\mathsf{df}}_{\mathtt{ITELoop}}$
- $\mathcal{I}_{\mathtt{ITELoop}} := \mathcal{I}^{\mathsf{cf}}_{\mathtt{ITELoop}} \wedge \mathcal{I}^{\mathsf{df}}_{\mathtt{ITELoop}}$
- $\mathcal{T}_{\mathtt{ITELoop}} := \mathcal{T}^{\mathsf{cf}}_{\mathtt{ITELoop}} \wedge \mathcal{T}^{\mathsf{df}}_{\mathtt{ITELoop}}$

Figure 4.11 and Figure 4.10 are the state transition diagrams of $\mathcal{K}_{\mathsf{ITELoop}}$ and $\mathcal{K}^{\mathsf{cf}}_{\mathsf{ITELoop}}$. Again, the yellow region covers the reachable states, and the green nodes are the states satisfying `i[0]`, while the remaining ones violate `i[0]`.

By the control-flow guided clause generation method, we prove that $\Phi :=$ `i[0]` holds on all reachable states of $\mathcal{K}_{\mathsf{ITELoop}}$ as follows:

- `i[0]` holds in both the initial state s_1 and its successor s_3. The initial state is not inductive since the successor s_3 of the initial state s_1 is not an initial state. Since

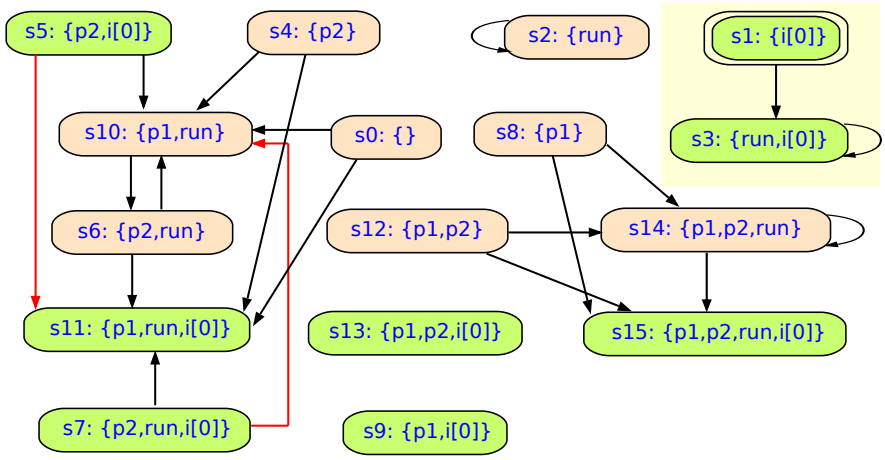

Figure 4.10: State Transition Diagram of $\mathcal{K}_{\texttt{ITELoop}}$ for Module `ITELoop` with `N` := 1

there are transitions from states where $\Phi := \texttt{i[0]}$ holds to states where Φ does not hold, Φ is also not inductive. So, we set up the first Ψ-sequence of clauses sets for $k := 1$:

$$\Psi_0 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p1}\}, \{\neg\texttt{p2}\}, \{\neg\texttt{run}\}\}$$
$$\Psi_1 := \{\{\texttt{i[0]}\}\}$$

- Let $[\![\Psi_i]\!]_{\mathcal{K}_{\texttt{ITELoop}}}$ represent the set of states satisfying Ψ_i. Looking at Figure 4.10, we see the following:

$$[\![\Psi_0]\!]_{\mathcal{K}_{\texttt{ITELoop}}} := \{s_1\}$$
$$[\![\Psi_1]\!]_{\mathcal{K}_{\texttt{ITELoop}}} := \{s_1, s_3, s_5, s_7, s_9, s_{11}, s_{13}, s_{15}\}$$

  s_5 and s_7 are CTIs since these states belong to $[\![\Psi_1]\!]_{\mathcal{K}_{\texttt{ITELoop}}}$ but have successors violating `i[0]`.

- We choose the CTI with the smallest index, i.e., s_5. It can be represented as cube $\neg\texttt{p1}\wedge\texttt{p2}\wedge\neg\texttt{run}\wedge\texttt{i[0]}$, whose control-flow part is $\neg\texttt{p1}\wedge\texttt{p2}\wedge\neg\texttt{run}$ which corresponds to states s_4 and s_5. There is no corresponding node in the EFSM labeled with $\{\texttt{p2}\}$, and thus, none of the states s_4 and s_5 are reachable in $\mathcal{K}^{\texttt{cf}}_{\texttt{ITELoop}}$, and thus neither in $\mathcal{K}_{\texttt{ITELoop}}$. Therefore, unreachability of s_4 and s_5 follows directly, and we could add clause $\{\texttt{p1}, \neg\texttt{p2}, \texttt{run}\}$ to Ψ_0 and Ψ_1.

- We next explore the lattice of the subclauses of $\{\texttt{p1}, \neg\texttt{p2}, \texttt{run}\}$ to generalize the clause. The minimal inductive subclause $\{\neg\texttt{p2}\}$ can be extracted. We then conjoin $\{\neg\texttt{p2}\}$ and obtain the following Ψ-sequence:

$$\Psi_0 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p1}\}, \{\neg\texttt{p2}\}, \{\neg\texttt{run}\}\}$$
$$\Psi_1 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p2}\}\}$$

  Now we have $[\![\Psi_1]\!]_{\mathcal{K}_{\texttt{ITELoop}}} = \{s_1, s_3, s_9, s_{11}\}$.

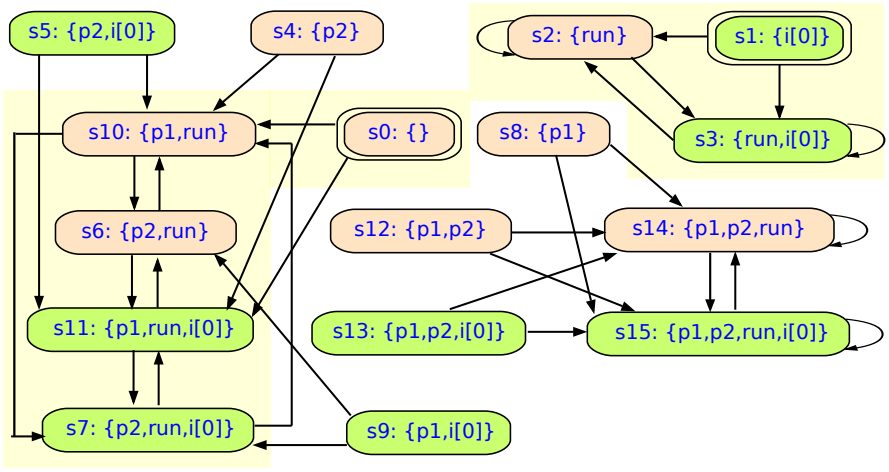

Figure 4.11: State Transition Diagram of $\mathcal{K}^{\mathsf{cf}}_{\texttt{ITELoop}}$ for Module `ITELoop` with $\texttt{N} := 1$

- Since now all successors of Ψ_1 satisfy `i[0]`, PDR increments the trace and propagates clauses as usual[1] in PDR. The following Ψ-sequence of clause sets for $k := 2$ is obtained:

$$\Psi_0 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p1}\}, \{\neg\texttt{p2}\}, \{\neg\texttt{run}\}\}$$
$$\Psi_1 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p1}\}, \{\neg\texttt{p2}\}\}$$
$$\Psi_2 := \{\{\texttt{i[0]}\}, \{\neg\texttt{p1}\}, \{\neg\texttt{p2}\}\}$$

  Note that $\{\neg\texttt{run}\}$ cannot be propagated to Ψ_1 since it does not hold on s_3, but $\{\neg\texttt{p1}\}$ propagates from Ψ_0 to Ψ_1 since it holds on all successors of Ψ_0, i.e., on s_3.

- We now have $\Psi_1 = \Psi_2$ (syntactic equality) with $[\![\Psi_1]\!]_{\mathcal{K}_{\texttt{ITELoop}}} = [\![\Psi_2]\!]_{\mathcal{K}_{\texttt{ITELoop}}} = \{s_1, s_3\}$, so we found a proof and conclude that `i[0]` holds on all reachable states of $\mathcal{K}_{\texttt{ITELoop}}$ by the control-flow guided clause generation method.

In general, the set of boolean variables of module `ITELoop` is the following:

$$\mathcal{V}_{\texttt{N}} := \underbrace{\{\texttt{i[0]}, \ldots, \texttt{i[N-1]}\}}_{\mathcal{V}^{\mathsf{df}}} \cup \underbrace{\{\texttt{p1}, \texttt{p2}, \texttt{run}\}}_{\mathcal{V}^{\mathsf{cf}}}$$

In the worst case, $\mathcal{C}$ contains $\texttt{N} + 3$ literals over $\mathcal{V}_{\texttt{N}}$. Omitting the dataflow literals, only three literals over $\mathcal{V}^{\mathsf{cf}}$ remain. By our control-flow guided clause generalization method, starting from those CTIs that cannot be mapped to any nodes in the EFSM, the PDR method benefits from the following two aspects:

- The unreachability of those CTIs can be proved directly by the control-flow labels of the EFSM nodes.

[1] A clause c of Ψ_i is propagated to Ψ_{i+1} if $\Psi_i \rightarrow \Box c$ is valid.

- It is sufficient to narrow the reachable state approximations with the generalized clause $\mathcal{C}'$ obtained from omitting the dataflow literals. In this way, the traditional generalization of clauses, which may yield $2^{\mathtt{N}+3}$ queries to a SAT/SMT solver, can be avoided.
- The generalized clause $\mathcal{C}'$ may not be relative inductive. If it is relative inductive, then deriving a minimal inductive subclause from it can exclude more unreachable states, which requires at most 2^3 times relative inductiveness reasoning.

The proposed control-flow guided method can safely omit dataflow literals for clause generation so that the more expensive clause generalization of PDR will only be called when it is really needed. The generalized clause generated by our method excludes all states of the transition system that refer to the same control-flow states.

Chapter 5

Experimental Evaluation

Contents

In this section, we use five synchronous and hybrid Quartz programs to demonstrate the feasibility of the presented VCG methods in Chapter 3. To simplify the verification goals, all ODEs in the VCs are solved with the aid of a computer algebra tool like Mathematica. Let v_i represent the i-th VC, v_i is valid if and only if $\neg v_i$ is unsatisfiable.

For programs where the underlying satisfiability program is decidable, we could either check each $\neg v_i$ by the SMT solvers directly, or combine them to a single VC before using the solvers. Thus, VCs can be verified in the following different formats:

- $\sum$-Format: Check each formula $\neg v_i$ independently, $\tau_{\sum}$ is the sum of the execution time for each $\neg v_i$.
- $\bigwedge$-Format: Check the reorganized formula in the form of $\neg \bigwedge_i v_i$, $\tau_{\bigwedge}$ represents the execution time.
- $\bigvee$-Format: Check the reorganized formula in the form of $\bigvee_i \neg v_i$, the execution time is $\tau_{\bigvee}$.

Both Z3 and iSAT, which can be called as external programs, are used to prove the VCs. Variable ranges must be provided for iSAT, and obviously the provided variable ranges affect the runtime. The implementation is written in F#, which is one of the main languages of the Mono/.NET framework. Meanwhile, Z3 API can be accessed in Mono/.Net, and additionally, it provides a parallelized version using the '*async*' computation expression. Therefore, we use Z3 API and its parallelized version to check VCs as well. We also try another SMT solver, namely CVC4, however, according to our experiments, CVC4's non-linear real and non-linear integer arithmetic support is currently not sufficient for our needs.

All experiments were performed on a machine with 3.2GHz Intel Core i5-3470 processor, 8.1GB RAM, and 64-bit Ubuntu 17.04. We compare the proved VC number with the total VC number, and collect the following execution time (average time over 5 iterations in milliseconds) consumed by different formats and different SMT solvers:

- EFSM-Inv Time: The computation time for the preparation work that includes parsing input files, generating the EFSM, and mapping SCC or loop invariant(s) to the EFSM nodes.
- VCG Time: The computation time to generate VCs by the provided VCG method.
- SMT Time: The computation time to check VCs by the provided SMT solver.

5.1 Synchronous Quartz Program `SearchZeros`

The synchronous program `SearchZero` in Figure 5.1 was first used as an introductory example in [AO09]. Its EFSM is depicted in Figure 5.2.

5.1.1 Module `SearchZeros` and its EFSM

Given a function `f` : **`int`** $\rightarrow$ **`int`** defined by a macro, the program starts two searches: one from 0,1,2,... and another from $-1,-2,-3,...$ until the first integer `x` is found for

```
1  // define the function f(x)
2  macro f(x) = (x−2)∗(x−2)+1;
3  module SearchZero(bool found, int y){
4    int x0,x1;
5    weak immediate abort{
6      { // one branch searches from 0 to the positive infinity
7        x0 = 0;
8        while(f(x0) != 0){
9          next(x0) = x0+1;
10         w0: pause;}
11       found = true;
12     }||
13     { // the other branch searches from −1 to the negative infinity
14       x1 = −1;
15       while(f(x1) != 0) {
16         next(x1) = x1−1;
17         w1: pause;}
18       found = true;
19     }
20   } when(found);
21   // return the value of x for f(x) = 0
22   if (f(x0) == 0) y = x0; else y = x1;
23 }
```

Figure 5.1: Synchronous Quartz Module SearchZero

which f(x) = 0 holds. The correctness of the program is ensured by the following invariant:

$$\Phi_{\text{SearchZero}} := \texttt{found} \rightarrow \Big((\texttt{f(y)} = 0) \ \wedge \ (\texttt{y} \geq 0 \rightarrow \forall \texttt{i} \in [-\texttt{y}, \texttt{y}-1].\texttt{f(i)} \neq 0) \ \wedge \ (\texttt{y} < 0 \rightarrow \forall \texttt{i} \in [\texttt{y}+1, -\texttt{y}-1].\ \texttt{f(i)} \neq 0) \Big)$$

In our example, i.e., $\texttt{f(x)} = (\texttt{x}-2)^2+1$, all model checking algorithms will not terminate since there is no x that satisfies $\texttt{f(x)} = 0$. which ensures the correctness of this program.

5.1.2 VCG using TransBased for SearchZero

The EFSM of module SearchZero in Figure 5.2 has five states. We use $\Phi_{\text{SearchZero}}$ as control-flow invariant α_i for each node s_i, and then apply the TransBased method to generate the following subgoals:

- $\mathcal{D}(s_0) \rightarrow \Phi_{\text{SearchZero}}$
- $\alpha_0 \rightarrow \Phi_{\text{SearchZero}}$
- $\alpha_1 \rightarrow \Phi_{\text{SearchZero}}$
- $\alpha_2 \rightarrow \Phi_{\text{SearchZero}}$
- $\alpha_3 \rightarrow \Phi_{\text{SearchZero}}$
- $\alpha_4 \rightarrow \Phi_{\text{SearchZero}}$
- $\mathsf{SafeTrans}((s_0, s_1), \alpha_0, true, \alpha_1)$
- $\mathsf{SafeTrans}((s_0, s_2), \alpha_0, true, \alpha_2)$
- $\mathsf{SafeTrans}((s_0, s_3), \alpha_0, true, \alpha_3)$
- $\mathsf{SafeTrans}((s_0, s_4), \alpha_0, true, \alpha_4)$
- $\mathsf{SafeTrans}((s_1, s_1), \alpha_1, true, \alpha_1)$
- $\mathsf{SafeTrans}((s_1, s_2), \alpha_1, true, \alpha_2)$
- $\mathsf{SafeTrans}((s_1, s_3), \alpha_1, true, \alpha_3)$
- $\mathsf{SafeTrans}((s_1, s_4), \alpha_1, true, \alpha_4)$
- $\mathsf{SafeTrans}((s_2, s_2), \alpha_2, true, \alpha_2)$
- $\mathsf{SafeTrans}((s_2, s_4), \alpha_2, true, \alpha_4)$

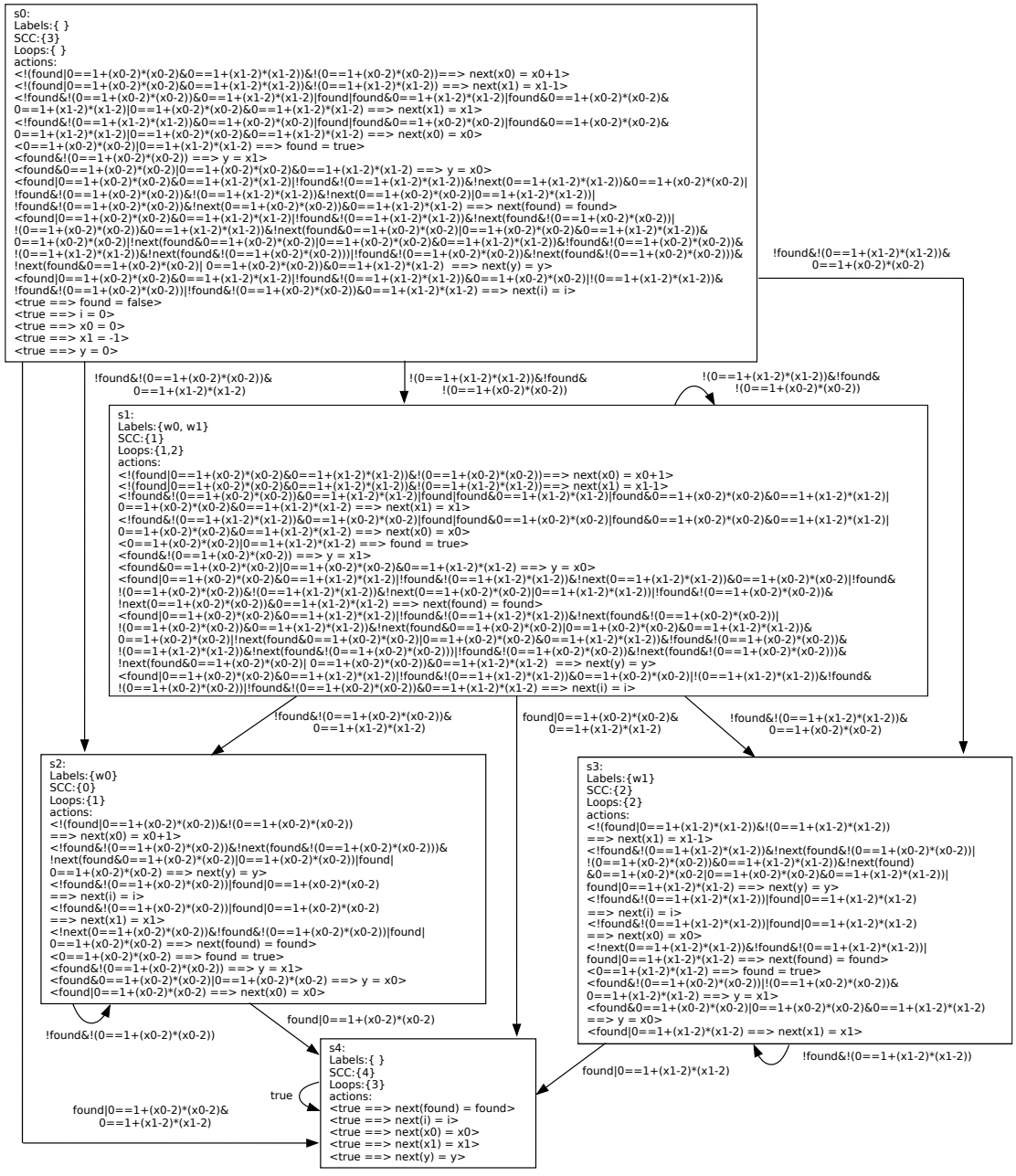

Figure 5.2: EFSM of Module `SearchZero`

- $\mathsf{SafeTrans}((s_3, s_3), \alpha_3, true, \alpha_3)$
- $\mathsf{SafeTrans}((s_3, s_4), \alpha_3, true, \alpha_4)$
- $\mathsf{SafeTrans}((s_4, s_4), \alpha_4, true, \alpha_4)$

Each subgoal represents a VC that can be converted to SMT-LIB format for SMT solvers. To have some idea on how the VCs look like, we give the following input file in SMT-LIB format in Figure 5.3 that describes the subgoal $\mathcal{D}(s_0) \rightarrow \Phi_{\text{SearchZero}}$:

```
1  (declare-fun y_0 () Int)
2  (declare-fun i_0 () Int)
3  (declare-fun found_0 () Bool)
4  (declare-fun x1_0 () Int)
5  (declare-fun x0_0 () Int)
6  (declare-fun w1_0 () Bool)
7  (declare-fun w0_0 () Bool)
8  (assert (let ((a!1 (ite (bvslt #b0 #b0) (- (bv2int #b0) 2) (bv2int #b0)))
9       (a!2 (ite (bvslt #b01 #b00) (- (bv2int #b01) 4) (bv2int #b01)))
10      (a!3 (ite (bvslt #b010 #b000) (- (bv2int #b010) 8) (bv2int #b010))))
11 (let ((a!4 (= a!1 (+ a!2 (* (- x0_0 a!3) (- x0_0 a!3)))))
12      (a!5 (= a!1 (+ a!2 (* (- x1_0 a!3) (- x1_0 a!3)))))
13      (a!7 (or (<= i_0 (- (- a!1 y_0) a!2)) (<= (+ y_0 a!2) i_0)))
14      (a!8 (not (= a!1 (* (- i_0 a!3) (- i_0 a!3)))))
15      (a!10 (= a!1 (+ a!2 (* (- y_0 a!3) (- y_0 a!3))))))
16 (let ((a!6 (and (not w0_0)
17                (not w1_0)
18                (=> (and found_0 (not a!4)) (= y_0 x1_0))
19                (=> (or (and found_0 a!4) (and a!4 a!5)) (= y_0 x0_0))
20                (=> (or a!4 a!5) (= found_0 true))
21                (= found_0 false)
22                (= y_0 a!1)
23                (= x0_0 a!1)
24                (= x1_0 a!1)
25                (= i_0 a!1)))
26      (a!9 (=> (or (<= i_0 (- y_0 a!2)) (<= (- a!1 y_0) i_0)) a!8)))
27 (let ((a!11 (=> found_0
28                (and (=> (< y_0 a!1) (=> a!7 a!8)) (=> (<= a!1 y_0) a!9) a!10))))
29   (not (=> a!6 a!11)))))))
```

Figure 5.3: $\mathcal{D}(s_0) \rightarrow \Phi_{\text{SearchZero}}$ in SMT-LIB Format

5.1.3 Experiment Results for SearchZero

Except for the root state, all the other states in the EFSM as shown in Figure 5.2 are nontrivial SCCs. The loop indices map them accordingly to the **while**(σ) S statements in the program. We can use $\Phi_{\text{SearchZero}}$ as SCC invariants and loop invariants for the other four VCG methods. The fifth column in Table 5.1 shows the proved VC number compared to the total VC number: We notice that providing the same SCC or loop invariants, iSAT cannot prove all VCs, but Z3 and Z3 API can do that. In general, for all five VCG methods, the parallelized version of Z3 API consumes less execution time than Z3 API and Z3. No matter whether we choose Z3 or Z3 API, there is barely a difference of the SMT checking time for the $\sum$-Format, $\bigvee$-Format, and $\bigwedge$-Format. The execution time to check VCs depends on the SMT tool we choose, and it does not rely

on the formats we use for this module. Moreover, we notice that the execution time spent on the preparation work is longer than the execution time to generate VCs, which is reasonable, since parsing input files usually takes a long time.

5.2 Synchronous Quartz Program `VectorLengthN`

Module `VectorLengthN` computes the euclidean length of a `N`-dimensional vector `v`,

$$\text{i.e., } \sqrt{\sum_{\texttt{i}=0}^{\texttt{N}-1} \texttt{v[i]}^2} \text{ for integer coordinates } \texttt{v[i]}.$$

We thereby compute just an integer approximation to the square root, which is the largest integer less than the real square root.

5.2.1 Module `VectorLengthN` and its EFSM

In Figure 5.4, the dimension of the vector is defined as a macro in the second line. The vector `v` in line 3 is an input variable, while `len` and `rdy` are output variables. The local variables `p`, `x`, and `y` are declared in line 4. From line 6 to line 8, the `N` loops compute the squares of each dimension `v[i]` in parallel. After that, the square of all dimensions will be summed up as shown in line 10, so that the square root of `v` can be computed from line 12 to line 15. Once the program terminates, the output variable `len` will be assigned to the value of the length of the vector `v[i]`, while the other output variable `rdy` will be set to **`true`**. The correctness of the program is ensured by the formula shown in the last two lines.

```
1  // N-dimentional Vector
2  macro N = ?;
3  module VectorLengthN([N]nat ?v, nat !len, event !rdy ) {
4    [N]nat p, x, y;
5    // compute the square of each v[i] in p[i]
6    for(i = 0..N-1) do || {
7      x[i] = v[i]; y[i] = v[i];
8      while(y[i] > 0){next(p[i]) = p[i]+x[i]; next(y[i]) = y[i]-1; w0: pause;}}
9    // compute the sum of all p[i] in p[0]
10   next(p[0]) = sum(i = 0..N-1) p[i]; w1: pause;
11   // compute the integer square root of p[0]
12   x[0] = 0; y[0] = 1;
13   while(y[0] <= p[0]) {
14     // invariant: y[0] = (x[0]+1)^2
15     next(x[0]) = x[0]+1; next(y[0]) = y[0]+2*x[0]+3; w2: pause;}
16   // return the length of the vector v
17   emit(rdy); len = x[0];}
18 satisfies{// the correctness of the program is ensured by the following formula
19   assert A G (rdy -> sum(i=0..N-1) exp(v[i],2) <= exp(len+1,2)
20                      and exp(len,2) <= sum(i=0..N-1) exp(v[i],2));}
```

Figure 5.4: Synchronous Quartz Module `VectorLengthN`

Table 5.1: Experiment Results - Module `SearchZero`

VCG Method	SMT Solver	Range	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	iSAT	[-100..100]	$\tau_{\sum}$	17/19	371.56	52.08	156.90
	Z3		$\tau_{\sum}$	19/19	366.32	51.99	404.99
	Z3		$\tau_{\bigwedge}$	1/1	368.06	52.12	334.42
	Z3		$\tau_{\bigvee}$	1/1	356.44	52.77	314.83
	Z3 API		$\tau_{\sum}$	19/19	357.95	53.38	279.97
	Z3 API		$\tau_{\bigwedge}$	1/1	354.87	51.57	234.43
	Z3 API		$\tau_{\bigvee}$	1/1	354.58	51.70	234.17
	Z3 API *async*		$\tau_{\sum}$	19/19	357.88	52.43	169.32
SCCTrans	iSAT	[-100..100]	$\tau_{\sum}$	18/19	355.35	55.81	116.35
	Z3		$\tau_{\sum}$	19/19	356.27	55.67	315.97
	Z3		$\tau_{\bigwedge}$	1/1	353.18	55.72	317.40
	Z3		$\tau_{\bigvee}$	1/1	354.02	55.63	317.97
	Z3 API		$\tau_{\sum}$	19/19	355.03	55.81	233.79
	Z3 API		$\tau_{\bigwedge}$	1/1	357.21	55.69	234.18
	Z3 API		$\tau_{\bigvee}$	1/1	352.17	55.61	235.59
	Z3 API *async*		$\tau_{\sum}$	19/19	354.68	55.88	167.86
SCCPath	iSAT	[-100..100]	$\tau_{\sum}$	18/19	356.48	58.23	121.12
	Z3		$\tau_{\sum}$	19/19	354.18	57.60	316.75
	Z3		$\tau_{\bigwedge}$	1/1	354.17	59.77	314.89
	Z3		$\tau_{\bigvee}$	1/1	352.81	57.50	313.54
	Z3 API		$\tau_{\sum}$	19/19	353.73	57.71	237.87
	Z3 API		$\tau_{\bigwedge}$	1/1	352.20	57.74	237.04
	Z3 API		$\tau_{\bigvee}$	1/1	356.53	57.68	238.40
	Z3 API *async*		$\tau_{\sum}$	19/19	352.96	57.92	171.99
LoopTrans	iSAT	[-100..100]	$\tau_{\sum}$	19/20	353.48	59.69	126.92
	Z3		$\tau_{\sum}$	20/20	356.77	59.82	342.40
	Z3		$\tau_{\bigwedge}$	1/1	352.51	59.62	342.04
	Z3		$\tau_{\bigvee}$	1/1	353.81	59.58	343.80
	Z3 API		$\tau_{\sum}$	20/20	355.57	59.87	277.82
	Z3 API		$\tau_{\bigwedge}$	1/1	359.04	60.04	277.85
	Z3 API		$\tau_{\bigvee}$	1/1	354.87	59.81	276.21
	Z3 API *async*		$\tau_{\sum}$	20/20	353.69	59.62	187.72
LoopPath	iSAT	[-100..100]	$\tau_{\sum}$	19/20	359.38	61.67	125.71
	Z3		$\tau_{\sum}$	20/20	353.83	60.91	343.08
	Z3		$\tau_{\bigwedge}$	1/1	354.20	61.09	337.69
	Z3		$\tau_{\bigvee}$	1/1	353.17	61.08	343.39
	Z3 API		$\tau_{\sum}$	20/20	353.94	61.07	277.00
	Z3 API		$\tau_{\bigwedge}$	1/1	354.58	60.95	278.28
	Z3 API		$\tau_{\bigvee}$	1/1	356.00	61.29	278.38
	Z3 API *async*		$\tau_{\sum}$	20/20	353.81	61.15	187.00

Module `VectorLengthN` can be scaled to evaluate the VCG methods. In the next two sections, we first explain how to generate VCs by SCCPath and LoopPath methods to prove the correctness of the 2-dimensional case when

$$\Phi_{\text{VectorLength2D}} \;:=\; \texttt{rdy} \rightarrow (\texttt{len}^2 \leq (\texttt{v[0]}^2 + \texttt{v[1]}^2) < (\texttt{len}+1)^2)$$

and then show experimental results up to 5-dimensional instances in Section 5.2.4.

5.2.2 VCG using SCCPath for `VectorLengthN` with `N` := 2

As shown in Figure 5.5, the EFSM of the 2-dimensional instance has seven control-flow states (s_0 is the root state and s_6 is the state for termination), which form five nontrivial SCCs and two trivial SCCs, i.e., each SCC has only one state. We thereby use the following SCC-invariants for nontrivial and trivial SCCs:

- $\mathcal{I}_0 := (\texttt{p[0]} + \texttt{x[0]} \cdot \texttt{y[0]} = \texttt{v[0]}^2) \wedge (0 \leq \texttt{y[0]}) \wedge (\texttt{p[1]} = \texttt{v[1]}^2) \wedge (\texttt{len} = 0) \wedge \neg\texttt{rdy}$
- $\mathcal{I}_1 := (\texttt{p[0]} + \texttt{x[0]} \cdot \texttt{y[0]} = \texttt{v[0]}^2) \wedge (0 \leq \texttt{y[0]}) \wedge (0 \leq \texttt{y[1]}) \wedge (\texttt{p[1]} + \texttt{x[1]} \cdot \texttt{y[1]} = \texttt{v[1]}^2) \wedge (\texttt{len} = 0) \wedge \neg\texttt{rdy}$
- $\mathcal{I}_2 := (\texttt{p[1]} + \texttt{x[1]} \cdot \texttt{y[1]} = \texttt{v[1]}^2) \wedge (0 \leq \texttt{y[1]}) \wedge (\texttt{p[0]} = \texttt{v[0]}^2) \wedge (\texttt{len} = 0) \wedge \neg\texttt{rdy}$
- $\mathcal{I}_3 := (\texttt{y[0]} = (\texttt{x[0]}+1)^2) \wedge (\texttt{p[0]} = \texttt{v[0]}^2 + \texttt{v[1]}^2) \wedge ((\texttt{len} = 0) \wedge \neg\texttt{rdy} \wedge (\texttt{y[0]} \leq \texttt{p[0]}) \vee (\texttt{y[0]} > \texttt{p[0]}) \wedge (\texttt{p[0]} \geq \texttt{x[0]}^2) \wedge \texttt{rdy} \wedge (\texttt{len} = \texttt{x[0]}))$
- $\mathcal{I}_6 := \Phi_{\text{VectorLength2D}}$
- $\mathcal{I}_t := (\neg\texttt{w0<0>} \wedge \neg\texttt{w0<1>} \wedge \neg\texttt{w1} \wedge \neg\texttt{w2} \rightarrow (\texttt{len} = 0) \wedge (\texttt{p[0]} = 0) \wedge (\texttt{p[1]} = 0) \wedge \neg\texttt{rdy} \wedge (\texttt{x[0]} = \texttt{v[0]}) \wedge (\texttt{y[1]} = \texttt{v[0]}) \wedge (\texttt{x[1]} = \texttt{v[1]}) \wedge (\texttt{y[1]} = \texttt{v[1]})) \wedge \big(\texttt{w1} \rightarrow (\texttt{p[0]} = \texttt{v[0]}^2 + \texttt{v[1]}^2) \wedge (\texttt{x[0]} = 0) \wedge (\texttt{y[0]} = 1) \wedge (\neg\texttt{rdy} \wedge (\texttt{len} = 0) \wedge (\texttt{y[0]} \leq \texttt{p[0]}) \vee (\texttt{y[0]} > \texttt{p[0]}) \wedge (\texttt{p[0]} \geq \texttt{x[0]}^2) \wedge \texttt{rdy} \wedge (\texttt{len} = \texttt{x[0]}))\big)$

As explained in Section 3.3.2, we now set up induction bases and induction steps to prove that the above invariants hold by the SCC-Path method:

- $\mathcal{I}_0$ for $\mathcal{C}_0$:
  - Induction Base:
    - SafePath$((s_0, s_4), (true), \mathcal{I}_t, true, \mathcal{I}_0)$
    - SafePath$((s_5, s_4), (true), \mathcal{I}_1, true, \mathcal{I}_0)$
  - Induction Step:
    - SafePath$((s_4, s_4), (true), \mathcal{I}_0, true, \mathcal{I}_0)$
- $\mathcal{I}_1$ for $\mathcal{C}_1$:
  - Induction Base:
    - SafePath$((s_0, s_5), (true), \mathcal{I}_t, true, \mathcal{I}_1)$
  - Induction Step:
    - SafePath$((s_5, s_5), (true), \mathcal{I}_1, true, \mathcal{I}_1)$

Figure 5.5: EFSM of Module `VectorLengthN` with `N` := 2

- $\mathcal{I}_2$ for $\mathcal{C}_2$:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_3), (true), \mathcal{I}_t, true, \mathcal{I}_2)$
    - $\mathsf{SafePath}((s_5, s_3), (true), \mathcal{I}_1, true, \mathcal{I}_2)$
  - Induction Step:
    - $\mathsf{SafePath}((s_3, s_3), (true), \mathcal{I}_2, true, \mathcal{I}_2)$
- $\mathcal{I}_3$ for $\mathcal{C}_3$:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_1, s_2), (true, true), \mathcal{I}_t, \mathcal{I}_t, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_3, s_1, s_2), (true, true), \mathcal{I}_2, \mathcal{I}_t, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_4, s_1, s_2), (true, true), \mathcal{I}_0, \mathcal{I}_t, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_5, s_1, s_2), (true, true), \mathcal{I}_1, \mathcal{I}_t, \mathcal{I}_3)$
  - Induction Step:
    - $\mathsf{SafePath}((s_2, s_2), (true), \mathcal{I}_3, true, \mathcal{I}_3)$
- $\mathcal{I}_6$ for $\mathcal{C}_6$:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_1, s_6), (true, true), \mathcal{I}_t, \mathcal{I}_t, \mathcal{I}_6)$
    - $\mathsf{SafePath}((s_3, s_1, s_6), (true, true), \mathcal{I}_2, \mathcal{I}_t, \mathcal{I}_6)$
    - $\mathsf{SafePath}((s_4, s_1, s_6), (true, true), \mathcal{I}_0, \mathcal{I}_t, \mathcal{I}_6)$
    - $\mathsf{SafePath}((s_5, s_1, s_6), (true, true), \mathcal{I}_1, \mathcal{I}_t, \mathcal{I}_6)$
    - $\mathsf{SafePath}((s_2, s_6), (true), \mathcal{I}_3, true, \mathcal{I}_6)$
  - Induction Step:
    - $\mathsf{SafePath}((s_6, s_6), (true), \mathcal{I}_6, true, \mathcal{I}_6)$

Each SafePath condition represents a VC. We need some more VCs to ensure that the safety property $\Phi_{\text{VectorLength2D}}$ holds in the root state s_0, and the trivial/nontrivial SCC invariants implies $\Phi_{\text{VectorLength2D}}$ as well.

5.2.3 VCG using LoopPath for `VectorLengthN` with `N` := 2

As can be seen in Figure 5.4, the program has three **`while`**(σ) `S` statements, and the program can terminate. Therefore, except for $\mathcal{I}_\emptyset$, we provide each **`while`**(σ) `S` statement a loop invariant, i.e., $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$, together with an additional one $\mathcal{I}_4$ for the hidden loop when the program terminates.

- $\mathcal{I}_1 := \texttt{w0<0>} \wedge (\texttt{len} = 0) \wedge \neg\texttt{rdy} \wedge (\texttt{p[0]} + \texttt{x[0]} \cdot \texttt{y[0]} = \texttt{v[0]}^2) \wedge (0 \leq \texttt{y[0]}) \wedge (\texttt{w0<1>} \rightarrow (\texttt{y[1]} = 0) \wedge (\texttt{p[1]} = \texttt{v[1]}^2))$
- $\mathcal{I}_2 := \texttt{w0<1>} \wedge (\texttt{len} = 0) \wedge \neg\texttt{rdy} \wedge (\texttt{p[1]} + \texttt{x[1]} \cdot \texttt{y[1]} = \texttt{v[1]}^2) \wedge (0 \leq \texttt{y[1]}) \wedge (\texttt{w0<0>} \rightarrow (\texttt{y[0]} = 0) \wedge (\texttt{p[0]} = \texttt{v[0]}^2))$
- $\mathcal{I}_3 := \texttt{w2} \wedge (\texttt{y[0]} = (\texttt{x[0]} + 1)^2) \wedge (\texttt{p[0]} = \texttt{v[0]}^2 + \texttt{v[1]}^2) \wedge (\neg\texttt{rdy} \wedge (\texttt{len} = 0) \wedge (\texttt{y[0]} \leq \texttt{p[0]}) \vee (\texttt{y[0]} > \texttt{p[0]}) \wedge (\texttt{p[0]} \geq \texttt{x[0]}^2) \wedge \texttt{rdy} \wedge (\texttt{len} = \texttt{x[0]}))$
- $\mathcal{I}_4 := \Phi_{\text{VectorLength2D}}$
- $\mathcal{I}_\emptyset := \Phi_{\text{VectorLength2D}}$

As explained in Section 3.4.1, we now set up the following induction bases and induction

steps to prove that the above invariants hold in the loops by Loop-Path method:

- $\mathcal{I}_1$ for $\mathcal{L}_1$ with control-flow label `w0<0>`:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_5), (true), \mathcal{I}_\emptyset, true, \mathcal{I}_1)$
    - $\mathsf{SafePath}((s_0, s_4), (true), \mathcal{I}_\emptyset, true, \mathcal{I}_1)$
  - Induction Step:
    - $\mathsf{SafePath}((s_5, s_5), (true), \mathcal{I}_1, true, \mathcal{I}_1)$
    - $\mathsf{SafePath}((s_5, s_4), (true), \mathcal{I}_1, true, \mathcal{I}_1)$
    - $\mathsf{SafePath}((s_4, s_4), (true), \mathcal{I}_1, true, \mathcal{I}_1)$
- $\mathcal{I}_2$ for $\mathcal{L}_2$ with control-flow label `w0<1>`:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_5), (true), \mathcal{I}_\emptyset, true, \mathcal{I}_2)$
    - $\mathsf{SafePath}((s_0, s_3), (true), \mathcal{I}_\emptyset, true, \mathcal{I}_2)$
  - Induction Step:
    - $\mathsf{SafePath}((s_5, s_5), (true), \mathcal{I}_2, true, \mathcal{I}_2)$
    - $\mathsf{SafePath}((s_5, s_3), (true), \mathcal{I}_2, true, \mathcal{I}_2)$
    - $\mathsf{SafePath}((s_3, s_3), (true), \mathcal{I}_2, true, \mathcal{I}_2)$
- $\mathcal{I}_3$ for $\mathcal{L}_3$ with control-flow label `w2`:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_1, s_2), (true, true), \mathcal{I}_\emptyset, \mathcal{I}_\emptyset, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_3, s_1, s_2), (true, true), \mathcal{I}_2, \mathcal{I}_\emptyset, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_4, s_1, s_2), (true, true), \mathcal{I}_1, \mathcal{I}_\emptyset, \mathcal{I}_3)$
    - $\mathsf{SafePath}((s_5, s_1, s_2), (true, true), \mathcal{I}_1 \wedge \mathcal{I}_2, \mathcal{I}_\emptyset, \mathcal{I}_3)$
  - Induction Step:
    - $\mathsf{SafePath}((s_4, s_4), (true), \mathcal{I}_3, true, \mathcal{I}_3)$
- $\mathcal{I}_4$ for $\mathcal{L}_4$ for termination:
  - Induction Base:
    - $\mathsf{SafePath}((s_0, s_1, s_6), (true, true), \mathcal{I}_\emptyset, \mathcal{I}_\emptyset, \mathcal{I}_4)$
    - $\mathsf{SafePath}((s_3, s_1, s_6), (true, true), \mathcal{I}_2, \mathcal{I}_\emptyset, \mathcal{I}_4)$
    - $\mathsf{SafePath}((s_4, s_1, s_6), (true, true), \mathcal{I}_1, \mathcal{I}_\emptyset, \mathcal{I}_4)$
    - $\mathsf{SafePath}((s_5, s_1, s_6), (true, true), \mathcal{I}_1 \wedge \mathcal{I}_2, \mathcal{I}_\emptyset, \mathcal{I}_4)$
    - $\mathsf{SafePath}((s_2, s_6), (true), \mathcal{I}_3, true, \mathcal{I}_6)$
  - Induction Step:
    - $\mathsf{SafePath}((s_6, s_6), (true), \mathcal{I}_4, true, \mathcal{I}_4)$

Each SafePath condition is a single VC. Again, we need some more VCs to ensure that the safety property $\Phi_{\text{VectorLength2D}}$ holds in the root state s_0, and all loop invariants implies $\Phi_{\text{VectorLength2D}}$ as well.

5.2.4 Scalability

In general, it is not recommended to use the Transition-based method since it becomes harder to determine a control-flow state assertion for each node when the size of the program grows. Considering the VCG methods using SCC assertions, the number of SCCs can grow exponentially with the size of the program as well. The Quartz program

Table 5.2: Experiment Results - Module `VectorLengthN` with `N` := 2

VCG Method	SMT Solver	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	Z3	$\tau_{\sum}$	25/25	434.44	50.79	709.25
	Z3	$\tau_{\bigwedge}$	1/1	382.84	51.00	564.00
	Z3	$\tau_{\bigvee}$	1/1	378.45	50.90	641.87
	Z3 API	$\tau_{\sum}$	25/25	382.48	50.88	410.83
	Z3 API	$\tau_{\bigwedge}$	1/1	384.83	50.80	408.53
	Z3 API	$\tau_{\bigvee}$	1/1	371.57	51.27	412.34
	Z3 API *async*	$\tau_{\sum}$	25/25	387.97	50.89	158.65
SCCTrans	Z3	$\tau_{\sum}$	25/25	389.60	55.12	645.11
	Z3	$\tau_{\bigwedge}$	1/1	380.53	55.32	630.03
	Z3	$\tau_{\bigvee}$	1/1	377.09	54.98	642.14
	Z3 API	$\tau_{\sum}$	25/25	382.67	55.02	422.02
	Z3 API	$\tau_{\bigwedge}$	1/1	389.04	55.49	422.49
	Z3 API	$\tau_{\bigvee}$	1/1	390.17	55.24	422.08
	Z3 API *async*	$\tau_{\sum}$	25/25	386.69	55.08	158.56
SCCPath	Z3	$\tau_{\sum}$	27/27	388.19	58.33	785.10
	Z3	$\tau_{\bigwedge}$	1/1	389.12	58.61	753.49
	Z3	$\tau_{\bigvee}$	1/1	380.52	58.69	745.05
	Z3 API	$\tau_{\sum}$	27/27	385.67	58.29	460.67
	Z3 API	$\tau_{\bigwedge}$	1/1	390.92	58.60	465.76
	Z3 API	$\tau_{\bigvee}$	1/1	391.27	58.60	458.39
	Z3 API *async*	$\tau_{\sum}$	27/27	386.93	58.45	181.16
LoopTrans	Z3	$\tau_{\sum}$	25/25	388.22	58.89	690.87
	Z3	$\tau_{\bigwedge}$	1/1	380.27	59.21	823.10
	Z3	$\tau_{\bigvee}$	1/1	386.84	59.08	852.89
	Z3 API	$\tau_{\sum}$	25/25	382.31	59.41	468.60
	Z3 API	$\tau_{\bigwedge}$	1/1	387.50	58.98	470.19
	Z3 API	$\tau_{\bigvee}$	1/1	385.69	59.00	466.84
	Z3 API *async*	$\tau_{\sum}$	25/25	388.13	58.98	156.84
LoopPath	Z3	$\tau_{\sum}$	27/27	389.10	61.80	997.55
	Z3	$\tau_{\bigwedge}$	1/1	359.30	61.92	948.14
	Z3	$\tau_{\bigvee}$	1/1	381.77	61.45	841.53
	Z3 API	$\tau_{\sum}$	27/27	388.18	61.55	499.93
	Z3 API	$\tau_{\bigwedge}$	1/1	386.75	61.49	499.85
	Z3 API	$\tau_{\bigvee}$	1/1	388.97	61.86	501.43
	Z3 API *async*	$\tau_{\sum}$	27/27	375.22	61.86	172.93

Table 5.3: Experiment Results - Module `VectorLengthN` with `N` := 3

VCG Method	SMT Solver	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	Z3	$\tau_{\sum}$	51/51	456.96	62.16	2072.82
	Z3	$\tau_{\bigwedge}$	1/1	393.61	61.85	2124.78
	Z3	$\tau_{\bigvee}$	1/1	393.32	62.60	2380.60
	Z3 API	$\tau_{\sum}$	51/51	397.72	61.55	1198.24
	Z3 API	$\tau_{\bigwedge}$	1/1	401.40	61.84	1195.53
	Z3 API	$\tau_{\bigvee}$	1/1	402.37	61.65	1190.69
	Z3 API *async*	$\tau_{\sum}$	51/51	380.62	61.74	341.45
SCCTrans	Z3	$\tau_{\sum}$	51/51	388.45	66.17	2139.24
	Z3	$\tau_{\bigwedge}$	1/1	389.13	66.53	2440.72
	Z3	$\tau_{\bigvee}$	1/1	399.76	66.83	1848.32
	Z3 API	$\tau_{\sum}$	51/51	395.19	66.50	1263.92
	Z3 API	$\tau_{\bigwedge}$	1/1	401.25	66.40	1221.20
	Z3 API	$\tau_{\bigvee}$	1/1	402.28	66.27	1224.28
	Z3 API *async*	$\tau_{\sum}$	51/51	399.82	66.43	342.55
SCCPath	Z3	$\tau_{\sum}$	57/57	401.58	72.74	2698.91
	Z3	$\tau_{\bigwedge}$	1/1	391.59	72.25	2197.59
	Z3	$\tau_{\bigvee}$	1/1	389.05	72.52	2314.93
	Z3 API	$\tau_{\sum}$	57/57	403.90	73.34	1337.19
	Z3 API	$\tau_{\bigwedge}$	1/1	401.32	72.51	1398.82
	Z3 API	$\tau_{\bigvee}$	1/1	411.27	73.87	1346.50
	Z3 API *async*	$\tau_{\sum}$	57/57	402.83	72.30	419.47
LoopTrans	Z3	$\tau_{\sum}$	59/59	385.88	76.04	3237.78
	Z3	$\tau_{\bigwedge}$	1/1	402.39	76.24	3286.25
	Z3	$\tau_{\bigvee}$	1/1	400.03	76.59	3370.92
	Z3 API	$\tau_{\sum}$	59/59	403.35	76.30	1849.85
	Z3 API	$\tau_{\bigwedge}$	1/1	402.07	76.19	1848.24
	Z3 API	$\tau_{\bigvee}$	1/1	399.97	76.05	1845.13
	Z3 API *async*	$\tau_{\sum}$	59/59	394.68	76.90	423.77
LoopPath	Z3	$\tau_{\sum}$	65/65	394.65	83.12	3896.60
	Z3	$\tau_{\bigwedge}$	1/1	397.76	84.76	3904.41
	Z3	$\tau_{\bigvee}$	1/1	395.51	83.17	4108.24
	Z3 API	$\tau_{\sum}$	65/65	395.87	83.39	1969.30
	Z3 API	$\tau_{\bigwedge}$	1/1	401.41	83.03	1963.83
	Z3 API	$\tau_{\bigvee}$	1/1	370.14	82.95	2083.27
	Z3 API *async*	$\tau_{\sum}$	65/65	397.51	83.77	500.81

Table 5.4: Experiment Results - Module `VectorLengthN` with `N` := 4

VCG Method	SMT Solver	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	Z3	$\tau_{\bigvee}$	121/121	427.17	135.39	9618.95
	Z3	$\tau_{\bigwedge}$	1/1	436.06	135.94	9723.99
	Z3	$\tau_{\bigvee}$	1/1	429.95	136.28	9808.30
	Z3 API	$\tau_{\bigvee}$	121/121	439.75	135.79	4839.75
	Z3 API	$\tau_{\bigwedge}$	1/1	443.34	134.84	4921.77
	Z3 API	$\tau_{\bigvee}$	1/1	433.04	135.41	4899.68
	Z3 API *async*	$\tau_{\bigvee}$	121/121	440.78	135.70	1159.25
SCCTrans	Z3	$\tau_{\bigvee}$	121/121	439.86	141.50	9352.39
	Z3	$\tau_{\bigwedge}$	1/1	441.27	140.55	9593.78
	Z3	$\tau_{\bigvee}$	1/1	438.31	139.99	9345.50
	Z3 API	$\tau_{\bigvee}$	121/121	437.07	140.18	4954.51
	Z3 API	$\tau_{\bigwedge}$	1/1	438.33	140.40	4910.09
	Z3 API	$\tau_{\bigvee}$	1/1	428.46	140.62	4978.46
	Z3 API *async*	$\tau_{\bigvee}$	121/121	440.34	142.36	1142.49
SCCPath	Z3	$\tau_{\bigvee}$	135/135	431.37	160.62	11266.91
	Z3	$\tau_{\bigwedge}$	1/1	426.61	159.26	11201.50
	Z3	$\tau_{\bigvee}$	1/1	436.90	158.93	11236.71
	Z3 API	$\tau_{\bigvee}$	135/135	440.94	159.94	5419.32
	Z3 API	$\tau_{\bigwedge}$	1/1	442.49	159.05	5358.18
	Z3 API	$\tau_{\bigvee}$	1/1	440.99	158.72	5190.73
	Z3 API *async*	$\tau_{\bigvee}$	135/135	441.45	160.60	1325.46
LoopTrans	Z3	$\tau_{\bigvee}$	169/169	429.61	231.88	18066.73
	Z3	$\tau_{\bigwedge}$	1/1	438.22	233.37	17561.73
	Z3	$\tau_{\bigvee}$	1/1	440.40	232.38	17839.58
	Z3 API	$\tau_{\bigvee}$	169/169	437.83	231.13	9062.71
	Z3 API	$\tau_{\bigwedge}$	1/1	441.98	234.04	9027.34
	Z3 API	$\tau_{\bigvee}$	1/1	438.16	232.26	9094.41
	Z3 API *async*	$\tau_{\bigvee}$	169/169	441.94	233.49	1974.69
LoopPath	Z3	$\tau_{\bigvee}$	183/183	439.62	253.34	22106.42
	Z3	$\tau_{\bigwedge}$	1/1	429.38	250.32	21683.81
	Z3	$\tau_{\bigvee}$	1/1	424.95	247.36	21974.05
	Z3 API	$\tau_{\bigvee}$	183/183	437.09	248.19	9389.55
	Z3 API	$\tau_{\bigwedge}$	1/1	438.14	246.70	9503.84
	Z3 API	$\tau_{\bigvee}$	1/1	425.25	254.30	9563.78
	Z3 API *async*	$\tau_{\bigvee}$	183/183	441.99	248.23	2021.59

Table 5.5: Experiment Results - Module `VectorLengthN` with `N` := 5

VCG Method	SMT Solver	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	Z3	$\tau_{\bigwedge}$	315/315	574.41	682.01	46873.32
	Z3	$\tau_{\bigwedge}$	1/1	566.48	684.38	46106.28
	Z3	$\tau_{\bigvee}$	1/1	590.54	688.51	46531.15
	Z3 API	$\tau_{\bigwedge}$	315/315	565.96	677.23	21633.23
	Z3 API	$\tau_{\bigwedge}$	1/1	588.88	763.28	21620.11
	Z3 API	$\tau_{\bigvee}$	1/1	580.02	682.30	21383.04
	Z3 API *async*	$\tau_{\bigwedge}$	315/315	589.34	679.88	6493.50
SCCTrans	Z3	$\tau_{\bigwedge}$	315/315	565.73	692.81	46034.93
	Z3	$\tau_{\bigwedge}$	1/1	581.24	699.39	46845.68
	Z3	$\tau_{\bigvee}$	1/1	433.18	140.28	9912.47
	Z3 API	$\tau_{\bigwedge}$	315/315	587.99	694.24	21466.12
	Z3 API	$\tau_{\bigwedge}$	1/1	587.15	693.59	21366.60
	Z3 API	$\tau_{\bigvee}$	1/1	441.48	145.01	4985.22
	Z3 API *async*	$\tau_{\bigwedge}$	315/315	582.07	692.20	5514.18
SCCPath	Z3	$\tau_{\bigwedge}$	345/345	587.88	768.29	52431.88
	Z3	$\tau_{\bigwedge}$	1/1	572.04	760.37	52320.02
	Z3	$\tau_{\bigvee}$	1/1	568.60	757.26	52447.26
	Z3 API	$\tau_{\bigwedge}$	345/345	582.43	765.23	22875.73
	Z3 API	$\tau_{\bigwedge}$	1/1	572.60	759.60	23071.09
	Z3 API	$\tau_{\bigvee}$	1/1	585.37	818.27	22850.36
	Z3 API *async*	$\tau_{\bigwedge}$	345/345	590.17	764.28	6022.41
LoopTrans	Z3	$\tau_{\bigwedge}$	531/531	577.48	1618.27	100904.19
	Z3	$\tau_{\bigwedge}$	1/1	565.51	1583.05	100088.78
	Z3	$\tau_{\bigvee}$	1/1	571.38	1616.93	101040.08
	Z3 API	$\tau_{\bigwedge}$	531/531	584.20	1547.86	48994.69
	Z3 API	$\tau_{\bigwedge}$	1/1	586.39	1538.83	47898.58
	Z3 API	$\tau_{\bigvee}$	1/1	586.81	1555.25	47872.01
	Z3 API *async*	$\tau_{\bigwedge}$	531/531	584.99	1541.33	17717.09
LoopPath	Z3	$\tau_{\bigwedge}$	561/561	588.51	1632.54	126123.35
	Z3	$\tau_{\bigwedge}$	1/1	588.77	1659.76	130354.01
	Z3	$\tau_{\bigvee}$	1/1	561.69	1608.75	129769.30
	Z3 API	$\tau_{\bigwedge}$	561/561	586.21	1611.08	49058.29
	Z3 API	$\tau_{\bigwedge}$	1/1	586.36	1623.36	50079.27
	Z3 API	$\tau_{\bigvee}$	1/1	587.03	1677.65	50091.06
	Z3 API *async*	$\tau_{\bigwedge}$	561/561	574.64	1605.33	13219.31

`VectorLengthN` is one of these worst case examples. Indeed, when all `N` **`while`**(σ) `S` statements run in parallel, we have to consider all $2^{\mathtt{N}}$ different situations where one of them terminated while the others were still running. As a consequence, we will have to set up exponentially many induction proofs for the SCC methods, since the worst case of the `VectorLengthN` program is caused by the parallel composition of **`while`**(σ) `S` statements.

Meanwhile, proving assertions for **`while`**(σ) `S` statements will only require a linear number of these induction proofs. Notice that for those **`while`**(σ) `S` statements run in parallel, their loop invariants should contain other **`while`**(σ) `S` statements' termination conditions. For example, the following formula is the loop invariant $\mathcal{I}_1$ defined in Section 5.2.3:

$$\underbrace{\mathtt{w0{<}0{>}} \wedge (\mathtt{len} = 0) \wedge \neg \mathtt{rdy} \wedge (\mathtt{p[0]} + \mathtt{x[0]} \cdot \mathtt{y[0]} = \mathtt{v[0]}^2) \wedge (0 \leq \mathtt{y[0]})}_{\text{Computation of } \mathtt{v[0]}^2 \text{ does not terminate yet!}}$$

$$\wedge \quad \underbrace{(\mathtt{w0{<}1{>}} \rightarrow (\mathtt{y[1]} = 0) \wedge (\mathtt{p[1]} = \mathtt{v[1]}^2))}_{\text{Computation of } \mathtt{v[1]}^2 \text{ has terminated!}}$$

Without the subformula describes the computation of $\mathtt{v[1]}^2$ has terminated, the induction bases of $\mathcal{I}_3$ and $\mathcal{I}_4$ generated by `LoopPath` method can not be proved.

We scale the `VectorLengthN` program up to five dimensions to evaluate the five induction-based VCG methods. It is very likely that the number of boolean variables has great impact on iSAT, so that it ran out of memory quite often for this module's instances. Therefore, only the execution time using Z3 and Z3 API has been collected for those instances.

Tables 5.2, 5.3, 5.4, and 5.5 show the experimental results. After comparing the execution time to check VCs, again we found that accessing the parallelized version of Z3 API in Mono/.NET leads to the best performance, and it does not rely on the formats we use.

5.3 Hybrid Quartz Program `WaterTank`

We illustrate the VCG methods with a parameterized water tank system in this section.

5.3.1 Module `WaterTank` and its EFSM

Module `WaterTank` in Figure 5.6 describes the following scenario: A water tank regulates water level `y` by filling or emptying the water tank. The initial water level is `y` = 5. `inV` and `outV` are parameters that describe the water level changes, and the module can either set **`drv`**`(y)<-` `inV` or **`drv`**`(y)<-` `-outV` to fill or empty the water tank. The reaction of the system is delayed, so that both the filling and emptying procedures are extended by `delta`-unit time. While `delta` is real, `y` is hybrid real.

```
1  module WaterTank(real ?delta, nat ?inV, ?outV){
2    hybrid real y; y = 5;
3    // the filling and emptying procedures happen continuously
4    loop{
5      // the water level increases with inV m/s until y = 10.
6      fl1,fl2:flow{
7        drv(y) <- inV;
8      }until(cont(y) >= 10);
9      // the water level still increases after delta-unit time
10     fl3,fl4:flow{
11       drv(y) <- inV;
12     }until(cont(y)-y >= delta*inV);
13     // the water level decreases with outV m/s until y = 5
14     fl5,fl6:flow{
15       drv(y) <- -outV;
16     }until(cont(y) <= 5);
17     // the water level still decreases after delta-unit time
18     fl7,fl8:flow{
19       drv(y) <- -outV;
20     }until(y-cont(y) >= delta*outV);
21   }
22 }
23 satisfies{// the water level will not exceed a certain range.
24   assert A G (inV > 0 and outV >0 and delta > 0) ->
25                 (5-outV*delta <= cont(y)) and (cont(y) <= 10+inV*delta);}
```

Figure 5.6: Hybrid Quartz Module WaterTank

The safety property of the water tank system states that the water level will not exceed a certain range. Hence, we check the following safety property:

$$\Phi_{\text{WaterTank}} \;:=\; (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y})) \;\wedge\; (\textbf{cont}(\texttt{y}) \leq 10 + \texttt{inV} * \texttt{delta})$$

The program has four flow statements, and the filling and emptying procedures run in turn, so that the program will not terminate. The corresponding EFSM is displayed in Figure 5.7. It has six nodes in total, one trivial SCC (the root node s_0), and one nontrivial SCC that contains the remaining five nodes. To prove $\Phi_{\text{WaterTank}}$ by the VCG methods, we give the following continuous invariants for each node s_i:

- $\Psi_0 \;:=\; (\textbf{cont}(\texttt{y}) \leq 10) \;\wedge\; (5 \leq \textbf{cont}(\texttt{y}))$
- $\Psi_1 \;:=\; (\textbf{cont}(\texttt{y}) \leq 10) \;\wedge\; (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y}))$
- $\Psi_2 \;:=\; (10 \leq \textbf{cont}(\texttt{y})) \;\wedge\; (\textbf{cont}(\texttt{y}) \leq 10 + \texttt{inV} * \texttt{delta})$
- $\Psi_3 \;:=\; (5 \leq \textbf{cont}(\texttt{y})) \;\wedge\; (\textbf{cont}(\texttt{y}) \leq 10 + \texttt{inV} * \texttt{delta})$
- $\Psi_4 \;:=\; (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y})) \;\wedge\; (\textbf{cont}(\texttt{y}) \leq 5)$
- $\Psi_5 \;:=\; (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y})) \;\wedge\; (\textbf{cont}(\texttt{y}) \leq 10)$

property, and then give the complete experimental results in Section 5.3.4.

5.3.2 VCG using SCCTrans for WaterTank

The SCC invariants provided for the SCCTrans method are as follows:

- $\mathcal{I}_1 := \neg\texttt{fl1} \wedge \neg\texttt{fl2} \wedge \neg\texttt{fl3} \wedge \neg\texttt{fl4} \wedge \neg\texttt{fl5} \wedge \neg\texttt{fl6} \wedge \neg\texttt{fl7} \wedge \neg\texttt{fl8} \wedge \big((\texttt{inV} > 0) \;\wedge\; (\texttt{outV} > 0) \wedge (\texttt{delta} > 0) \rightarrow (\textbf{cont}(\texttt{y}) \leq 10) \wedge (5 \leq \textbf{cont}(\texttt{y}))\big)$

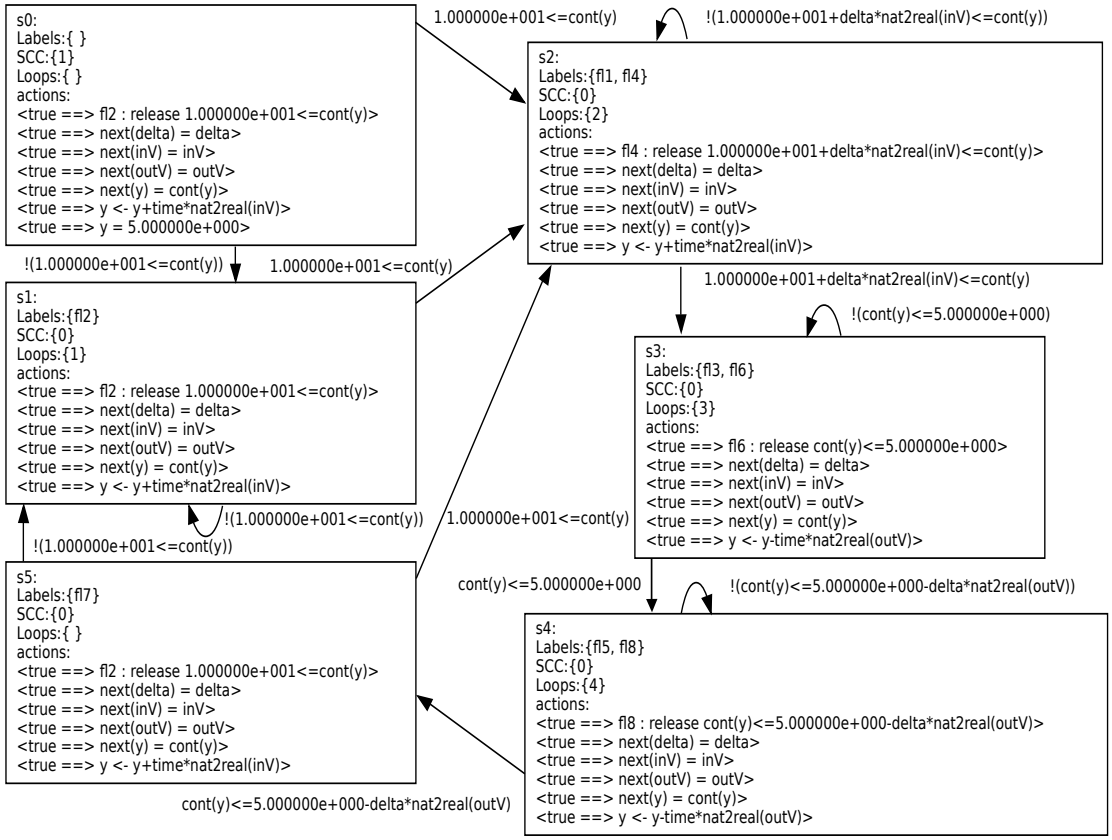

Figure 5.7: EFSM of Hybrid Quartz Module `WaterTank`

- $\mathcal{I}_0 := \Big(\neg\mathtt{fl1} \wedge \mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8} \rightarrow \big((\mathtt{inV} > 0) \ \wedge$
  $(\mathtt{outV} > 0) \wedge (\mathtt{delta} > 0) \rightarrow (\mathbf{cont}(\mathtt{y}) \leq 10) \wedge (5 - \mathtt{outV} * \mathtt{delta} \leq \mathbf{cont}(\mathtt{y}))\big)\Big)$
  $\wedge \Big(\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8} \rightarrow \big((\mathtt{inV} > 0) \ \wedge$
  $(\mathtt{outV} > 0) \wedge (\mathtt{delta} > 0) \rightarrow (10 \leq \mathbf{cont}(\mathtt{y})) \wedge (\mathbf{cont}(\mathtt{y}) \leq 10 + \mathtt{inV} * \mathtt{delta})\big)\Big)$
  $\wedge \Big(\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8}) \rightarrow \big((\mathtt{inV} > 0) \ \wedge$
  $(\mathtt{outV} > 0) \wedge (\mathtt{delta} > 0) \rightarrow (5 \leq \mathbf{cont}(\mathtt{y})) \wedge (\mathbf{cont}(\mathtt{y}) \leq 10 + \mathtt{inV} * \mathtt{delta})\big)\Big)$
  $\wedge \Big(\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \mathtt{fl8} \rightarrow \big((\mathtt{inV} > 0) \ \wedge$
  $(\mathtt{outV} > 0) \wedge (\mathtt{delta} > 0) \rightarrow (5 - \mathtt{outV} * \mathtt{delta} \leq \mathbf{cont}(\mathtt{y})) \wedge (\mathbf{cont}(\mathtt{y}) \leq 5)\big)\Big)$
  $\wedge \Big(\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \mathtt{fl7} \wedge \neg\mathtt{fl8} \rightarrow \big((\mathtt{inV} > 0) \ \wedge$
  $(\mathtt{outV} > 0) \wedge (\mathtt{delta} > 0) \rightarrow (5 - \mathtt{outV} * \mathtt{delta} \leq \mathbf{cont}(\mathtt{y})) \wedge (\mathbf{cont}(\mathtt{y}) \leq 10)\big)\Big)$
  $\wedge \big(\neg\mathtt{fl1} \wedge \mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8} \ \vee$
  $\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \ \mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8} \ \vee$
  $\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \neg\mathtt{fl8} \ \vee$
  $\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \neg\mathtt{fl7} \wedge \mathtt{fl8} \ \vee$
  $\neg\mathtt{fl1} \wedge \neg\mathtt{fl2} \wedge \neg\mathtt{fl3} \wedge \neg\mathtt{fl4} \wedge \neg\mathtt{fl5} \wedge \neg\mathtt{fl6} \wedge \mathtt{fl7} \wedge \neg\mathtt{fl8}\big)$

As explained in Section 3.3.2, the following induction bases and steps of $\mathcal{I}_0$ are set up for the nontrivial SCC $\mathcal{C}_0$ to prove the validity of $\Phi_{\text{WaterTank}}$ by the SCC-Trans method:

- Induction Base of $\mathcal{I}_0$ for $\mathcal{C}_0$:

$$\mathsf{SafeTrans}((s_0,s_1),(\Psi_0),\mathcal{I}_1,true,\mathcal{I}_0) \qquad \mathsf{SafeTrans}((s_0,s_2),(\Psi_0),\mathcal{I}_1,true,\mathcal{I}_0)$$

- Induction Step of $\mathcal{I}_0$ for $\mathcal{C}_0$:

$$\begin{array}{ll}
\mathsf{SafeTrans}((s_1,s_1),(\Psi_1),\mathcal{I}_0,true,\mathcal{I}_0) & \mathsf{SafeTrans}((s_3,s_4),(\Psi_3),\mathcal{I}_0,true,\mathcal{I}_0)\\
\mathsf{SafeTrans}((s_1,s_2),(\Psi_1),\mathcal{I}_0,true,\mathcal{I}_0) & \mathsf{SafeTrans}((s_4,s_4),(\Psi_4),\mathcal{I}_0,true,\mathcal{I}_0)\\
\mathsf{SafeTrans}((s_2,s_2),(\Psi_2),\mathcal{I}_0,true,\mathcal{I}_0) & \mathsf{SafeTrans}((s_4,s_5),(\Psi_4),\mathcal{I}_0,true,\mathcal{I}_0)\\
\mathsf{SafeTrans}((s_2,s_3),(\Psi_2),\mathcal{I}_0,true,\mathcal{I}_0) & \mathsf{SafeTrans}((s_5,s_1),(\Psi_5),\mathcal{I}_0,true,\mathcal{I}_0)\\
\mathsf{SafeTrans}((s_3,s_3),(\Psi_3),\mathcal{I}_0,true,\mathcal{I}_0) & \mathsf{SafeTrans}((s_5,s_2),(\Psi_5),\mathcal{I}_0,true,\mathcal{I}_0)
\end{array}$$

Each SafeTrans condition is thereby a single VC. Again we need some more VCs to ensure that the safety property $\Phi_{\text{WaterTank}}$ holds in the root state s_0, and that the trivial/nontrivial SCC invariants together with the continuous invariants of each node imply $\Phi_{\text{WaterTank}}$ as well. To have some idea on how the VCs look like, we give the input file in SMT-LIB format in Figure 5.8 that describe $\mathcal{D}(s_0) \rightarrow \Phi_{\text{WaterTank}}$.

5.3.3 VCG using LoopTrans for `WaterTank`

The loop invariants provided for the LoopTrans method are the following:

- $\mathcal{I}_1 := \texttt{fl2} \wedge (\textbf{cont}(\texttt{y}) \leq 10) \wedge (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y}))$
- $\mathcal{I}_2 := \texttt{fl4} \wedge (10 \leq \textbf{cont}(\texttt{y})) \wedge (\textbf{cont}(\texttt{y}) \leq 10 + \texttt{inV} * \texttt{delta})$
- $\mathcal{I}_3 := \texttt{fl6} \wedge (5 \leq \textbf{cont}(\texttt{y})) \wedge (\textbf{cont}(\texttt{y}) \leq 10 + \texttt{inV} * \texttt{delta})$
- $\mathcal{I}_4 := \texttt{fl8} \wedge (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y})) \wedge (\textbf{cont}(\texttt{y}) \leq 5)$
- $\mathcal{I}_\emptyset := \texttt{fl7} \wedge (5 - \texttt{outV} * \texttt{delta} \leq \textbf{cont}(\texttt{y})) \wedge (\textbf{cont}(\texttt{y}) \leq 10) \vee \neg\texttt{run} \wedge ((\textbf{cont}(\texttt{y}) \leq 10) \wedge (5 \leq \textbf{cont}(\texttt{y})))$

As explained in Section 3.4.2, our tool automatically sets up the following induction bases and induction steps to prove the validity of $\Phi_{\text{WaterTank}}$ by the Loop-Trans method:

- $\mathcal{I}_1$ for $\mathcal{L}_1$ with control-flow label `fl2`:
  - Induction Base:
    - $\mathsf{SafeTrans}((s_0,s_1),(\Psi_0),\mathcal{I}_\emptyset,true,\mathcal{I}_1)$
    - $\mathsf{SafeTrans}((s_5,s_1),(\Psi_5),\mathcal{I}_\emptyset,true,\mathcal{I}_1)$
  - Induction Step:
    - $\mathsf{SafeTrans}((s_1,s_1),(\Psi_1),\mathcal{I}_1,true,\mathcal{I}_1)$
- $\mathcal{I}_2$ for $\mathcal{L}_2$ with control-flow label `fl4`:
  - Induction Base:
    - $\mathsf{SafeTrans}((s_0,s_2),(\Psi_0),\mathcal{I}_\emptyset,true,\mathcal{I}_2)$
    - $\mathsf{SafeTrans}((s_1,s_2),(\Psi_1),\mathcal{I}_1,true,\mathcal{I}_2)$
    - $\mathsf{SafeTrans}((s_5,s_2),(\Psi_5),\mathcal{I}_\emptyset,true,\mathcal{I}_2)$
  - Induction Step:
    - $\mathsf{SafeTrans}((s_2,s_2),(\Psi_2),\mathcal{I}_2,true,\mathcal{I}_2)$
- $\mathcal{I}_3$ for $\mathcal{L}_3$ with control-flow label `fl6`:
  - Induction Base:

```
1  (declare-fun delta () Real)
2  (declare-fun |cont(y)| () Real)
3  (declare-fun inV () Int)
4  (declare-fun outV () Int)
5  (declare-fun fl2 () Bool)
6  (declare-fun fl1 () Bool)
7  (declare-fun fl3 () Bool)
8  (declare-fun fl4 () Bool)
9  (declare-fun fl5 () Bool)
10 (declare-fun fl6 () Bool)
11 (declare-fun fl7 () Bool)
12 (declare-fun fl8 () Bool)
13 (assert (let ((a!1 (and (< (bv2int #b0) inV)
14              true
15              (<= 0 inV)
16              (< (bv2int #b0) outV)
17              true
18              (<= 0 outV)
19              (< 0.0 delta)
20              true
21              (<= 0.0 delta)))
22     (a!3 (<= (- 5.0 (* delta (to_real outV))) |cont(y)|))
23     (a!4 (<= |cont(y)| (+ 10.0 (* delta (to_real inV))))))
24 (let ((a!2 (and (not fl8)
25              (not fl7)
26              (not fl6)
27              (not fl5)
28              (not fl4)
29              (not fl3)
30              (not fl1)
31              (not fl2)
32              (=> a!1
33                  (and (<= 5.0 |cont(y)|)
34                       true
35                       (<= 0.0 |cont(y)|)
36                       (<= |cont(y)| 10.0)
37                       (<= 0.0 |cont(y)|)
38                       true))))
39     (a!5 (=> a!1
40              (and a!3
41                  (and true true true (<= 0.0 delta) true)
42                  (<= 0.0 |cont(y)|)
43                  a!4
44                  (<= 0.0 |cont(y)|)
45                  (and true true true (<= 0.0 delta) true)))))
46  (not (=> a!2 a!5)))))
```

Figure 5.8: $\mathcal{D}(s_0) \rightarrow \Phi_{\text{WaterTank}}$ in SMT-LIB Format

  - $*$ $\mathsf{SafeTrans}((s_2, s_3), (\Psi_2), \mathcal{I}_2, true, \mathcal{I}_3)$
- Induction Step:
  - $*$ $\mathsf{SafeTrans}((s_3, s_3), (\Psi_3), \mathcal{I}_3, true, \mathcal{I}_3)$

- $\mathcal{I}_4$ for $\mathcal{L}_4$ with control-flow label `fl7`:
  - Induction Base:
    - $*$ $\mathsf{SafeTrans}((s_3, s_4), (\Psi_3), \mathcal{I}_3, true, \mathcal{I}_4)$
  - Induction Step:

  - $*$ SafeTrans$((s_4, s_4), (\Psi_4), \mathcal{I}_4, true, \mathcal{I}_4)$
- $\mathcal{I}_\emptyset$: SafeTrans$((s_4, s_5), (\Psi_4), \mathcal{I}_3, true, \mathcal{I}_\emptyset)$

Each SafeTrans condition is thereby a single VC. We need some more VCs to ensure that the safety property $\Phi_{\text{WaterTank}}$ holds in the root state s_0, the loop invariants and all continuous invariants imply $\Phi_{\text{WaterTank}}$ as well.

5.3.4 Experiment Results for WaterTank

Notice that both inV and outV are $\mathbb{Z}$-variables, all the others are $\mathbb{R}$-variables, therefore, the underlying satisfiability problem contains MINLP problems as defined in Section 2.5.2. From the fifth column of Table 5.6, we know that iSAT can prove at most half of the total VCs, and also, the external version of Z3 is unstable in the 5-times iteration experiment, since the proved VC number is not equal to the total VC number when using either SCC or loop assertions in $\sum$-Format. Again, accessing the parallelized version of Z3 API in Mono/.NET leads to the best performance comparing the execution time to check VCs, it does not rely on the formats we use for this module.

5.4 Hybrid Quartz Program SlowDown

SlowDown behavior is a safety control component of an autonomous mobile robot. It requires the vehicle always keeps a certain distance from the obstacle [Rop+16].

```
1  macro delta= ?, max_velocity_v = ?, protect_distance = ?;
2  module SlowDown(real ? obstacle_x, ? vehicle_xi) {
3    real max_velocity_a, slow_down_a, distance_front, vehicle_v;
4    hybrid real vehicle_x; vehicle_x = vehicle_xi;
5    // the movement of the vehicle towards the obstacle
6    loop {
7      // detect the distance between the vehicle and the obstacle
8      distance_front = obstacle_x-vehicle_x;
9      // decide the velocity for the movement
10     if (distance_front <= protect_distance) slow_down_a = 1.0;
11     else slow_down_a = 0.0;
12     max_velocity_a = 1.0-slow_down_a;
13     vehicle_v = max_velocity_v*max_velocity_a;
14     // the vehicle goes forward for delta-unit time
15     w0,w1: flow{
16         drv(vehicle_x) <- vehicle_v;
17     }until(cont(time)-time >= delta);
18   }
19 }
20 satisfies{
21     assume ((protect_distance >= 0) and (delta > 0) and (max_velocity_v > 0));
22     // for bounded initial distance
23     assume (abs(vehicle_xi-obstacle_x) >= protect_distance);
24     assume (abs(max_velocity_v * delta) <= protect_distance);
25     // the vehicle keeps a certain distance from the obstacle
26     assert A G (abs(obstacle_x-vehicle_x) >= protect_distance-vehicle_v*delta);}
```

Figure 5.9: Hybrid Quartz Module SlowDown

Table 5.6: Experiment Results - Module `WaterTank`

VCG Method	SMT Solver	Range	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	iSAT	[-100..100]	$\tau_{\sum}$	12/25	364.79	68.80	111.79
	Z3		$\tau_{\sum}$	25/25	364.84	69.31	14673.97
	Z3		$\tau_{\bigwedge}$	1/1	358.52	68.71	345.36
	Z3		$\tau_{\bigvee}$	1/1	389.70	73.89	350.83
	Z3 API		$\tau_{\sum}$	25/25	387.63	72.46	208.80
	Z3 API		$\tau_{\bigwedge}$	1/1	359.66	68.85	197.08
	Z3 API		$\tau_{\bigvee}$	1/1	364.46	68.69	199.73
	Z3 API *async*		$\tau_{\sum}$	25/25	363.68	69.18	169.80
SCCTrans	iSAT	[-100..100]	$\tau_{\sum}$	8/21	360.20	73.67	123.41
	Z3		$\tau_{\sum}$	20/21	363.74	74.11	315.83
	Z3		$\tau_{\bigwedge}$	1/1	363.15	74.17	318.52
	Z3		$\tau_{\bigvee}$	1/1	361.54	74.07	316.04
	Z3 API		$\tau_{\sum}$	21/21	361.53	73.79	189.94
	Z3 API		$\tau_{\bigwedge}$	1/1	362.62	73.93	188.74
	Z3 API		$\tau_{\bigvee}$	1/1	361.37	74.00	190.97
	Z3 API *async*		$\tau_{\sum}$	21/21	363.46	74.19	166.46
SCCPath	iSAT	[-100..100]	$\tau_{\sum}$	8/21	359.91	74.76	124.79
	Z3		$\tau_{\sum}$	20/21	360.93	79.97	371.50
	Z3		$\tau_{\bigwedge}$	1/1	386.39	74.60	344.10
	Z3		$\tau_{\bigvee}$	1/1	382.02	74.29	328.80
	Z3 API		$\tau_{\sum}$	21/21	384.92	74.22	186.68
	Z3 API		$\tau_{\bigwedge}$	1/1	387.05	74.58	188.56
	Z3 API		$\tau_{\bigvee}$	1/1	376.75	74.41	189.78
	Z3 API *async*		$\tau_{\sum}$	21/21	364.64	74.48	134.61
LoopTrans	iSAT	[-100..100]	$\tau_{\sum}$	12/24	388.62	75.91	112.93
	Z3		$\tau_{\sum}$	23/24	377.65	76.03	326.99
	Z3		$\tau_{\bigwedge}$	1/1	387.95	76.58	352.91
	Z3		$\tau_{\bigvee}$	1/1	386.29	76.28	321.39
	Z3 API		$\tau_{\sum}$	24/24	383.08	76.11	187.13
	Z3 API		$\tau_{\bigwedge}$	1/1	355.54	75.67	187.40
	Z3 API		$\tau_{\bigvee}$	1/1	387.72	75.97	187.70
	Z3 API *async*		$\tau_{\sum}$	24/24	388.06	76.24	148.69
LoopPath	iSAT	[-100..100]	$\tau_{\sum}$	11/23	371.64	78.61	115.04
	Z3		$\tau_{\sum}$	22/23	387.56	78.23	324.17
	Z3		$\tau_{\bigwedge}$	1/1	378.44	78.16	311.40
	Z3		$\tau_{\bigvee}$	1/1	380.23	78.03	335.55
	Z3 API		$\tau_{\sum}$	23/23	379.96	79.12	183.44
	Z3 API		$\tau_{\bigwedge}$	1/1	391.08	78.86	184.32
	Z3 API		$\tau_{\bigvee}$	1/1	387.50	78.02	182.17
	Z3 API *async*		$\tau_{\sum}$	23/23	383.58	78.36	141.28

5.4.1 Module SlowDown and its EFSM

Module SlowDown in Figure 5.9 with its EFSM in Figure 5.10 is a discretization model for the SlowDown behavior in horizontal **x** dimension, which can be extended to two dimensions.

The sample time delta, the maximum forward speed max_velocity_v, and a default distance protect_distance to avoid the collision, are declared in the **macro** part. The main **module** starts with the declaration of the input variables. The initial positions of the obstacle and the vehicle are denoted as obstacle_x and vehicle_xi, respectively. The local variables max_velocity_a, slow_down_a, and distance_front are used to determine vehicle_v, i.e., the speed of the vehicle. vehicle_x is the position of the vehicle that changes continuously.

The movement of the vehicle is represented by a **loop** statement, which contains:

- One discrete macro step to set the velocity of the vehicle, as shown from line 9 to line 13.
- Another continuous macro step from line 15 to line 17 that describes the vehicle goes forward with the speed vehicle_v for delta unit time.

The assertion in line 26 of the last **satisfies** part specifies that the vehicle always keeps a certain distance from the obstacle, provided the prerequisite conditions in line 20-25 which states:

- The vehicle starts from a bounded distance from the obstacle.
- The vehicle can proceed at most a certain distance with the maximum speed in one sample time.

5.4.2 Experiment Results for SlowDown

All variables are real-valued, therefore the underlying satisfiability problem contains NLP problems as defined in Section 2.5.2.

From the fifth column of Table 5.7, we notice that iSAT can only prove less than half of the total VCs. Also, it can be deduced from the fifth column that all variants of Z3 — i.e., external Z3, Z3 API, and the parallelized version of Z3 API — are unstable according to our 5-times iteration experiment: The proved VC number is not equal to the total VC number when using $\sum$-Format.

Only by either $\bigwedge$-Format or $\bigvee$-Format with external Z3 or Z3 API can we prove the validity of all VCs. And, once more, the execution time consumed by Z3 API is shorter than the one by external Z3.

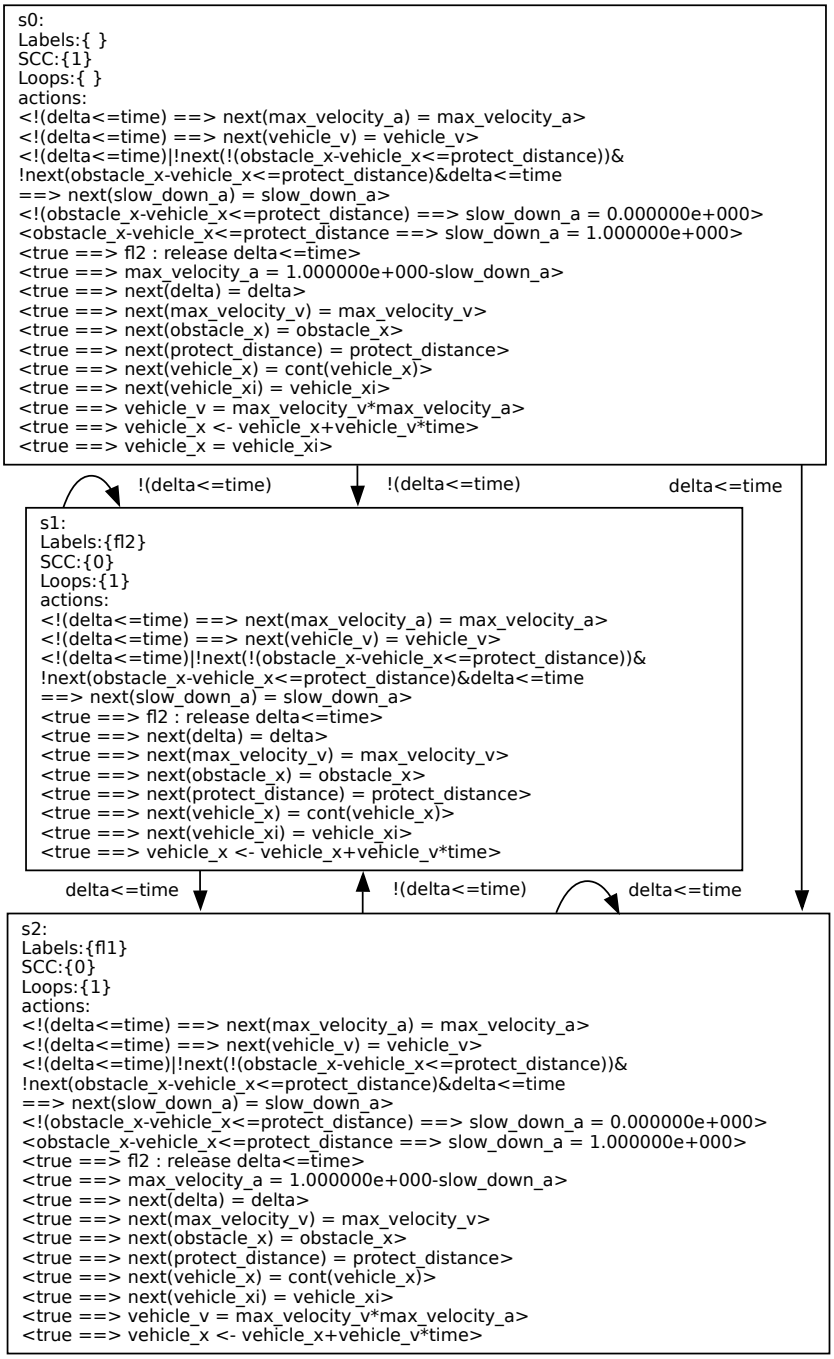

Figure 5.10: EFSM of Module `SlowDown`

Table 5.7: Experiment Results - Module `SlowDown`

VCG Method	SMT Solver	Range	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	iSAT	[-100..100]	$\tau_{\sum}$	5/13	342.66	65.73	636.04
	Z3		$\tau_{\sum}$	11/13	369.22	65.87	200.97
	Z3		$\tau_{\bigwedge}$	1/1	360.53	66.03	209.36
	Z3		$\tau_{\bigvee}$	1/1	371.01	66.05	193.04
	Z3 API		$\tau_{\sum}$	11/13	364.13	65.94	119.73
	Z3 API		$\tau_{\bigwedge}$	1/1	369.81	65.68	120.10
	Z3 API		$\tau_{\bigvee}$	1/1	361.88	65.98	120.79
	Z3 API *async*		$\tau_{\sum}$	11/13	375.23	66.24	117.93
SCCTrans	iSAT	[-100..100]	$\tau_{\sum}$	4/12	344.08	69.25	3931.05
	Z3		$\tau_{\sum}$	10/12	363.83	69.05	184.25
	Z3		$\tau_{\bigwedge}$	1/1	360.91	69.10	192.67
	Z3		$\tau_{\bigvee}$	1/1	363.15	69.03	186.44
	Z3 API		$\tau_{\sum}$	10/12	366.96	68.94	113.78
	Z3 API		$\tau_{\bigwedge}$	1/1	371.05	69.14	113.91
	Z3 API		$\tau_{\bigvee}$	1/1	364.90	69.81	114.41
	Z3 API *async*		$\tau_{\sum}$	10/12	361.03	69.23	122.63
SCCPath	iSAT	[-100..100]	$\tau_{\sum}$	4/12	370.12	71.55	3875.76
	Z3		$\tau_{\sum}$	10/12	364.16	70.60	198.77
	Z3		$\tau_{\bigwedge}$	1/1	374.97	70.38	189.04
	Z3		$\tau_{\bigvee}$	1/1	369.46	70.12	188.95
	Z3 API		$\tau_{\sum}$	10/12	371.88	70.25	113.50
	Z3 API		$\tau_{\bigwedge}$	1/1	343.44	70.39	113.48
	Z3 API		$\tau_{\bigvee}$	1/1	367.49	70.19	120.03
	Z3 API *async*		$\tau_{\sum}$	10/12	357.21	70.05	116.43
LoopTrans	iSAT	[-100..100]	$\tau_{\sum}$	4/12	371.81	72.40	3951.95
	Z3		$\tau_{\sum}$	10/12	365.53	72.28	183.97
	Z3		$\tau_{\bigwedge}$	1/1	377.37	73.23	188.02
	Z3		$\tau_{\bigvee}$	1/1	359.14	72.26	180.12
	Z3 API		$\tau_{\sum}$	10/12	363.25	72.14	113.54
	Z3 API		$\tau_{\bigwedge}$	1/1	372.68	72.54	113.71
	Z3 API		$\tau_{\bigvee}$	1/1	371.09	72.50	112.96
	Z3 API *async*		$\tau_{\sum}$	10/12	342.71	72.43	118.61
LoopPath	iSAT	[-100..100]	$\tau_{\sum}$	4/12	360.02	72.73	3872.54
	Z3		$\tau_{\sum}$	10/12	366.12	73.19	191.00
	Z3		$\tau_{\bigwedge}$	1/1	365.05	73.77	233.90
	Z3		$\tau_{\bigvee}$	1/1	371.05	72.72	190.21
	Z3 API		$\tau_{\sum}$	10/12	368.16	72.74	112.90
	Z3 API		$\tau_{\bigwedge}$	1/1	358.80	73.05	113.45
	Z3 API		$\tau_{\bigvee}$	1/1	371.81	72.81	112.99
	Z3 API *async*		$\tau_{\sum}$	10/12	364.68	73.10	124.83

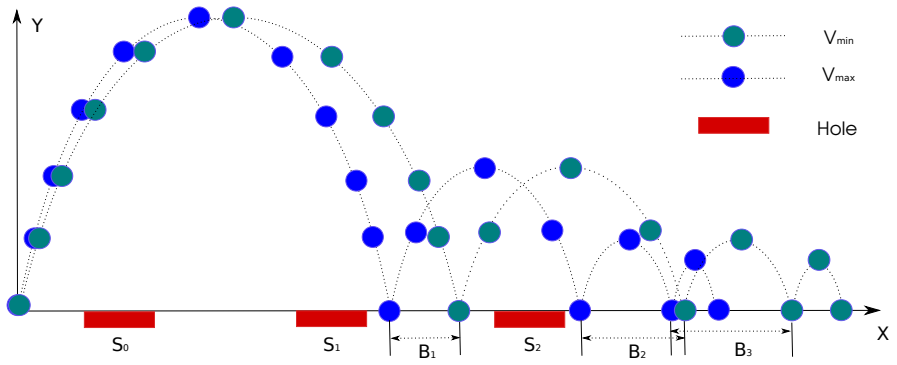

Figure 5.11: The Ball and Holes Scenario

5.5 Hybrid Quartz Program `ParametricBall`

Figure 5.11 describes the following scenario: Throwing a ball from the ground with a non-zero speed, the ball will bounce continuously according to the environment conditions, e.g. the wind and the gravity. However, there are some holes on the ground where the ball will not be able to bounce again if it falls into the holes.

5.5.1 Module `ParametricBall` and its EFSM

Module `ParametricBall` in Figure 5.12 with its EFSM in Figure 5.13 is an abstract model of the above scenario. It consists of three parts. The **`macro`** part gives the static hole region information, including $\mathtt{N} \in \mathbb{Z}$ for the hole number, and two real-valued arrays `S_min[N]` and `S_max[N]` for the hole locations. There are three holes in total, $\mathtt{S\_0} = [0.0, 8.0]$, $\mathtt{S\_1} = [10.0, 18.0]$, and $\mathtt{S\_2} = [22.5, 24.5]$.

The main **`module`** starts with the input $\mathbb{R}$-variables, including `a_x`, `Vx_init`, `Vy_init`, `Ts`, and `c`. Among them, `a_x` is the instant wind acceleration that is sampled by the given period `Ts`, and `a_x` keeps unchanged until the next sample period. `Vx_init` and `Vy_init` stand for the initial horizontal and vertical speed components. `Vx` is influenced by the wind, while `Vy` is controlled by the gravity, which means the acceleration in `Y` dimension is the constant value `g`. The ball may lose some energy in `Y` dimension after hitting the ground, where the energy loss coefficient `c` satisfies $0.0 \leq \mathtt{c} \leq 1.0$.

Local variables `B` and `H` encode the bounce trace of the two dimensions that are decided by the speed components `Vx` and `Vy`, respectively. The other time related variable `T` works as a timer to stimulate the sampling procedure, so that both `a` and `T` should be reset after every `Ts` time units. Variable `n` increases its value whenever the ball hits the ground. The bouncing procedure is represented by a **`loop`** statement, which contains one continuous macro step for the dynamic evolutions, and another discrete macro step for reseting either the wind acceleration at each sample time or the velocity of the ball when it hits the ground. Checking whether the ball could avoid all holes, is equal to verifying whether there exists a path in the future so that the ball may fall

into the hole region, as stated in the last **satisfies** part.

```
1  macro N = 3; macro S_min = [0.0,10.0,22.5]; macro S_max= [8.0,18.0,24.5];
2  module ParametricBall(real ?a_x, ?Vx_init, ?Vy_init, ?Ts ?c){
3    hybrid real Vx,Vy,T,B,H; real a; nat n;
4    // Initialization configuration
5    Vx = Vx_init; Vy = Vy_init; a = a_x;
6    // Ball Bounces
7    loop{
8      // Continuous dynamics
9      flow{
10         drv(B) <- cont(Vx); drv(Vx) <- a;
11         drv(H) <- cont(Vy); drv(Vy) <- -9.8;
12         drv(T) <- 1.0;
13     }until((cont(T) >= Ts) or ((cont(H) <= 0.0)&(cont(Vy) <= 0.0)));
14     if((T >= Ts) and ((cont(H) <= 0.0) and (cont(Vy) <= 0.0))){
15     //Case 1: the ball hits the ground at the sample time
16       next(a) = a_x; next(T) = 0.0;
17       next(Vy) = -c*Vy; next(n) = n+1;}
18     else{
19       if((T >= Ts) and !((cont(H) <= 0.0) and (cont(Vy) <= 0.0))){
20       //Case 2: sample time, the ball does not hit the ground
21         next(a)= a_x; next(T) = 0.0;}
22       else{
23         //Case 3: the ball hits the ground not at the sample time
24         next(Vy) = -c*Vy; next(n) = n+1;}}
25     pause;}}
26 satisfies{
27   assert E F (exists(i = 0..N-1) ((S_min[i] <= B) and (B <= S_max[i]) and (H <= 0.0));}
```

Figure 5.12: Hybrid Quartz Module `ParametricBall`

5.5.2 Validation by VCG Methods

Assume that the wind acceleration is constant and that the initial variable values are: `Vx_init` $= 0.0$, `Vy_init` $= 19.6$, and `c` is constantly equal to 0.5. The following safety regions for `a_x` is generated by a symbolic simulation algorithm shown in Appendix A:

$$2.25 < \texttt{a\_x} < 2.8125 \ \vee \ 3.0625 < \texttt{a\_x}$$

This safety region can be validated by the VCG methods. The underlying satisfiability problem contains MINLP problems as defined in Section 2.5.2, since except for $\texttt{n} \in \mathbb{Z}$, all the other local variables are $\mathbb{R}$-variables.

The experimental result is similar to module `SlowDown`. As shown in Table 5.8, iSAT can only prove no more than half of the total VCs, as shown in the fifth column. And also, all variants of Z3, i.e., external Z3, Z3 API, and the parallelized version of Z3 API, are unstable in our 5-times iteration experiment: The proved VC number is not equal to the total VC number when using $\sum$-Format.

Only by either $\bigwedge$-Format or $\bigvee$-Format with external Z3 or Z3 API can prove the validity of all VCs. And, as before, the execution time consumed by Z3 API is shorter than the one by external Z3.

Figure 5.13: EFSM of Module `ParametricBall`

Table 5.8: Experiment Results - Module `ParametricBall`

VCG Method	SMT Solver	Range	Format	VC Number	EFSM-Inv	VCG	SMT
TransBased	iSAT	[-100..100]	$\tau_{\sum}$	8/16	450.25	70.36	119.23
	Z3		$\tau_{\sum}$	11/16	415.25	70.37	268.03
	Z3		$\tau_{\bigwedge}$	1/1	417.02	70.67	280.20
	Z3		$\tau_{\bigvee}$	1/1	412.42	70.12	258.23
	Z3 API		$\tau_{\sum}$	11/16	417.02	70.69	167.41
	Z3 API		$\tau_{\bigwedge}$	1/1	419.89	70.40	166.92
	Z3 API		$\tau_{\bigvee}$	1/1	419.44	70.46	168.46
	Z3 API *async*		$\tau_{\sum}$	11/16	420.39	70.80	131.89
SCCTrans	iSAT	[-100..100]	$\tau_{\sum}$	6/14	405.82	73.50	112.99
	Z3		$\tau_{\sum}$	9/14	416.55	74.05	231.31
	Z3		$\tau_{\bigwedge}$	1/1	412.67	73.52	240.85
	Z3		$\tau_{\bigvee}$	1/1	416.70	73.74	271.23
	Z3 API		$\tau_{\sum}$	9/14	414.57	73.32	152.42
	Z3 API		$\tau_{\bigwedge}$	1/1	423.04	73.86	151.83
	Z3 API		$\tau_{\bigvee}$	1/1	419.99	73.13	152.07
	Z3 API *async*		$\tau_{\sum}$	9/14	422.27	73.98	118.19
SCCPath	iSAT	[-100..100]	$\tau_{\sum}$	6/14	392.89	74.77	113.68
	Z3		$\tau_{\sum}$	9/14	410.01	74.62	235.63
	Z3		$\tau_{\bigwedge}$	1/1	412.38	74.54	228.16
	Z3		$\tau_{\bigvee}$	1/1	411.33	74.61	233.63
	Z3 API		$\tau_{\sum}$	9/14	413.62	74.53	151.55
	Z3 API		$\tau_{\bigwedge}$	1/1	420.95	74.52	152.27
	Z3 API		$\tau_{\bigvee}$	1/1	420.60	74.92	152.65
	Z3 API *async*		$\tau_{\sum}$	9/14	421.59	74.50	117.96
LoopTrans	iSAT	[-100..100]	$\tau_{\sum}$	6/14	415.35	77.98	108.46
	Z3		$\tau_{\sum}$	9/14	414.51	77.89	239.85
	Z3		$\tau_{\bigwedge}$	1/1	413.36	77.56	242.65
	Z3		$\tau_{\bigvee}$	1/1	417.24	78.02	237.63
	Z3 API		$\tau_{\sum}$	9/14	415.02	77.55	151.76
	Z3 API		$\tau_{\bigwedge}$	1/1	423.20	78.03	151.65
	Z3 API		$\tau_{\bigvee}$	1/1	420.18	77.53	153.20
	Z3 API *async*		$\tau_{\sum}$	9/14	395.02	77.96	118.84
LoopPath	iSAT	[-100..100]	$\tau_{\sum}$	6/13	407.26	79.11	121.69
	Z3		$\tau_{\sum}$	8/13	414.67	78.82	238.29
	Z3		$\tau_{\bigwedge}$	1/1	414.80	78.72	250.37
	Z3		$\tau_{\bigvee}$	1/1	409.81	79.42	261.19
	Z3 API		$\tau_{\sum}$	8/13	409.52	78.96	156.12
	Z3 API		$\tau_{\bigwedge}$	1/1	421.29	78.77	155.65
	Z3 API		$\tau_{\bigvee}$	1/1	424.26	79.45	156.21
	Z3 API *async*		$\tau_{\sum}$	8/13	419.74	79.20	118.75

Chapter 6

Conclusion

This thesis develops induction-based techniques for the verification of safety property for synchronous and hybrid programs. The imperative synchronous language Quartz and its extension to hybrid systems are used to sustain the findings.

- As a first contribution, we used *Floyd's induction-based* approach to generate verification conditions for synchronous and hybrid programs. We then introduced five VCG methods for deductive verification that use inductive assertions (i.e., invariants) to decompose the overall proof goal.

  Given the right assertions, the proposed VCG methods can automatically generate a set of VCs that can then be checked by SMT solvers or automated theorem provers. The methods are proved sound and can be applied to any program with a state-based semantics. We also prove relative completeness for the methods, provided that the underlying assertion language is expressive enough.

  The last experimental part demonstrates the feasibility of the induction-based VCG methods by several synchronous and hybrid Quartz programs.

- As the second contribution, we optimize the PDR method by using the distinction between the control- and dataflow of imperative synchronous programs.

  We present two methods to compute additional control-flow information that differ in how precisely they approximate the reachable control-flow states and, consequently, in their required runtime. Before calling the PDR method, the transition relation can be enhanced by the derived additional program control-flow information such that less CTIs will be generated. Many safety properties become inductive with respect to such an enhanced transition relation.

  After calling the PDR method, we can use the control-flow information to reason about the unreachability of counterexamples and to generalize them to all states with the same control-flow states. Thus, we avoid expensive clause generalizations required to narrow the over-approximations of the clause sets.

As potential future work, the following directions are worth addressing:

- Inductive invariants are one of the key elements to prove safety properties through the proposed VCG methods: The user is supposed to provide such prerequisite information. Indeed, automatic inductive invariant generation is not an easy task. Applying the PDR method in the framework of compositional verification can be used an alternative to automatically generating inductive invariants. This is the case since once the PDR method can prove a safety property of a program component it may also yield a better inductive invariant for that component: The Ψ-sequence of computed clause sets are inductive and, at the same time, refine the possible non-inductive safety property.

- The proposed PDR optimization includes the control-flow part of input imperative synchronous programs. The similar challenge of coming up with optimizations that exploit the dataflow part still stands. To solve this problem, one would need powerful SMT solvers or theorem provers, since the underlying satisfying problem is undecidable. According to our experimental results, it would be a good idea to try more than one backend tool with various input formats. Furthermore, integrating other effective inductive methods like differential induction can likely ease the difficulties faced when solving this kind of verification goals.

- Applying our proposed VCG methods to embedded systems used in realistic safety-critical applications is (due to the inherent complexity) currently infeasible. Hence, except for the module `SlowDown`, all our other examples did not originate in safety-critical applications. Module `SlowDown` is obtained by exploiting the modularity and network structure of behavior-based control systems where only the relevant behaviors are isolated and analyzed by techniques for hybrid system verification. To address such shortcomings, abstraction could be another research topic for reducing the complexity of real systems to high level models. This would allow for more practical experiments to be performed in order to further sustain the feasibility of the proposed verification techniques.

Bibliography

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. "TREX: A Tool for Reachability Analysis of Complex Systems". In: *Computer Aided Verification (CAV)*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. LNCS. Paris, France: Springer, 2001, pp. 368–372.

[AD94] R. Alur and D.L. Dill. "A theory of timed automata". In: *Theoretical Computer Science (TCS)* 126.2 (1994), pp. 183–235.

[ADI06] R. Alur, T. Dang, and F. Ivani. "Counter-example guided predicate abstraction of hybrid systems". In: *Theoretical Computer Science (TCS)* 354.2 (2006), pp. 250–271.

[AH97] R. Alur and T.A. Henzinger. "Modularity for Timed and Hybrid Systems". In: *Concurrency Theory (CONCUR)*. Ed. by A. Mazurkiewicz and J. Winkowski. Vol. 1243. LNCS. Warsaw, Poland: Springer, 1997, pp. 74–88.

[All70] F.E. Allen. "Control Flow Analysis". In: *ACM SIGPLAN Notices* 5.7 (1970), pp. 1–19.

[Alu+00] R. Alur, T.A. Henzinger, G. Lafferriere, and G.J. Pappas. "Discrete Abstractions of Hybrid Systems". In: *Proceedings of the IEEE* 88.7 (2000), pp. 971–984.

[Alu+93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems". In: *Hybrid Systems*. Ed. by R.L. Grossmann, A. Nerode, A.P. Ravn, and H. Rischel. Vol. 736. LNCS. Springer, 1993, pp. 209–229.

[Alu+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. "The Algorithmic Analysis of Hybrid Systems". In: *Theoretical Computer Science (TCS)* 138.1 (1995), pp. 3–34.

[Alu11] R. Alur. "Formal verification of hybrid systems". In: *Embedded Software (EMSOFT)*. Ed. by S. Chakraborty, A. Jerraya, S.K. Baruah, and S. Fischmeister. Taipei, Taiwan: ACM, 2011, pp. 273–278.

[AM71] E.A. Ashcroft and Z. Manna. *The translation of 'go to' programs to 'while' programs*. Technical Report CS-TR-71-188. Stanford, California, USA: Department of Computer Science, University of California, Jan. 1971.

[Ame+06] A.D. Ames, H. Zheng, R.D. Gregg, and S. Sastry. "Is there life after Zeno? Taking executions past the breaking (Zeno) point". In: *American Control Conference*. Minneapolis, MN, USA: IEEE Computer Society, 2006, pp. 2652–2657.

[And+09] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. "An Inverse Method for Parametric Timed Automata". In: *International Journal of Foundations of Computer Science* 20.5 (Oct. 2009), pp. 819–836.

[And09] E. André. "IMITATOR: A Tool for Synthesizing Constraints on Timing Bounds of Timed Automata". In: *Theoretical Aspects of Computing (ICTAC)*. Ed. by M. Leucker and C. Morgan. Vol. 5684. LNCS. Kuala Lumpur, Malaysia: Springer, 2009, pp. 336–342.

[AO09] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. 3rd ed. Springer, 2009.

[Apt81] K.R. Apt. "Ten Years of Hoare's Logic: A Survey-Part I". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3.4 (1981), pp. 431–483.

[Aud+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. "A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions". In: *Conference on Automated Deduction (CADE)*. Ed. by A. Voronkov. Vol. 2392. LNCS. Copenhagen, Denmark: Springer, 2002, pp. 195–210.

[Bau12] K. Bauer. "A New Modelling Language for Cyber-physical Systems". PhD. PhD thesis. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Germany, Jan. 2012.

[BB91] A. Benveniste and G. Berry. "The synchronous approach to reactive real-time systems". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.

[BC85] G. Berry and L. Cosserat. "The Esterel Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency (CONCUR)*. Ed. by S.D. Brookes, A.W. Roscoe, and G. Winskel. Vol. 197. LNCS. Pittsburgh, Pennsylvania, USA: Springer, 1985, pp. 389–448.

[BDS02] C.W. Barrett, D.L. Dill, and A. Stump. "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT". In: *Computer Aided Verification (CAV)*. Ed. by E. Brinksma and K.G. Larsen. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, 2002, pp. 236–249.

[Bel+13] P. Belotti, C. Kirches, S. Leyffer, J.T. Linderoth, J. Luedtke, and A. Mahajan. “Mixed-Integer Nonlinear Optimization”. In: *Acta Numerica*. Ed. by Arieh Iserles. Vol. 22. Cambridge University Press, 2013, pp. 1–131.

[Ber00] G. Berry. *The Esterel v5 Language Primer*. July 2000.

[Ber99] G. Berry. *The Constructive Semantics of Pure Esterel*. July 1999.

[BFT15] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015.

[BGS10] K. Bauer, R. Gentilini, and K. Schneider. “A Uniform Approach to Three-Valued Semantics for mu-Calculus on Abstractions of Hybrid Automata”. In: *Software Tools for Technology Transfer (STTT)* 12.2 (May 2010), pp. 1–15.

[BHZ06] L. Bordeaux, Y. Hamadi, and L. Zhang. “Propositional Satisfiability and Constraint Programming: A Comparative Survey”. In: *ACM Computing Surveys (CSUR)* 38.4 (Dec. 2006).

[Bie+03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. “Bounded Model Checking”. In: *Advances in Computers*. Ed. by M. Zelkowitz. Vol. 58. Academic Press, 2003, pp. 118–149.

[BKS03] G. Berry, M. Kishinevsky, and S. Singh. “System Level Design and Verification Using a Synchronous Language”. In: *International Conference on Computer-Aided Design (ICCAD)*. San Jose, California, USA: ACM/IEEE Computer Society, 2003, pp. 433–440.

[BM07] A.R. Bradley and Z. Manna. “Checking Safety by Inductive Generalization of Counterexamples to Induction”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Austin, Texas, USA: IEEE Computer Society, 2007, pp. 173–180.

[BM10] R. Brayton and A. Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *Computer Aided Verification (CAV)*. Ed. by T. Touili, B. Cook, and P. Jackson. Vol. 6174. LNCS. Edinburgh, Scotland, UK: Springer, 2010, pp. 24–40.

[Bon+08] P. Bonami, L. Biegler, A. Conn, G. CornuéJols, I. Grossmann, C. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya, and A. Wächter. “An Algorithmic Framework for Convex Mixed Integer Nonlinear Programs”. In: *Discret. Optim.* 5.2 (May 2008), pp. 186–204. ISSN: 1572-5286.

[Bon10] M.P. Bonacina. “On Theorem Proving for Program Checking: Historical Perspective and Recent Developments”. In: *Principles and Practice of Declarative Programming (PPDP)*. Hagenberg, Austria: ACM, 2010, pp. 1–11.

[BPT07] A. Bauer, M. Pister, and M. Tautschnig. “Tool-support for the analysis of hybrid systems and models”. In: *Design, Automation and Test in Europe (DATE)*. Ed. by R. Lauwereins and J. Madsen. Nice, France: IEEE Computer Society, 2007, pp. 924–929.

[Bra11] A.R. Bradley. “SAT Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by R. Jhala and D.A. Schmidt. Vol. 6538. LNCS. Austin, Texas, USA: Springer, 2011, pp. 70–87.

[Bra12a] A.R. Bradley. “IC3 and beyond: Incremental, Inductive Verification”. In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Vol. 7358. LNCS. Berkeley, California, USA: Springer, 2012, p. 4.

[Bra12b] A.R. Bradley. “Understanding IC3”. In: *Theory and Applications of Satisfiability Testing (SAT)*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. LNCS. Trento, Italy: Springer, 2012, pp. 1–14.

[Bro03] C.W. Brown. “An overview of QEPCAD B: a tool for real quantifier elimination and formula simplification”. In: *Journal of Japan Society for Symbolic and Algebraic Computation* 10.1 (2003), pp. 13–22.

[BS11] K. Bauer and K. Schneider. “Transferring Causality Analysis from Synchronous Programs to Hybrid Programs”. In: *International Modelica Conference*. Ed. by C. ClauSS. Vol. 63. Linköping Electronic Conference Proceedings. Dresden, Germany: Linköping University Electronic Press, 2011, pp. 207–217.

[BT79] S.L. Bloom and R. Tindell. “Algebraic and graph theoretic characterizations of structured flowchart schemes”. In: *Theoretical Computer Science (TCS)* 9.3 (Oct. 1979), pp. 265–286.

[Bu+08] L. Bu, Y. Li, L. Wang, and X. Li. “BACH: Bounded Reachability Checker for Linear Hybrid Automata”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Portland, Oregon, USA: IEEE Computer Society, 2008, pp. 1–4.

[Bur+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Logic in Computer Science (LICS)*. Washington, District of Columbia, USA: IEEE Computer Society, 1990, pp. 1–33.

[Bur+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Information and Computation* 98.2 (June 1992), pp. 142–170.

[Bur+93] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. *Symbolic Model Checking for Sequential Circuit Verification*. Tech. rep. CMU-CS-93-211. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, July 1993.

[Car+06] L.P. Carloni, R. Passerone, A. Pinto, and A.L. Sangiovanni-Vincentelli. "Languages and Tools for Hybrid Systems Design". In: *Foundations and Trends in Electronic Design Automation* 1.1/2 (2006), pp. 1–193.

[CG12] A. Cimatti and A. Griggio. "Software Model Checking via IC3". In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Vol. 7358. LNCS. Berkeley, California, USA: Springer, 2012, pp. 277–293.

[Cha+16] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. "The Kind2 Model Checker". In: *Computer Aided Verification (CAV)*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. LNCS. Toronto, ON, Canada: Springer, 2016, pp. 510–517.

[Cha14] A. Champion. "Collaboration of Formal Techniques for the Verification of Safety Properties over Transition Systems". PhD. PhD thesis. Toulouse, France: Institut Supérieur de l'Aeronautique et de l'Espace (ISAE), Université de Toulouse, Jan. 2014.

[Cho+11] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. "Incremental formal verification of hardware". In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by P. Bjesse and A. Slobodová. Austin, Texas, USA: IEEE Computer Society, 2011, pp. 135–143.

[Cim+13] A. Cimatti, A. Griggio, B. Joost Schaafsma, and R. Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by N. Piterman and S.A. Smolka. Vol. 7795. LNCS. Rome, Italy: Springer, 2013, pp. 93–107.

[Cim+14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. "IC3 Modulo Theories via Implicit Predicate Abstraction". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by E. Ábrahám and K. Havelund. Vol. 8413. LNCS. Grenoble, France: Springer, 2014, pp. 46–61.

[Cla+01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design (FMSD)* 19.1 (July 2001), pp. 7–34.

[Cla+03] E.M. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. "Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems". In: *International Journal of Foundations of Computer Science (IJFCS)* 14.4 (Aug. 2003), pp. 583–604.

[CMT11] A. Cimatti, S. Mover, and S. Tonetta. "HyDI: A Language for Symbolic Hybrid Systems with Discrete Interaction". In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE Computer Society, 2011, pp. 275–278.

[CW96] E.M. Clarke and J.M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[DBCB04] S. Dajani-Brown, D. Cofer, and A. Bouali. “Formal Verification of an Avionics Sensor Voter Using SCADE”. In: *Formal Modeling and Analysis of Timed Systems (FORMATS)*. Ed. by Y. Lakhnech and S. Yovine. Vol. 3253. LNCS. Grenoble, France: Springer, 2004, pp. 5–20.

[Det+14] M. Deters, A. Reynolds, T. King, C.W. Barrett, and C. Tinelli. “A tour of CVC4: How it works, and how to use it”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Lausanne, Switzerland: IEEE Computer Society, 2014, p. 7.

[DLSV12] P. Derler, E.A. Lee, and A. Sangiovanni-Vincentelli. “Modeling Cyber-Physical Systems”. In: *Proceedings of the IEEE* 100.1 (2012), pp. 13–28.

[DS96] A. Dolzmann and T. Sturm. *Redlog user manual*. Universität Passau. Fakultät für Mathematik und Informatik, 1996.

[EFH08] A. Eggers, M. Fränzle, and C. Herde. “SAT Modulo ODE: A Direct SAT Approach to Hybrid Systems”. In: *Automated Technology for Verification and Analysis (ATVA)*. Ed. by S.D. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan. Vol. 5311. LNCS. Seoul, South Korea: Springer, 2008, pp. 171–185.

[Elg76] C.C. Elgot. “Structured Programming With and Without ‘go to’ Statements”. In: *IEEE Transactions on Software Engineering (T-SE)* SE-2.1 (Mar. 1976), pp. 41–54.

[EMB11] N. Eén, A. Mishchenko, and R.K. Brayton. “Efficient implementation of property directed reachability”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by P. Bjesse and A. Slobodová. Austin, Texas, USA: IEEE Computer Society, 2011, pp. 125–134.

[FH05] M. Fränzle and C. Herde. “Efficient Proof Engines for Bounded Model Checking of Hybrid Systems”. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 133 (2005), pp. 119–137.

[FH07] M. Fränzle and C. Herde. “HySAT: An efficient proof engine for bounded model checking of hybrid systems”. In: *Formal Methods in System Design (FMSD)* 30 (2007), pp. 179–198.

[FK11] L. Fribourg and U. Kühne. “Parametric Verification and Test Coverage for Hybrid Automata Using the Inverse Method”. In: *Reachability Problems (RP)*. Ed. by G. Delzanno and I. Potapov. Vol. 6945. LNCS. Genoa, Italy: Springer, 2011, pp. 191–204.

[Flo67] R.W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science*. Ed. by J.T. Schwartz. Vol. 19. Symposia in Applied Mathematics. Providence, Rhode Island, USA: American Mathematical Society, 1967, pp. 19–32.

[Fre+11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. "SpaceEx: Scalable Verification of Hybrid Systems". In: *Computer Aided Verification (CAV)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Snowbird, Utah, USA: Springer, 2011, pp. 379–395.

[Fre05] G. Frehse. "Compositional Verification of Hybrid Systems using Simulation Relations". PhD. PhD thesis. Nijmegen, The Netherlands: Institute for Programming research and Algorithmics (IPA), Radboud University, 2005.

[Frä+07] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. "Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure". In: *Journal on Satisfiability, Boolean Modeling and Computation* 1.3-4 (May 2007), pp. 209–236.

[GAC12a] S. Gao, J. Avigad, and E.M. Clarke. "δ-Complete Decision Procedures for Satisfiability over the Reals". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by B. Gramlich, D. Miller, and U. Sattler. Vol. 7364. LNCS. Manchester, UK: Springer, 2012, pp. 286–300.

[GAC12b] S. Gao, J. Avigad, and E.M. Clarke. "Delta-Decidability over the Reals". In: *Logic in Computer Science (LICS)*. Dubrovnik, Croatia: IEEE Computer Society, 2012, pp. 305–314.

[Ges14] M. Gesell. "Interactive Verification of Synchronous Systems". PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, 2014.

[Gor86] M.J.C. Gordon. "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware". In: *Formal Aspects of VLSI Design*. Ed. by G.J. Milne and P.A. Subrahmanyam. Cambridge, England, UK: North-Holland, 1986, pp. 153–177.

[GR16] A. Griggio and M. Roveri. "Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 35.6 (June 2016), pp. 1026–1039.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[GS12] M. Gesell and K. Schneider. "A Hoare calculus for the verification of synchronous languages". In: *Programming Languages meets Program Verification (PLPV)*. Ed. by K. Claessen and N. Swamy. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 37–48.

[GS13a] M. Gesell and K. Schneider. "An Interactive Verification Tool for Synchronous/Reactive Systems". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by C. Haubelt and D. Timmermann. Warnemünde, Germany: University of Rostock, 2013, pp. 267–277.

[GS13b] M. Gesell and K. Schneider. “Translating synchronous guarded actions to interleaved guarded actions”. In: *Formal Methods and Models for Codesign (MEMOCODE)*. Portland, OR, USA: IEEE Computer Society, 2013, pp. 167–176.

[GSM07] R. Gentilini, K. Schneider, and B. Mishra. “Successive Abstractions of Hybrid Automata for Monotonic CTL Model Checking”. In: *Logical Foundations of Computer Science (LFCS)*. Ed. by S.N. Artemov and A. Nerode. Vol. 4514. LNCS. New York, New York, USA: Springer, 2007, pp. 224–240.

[Gup92] A. Gupta. “Formal Hardware Verification Methods: A Survey”. In: *Formal Methods in System Design (FMSD)* 1.2-3 (1992), pp. 151–238.

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The Synchronous Dataflow Programming Language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[HB12] K. Hoder and N. Bjørner. “Generalized Property Directed Reachability”. In: *Theory and Applications of Satisfiability Testing (SAT)*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. LNCS. Trento, Italy: Springer, 2012, pp. 157–171.

[HBS12] Z. Hassan, A.R. Bradley, and F. Somenzi. “Incremental, Inductive CTL Model Checking”. In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Vol. 7358. LNCS. Berkeley, California, USA: Springer, 2012, pp. 532–547.

[HBS13] Z. Hassan, A.R. Bradley, and F. Somenzi. “Better generalization in IC3”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by B. Jobstmann and S. Ray. Portland, Oregon, USA: IEEE Computer Society, 2013, pp. 157–164.

[Hea84] A.C. Hearn. “REDUCE User’s Manual: Version 3.1 The Rand Corporation”. In: *Santa Monica* (1984).

[Hen96] T.A. Henzinger. “The Theory of Hybrid Automata”. In: *Logic in Computer Science (LICS)*. New Brunswick, New Jersey, USA: IEEE Computer Society, 1996, pp. 278–292.

[Hoa69] C.A.R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM (CACM)* 12.10 (1969), pp. 576–580.

[KG99] C. Kern and M.R. Greenstreet. “Formal Verification in Hardware Design: A Survey”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 4.2 (1999), pp. 123–193.

[KM08] M. Kaufmann and J.S. Moore. “An ACL2 Tutorial”. In: *Theorem Proving in Higher Order Logics (TPHOL)*. Ed. by O. Ait Mohamed, C. Muñoz, and S. Tahar. Vol. 5170. LNCS. Montréal, Québec, Canada: Springer, 2008, pp. 17–21.

[KV99] O. Kupferman and M.Y. Vardi. “Model Checking of Safety Properties”. In: *Computer Aided Verification (CAV)*. Ed. by N. Halbwachs and D. Peled. Vol. 1633. LNCS. Trento, Italy: Springer, 1999, pp. 172–183.

[Lam80] L. Lamport. “The ‘ Hoare Logic’ of Concurrent Programs”. In: *Acta Informatica* 14 (1980), pp. 21–37.

[LBS13] X. Li, K. Bauer, and K. Schneider. “Interactive Verification of Cyber-physical Systems: Interfacing Averest and KeYmaera”. In: *International Workshop on Cyber-Physical Systems (IWCPS)*. Kraków, Poland: IEEE Computer Society, 2013, pp. 1447–1454.

[LNN15] T. Lange, M.R. NeuhäuSSer, and T. Noll. “IC3 Software Model Checking on Control Flow Automata”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by R. Kaivola and T. Wahl. Austin, Texas, USA: IEEE Computer Society, 2015, pp. 97–104.

[LS14] X. Li and K. Schneider. “Interactive Verification of Hybrid Systems”. In: *Automated Verification of Critical Systems (AVoCS)*. Ed. by M. Huisman and J. van de Pol. Vol. 70. Enschede, The Netherlands: EASST, 2014, pp. 265–266.

[LS15a] X. Li and K. Schneider. “A Counterexample-Guided Approach to Symbolic Simulation of Hybrid Systems”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by U. Heinkel and M. RöSSler. Chemnitz, Germany, 2015.

[LS15b] X. Li and K. Schneider. “An SMT-based Approach to analyze Non-Linear Relations of Parameters for Hybrid Systems”. In: *SyDe Summer School*. http://www.informatik.uni-bremen.de/syde/index.php?summerschool-2. Bremen, Germany, 2015.

[LS15c] X. Li and K. Schneider. “Verification Condition Generation for Hybrid Systems”. In: *Formal Methods and Models for Codesign (MEMOCODE)*. Ed. by A. Gerstlauer, C. Heitmeyer, and E. Leonard. Austin, Texas, USA: IEEE Computer Society, 2015, pp. 238–247.

[LS16a] X. Li and K. Schneider. “Control-flow Guided Clause Generation for Property Directed Reachability”. In: *High-Level Design Validation and Test Workshop (HLDVT)*. Ed. by M. Vechev. Santa Cruz, California, USA: IEEE Computer Society, 2016.

[LS16b] X. Li and K. Schneider. “Control-flow Guided Property Directed Reachability for Synchronous Programs”. In: *Formal Methods and Models for Codesign (MEMOCODE)*. Ed. by E. Leonard and K. Schneider. Kanpur, India: IEEE Computer Society, 2016.

[LT79] T. Lengauer and R.E. Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (July 1979), pp. 121–141.

[Mat+06] J. Matthews, J. Strother Moore, S. Ray, and D. Vroon. "Verification Condition Generation Via Theorem Proving". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by M. Hermann and A. Voronkov. Vol. 4246. LNCS. Phnom Penh, Cambodia: Springer, 2006, pp. 362–376.

[MB08] L. Mendonça de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by C.R. Ramakrishnan and J. Rehof. Vol. 4963. LNCS. Budapest, Hungary: Springer, 2008, pp. 337–340.

[McM03] K.L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification (CAV)*. Ed. by W.A. Hunt and F. Somenzi. Vol. 2725. LNCS. Boulder, Colorado, USA: Springer, 2003, pp. 1–13.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. "From Timed to Hybrid Systems". In: *Real-Time: Theory in Practice*. Ed. by J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg. Vol. 600. LNCS. Mook, The Netherlands: Springer, 1992, pp. 447–484.

[NM09] G. Nicolescua and P.J. Mosterman. *Model-based design for embedded systems*. CRC Press, 2009.

[OG76a] S.S. Owicki and D. Gries. "An axiomatic proof technique for parallel programs I". In: *Acta Informatica* 6.4 (1976), pp. 319–340.

[OG76b] S.S. Owicki and D. Gries. "Verifying properties of parallel programs: An axiomatic approach". In: *Communications of the ACM (CACM)* 19.5 (1976), pp. 279–284.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. "PVS: A Prototype Verification System". In: *Conference on Automated Deduction (CADE)*. Ed. by D. Kapur. Vol. 607. LNCS. Saratoga Springs, New York, USA: Springer, 1992, pp. 748–752.

[Pau12] L.C. Paulson. "MetiTarski: Past and Future". In: *Interactive Theorem Proving (ITP)*. Ed. by L. Beringer and A.P. Felty. Vol. 7406. LNCS. Princeton, New Jersey, USA: Springer, 2012, pp. 1–10.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Vol. 828. LNCS. Springer, 1994.

[PC08] A. Platzer and E.M. Clarke. "Computing Differential Invariants of Hybrid Systems as Fixpoints". In: *Computer Aided Verification (CAV)*. Ed. by A. Gupta and S. Malik. Vol. 5123. LNCS. Princeton, New Jersey, USA: Springer, 2008, pp. 176–189.

[PC09] A. Platzer and E.M. Clarke. "Computing differential invariants of hybrid systems as fixedpoints". In: *Formal Methods in System Design (FMSD)* 35.1 (Aug. 2009), pp. 98–120.

[PJ04] S. Prajna and A. Jadbabaie. “Safety Verification of Hybrid Systems Using Barrier Certificates”. In: *Hybrid Systems: Computation and Control (HSCC)*. Ed. by R. Alur and G.J. Pappas. Vol. 2993. LNCS. Philadelphia, Pennsylvania, USA: Springer, 2004, pp. 477–492.

[Pla08] A. Platzer. “Differential-algebraic Dynamic Logic for Differential-algebraic Programs”. In: *Journal of Logic and Computation* 20.1 (Nov. 2008), pp. 309–352.

[Pla10] A. Platzer. *Logical Analysis of Hybrid Systems – Proving Theorems for Complex Dynamics*. Springer, 2010.

[Plo81] G.D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. FN-19. Århus, Denmark: DAIMI, 1981.

[PQ08] A. Platzer and J.-D. Quesel. “KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description)”. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by A. Armando, P. Baumgartner, and G. Dowek. Vol. 5195. LNCS. Sydney, New South Wales, Australia: Springer, 2008, pp. 171–178.

[Pre30] M. Presburger. “über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt”. In: *Sprawozdanie z I Kongresu Matematykòw Krajòw Sowiaskich (Comptes-rendus du I Congrès des Mathèmaticiens des Pays Slaves)*. Ed. by F. Leja. Warsaw, Poland, 1930, pp. 92–101.

[RKL14] M. Reza Shoaei, L. Kovács, and B. Lennartson. “Supervisory Control of Discrete-Event Systems via IC3”. In: *Haifa Verification Conference (HVC)*. Ed. by E. Yahav. Vol. 8855. LNCS. Haifa, Israel: Springer, 2014, pp. 252–266.

[Rop+16] T. Ropertz, K. Berns, X. Li, and K. Schneider. “Verification of Behavior-Based Control Systems in their Physical Environment”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Freiburg, Germany, 2016, accepted for publication.

[SB11] F. Somenzi and A.R. Bradley. “IC3: where monolithic and incremental meet”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by P. Bjesse and A. Slobodová. Austin, Texas, USA: IEEE Computer Society, 2011, pp. 3–8.

[SB14] K. Scheibler and B. Becker. “Implication Graph Compression inside the SMT Solver iSAT3”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by J. Ruf, D. Allmendinger, and M. Michel. IBM Deutschland, Böblingen, Germany: Cuvillier, 2014, pp. 25–36.

[SBS06] K. Schneider, J. Brandt, and T. Schüle. “A Verified Compiler for Synchronous Programs with Local Declarations”. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 153.4 (2006), pp. 71–97.

[Sch00] K. Schneider. “A Verified Hardware Synthesis for Esterel”. In: *Distributed and Parallel Embedded Systems (DIPES)*. Ed. by F.J. Rammig. SchloSS Ehringerfeld, Germany: Kluwer, 2000, pp. 205–214.

[Sch01] K. Schneider. “Embedding Imperative Synchronous Languages in Interactive Theorem Provers”. In: *Application of Concurrency to System Design (ACSD)*. Newcastle Upon Tyne, England, UK: IEEE Computer Society, 2001, pp. 143–154.

[Sch02] K. Schneider. “Proving the Equivalence of Microstep and Macrostep Semantics”. In: *Theorem Proving in Higher Order Logics (TPHOL)*. Ed. by V. Carreño, C. Muñoz, and S. Tahar. Vol. 2410. LNCS. Hampton, Virginia, USA: Springer, 2002, pp. 314–331.

[Sch03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[Sch09] K. Schneider. *The Synchronous Programming Language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.

[SKB13] K. Scheibler, S. Kupferschmid, and B. Becker. “Recent Improvements in the SMT Solver iSAT”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by C. Haubelt and D. Timmermann. Warnemünde, Germany: University of Rostock, 2013, pp. 231–241.

[Sti88] C. Stirling. “A generalization of Owicki-Gries’s Hoare logic for a concurrent while language”. In: *Theoretical Computer Science (TCS)* 58.1-3 (June 1988), pp. 347–359.

[Tar36] A. Tarski. “Der Wahrheitsbegriff in formalisierten Sprachen”. In: *Studia Philosophica* 1 (1936).

[Tiw12] A. Tiwari. “HybridSAL Relational Abstracter”. In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Vol. 7358. LNCS. Berkeley, California, USA: Springer, 2012, pp. 725–731.

[VGS12] Y. Vizel, O. Grumberg, and S. Shoham. “Lazy Abstraction and SAT-based Reachability in Hardware Model Checking”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by G. Cabodi and S. Singh. Cambridge, UK: ACM and IEEE Computer Society, 2012, pp. 173–181.

[WK13] T. Welp and A. Kuehlmann. “QF BV model checking with property directed reachability”. In: *Design, Automation and Test in Europe (DATE)*. Ed. by E. Macii. Grenoble, France: EDA Consortium/ACM, 2013, pp. 791–796.

[WK14] T. Welp and A. Kuehlmann. “Property directed invariant refinement for program verification”. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. Dresden, Germany: IEEE Computer Society, 2014, pp. 1–6.

[ZLA06] H. Zheng, E.A. Lee, and A.D. Ames. “Beyond Zeno: Get on with It!” In: *Hybrid Systems: Computation and Control (HSCC)*. Ed. by J. Hespanha and A. Tiwari. Vol. 3927. LNCS. Santa Barbara, California, USA: Springer, 2006, pp. 568–582.

[Men+04] L. Mendonça de Moura, S. Owre, H. RueSS, J.M. Rushby, and N. Shankar. “The ICS Decision Procedures for Embedded Deduction”. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by D.A. Basin and M. Rusinowitch. Vol. 3097. LNCS. Cork, County Cork, Ireland: Springer, 2004, pp. 218–222.

[de +04] L.M. de Moura, S. Owre, H. RueSS, J.M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. “SAL 2”. In: *Computer Aided Verification (CAV)*. Ed. by R. Alur and D.A. Peled. Vol. 3114. LNCS. Boston, Massachusetts, USA: Springer, 2004, pp. 496–500.

Appendices

Appendix A

A Symbolic Simulation Algorithm

Here we describe the symbolic simulation algorithm that we introduced in [LS15a] and which we use to verify `ParametricBall` in Section 5.5.

Given a system's symbolic representation $\mathcal{G}$, specification C_b, and an initial state set I, the algorithm reuses the solution returned by Bonmin that violates the given specification. The recursive function *ValueRange* computes two subsets of the range constraint vector: Δ_f has range constraint vectors that lead to an over-approximation of the reachable states violating the given specification, while Δ_u has not yet found any parameter valuation that will go against the specification.

Function *Extend* in Line 5 generates a new range constraint vector $\vec{\Delta}_i$ by extending $\vec{v}_i$. For each parameter, the lower and upper bounds of $\vec{\Delta}_i$ are obtained by subtracting and adding $(j+1)$ times ϵ to the value according to $\vec{v}_i$, respectively. Function *Valuation* reorganizes the solution produced by Bonmin for this new range constraint vector. Whenever a parameter valuation vector $\vec{v}$ that violates the specification is found, we first exclude $\vec{v}_i$ from the initial state set, and then classify $\vec{\Delta}_i$ to Δ_f by function *Merge* iff $j = 0$ holds, as shown from Line 8 to Line 11. Otherwise, if function *Valuation* returns an empty set, then $\vec{\Delta}_i$ is classified to Δ_u by function *Merge* in Line 13. The algorithm executes the above steps for all the elements in the initial state set, before performing a recursive call. The recursive procedure terminates when either the initial state set becomes empty or the maximum recursion step N is reached.

Both Δ_f and Δ_u are obtained by extending each parameter valuation in the initial state set to a range constraint vector until some parameter valuation violates the property. The range constraint vector set Δ_f may include some range vectors that should belong to Δ_u, due to the introduced inaccuracy ϵ, while Δ_u provides the candidate ranges for parameters that meet the given specifications. Engineers could adapt the value of ϵ to get more accurate results. However, high accuracy is sometimes troublesome since it generates infeasible problems for Bonmin. For the moment, there is no general suggestion to avoid this problem.

Algorithm A.1 Computing Ranges for Input Parameters

Input:

- Initial State Set: $I = \{\vec{v}_i \mid i \in \mathbb{N}\}$
- Specification: C_b
- System's Symbolic Representation: $\mathcal{G}$
- Iteration Length: N

Output:

- Range Constraint Vector Set: Δ_u
- Range Constraint Vector Set: Δ_f

Local:

- Step Size: ϵ
- Parameter Vector: $\vec{p}_r$
- Parameter Valuation Vector: $\vec{v}$
- Iteration Counter : j

```
1: Δ_u ← ∅, Δ_f ← ∅, j ← 0
2: procedure ValueRange(I, C_b, G, N, Δ_u, Δ_f, j)
3:     if I ≠ {} then
4:         for all v_i ∈ I do
5:             Δ⃗_i ← Extend(v_i, ε, j)
6:             {v⃗} ← Valuation(Δ⃗_i, C_b, G, p⃗_r)
7:             if {v⃗} ≠ {} then
8:                 I ← I\v_i
9:                 if j = 0 then
10:                    Δ_f ← Merge(Δ_f, Δ⃗_i)
11:                end if
12:            else
13:                Δ_u ← Merge(Δ_u, Δ⃗_i)
14:            end if
15:        end for
16:        j ← j + 1
17:        if j < N then
18:            ValueRange(I, C_b, G, N, Δ_u, Δ_f, j)
19:        else
20:            return (Δ_u, Δ_f)
21:        end if
22:    else
23:        return (Δ_u, Δ_f)
24:    end if
25: end procedure
```

Appendix B

Curriculum Vitae

Persönliche Daten

Name	Xian Li
Geburtsdatum	April 1989
Geburtsort	Fujian, China
Staatsangehörigkeit	chinesisch

Bildungsweg

Sep 13 — Sep 17	**PhD: Computer Science** TU Kaiserslautern, Kaiserslautern, Germany
Oct 12 — Aug 13	**PhD: Qualification Phase** TU Kaiserslautern, Kaiserslautern, Germany
Sep 10 — Jul 12	**Master of Engineering:** **Traffic Information Engineering & Control** Beijing Jiaotong University, Beijing, China
Sep 06 — Jul 10	**Bachelor of Engineering:** **Communication Engineering (Science Experimental Class)** Beijing Jiaotong University, Beijing, China
Sep 03 — Jul 06	**Senior Secondary Education** Shishi Shiguang Highschool, Fujian, China