2018

Model-based compositional verification approaches and tools development for cyber-physical systems

Hao Ren
Iowa State University

Recommended Citation

Ren, Hao, "Model-based compositional verification approaches and tools development for cyber-physical systems" (2018). *Graduate Theses and Dissertations*. 16444.
https://lib.dr.iastate.edu/etd/16444

Model-based compositional verification approaches and tools development for cyber-physical systems

by

Hao Ren

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Electrical Engineering

Program of Study Committee:
Ratnesh Kumar, Major Professor
Gianfranco Ciardo
Manimaran Govindarasu
Joseph Zambreno
Samik Basu

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this dissertation to my wife Hui Dong and to my son Tianyi without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and financial assistance during the writing of this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation. First and foremost, Dr. Ratnesh Kumar for his guidance, patience, and support throughout this research and the writing of this dissertation. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Ciardo Gianfranco, Dr. Manimaran Govindarasu, Dr. Joseph Zambreno, and Dr. Samik Basu. I would additionally like to thank Matthew Clark of Air Force Research Lab for collaboration and the guidance of part of my work and Dr. Stanley Bak of Air Force Research Lab for the guidance of part of my work.

ABSTRACT

The model-based design for embedded real-time systems utilizes the verifiable reusable components and proper architectures, to deal with the scalability problem caused by state-explosion. In this dissertation, we address verification approaches for both low-level individual component correctness and high-level system correctness, which are equally important under this scheme. Three prototype tools are developed, implementing our approaches and algorithms accordingly.

For the component-level design-time verification, we developed a symbolic verifier, LhaVrf, for the reachability verification of concurrent linear hybrid systems (LHA). It is unique in translating a hybrid automaton into a transition system that preserves the discrete transition structure, possesses no continuous dynamics, and preserves reachability of discrete states. Afterwards, model-checking is interleaved in the counterexample fragment based specification relaxation framework. We next present a simulation-based bounded-horizon reachability analysis approach for the reachability verification of systems modeled by hybrid automata (HA) on a run-time basis. This framework applies a dynamic, on-the-fly, repartition-based error propagation control method with the mild requirement of Lipschitz continuity on the continuous dynamics. The novel features allow state-triggered discrete jumps and provide eventually constant over-approximation error bound for incremental stable dynamics. The above approaches are implemented in our prototype verifier called HS^{3}V. Once the component properties are established, the next thing is to establish the system-level properties through compositional verification. We present our work on the role and integration of quantifier elimination (QE) for property composition and verification. In our approach, we derive in a single step, the strongest system property from the given component properties for both time-independent and time-dependent scenarios. The system initial condition can also be composed, which, alongside the strongest system property, are used to verify a postulated system property through induction. The above approaches are implemented in our prototype tool called ReLIC.

CHAPTER 1. OVERVIEW

As embedded real-time systems become more and more complex and mission- or safety-critical, there is a recent trend to raise the level of abstraction to the model-level. Model-based design utilizes the verifiable reusable components and proper architectures, to deal with the verification scalability problem caused by state-explosion. Thus, ensuring low-level individual component correctness and high-level system correctness are equally important under this scheme.

For example, in a distributed Cyber-Physical System (CPS) as shown in Figure 1.1, control units are distributed, collecting sensor measurements driven by the underlying physical dynamics, commanding target actuators, while interacting/communicating through an embedded bus/network. To formally specify the entire system architecture, AADL (standardized by SAE) [1] can capture the architecture of software, computing/communication hardware/medium, and physical components, together with their behavior models, and other constraints.

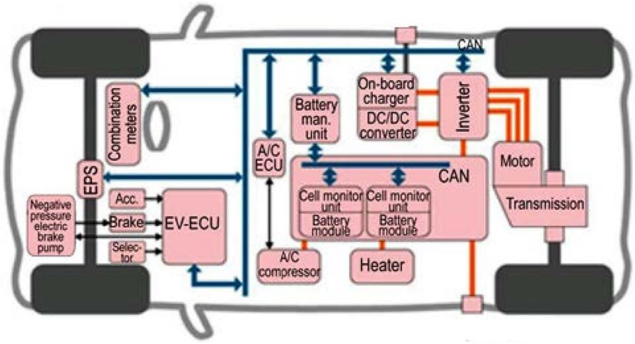


Figure 1.1: A distributed CPS

Verification techniques of such CPSs are manifold and varying across different types of components and the system according to the modeling and design/operation stages. In this dissertation, we present our work on (i) component-level design-time verification, (ii) component-level runtime verification, and (iii) system-level design-time compositional reasoning.

In a model-based system, examples of mathematical objects often used to model system components in design-time are: Extended/Timed Automata for discrete real-time behaviors in the software/computation/communication components and Stochastic Hybrid Automata for the physical components (those are subject to noise), etc. We have developed a symbolic verifier, **LhaVrf**, for the symbolic reachability verification of concurrent linear hybrid systems. A concurrent linear hybrid automaton is composed of a set of linear hybrid automata that interact through shared variables and/or events. A linear hybrid automaton is first translated to a purely discrete linear transition system that preserves the reachability of discrete states (locations). Its analysis can be conducted in the proposed counterexample fragment based specification relaxation (CEFSR) framework, where an invalid fragment (a subsequence) of a counterexample is used to eliminate the entire set of counterexamples sharing the same fragment, by way of specification relaxation (as opposed to the more traditional, model refinement). In the concurrent system setting, we propose further enhancement towards scalability as follows. For each spurious counterexample, an unsatisfiable core associated with it that makes the counterexample invalid, is identified and used for specification relaxation. This results in eliminating the entire set of spurious counterexamples sharing the same unsatisfiable core in a single iteration. Our implementation of **LhaVrf** adopts the above key ideas, with the capability of automatically translating the hybrid automata into discrete transition system, composing the concurrent model, and using SMT solver for validating counterexamples and fast-searching for the unsatisfiable core. The verification approach and the prototype verifier are illustrated via an application to the Fischer mutual exclusion protocol in Chapter 2.

In all safety-critical applications, we need to know on a run-time basis, whether the system under consideration will remain safe in a bounded future (corresponding to the reaction time required to take any corrective actions). One approach to bounded-time safety is the computation of the reach set over that horizon. Hybrid system verification tools based on reachability analysis incur over-approximation errors or have restrictions on the class of systems they can handle. In Chapter 3, we present a simulation-based bounded-horizon verification framework for general systems modeled by hybrid automata, with a mild requirement of Lipschitz continuity on the continuous dynamics. In this framework, the bounded initial set is covered by a finite set of representative states, whose forward simulations are used to generate an overapproximation of all the reachable states of the initial states. A novel feature of our approach is that the representative states are generated dynamically, on-the-fly, while the forward simulations are being performed. This is a key innovation introduced in the paper that refines the current "reachability-face" by a new partition only when needed. Our approach works for general class of hybrid systems with state-triggered discrete jumps and also allows piecewise constant bounded inputs, extending the existing work that applied to switched systems, that did not have state-triggered discrete jumps (switching was time-based and known a priori), and had no inputs. Additionally, when a dynamical system is incremental (input-to-state) stable, our algorithm that uses a simple Lipschitz-based discrepancy function provides an eventually constant error bound of over-approximation. This is indeed of practical significance since a Lipschitz-based discrepancy function is easily computable, while a more precise discrepancy function may not be available, but if it is, our algorithm can also utilize it. As a consequence of the constant error bound, owing to incremental (input-to-state) stability, the number of representative simulations also converges to a constant. We have developed a prototype verifier, HS3V, implementing our algorithms and providing verification results from several benchmarks to show its effective performance.

Last by not least, one central goal of model-based development is to enable analysis of the system. System-level properties are often expected to be derived from the compositional reasoning of the requirements on the components to be implemented and/or the verified properties of reusable elements. To achieve that, we look into taking advantage of state-of-art technologies on compositional verification, currently focusing on discrete-state systems (cyber part of CPS). Quantifier elimination (QE) is a powerful technique for gaining insight, through simplification, into problems involving logic expressions in various theories. We present our work in Chapter 4 on the role and integration of QE for compositional verification. In contrast to the current proof-based approach as for example in `AGREE` [2], in our QE-based approach, we derive in a single step, the strongest system contract from the given component contracts. This formalism is first developed for time-independent contracts, and later also extended to the case of time-dependent property composition. The extension to the time-dependent/temporal case requires additional work, namely, replicating the given properties by shifting those along time so the entire time-horizon of interest is captured. We show that the time-horizon (or order) of a system property is bounded by the sum of the time-horizons (or orders) of the component properties. The initial conditions of the components are also composed to obtain the system-level initial condition, which, alongside the inferred strongest system property, is used to verify a postulated system property through induction. We implemented our above approaches in a new prototype tool called ReLIC (Reduced Logic Inference for Composition) and demonstrated it through several examples. Additionally, in a k-induction [3] based model-checking approach, the verification problems of the base and inductive steps can be reduced to the QE problems in which all the variables are existentially quantified. Thus such a model-checker can be extended with QE tools to provide back-end solver options. Our implementation of integrating the QE tool Redlog [4] with the model-checker JKind showed the successful resolution of a nonlinear computation (emanating from a fuzzy logic operation) that the SMT-solver supported JKind [5] was unable to resolve. Thus the QE-integrated solver can provide an alternative option to a model-checker.

Chapter 5 gives the conclusion.

CHAPTER 2. VERIFICATION USING COUNTEREXAMPLE FRAGMENT BASED SPECIFICATION RELAXATION: CASE OF MODULAR/CONCURRENT LINEAR HYBRID AUTOMATA

2.1 Introduction

A typical cyber-physical system (CPS) exhibits complex behaviors caused by interleaving discrete transitions guided by discrete control actions or constraints, and continuous dynamics governed by underlining ordinary differential equations. *Hybrid automaton (HA)* [6, 7] has become a widely accepted choice for CPSs by integrating discrete transitions with continuous dynamics. An HA consists of multiple discrete *modes*. In each mode, a set of real-valued variables evolve according to specified flow dynamics and invariant, while the mode switches and value resets are guided by guards and jumps. An HA is called *Linear Hybrid Automaton (LHA)* when its flow dynamics, flow invariants, transition guards and jump conditions are all constrained by linear predicates.

In the domain of safety verification of hybrid systems, one seeks to prove or disprove a system's reachability to a set of unsafe states, given its initial states and dynamics. Some systems require safety at each moment, while for some other systems, the safe behavior is required to be maintained over a bounded time, but on a runtime basis. Normally, the reachability analysis requires the computation of the reachable set, and in the presence of continuous dynamics, this is typically undecidable. However, semi-decision algorithms can be developed under *counterexample-guided abstraction refinement (CEGAR)* [8] framework, by employing symbolic model checking over the finite-state abstractions, together with the iterative refinements to eliminate the spurious counterexamples. In case of the termination of a CEGAR algorithm, either the system is proven to be safe or a concrete counterexample is produced.

Parts of the work presented here was first reported in [9, 10], whereas we introduced an enhanced CEGAR-based algorithm for the verification of a class of hybrid systems modeled by LHAs.

Our work is unique in translating an LHA into a linear transition system (LTS) that preserves the discrete transition structure, possesses no continuous dynamics, and also preserves the reachability of discrete states (locations). Note checking the reachability of discrete locations suffices for verifying a class of safety specifications for which the model can be "refined against the specification" to reduce it to checking the unreachability of certain unsafe locations (locations, whose reachability implies the specification violation). The extension of the translation approach to general hybrid systems with inputs and outputs was later presented in [11]. The translation reduces the discrete state reachability problem of the original LHA to the one of the corresponding LTS. The safety property of the LTS is examined starting with the underlying discrete state automaton as the abstract model. There are several advantages of specification relaxation as opposed to the standard abstract model refinement: (i) Firstly, specification relaxation is more straightforward than model refinement, since all that needs to be done is to encode the counterexample to relax the specification, so in a next round it can no longer remain a counterexample; (ii) in fact in our approach we identify an *unsatisfiable core (unsat-core)* that is the *minimal conflicting constraint set* (see [10, 12]) over the constraints derived from a counterexample path. By using the encoding of the "unsat-core" to relax the specification, we are able to rule out an entire set of counterexamples possessing the same "unsat-core"; (iii) since there is no change in the underlying abstract model, the counterexamples of newer iterations can still be interpreted over the same underlying discrete graph structure, aiding the understanding of the counterexample. We refer to our approach *counterexample fragment based specification relaxation (CEFSR)*. We also developed a prototype tool `LhaVrf`, based on the approach described above and extending the initial development reported in [12]: The symbolic model checker `NuSMV` is integrated for model checking, whereas the SMT (satisfiability modulo theory) solver `Z3` [13] is integrated for counterexample validation and unsat-core identification. Our contributions that integrate past works and recent enhancements are summarized as follows:

- We provide a systematic method for the translation from LHA to a purely discrete abstraction LTS, that preserves the discrete behaviors (sequences of discrete states visited), and thereby also the safety and reachability properties.

- We have developed a framework for the safety verification of LHA, utilizing the LHA to LTS translation, called CEFSR (counterexample fragment based specification relaxation), in which a counterexample fragment is used for relaxing the safety specification as opposed to for refining the abstract model, and as explained in a next bullet multiple counterexamples are eliminated in a single iteration.
- To enhance scalability, an unsatisfiable core of a counterexample is identified, and encoded for specification relaxation so as to eliminate an entire set of spurious counterexamples that possess the same unsatisfiable core constraint set.
- We have developed an automated safety verification tool `LhaVrf` for concurrent LHA, based on the above techniques. The tool is integrated with `NuSMV` for model-checking, and with SMT solver `Z3` for counterexample validation and unsatisfiable core identification.
- The tool is validated with a system possessing 10 components, each with 4 locations, implying an overall locations space of size $4^{10} = \sim 1M$, and about $\sim 13M$ discrete transitions, validating the scalability of the approach and the tool.
- Finally, while the algorithm is presented in the domain of LHA, the tool's capability is extended to more general hybrid automaton allowing nonlinear guard/jump conditions (flows are still constrained by rectangular predicates).

This chapter extends the conference version [12] in multiple ways. First, a new section on related works from the literature is added, covering recent and historical development of algorithms and tools with respect to both (variable) state and location reachability analysis for hybrid systems. Secondly, we upgraded the tool `LhaVrf` by integrating the state-of-the-art SMT solver `Z3`. `Z3` is used for counterexample validation to replace the previously implemented Linear Programming solver via Microsoft Solver Foundation, allowing `LhaVrf` to be more scalable and extend its capability to handle nonlinear constraints for guards/jumps. `Z3` is further used for identifying the unsatisfiable core for a spurious counterexample using its built-in command "unsat-core", replacing the previous bisection based search algorithm, making the tool much more computationally efficient. Lastly, an experiment on a 10-process Fischer mutual exclusion protocol is added to show the scalability.

Rest of the chapter is organized as follows. Section 2.2 presents related works from the literature. Section 2.3 provides notations and preliminaries. Section 2.4 gives the implementation details of the `LhaVrf`. An illustrative example of its application to the Fischer mutual exclusion protocol is provided in Section 2.4.2.

2.2 Related Works

2.2.1 Symbolic Set-based Reachability Analysis

It is usually undecidable to compute the *reachable set*, except for certain subclass of LHAs that for example include timed automata, and initialized rectangular automata, etc. For clarification, the flow dynamics of LHA in this dissertation is *rectangular*, i.e., of the form $\dot{x} \in [a, b]$, where a, b are constants. Polyhedral flow representation is also used, e.g., first by the tool `HyTech` [14], for simple reason that polyhedral regions are invariant under linear discrete and continuous transitions.

For continuous dynamics described by linear differential equations of form $\dot{x} = ax + b$, where a, b are constant, the analytical solution for x is an exponential curve. Some literature [7, 15, 16, 17] employ the term *Linear Hybrid System (LHS)* to denote a hybrid automaton with such flow and linear invariants, guards and jumps. For LHS, although the reachable set, starting from a polyhedron, is also a polyhedron at any given moment, but the entire region of reachable states over time is not. Systems with nonlinear dynamics have more complicated continuous behaviors that often rule out analytical solutions. It is possible to use polyhedron representation for over-approximating the reachable set for both LTSs and nonlinear hybrid automata. In such scenarios, the over-approximation of overall reachable set grows by either expanding the current polyhedron zone, or adding new polyhedra for each time interval to construct flow pipes/tubes. Usually, better precision can be achieved by lowering the size of time intervals. In practice, the choice of time interval usually depends on the trade-off between demanded accuracy and computational cost.

The tool `d/dt` [18] tries to expand the polyhedron that contains the overall reachable set by lifting its faces outwards for a certain distance based on the vector field along the faces for each time interval [19]. However, the expansion may produce unboundedly large over-approximation. Instead of computing the over-approximation of the overall reachable set, the later tools focused on the reachable sets in the sequential time intervals. For each time interval $[t_k, t_{k+1}]$, CheckMate [20] first computes the convex hull of the polyhedron vertices at t_k and t_{k+1}, then bloats this convex hull along the directions of the normal vectors to its faces to compute the minimal convex set containing the reachable set in $[t_k, t_{k+1}]$. This reduces to a non-convex optimization problem solved numerically by MATLAB numerical package. C2E2 [21, 22], developed for verification of annotated hybrid systems and Stateflow models, blows a simulation trace to a reachtube (consecutive polyhedron over-approximation for sequential time intervals) to represent a group of executions within a close enough neighborhood loosely bounded by Lipschitz property of the continuous dynamics. If possible, annotations to the dynamics are used to derive so-called *discrepancy functions* that provide less conservative bounds on the error of the reachtube than Lipschitz-based ones. Recent version of C2E2 supports on-the-fly discrepancy function computation based on local optimization [23, 24].

Support function [25, 26] is another form for representing the reachable set, developed since 2009 for hybrid systems with piecewise affine continuous dynamics. Given the fact that any convex set is the intersection of a set of half-spaces, represented using normal vectors and distance values, a large class of sets such as unit ball, ellipsoid, zonotope, etc. can be represented by support functions in a compact manner. Further, more complex sets can be obtained using linear mapping, convex hull, Minkowski sum operations etc. on elementary convex sets. These operations on support functions are used to compute the post-image of a currently computed reachable set. Due to the expressive compactness and computational effectiveness of the support functions, SpaceEx [27, 28], that adopted this technique, has increased the scope of linear systems that can be verified to several hundreds of state variables. SpaceEx recently combines Zonotope representation for the input solution of a linear system with the support function representation of the affine solution [29], to balance the approximation error and scalability during flowpipe construction, as well as reduce complexity of handling time-triggered switching. For state-triggered discrete transitions, support functions are not efficient on intersection and deciding containment – operations often used in determining if a guard is triggered or an invariant/safety condition is violated. Thus, SpaceEx has to also mix in the polyhedral representation into the verification algorithm, making the extra over-approximating translations during the discrete transitions. Errors introduced during frequent discrete transitions may become a major bottleneck for precision in SpaceEx. Improvement has been done on eliminating spurious discrete transitions caused by conservative over-approximation involving template polyhedra, and checking the existence of a hyperplane that separates the guard set from the flowpipe [30]. A model translation technique implemented by Hyst [31, 32] is used to alleviate the same problem using a different approach. Hyst over-approximates the original frequently switching continuously-controlled system with a continuously-controlled system with an additional bounded non-deterministic input, resulting in the so-called *continuization*, which eliminates a large number of discrete transitions, thereby eliminating error growth caused by frequent translations between the aforementioned two types of representations occurring at the discrete transitions.

Taylor Model (TM) [33, 34], also developed since 2009, tightly encloses a flow of a differentiable function, expressed by its Taylor polynomial of degree up to k, bloated by an interval representing higher order reminder terms over a time interval. With proper interval-based techniques and semantically derived TM arithmetic operations, the tool Flow* [35] is able to construct TM flowpipes, i.e., the reachable sets of continuous nonlinear dynamics in sequential time intervals. As a result, TMs can provide guaranteed over-approximation to the solutions of ordinary differential equations (ODEs). For nonlinear dynamics, high accuracy demands higher order TMs, which in turn increases the number of parameters to be computed, often consuming significant resources. Instead of using an unified and fixed order for all state variables, one improvement is to adjust the TM order on-the-fly depending on the local varying rate for each state variable independently. Adaptive TM orders, along with adaptive time intervals were added as new features in the recent version of Flow* [36]. Because of the similar complications as SpaceEx for handling the intersections operation, Flow* also needs to translate between the TMs and the other effective representations back and forth. It is possible that Flow* could also benefit from using Hyst as a model preprocessing tool to alleviate this translational problem, especially for in case of frequently switching dynamics.

Despite the diversiform representation of reachable set, most state-of-the-art verification tools have been advancing by digging into the knowledge of nonlinear dynamics in Control Theory, searching for annotations and parameters that diminish the over-approximation error bound without exhausting computation resources. Readers can refer to [23, 24, 36, 37, 38] for further technical details. In contrast, a recently developed tool HS^{3}V [39] (see Chapter 3) for verification of general nonlinear hybrid automata proposed a different approach on over-approximation error propagation control. HS^{3}V is built upon simulation-based over-approximation framework. Its unique idea is to suppress the existing error periodically by dynamic repartitions and simulations on-the-fly. Because a repartition makes the existing error covered by multiple simulations (therefore more finely gridded) instead of the only one from the initial partition, even with very conservative error bound based on Lipschitz property which grows exponentially in between repartitions, the accumulated error "tightened up", or even gets suppressed to zero as time approaches infinity if the dynamics happens to be convergent. Regardless of how the error accumulates, this feature often results in a good error bound not only for complex nonlinear hybrid systems without any (usually hard-to-get) annotations from the users, but also for frequently switching dynamics.

2.2.2 Model Checking and Counterexample-Guided Abstraction Refinement

Another thread in hybrid system verification focuses on systems where the unsafe behavior is represented by a set of discrete unsafe locations. The major effort in safety verification is discretization-based over-approximation and the reliance on symbolic model checking [40] based on BDD and SAT [41]. In this context, CEGAR [8] has been proved to be successful. It involves iterations consisting of system abstraction, symbolic model checking, counterexample validation and abstraction-refinement. While there are some powerful tools for symbolic model checking, e.g., NuSMV and SPIN, which are commonly integrated in many CEGAR algorithms, these algorithms may vary on the abstraction-refinement schemes. In [16, 42], the abstraction of an LHA is a low-dimensional LHA constructed using a subset of the continuous variables from the original LHA. Model checking is done on its discrete transition graph. Once a spurious counterexample is identified by solving the constraints along the path of the original LHA using Linear Programing, a subset of the variables that preserves the infeasibility of the counterexample is selected then added to the set of variables used thus far to construct a new abstracted LHA. Abstraction gets refined in each iteration and excludes previously discovered spurious counterexamples. This incremental variable set abstraction allows users to diagnose and twist design parameters so as to eliminate undesirable behaviors. The tool HARE [43] abstracts a rectangular hybrid automaton via the following operations: collapsing the control states and transitions, dropping the continuous variables and scaling the variables. In the end, the abstract model is an *initialized* rectangular automaton and may have different discrete transition graph from the original model. HARE makes calls to HyTech to analyze abstract model and generate counterexamples. Since the reachability problem on the initialized rectangular automata is proven to be decidable, HyTech guarantees termination on the abstract model. Then upon the analysis to the spurious counterexample, HARE refines the abstraction by splitting control states/transitions, and/or adding variables that may have new dynamics (due to scaling). In general, HARE carries out a CEGAR framework with symbolic reachability analysis replacing symbolic model checking.

The complex behaviors of hybrid systems has attracted researchers with varying approaches, beyond the scope of methodologies and tools covered in the section above. These include phase-portrait adopted by PHAVer [44], theorem proving by KeYmaera [45], SMT-based techniques (e.g. BMC, k-induction) by HyComp [17] and Passel [46], to name a few. [15, 47, 48] further provide literature review on hybrid system verification technologies and tools.

2.3 Notation and Preliminary

2.3.1 Linear Hybrid Automaton (LHA)

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of real-valued variables and $\vec{v} = (v_1, v_2, \ldots, v_n)$ be their vector representation. A convex linear predicate over V is a finite boolean conjunction of linear inequalities over V. For a linear predicate ϕ and a valuation $\vec{a}$ over $\vec{v}$, we write $\phi[\vec{v} := \vec{a}]$ for the truth value obtained by evaluating ϕ with the constant a_i replacing in ϕ all occurrences of the variable v_i. Every linear predicate ϕ over $\vec{v}$ defines a set $[\phi] \subseteq \mathbb{R}^n$ of valuations such that $[\phi] = \{\vec{a} \in \mathbb{R}^n \mid \phi[\vec{v} := \vec{a}] = true\}$.

Definition 1. *A linear hybrid automaton is a tuple* $A = (L, V, \Sigma, E, Init, flow, inv, guard, jump)$ *consisting of the following components:*

- L is a finite set of locations,
- $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of real-valued variables. The state space of A is $L \times \mathbb{R}^n$. Each state thus has the form $(l, \vec{a})$, where $l \in L$ is the discrete part of the state and $\vec{a} \in \mathbb{R}^n$ is the continuous part,
- Σ is a finite set of events,
- $E \subseteq L \times \Sigma \times L$ is the set of discrete transitions,
- $Init \subseteq L \times \mathbb{R}^n$ is the set of initial states,
- $flow$ is the flow function that assigns each location $l \in L$ a convex linear predicate $flow(l)$ over $\dot{\vec{v}}$ that constrains the rates at which the values of variables change within the set $[flow(l)]$,
- inv is the invariant function that assigns each location $l \in L$ a convex linear predicate $inv(l)$ over $\vec{v}$ that constrains the values of variables within the set $[inv(l)]$,
- $guard$ is the guard function that assigns each transition $e = (l, \sigma, l') \in E$ a convex linear predicate $guard(e)$ over $\vec{v}$ such that e is enabled at state $(l, \vec{a})$ only if $\vec{a} \in [guard(e)]$,
- $jump$ is the jump function that assigns each transition $e = (l, \sigma, l') \in E$ a convex linear predicate $jump(e)$ over $\vec{v}$ and $\vec{v}'$ such that if e is taking place from state $(l, \vec{a})$ to state $(l', \vec{a}')$, then $(\vec{a}, \vec{a}') \in [jump(e)]$, i.e., $jump(e)[\vec{v} := \vec{a}, \vec{v}' := \vec{a}'] = true$.

We use subscript to denote the position of a specific element in a vector, and superscript to differentiate valuations of a variable at different occasions . A run in A is a (finite or infinite) sequence $r = (l^0, \vec{a}^0)(l^1, \vec{a}^1) \ldots (l^i, \vec{a}^i) \ldots$ such that there exists a sequence of events $\sigma^0 \sigma^1 \ldots \sigma^i \ldots$ satisfying following properties:

- $\forall i \geq 0$, $(l^i, \sigma^i, l^{i+1}) \in E$,
- There exists a sequence of non-negative real numbers $t^0 t^1 \ldots t^i \ldots$ and a sequence of functions $\dot{\vec{x}}^0 \dot{\vec{x}}^1 \ldots \dot{\vec{x}}^i \ldots$, where $\forall i \geq p$, $\dot{\vec{x}}^i : [0, t^i] \to \mathbb{R}^n$ such that

- $\forall i \geq 0$, $\forall t \in [0, t^i]$, $\dot{\vec{x}}^i(t) \in [flow(l^i)]$,
- $\forall i \geq 0$, $\forall t \in [0, t^i]$, $(\vec{a}^i + \int_0^{t^i} \dot{\vec{x}}^i(t)dt) \in [inv(l^i)]$,
- $\forall i \geq 0$, $(\vec{a}^i + \int_0^{t^i} \dot{\vec{x}}^i(t)dt) \in [guard((l^i, \sigma^i, l^{i+1}))]$,
- $\forall i \geq 0$, $jump((l^i, \sigma^i, l^{i+1}))[\vec{v} := \vec{a}^i + \int_0^{t^i} \dot{\vec{x}}^i(t)dt, \vec{v}' := \vec{a}^{i+1}] = true.$

Remark 1. *In this dissertation, the terminology linear hybrid automaton (LHA) is used to mean differently from a linear hybrid system (LHS): In LHA, the flow rates are linearly constrained. In contrast, by LHS we mean a system where the flow rate is governed by a linear ordinary differential equations (ODE). Such definitions are also adopted elsewhere, as [7, 15, 16, 17].*

Example 1. *The Fischer mutual exclusion protocol is used to guarantee mutual exclusion for shared resources in a concurrent system consisting of a number of processes. Each process_i is assumed to have a local clock modeled by the variable v_i. The global variable n is used to coordinate the access to the critical section. The LHA model Proc_i (to distinguish from the location A_i) for the* i^{th} *process is shown in Figure 2.1. There are two global positive real-valued parameters* D_1 *and* D_2 *in the model.* D_1 *represents the upper bound on the time that each process could take in changing the shared variable to its own number, and* D_2 *represents the lower bound for the time that each process must wait before it can check the variable value again.*

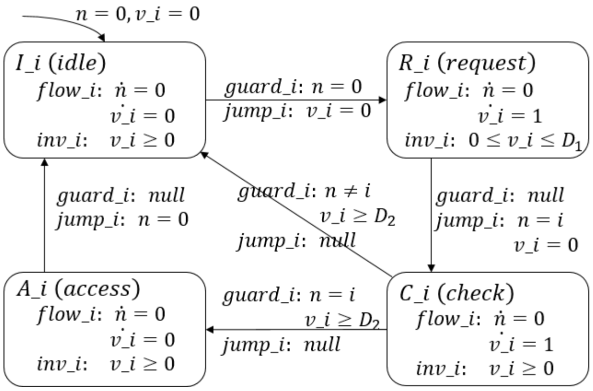


Figure 2.1: LHA model *Proc_i* for process_*i* in the Fischer protocol

$Proc\_i = (L\_i, V\_i, \Sigma\_i, E\_i, Init\_i, flow\_i, inv\_i, guard\_i, jump\_i)$ *is the LHA model for the* i^{th} *process, where*

- $L\_i = \{I\_i, R\_i, C\_i, A\_i\}$,
- $V\_i = \{v\_i, n\}$,
- $\Sigma\_i = \emptyset$,
- *E_i, $Init\_i$, $flow\_i$, inv_i, $guard\_i$ and $jump\_i$ are clear from Figure 2.1.*

2.3.2 Linear Transition System (LTS) and Translation of LHA into LTS

In this section we introduce the notion of a linear transition system (LTS) that is a purely discrete-state system, and present a translation from an LHA to an LTS to obtain a discrete-state abstraction that preserves the set of discrete behaviors (sequences of discrete states visited), including the reachability of discrete states (see Theorem 1). Note the LTS model does not possess any continuous dynamics. This is a key contribution of our approach. Under this equivalence, a reachability check over LHA can be reduced to an equivalent reachability/safety check over the LTS, while a counterexample validation is reduced to a standard linear programming (since LHAs are only endowed with linear predicates).

Definition 2. *A linear transition system is a tuple $S = (L, V, \Sigma, E, Init, P)$, where L, V, Σ, E and $Init$ are the same as in Definition 1, and P is the transition relation function that assigns each transition in E a convex linear predicate over $\{v_1, v_2, \ldots, v_n, v'_1, v'_2, \ldots, v'_n\}$, i.e., $\forall e = (l, \sigma, l') \in E, P : e \mapsto p_e(\vec{v}, \vec{v}')$, which relates the variable values at locations l and l'.*

A run in S is a (finite or infinite) sequence $r = (l^0, \vec{a}^0)(l^1, \vec{a}^1)\ldots(l^i, \vec{a}^i)\ldots$ such that there exists a sequence of $\sigma^0\sigma^1\ldots\sigma^i\ldots$, $\forall i \geq 0$, $(l^i, \sigma^i, l^{i+1}) \in E$ and $p_{(l^i,\sigma^i,l^{i+1})}[\vec{v} := \vec{a}^i, \vec{v}' := \vec{a}^{i+1}] = true$. In the following we discuss the translation from an LHA to its discrete-state abstracted LTS, in which continuous state vector $\vec{v}_-$ (respectively, $\vec{v}_+$) represents the values just before (respectively, after) the LTS executes a transition and enters the current location l, and t represents the time that the system spends at the previous location. The primed variables $\vec{v}'_-, \vec{v}'_+, t'$ correspond to the variables v_-, v_+, t at the next stage when a next transition is taken (see Figure 2.2). It then follows that a flow constraint $\dot{\vec{v}} \in [a, b]$ in an LHA is mapped to a linear constraint of the type,

$(\vec{v}'_- - \vec{v}_+)/t' \in [a,b] \wedge t' \geq 0$ in the translated LTS. The invariant, guard, and jump conditions are also translated accordingly as described below.

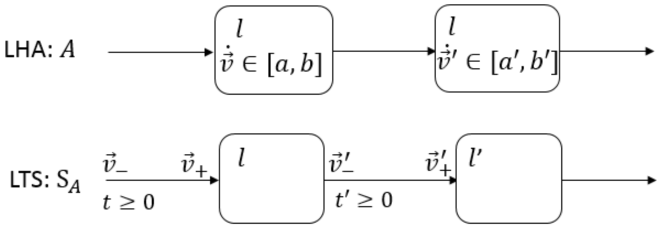


Figure 2.2: Translation from an LHA A (partial) to its discrete-state abstracted LTS S_A.

Given an LHA $A = (L, V, \Sigma, E, Init, flow, inv, guard, jump)$, we can construct the following LTS $S_A = (L, W, \Sigma, E, Init_S, P)$, where

- $W = V_- \cup V_+ \cup \{t\}$, with $V_- = \{v_{1-}, \ldots, v_{n-}\}$ and $V_+ = \{v_{1+}, \ldots, v_{n+}\}$,
- let $\vec{w} = (v_{1-}, \ldots, v_{n-}, v_{1+}, \ldots, v_{n+}, t)$ and $\vec{\beta}$ be a valuation of $\vec{w}$, then $(l^0, \vec{\beta}^0) \in Init_S$ with $\vec{\beta}^0 = (0, \ldots, 0, 0, a_1^0, \ldots, a_n^0)$,
- $\forall e = (l, \sigma, l') \in E$, $p(e) = flow(l)|_{\dot{\vec{v}} \leftarrow (\vec{v}'_- - \vec{v}_+)/t'} \wedge inv(l)|_{\vec{v} \leftarrow \vec{v}'_-} \wedge guard(e)|_{\vec{v} \leftarrow \vec{v}'_-}$
  $\wedge\, jump(e)|_{\vec{v} \leftarrow \vec{v}'_-, \vec{v}' \leftarrow \vec{v}'_+} \wedge inv(l')|_{\vec{v}' \leftarrow \vec{v}'_+} \wedge (t' \geq 0)$.

In order to handle the case of $t = 0$, every inequality in $flow(l)|_{\dot{\vec{v}} \leftarrow (\vec{v}'_- - \vec{v}_+)/t'}$ is modified by multiplying t' at both sides.

Example 2. *The LTS model* $S_{Proc\_i} = (L\_i, W\_i, \Sigma\_i, E\_i, Init_S\_i, P\_i)$ *derived from the LHA model* $Proc\_i$ *for* $process\_i$ *in the Fischer protocol is shown in Figure 2.3, where*

- $L\_i = \{I\_i, R\_i, C\_i, A\_i\}$,
- $W\_i = \{v\_i_-, v\_i_+, n_-, n_+, t\_i\}$,
- $\Sigma\_i = \emptyset$,
- $Init_S\_i = \{(I\_i, (0, 0, 0, 0, 0))\}$,
- *E_i is clear from the context of Figure 2.3,*

- *P_i defines the transition relation function as follows:*

$$\begin{aligned}
P\_i((I\_i, R\_i)) &= (n'_- = 0) \wedge (v\_i'_+ = 0) \wedge (0 \le v\_i'_+ \le D_1) \wedge (t\_i' \ge 0),\\
P\_i((R\_i, C\_i)) &= (v\_i'_- - v\_i_+ = t\_i') \wedge (0 \le v\_i'_- \le D_1) \wedge (n'_+ = i)\\
&\quad \wedge (v\_i'_+ = 0) \wedge (v\_i_+ \ge 0) \wedge (t\_i' \ge 0),\\
P\_i((C\_i, A\_i)) &= (v\_i'_- - v\_i_+ = t\_i') \wedge (0 \le v\_i'_-) \wedge (n'_- = i) \wedge (v\_i'_- \ge D_2)\\
&\quad \wedge (t\_i' \ge 0),\\
P\_i((A\_i, I\_i)) &= (n'_- = 0) \wedge (t\_i' \ge 0),\\
P\_i((C\_i, I\_i)) &= (v\_i'_- - v\_i_+ = t\_i') \wedge (0 \le v\_i'_-) \wedge (n'_+ \ne i) \wedge (v\_i'_- \ge D_2)\\
&\quad \wedge (t\_i' \ge 0).
\end{aligned}$$

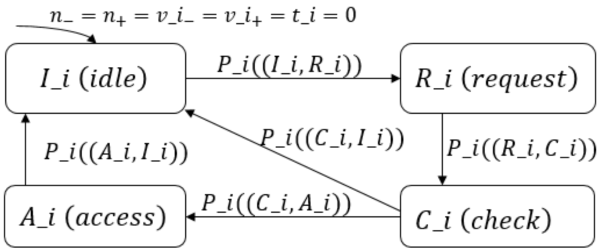


Figure 2.3: LTS model for process$_i$ in the Fischer protocol

For both A and S_A, the discrete transition trace of r is $(l^0, \sigma^0, l^1)(l^1, \sigma^1, l^2) \ldots (l^i, \sigma^i, l^{i+1}) \ldots$. Location l' is reachable in A (or S_A) if there exists a run in A (or S_A) with a transition trace $(l^0, \sigma^0, l^1)(l^1, \sigma^1, l^2) \ldots (l^i, \sigma^i, l')$. The set of transition traces associated with all finite-length runs in A (or S_A) is called the language of A (or S_A) and is denoted by $\mathcal{L}(A)$ (or $\mathcal{L}(S_A)$). It is obvious that $\mathcal{L}(A) \subseteq E^*$ (or $\mathcal{L}(S_A) \subseteq E^*$) and $\mathcal{L}(A)$ (or $\mathcal{L}(S_A)$) is prefix closed. There exists a precise equivalence between an LHA and its discrete-state abstracted LTS. (see Theorem 1 in [10] and its proof):

Theorem 1. *Given an LHA A as in Definition 1, an LTS S_A as in Definition 2 constructed using above procedures satisfies $\mathcal{L}(S_A) = \mathcal{L}(A)$.*

Theorem 1 shows that A and S_A have the same behaviors over the discrete locations, i.e., $\mathcal{L}(A) = \mathcal{L}(S_A)$. Therefore, the reachability problem of A can be reduced to the reachability problem of S_A.

Remark 2. *Note one can also encode the reachability of some safe set of continuous states in the form of the reachability of a certain discrete location, by first refining the original model with respect to the given safe set of continuous states. Suppose such a safe region is encoded as ϕ . Then we can introduce a new discrete state called "unsafe", and split each transition edge e with guard $guard(e)$ into a pair of transitions, one guarded by $guard(e) \wedge \phi$ that reaches the original successor of e, and another guarded by $guard(e) \wedge \neg\phi$ that reaches the newly added discrete location "unsafe".*

2.3.3 Safety Verification Algorithm of LHA

Given the notations introduced above, we have the following safety verification algorithm for an LHA under the proposed CEFSR framework.

Algorithm 1: Safety verification of LHA.

```
   Input: LHA $A$ and its unsafe locations.
1  $S_A \leftarrow$ Translate($A$);
2  $G_A \leftarrow$ Abstract($S_A$);
3  $spec_{safe} \leftarrow \neg Unsafe$;
4  repeat
5      $ce \leftarrow$ModelCheck($G_A, \mathbf{G}(spec_{safe})$);
6      if $ce = null$ then
7          Terminate("Safe.");
8      else if Validate($S_A, ce$) $= \top$ then
9          Terminate("Unsafe.", $ce$);
10     else
11         $unsat\text{-}core \leftarrow$ UnsatCore($S_A, ce$) ;
12         $spec_{safe} \leftarrow spec_{safe} \vee$ Encode($unsat\text{-}core$) ;
13     end
14 until running time limit is reached;
15 Terminate("Time out.");
```

In line 1, the function `Translate`(A) returns the discrete-behaviors equivalent LTS S_A constructed from LHA A following the procedures described in Section 2.3.2. In line 2, the function `Translate`(S_A) returns the discrete transition graph of S_A as its abstraction model. Line 3 initializes the safety specification with the atomic predicate "$\neg Unsafe$", where "$Unsafe$" holds at the given unsafe locations of A. Lines 4-14 form the CEFSR loop. In line 5, $\mathbf{G}(spec_{safe})$ is the temporal logic formula denoting the global safety specification, where “**G**” denotes the “globally” operator. The function `ModelCheck`$(G_A, \mathbf{G}(spec_{safe}))$ performs BDD-based model checking on G_A against $\mathbf{G}(spec_{safe})$. In case of satisfaction, `ModelCheck`$(G_A, \mathbf{G}(spec_{safe}))$ returns *null*, and in line 7 the function `Terminate`(“Safe.”) terminates the algorithm and prints “Safe.”. Otherwise, it returns a counterexample $ce = e^0 \dots e^n$ in the form of a sequence of consecutive edges of S_A, where only e^n is the edge leading to an unsafe location. ce is *concrete* in S_A if the constraint set along its path is satisfiable. Note that the constraints of ce also include the initial constraint P_{Init}. The validation of ce is reduced to the SMT problem of checking $P_{Init} \wedge \big(\bigwedge_{i=0}^{n} P(e^i) \big)$; this is performed by the function `Validate`(S_A, ce). If `Validate`(S_A, ce) returns true, the function `Terminate`(“Unsafe.”, ce) terminates the algorithm and prints “Unsafe.”, together with the concrete counterexample ce. Otherwise ce is called *spurious* if $P_{Init} \wedge \big(\bigwedge_{i=0}^{n} P(e^i) \big)$ is unsatisfiable. The function `UnsatCore`(S_A, ce) in line 11 computes an *unsatisfiable core (unsat-core)*, which is a minimal subset of constraints whose conjunction is still unsatisfiable. By mapping the unsat-core to the edges of S_A, the function `Encode`$(unsat\text{-}core)$ returns the *minimal invalid fragment (MIF)* of ce, denoted by f_{min}. Note not all constraints of an edge need to appear in the unsat-core for that edge to be included in MIF. Also note due to the minimality requirement, the edges that appear in MIF must be consecutive since any non-consecutive edges have no variables in common hence cannot conflict with each other.

We relax the specification to rule out *all* spurious counterexamples that share an unsat-core by encoding the unsat-core and disjuncting it with the current specification. Then in the next iteration, the relaxed specification accepts all counterexamples that contain the minimal invalid fragment before reaching the unsafe location. Let $f_{min} = e^j e^{j+1} \dots e^k$ with $0 \leq j \leq k \leq n$. There are two cases to consider. The first case is when P_{Init} is included in the unsat-core (implying $j = 0$, not vise versa). The consecutiveness property of the unsat-core implies that the MIF starts at the initial location. So the specification must be relaxed so as to accept any path for which MIF appears in the very beginning. On the other hand, if P_{Init} is not in the unsat-core, then the specification should be relaxed to accept the MIF to appear anywhere in a path, but prior to reaching an unsafe location. Accordingly, the temporal logic encoding of the unsat-core is given by:

$$\texttt{Encode}(\mathit{unsat\text{-}core}) = \begin{cases} \bigwedge_{i=0}^{k} \left(\mathbf{X}^i(l^i\mathbf{X}l^{i+1})\right) & P_{Init} \in \mathit{unsat\text{-}core} \\ (\neg \mathit{Unsafe})\mathbf{U}\Big(\bigwedge_{i=j}^{k} \left(\mathbf{X}^{i-j}(l^i\mathbf{X}l^{i+1})\right)\Big) & \mathit{otherwise,} \end{cases}$$

where "**U**" denotes the "until" operator, "**X**" denotes the "next" operator and $\mathbf{X}^i$ is the composition of i number of "**X**". We now establish the correctness of Algorithm 1 as follows.

Theorem 2. *Algorithm 1 is correct.*

Proof. Since G_A is the abstract model of S_A, $\mathcal{L}(S_A) \subseteq \mathcal{L}(G_A)$. So if Algorithm 1 terminates with "Safe.", then it implies that the set of bad locations is not reachable in G_A, thereby also not reachable in S_A or A. Otherwise if Algorithm 1 terminates with "Safe.", then because of the counterexample validation we know that the concrete counterexample ce is a valid run in S_A, which in turn implies that ce is a valid run in A following Theorem 1. □

2.3.4 Concurrent LHA

Definition 3. *Let* $A\_i = (L\_i, V\_i, \Sigma\_i, E\_i, Init\_i, flow\_i, inv\_i, guard\_i, jump\_i)$, $i = 1, \ldots, k$, *the concurrent LHA, that is synchronously composed of* $A\_1, \ldots, A\_k$, *is given by* $A = \|_{i=1}^{k} A\_i := (L, V, \Sigma, E, Init, flow, inv, guard, jump)$, *where*

- $L := \times_{i=1}^{k} L\_i$,
- $V := \cup_{i=1}^{k} V\_i$,
- $\Sigma := \times_{i=1}^{k} \overline{\Sigma}\_i$, *where* $\overline{\Sigma}\_i = \Sigma\_i \cup \{\epsilon\}$,
- $E := \times_{i=1}^{k} \overline{E}\_i$, *where* $\overline{E}\_i = E\_i \cup \{(l\_i, \epsilon, l\_i) \mid l\_i \in L\_i\}$,
- $((l\_1^0, \ldots, l\_k^0), \vec{a}^0) \in Init$, *iff* $(l\_i^0, \vec{a}\_i^0) \in Init\_i$, *where* $\vec{a}\_i^0$ *is* $\vec{a}^0$*'s projection on* V_i,
- $\forall l = (l\_1, \ldots, l\_k) \in L, inv(l) := \wedge_{i=1}^{k} inv\_i(l\_i)$,
- $\forall l = (l\_1, \ldots, l\_k) \in L$, *let* $I(l, v_j) = \{i \mid flow\_i(l\_i)(v_j) \neq null\}$ *be is the index set of component LHA that has flow constraint on* v_j *at* l, *define* $flow(l) := \wedge_{v_j \in V} f(l, v_j)$, *where*

  $$f(l, v_j) = \begin{cases} \wedge_{i \in I(l,v_j)} flow(l\_i)(v_j) & I(l, v_j) \neq \emptyset \\ (\dot{v}_j = 0) & I(l, v_j) = \emptyset \end{cases},$$

- $\forall e \in E$, *$guard(e) := \wedge_{i=1}^{k} guard\_i(e\_i) \wedge syn(e)$, where $syn : E \to \{\top, \bot\}$ is used to capture user-specified enabling constraints. When $guard(e) = \bot$, e is invalid,*
- $\forall e \in E$, *let $J(e, v_j) = \{i | jump\_i(e\_i)(v_j) \neq null\}$, where $J(l, v_j)$ is the index set of component LHA that has jump constraint on v_j at l, define $jump(e) := \wedge_{v_j \in V} j(e, v_j)$, where*
  $$j(e, v_j) = \begin{cases} \wedge_{i \in J(e,v_j)} jump(e\_i)(v_j) & J(e, v_j) \neq \emptyset \\ (v_j' = v_j) & J(e, v_j) = \emptyset \end{cases}.$$

The definition of the concurrent LHA above is a more compact version of the one in [9]. Note in the context of a concurrent LHA, the edges within a MIF are the edges of the composed LHA, and the unsat-core needs to be further mapped to the edges of underlying component LHAs. Each concurrent edge e^i in a fragment can be decomposed into a set of edges $(e\_1^i, \ldots, e\_k^i)$, where e_j^i denotes an edge from component LHA A_j at the i^{th} step. Then an unsat-core can further map to a set of sequence of edges of the component LHAs. Accordingly the encoding of unsat-core for a concurrent LHA, that maps down to minimal invalid fragments of individual component LHAs is given by:

$$\mathtt{Encode}(\textit{unsat-core}) = \begin{cases} \bigwedge_{i=0}^{k} \left(\mathbf{X}^i \bigwedge_{k \in K^i} (l\_k^i \mathbf{X} l\_k^{i+1})\right) & P_{Init} \in \textit{unsat-core} \\ (\neg \textit{Unsafe})\mathbf{U}\Big(\bigwedge_{i=j}^{k} \left(\mathbf{X}^{i-j} \bigwedge_{k \in K^i} (l\_k^i \mathbf{X} l\_k^{i+1})\right)\Big) & \textit{otherwise,} \end{cases}$$

where K^i is the index set of component LHAs that participate in the i^{th} step of the counterexample and contribute to the unsat-core. Note that this encoding of unsat-core matches to the encoding in case of a single LHA presented in the previous section. The following algorithm extends Algorithm 1 from the case of a single LHA to that of a concurrent LHA:

Algorithm 2: Safety verification of concurrent LHA

```
Input: Components LHA $A\_i, i = 1, \ldots, k$ and the unsafe concurrent locations.
$A \leftarrow \|_{i=1}^{k} A\_i$ ;                          // Construct the concurrent LHA $A$ from $A_i$s
Algorithm1($A$, unsafe concurrent locations) ;               // Call Algorithm 1
```

2.4 Implementation of `LhaVrf`

2.4.1 Architecture

The tool `LhaVrf` is implemented in the programming language F#. The architecture consists of six modules, and the data flow among them occurs along the arrows, as shown in Figure 2.4. Each of the modules is introduced separately in the following subsections.

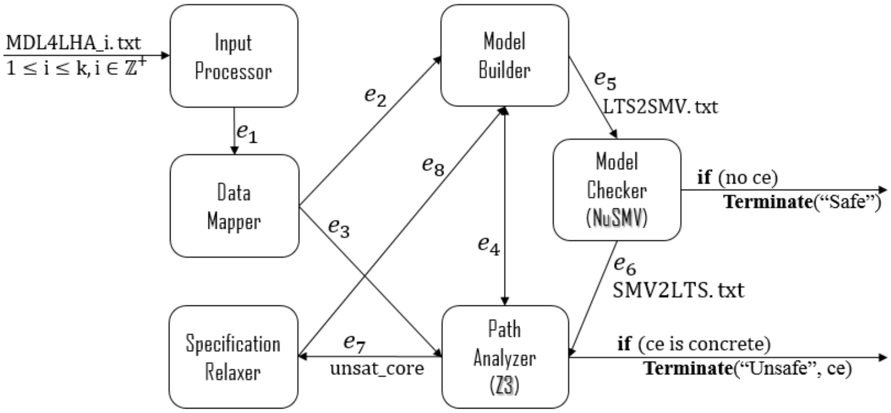


Figure 2.4: Architecture of `LhaVrf`

2.4.1.1 Input Processor

The Input Processor accepts a series of txt files as its inputs. Each txt file MDL4LHA_*i*.txt describes a component LHA *A_i* with the following syntax:

⟨*model*⟩ ::= ⟨*transition*⟩* ⟨*constraint*⟩* ⟨*var_set*⟩ ⟨*init_loc*⟩⟨*init_state*⟩* ⟨*unsafe_loc*⟩

⟨*transition*⟩ ::= '`$`' ⟨*location*⟩ ⟨*edge*⟩ ⟨*location*⟩

⟨*constraint*⟩ ::= ⟨*location*⟩ '`:`' ⟨*flow*⟩

| ⟨*location*⟩ '`:`' ⟨*invariant*⟩

| ⟨*edge*⟩ '`:`' ⟨*guard*⟩

| ⟨*edge*⟩ '`:`' ⟨*jump*⟩

⟨*var_set*⟩ ::= '`@`' ⟨*variable*⟩*

⟨*init_loc*⟩ ::= '`!`' ⟨*location*⟩

⟨*init_state*⟩ ::= '`!!`' ⟨*variable*⟩ ⟨*number*⟩

⟨*unsafe_loc*⟩ ::= '`#`' ⟨*location*⟩*

The Input Processor parses the syntactical lines in each input file and passes the data containing the description information of the LHA to the Data Mapper (e_1 in Figure 2.4).

2.4.1.2 Data Mapper

In the Data Mapper, each location or transition is assigned a unique id composed by its subsystem index and its serial number in the subsystem. Each atomic predicate is mapped to its source location or transition id. The unsafe locations are also stored here coupled with there own ids. These mapping data is sent to Model builder (e_2 in Figure 2.4) as well as Path Analyzer (e_3 in Figure 2.4) for later use.

2.4.1.3 Model Builder

The Model Builder collects all the data from the Data Mapper and computes the LTS model of the concurrent LHA: For every possible concurrent edge, the module collects the relevant predicates in the concurrent LHA model and converts them into the edge-predicate for the LTS model. Then, the Model Builder calls the Path Analyzer (e_4 in Figure 2.4) for validating the edge, and only when the edge is valid, it adds the edge to the LTS model.

The Model Builder also collects the unsafe locations and encodes them into the initial LTL specification $\mathbf{G}(\neg \mathit{Unsafe})$. Then it passes the abstract model and the LTL specification in LTS2SMV.txt to the Model Checker `NuSMV` (e_5 in Figure 2.4).

2.4.1.4 Model Checker (`NuSMV`)

`NuSMV` is a well known symbolic model checker. Once called by the Model Builder, it checks the abstract model in the LTS2SMV.txt file against its specification. If it is satisfied, the entire program terminates with the output stating that the system is safe. Otherwise, a counterexample is generated and written into SMV2LTS.txt and passed on to the Path Analyzer (e_6 in Figure 2.4).

2.4.1.5 Path Analyzer (SMT solver Z3)

The Path Analyzer accepts edge guards from the Model Builder and returns whether or not those are satisfiable. It also accepts a counterexample returned from NuSMV, and gathers the predicates of the counterexample edges from the Data Mapper. The SMT solver Z3 inside the module accepts the predicate constraints, solves for satisfiability, and returns "*sat*" along with the valid assignments to the variables or "*unsat*" along with an *unsat-core* (a set of constraint id). If the counterexample from SMV2LTS.txt is found valid, the entire program terminates with the output stating that the system is unsafe and reports the concrete counterexample and the valid assignments. Otherwise the unsat-core is sent to the Specification Relaxer (e_7 in Figure 2.4).

2.4.1.6 Specification Relaxer

Given an unsat-core received from the Path Analyzer, the Specification Relaxer relaxes the current specification formula by disjuncting it with the encoding of the unsat-core, using the mapping information created in the Data Mapper. The relaxed formula is then sent to the Model Builder (edge 8 in Figure 2.4) to start a new round of iteration. Thereby the edges $e_5 \rightarrow e_6 \rightarrow e_7 \rightarrow e_8$ form a verification loop.

2.4.2 An Illustrative Example

In the Fischer protocol case, assume there are two processes, and that at most one process can make a location transition at any given time under interleaving semantics [9, 49]. The LhaVrf first reads the model files then automatically translates each input file into its LHA model $Proc\_i$ and stores it in the Data Mapper. Next, all the LHA $Proc\_i$ are composed to form the concurrent LHA $Proc$, and converted to the LTS S_{Proc}, from which the abstract model as shown in Figure 2.5 (with unsafe location in shadow) is extracted.

For the specification, $Proc\_1$ and $Proc\_2$ are not allowed to be in the access state at the same time. More precisely, we have the following LTL specification: $\mathbf{G}(\neg \mathit{Unsafe})$, where $\mathit{Unsafe} = A\_1 \wedge A\_2$, representing that both $Proc\_1$ and $Proc\_2$ are in the access state at the same time.

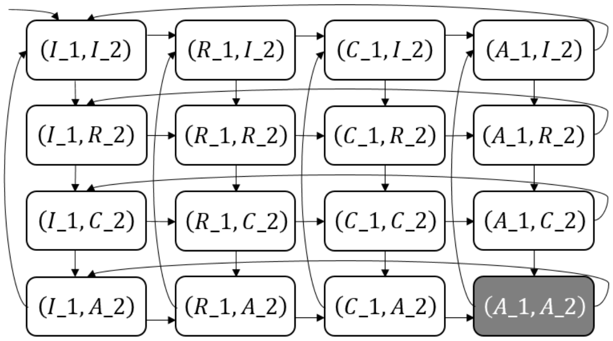


Figure 2.5: The abstract model for the two-process Fischer protocol

Note in the Fischer protocol case, there exists at most one edge between any pair of locations, and so any edge can be identified by a unique pair of locations. Therefore, we can express each counterexample/fragment as a sequence of locations for simplicity. When the parameters $D_1 = 4, D_2 = 3$, `NuSMV` generates a counterexample $ce = (I\_1, I\_2)(R\_1, I\_2)(R\_1, R\_2)$ $(R\_1, C\_2)(R\_1, A\_2)(C\_1, A\_2)(A\_1, A\_2)$. The set of predicate constraints, that are mapped to the locations and edges along the counterexample path, is sent to `Z3`. In this case, `Z3` returns "*sat*" and a valid solution, which means when $D_1 = 4, D_2 = 3$, the mutual exclusion is not guaranteed. On the other hand, when $D_1 = 2, D_2 = 3$, `NuSMV` generates a counterexample $ce = (I\_1, I\_2)(R\_1, I\_2)(C\_1, I\_2)(C\_1, R\_2)$ $(C\_1, C\_2)(C\_1, A\_2)(A\_1, A\_2)$. Upon SMT solving the mapped predicate constraint set, `Z3` returns "*unsat*" and an unsat-core that contains three atomic constraints $(n_v^2 = 1)$, $(n_u^3 = 0)$ and $(n_u^3 = n_v^2)$, where a superscript denotes the step number of the transition in the counterexample. The three constraints are mapped to $jump\_1((R\_1, C\_1))$, $guard\_2((I\_2, R\_2))$ and $guard\_2((I\_2, R\_2))$ respectively as shown below in bold:

$$\begin{array}{r} Proc\_1: \\[1em] Proc\_2: \end{array} \begin{pmatrix} I\_1 \\ I\_2 \end{pmatrix} \begin{pmatrix} \mathbf{R\_1} \\ I\_2 \end{pmatrix} \begin{pmatrix} \mathbf{C\_1} \\ \mathbf{I\_2} \end{pmatrix} \begin{pmatrix} C\_1 \\ \mathbf{R\_2} \end{pmatrix} \begin{pmatrix} C\_1 \\ C\_2 \end{pmatrix} \begin{pmatrix} C\_1 \\ A\_2 \end{pmatrix} \begin{pmatrix} A\_1 \\ A\_2 \end{pmatrix}.$$

Since the unsat-core doesn't include the initial condition constraint, its encoding is :

$$\texttt{Encode}(\textit{unsat-core}) = (\neg \textit{Unsafe})\mathbf{U}\big((R\_1\mathbf{X}C\_1)\mathbf{X}(I\_2\mathbf{X}R\_2)\big).$$

Accordingly, the relaxed specification formula for model checking in the next iteration is the disjunction of the current specification and the above encoding of the unsat-core:

$$\Big(\mathbf{G}(\neg \mathit{Unsafe})\Big)\Big|\Big((\neg \mathit{Unsafe})\mathbf{U}\big((R\_1\mathbf{X}C\_1)\mathbf{X}(I\_2\mathbf{X}R\_2)\big)\Big),$$

where "|" denotes the disjunction. The verifier repeats the execution loop as above for twelve iterations, finally showing that the unsafe state is not reachable. In the case of the 10-process Fischer mutual exclusion protocol, this corresponds to a space of $4^{10} = \sim 1 Million$ locations, and potentially $10 \times 5 \times 4^9 = \sim 13 Million$ edges (due to the user-specified enabling constraints), given an unsafe state *Unsafe*$= A\_1 \wedge A\_2$, the verifier also proves that the unsafe state is not reachable, and with almost the same running time as the 2-process case. By symmetry it follows that no other unsafe state (of the for, $A\_i \wedge A\_j, i, j \in \{1, \ldots, 10\}, i \neq j$, is reachable. One of the counterexamples is as follows, where as above the unsat-core is shown in bold:

$$\begin{array}{r} Proc\_1: \\ Proc\_2: \\ Proc\_3: \\ \cdots: \\ Proc\_10: \end{array} \begin{pmatrix} I\_1 \\ I\_2 \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} \mathbf{R\_1} \\ I\_2 \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} \mathbf{C\_1} \\ \mathbf{I\_2} \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} C\_1 \\ \mathbf{R\_2} \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} C\_1 \\ C\_2 \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} C\_1 \\ A\_2 \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix} \begin{pmatrix} A\_1 \\ A\_2 \\ I\_3 \\ \cdots \\ I\_10 \end{pmatrix}.$$

The unsat-core and its encoding are identical to the 2-process case. This shows that the algorithm has extremely good scalability due to the compactness of the unsat-core and its encoding even for a large state space system.

The constraints in the unsat-core shown above are all on the variable n at different steps, and the same holds for most of the other counterexamples we obtained. By observation, we know that n is a discrete finite valued variable with its value updated only on transitions. Thus, instead of treating n as a flow variable, we can treat it as a discrete state, with a finite set of locations ranging over the values n can take, with its transitions guarded by events that represent the discrete location transitions in the concurrent process model that update n. Such a model of n for the k-component case is shown in Figure 2.6, in which n varies among $k + 1$ discrete integer values $0, 1, \ldots, k$.

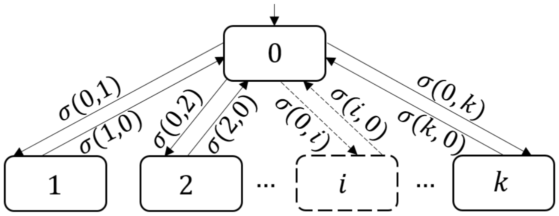


Figure 2.6: Transition system for n

An event set $\sigma(n, n')$ (with $n, n' \in \{0, 1, \ldots, k\}$) associated with an edge in this model can be derived from the original process model by looking at which discrete transition may cause the value change of n to n'. An example is the change from 0 to 1 that occurs if and only if when F_1 changes its discrete location from R_1 to C_1. Therefore when $k = 2$, $\sigma(0, 1)$ contains 4 valid events as follows:

$$\left\{\left(\begin{pmatrix}\mathbf{R\_1}\\ I\_2\end{pmatrix}\begin{pmatrix}\mathbf{C\_1}\\ I\_2\end{pmatrix}\right),\left(\begin{pmatrix}\mathbf{R\_1}\\ R\_2\end{pmatrix}\begin{pmatrix}\mathbf{C\_1}\\ R\_2\end{pmatrix}\right),\left(\begin{pmatrix}\mathbf{R\_1}\\ C\_2\end{pmatrix}\begin{pmatrix}\mathbf{C\_1}\\ C\_2\end{pmatrix}\right),\left(\begin{pmatrix}\mathbf{R\_1}\\ A\_2\end{pmatrix}\begin{pmatrix}\mathbf{C\_1}\\ A\_2\end{pmatrix}\right)\right\}.$$

Once the discrete transition model for n is created, the predicates over n can be discarded from each LHA model F_i, resulting in simpler LHA models $\tilde{F}\_i$. The concurrent model then is obtained by composing $\tilde{F}\_i, i = 1, \ldots, k$, and the transition model for n. Figure 2.7 shows the corresponding abstract model for the two-process case. A counterexample generated from this new model is given in [9] as: $(I\_1, I\_2, 0)$ $(I\_1, R\_2, 0)(R\_1, R\_2, 0)(R\_1, C\_2, 2)(R\_1, A\_2, 2)$ $(C\_1, A\_2, 1)(A\_1, A\_2, 1)$.

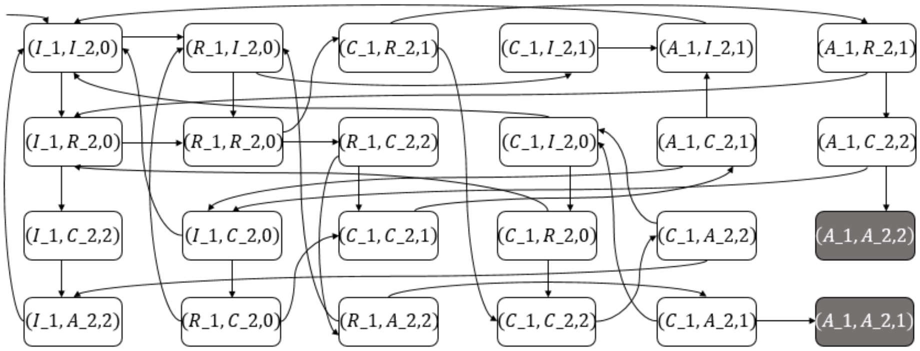


Figure 2.7: The new abstract model for Fischer protocol with two processes

Since n is no longer a flow variable, all the counterexamples related to n are eliminated. This reduces the number of iterations dramatically, and the length of the relaxed LTL specification reduces as a consequence. As a trade-off, the size of the abstract model gets larger (compare Figure 2.7 versus Figure 2.5). Never the less, since NuSMV has a proven scalability, the overall time-efficiency of the algorithm is enhanced.

So we can say that in general, if a concurrent linear hybrid system contains a variable that has finite domain, so it gets updated only along transitions (and not within the locations), then we can model the behavior of the variable as another transition system, and compose it with the other subsystems (with the constraints related to the variable suppressed). This corresponds to an "in-built" refinement of the model that automatically removes many of the invalid counterexample paths of the original model, and expedites the verification process.

2.4.2.1 Additional Benchmark Experiments

The illustrative example of Fischer protocol illustrated above can be generalized to be modeled as a timed automaton (TFischer and TFischer_10 in Table 2.1) and were analyzed using our tool. Additional experimental results on other benchmark examples were also analyzed and reported here; these were taken from Passel tool website (https://publish.illinois.edu/passel-tool/), including rectangular Fischers protocol (RFischer), the core component of the Small Aircraft Transportation System (SATS) landing protocol, and a purely discrete example (TMUX). All the experiments were performed on a laptop computer with 8G memory and Intel Core @2.30GHz processor. Table 2.1 summarizes performance of the prototype tool LhaVrf on these benchmarks.

Table 2.1: Table of Performance on benchmarks.

Benchmark	Safety property	Number of iteration.	Runtime(sec)
TFishcer	✓	12	4.85
TFischer_10	✓	12	5.12
RFischer	✓	12	5.01
RFischer (Buggy)	✗	2	2.15
SATS	✓	10	4.99
TMUX	✓	6	3.17

CHAPTER 3. SIMULATION BASED VERIFICATION OF BOUNDED-HORIZON SAFETY FOR HYBRID SYSTEM USING DYNAMIC NUMBER OF SIMULATIONS

3.1 Introduction

Testing and simulation techniques are often used for run-time monitoring of safety-critical systems such as air-traffic management system, aircraft and automobile control. Such systems are hybrid, possessing discrete states (also called locations), where different physical laws govern the evolution of the continuous states, while a set of predicates guard the transitions of the discrete states and constrain the reset values of the continuous states. However, testing and simulation are not comprehensive to fully discover system flaws. Alternatively, formal verification methods can be employed. As decribed in Chapter 2, a common design-time safety verification involves computing the reachable set over-approximation symbolically, and checking whether the over-approximation does not reach the unsafe region [9, 10, 12, 19, 20, 27, 36, 50]. In contrast, the alternative simulation-based verification approach, that can be applied in run-time, generates a finite set of simulation traces from current system states, then computes over-approximation sets around those simulation trajectories to cover for the infinite set of cases that were not simulated [51, 52, 53, 54, 55]. The computation of the over-approximation sets relies on certain continuity properties.

The reachable set is usually non-convex and hard to compute precisely, and so those are typically over-approximated using certain easy-to-compute representations. Examples are *support function* in `SpaceEx` [27], *Taylor model* in `Flow*` [36], polyhedra/balls in `C2E2` [53, 54, 55], and zonotope in `CORA` [56, 57]. In `SpaceEx` and `Flow*`, the reachable set over-approximation of a set S over each time-step is performed directly on S using symbolic representation and one-step evolution's approximated post-image formulas, and implemented through support function or Taylor Model. This introduces additional errors when computing discrete post-images, due to a lack of compatibility between the reachable state representation in continuous stage vs. guard/reset representation used for discrete transition (e.g. a hyperplane). This calls for representation conversion, which requires adoption of additional over-approximation, when connecting two consecutive stages of continuous evolution. Tools that require representation conversion usually do not perform well on models with frequent discrete jumps.

In contrast, in a bounded horizon *verification from simulation* approach, the correctness is established from multiple simulation runs of the system, and over-approximating tubes around those, whose initial face covers the entire initial set. Over time, the scope of study has expanded from continuous linear dynamics [51, 52] to those described by a richer class of hybrid nonlinear ones [53, 54, 55]. In [53], authors presented a framework that formally verifies the *bounded time* safety of a subclass of hybrid nonlinear systems, the so called *switched systems.* A switched system has time-triggered mode changes, determined by pre-specified switching signals indicating the switching times and destination locations. In contrast, a general hybrid system changes its discrete mode autonomously as determined from the guard predicates over states, and the sequence of discrete mode switches are not known a priori. A contribution in our work is to develop the simulation-based verification approach for this general class of hybrid dynamics.

In prior works, a simulation of *entire* time horizon is used as a reference or representative for a class of system behaviors starting close to the reference. *Discrepancy functions* [58, 59] are used to bound the state deviation from neighboring non-reference states. In [53], the use of discrepancy function requires users to provide control theoretic annotations on the dynamics. Moreover, the algorithms in [53] are based upon two assumptions on the hybrid systems, namely, (i) state resets are not allowed, and (ii) the executions starting from the same class of initial states/inputs must all experience the same discrete behaviors [54, 55]. Both assumptions are restrictive, e.g., the first assumption does not hold for a simple bouncing ball. On the other hand, for a general hybrid system, a simulation error may cause a deviation in the result of discrete-jump, violating the second assumption above. Once a location deviation occurs, the original simulation is no longer a reliable estimate, meaning that a simulation of the entire time horizon may not always qualify as a reliable reference. In addition, determining a finite set of representative initial states, that represent executions that visit the same sequence of discrete locations within a given time horizon, is not yet proven to be computable. Here, we proposed a simulation-based verification algorithm for bounded-time safety that can deal with general nonlinear hybrid systems, while relaxing the aforementioned assumptions, and also bypassing the computability issue of the representative set of initial states, by way of their dynamic, on-the-fly, and as-needed computation.

A key concept in the simulation-based approaches is the discrepancy function that is used for bounding the deviation between two neighboring trajectories as a function of their initial distance and time. [53] has shown that some proof certificates routinely used in stability analysis of dynamical systems are in fact discrepancy functions. For example, Lipschitz property of a flow function lends an exponentially growing bound among the system trajectories. On the other hand, incremental stability of a system specifies that the distance between two trajectories remains bounded by a $\mathcal{KL}$ function of their initial distance and time. An even stronger property exists for system with a contraction metric, whose trajectories converge exponentially with time. In [60], under the assumptions ensuring incremental stability of a switched system, it is possible to construct a symbolic model that is approximately bisimilar to the original switched system with a precision that can be chosen a priori. Algorithmic controller syndissertation then can be applied on the symbolic model, which is supposedly easier to handle than the original model. In the simulation-based reachability verification, discrepancy functions based on the $\mathcal{KL}$ function, are shown to be able to diminish the over-approximation error bound [53]. However, simply assuming the incremental stability is not enough, one has to actually obtain a $\mathcal{KL}$ discrepancy function prior to a computation based on it. As [53] pointed out, obtaining such a bounded relationship among trajectories is computationally intractable in general. Aside from using better discrepancy functions, several other heuristic techniques such as using localized exponential discrepancy functions during each time interval, have been adopted by C2E2 to slow down the error growth [23, 24, 38]. In this chapter, we have developed a practical approach that bypasses the intractable pre-verification analysis of finding a $\mathcal{KL}$ function if it exists, uses the easily computable Lipschitz constant based discrepancy function, yet also alleviates the fast growth of over-approximation errors for stable systems. Lipschitz constants can be computed algorithmically for linear, polynomial, and certain classes of trigonometric functions. For more general classes, empirical techniques can estimate it over closed subsets [61].

Briefly, in our approach, the bounded initial set is covered by a partition whose cells are γ-hypercubes (so distance from center of hypercube to a side is γ, and where γ is a user-defined partition size parameter). The center of a hypercube is chosen as its representative state. Each such representative state is forward simulated in discrete time-steps of user chosen intervals, and the corresponding error is tracked using a bound. After a finite number of time-steps (which is another user-defined parameter), the states reached at the current final time are covered by a new partition with cells that are again γ-hypercubes. Also, when a discrete jump occurs, the newly reached states after the jump are again covered by a partition with γ-hypercubes cells. Upon each such partition, the process of selecting the representative states, one per cell, and their forward simulations, continues, until the required time for forward simulation bound is elapsed. We refer to this type of on-the-fly partitioning as “dynamic partitioning”. In this way, the exponential error growth among trajectories remains confined to short time-intervals, as opposed to the entire simulation horizon. As a result, the over-approximation bounds evolve at a much less growth rate (or even at a decaying rate for the case of converging trajectories). We have proven that, for incremental (input-to-state) stable dynamics, the two growth trends culminate into a bounded deviation, which can be made arbitrarily small with the choice of simulation/partition parameters. This boundedness of the trajectory deviation estimation using only a Lipschitz constant based discrepancy function, is a novel feature of our approach. This makes our approach practical since a Lipschitz-based discrepancy function is easily computable, while a more precise discrepancy function may not be available since there is no known algorithm to compute it in general. (Certainly if a more precise discrepancy function is available, our algorithm can also utilize it.) Our contributions are summarized as follows:

- Our dynamic partitioning approach supports state-triggered discrete jumps with guard/reset conditions. Errors in simulation can alter discrete behaviors rendering a simulation-trace obsolete, when state-triggered discrete transitions are allowed. Our approach circumvents this problem by introducing new simulations as needed, and on-the-fly.

- Our dynamic partitioning scheme controls the error growth of over-approximation, so that even for a Lipschitz constant based discrepancy function, it guarantees that the error converges to a constant bound for incremental stable systems.
- This guarantees that the number of representative simulations also converge for an incremental stable system, and in fact may become smaller in number as forward simulations are carried out.
- Our approach relies on discrepancy function based on Lipschitz constant, that is easy to compute in practice, without sacrificing accuracy due to our use of dynamic partitioning. More precise discrepancy function are not practically computable – their computation may even be undecidable. But those can be used in our approach if available.
- Our approach allows piecewise constant bounded inputs.
- Our approach only partitions the "reachability boundary" that reduces the number of representative simulations while maintaining verification correctness.
- We have implemented a prototype verifier `HS`^{3}`V` incorporating our innovations, and have tested an compared it on a variety of benchmarks.

This chapter extends the conference version [39] in multiple ways. Firstly, a new section on error growth control is added, explaining the dynamic partitioning scheme, providing a correctness theorem and its corollaries regarding the boundedness of trajectory deviation errors, and discussing its novelty and benefits. Secondly, we updated the tool `HS`^{3}`V` implementing the above new ideas. Lastly, more experimental results are presented to show the improvement of the error growth control for both continuous and hybrid dynamic evolution.

Rest of the chapter is organized as follows. Notations and preliminaries are given in Section 3.2. The stepwise computation of the reachable set over-approximation regarding continuous evolution is described in Section 3.3. The dynamic partitioning scheme for error growth control is given in Section 3.4. The complete algorithm is in Section 3.5. The implementation of our prototype tool and experimental results for several benchmarks are given in Section 3.6.

3.2 Notation and Preliminary

For a vector $\vec{v} \in \mathbb{R}^n$, the notation $\|\vec{v}\|$ denotes its ℓ^∞ norm. The *diameter* of a bounded set S of a metric space is denoted as $D(S) = sup\{\|\vec{v} - \vec{v}'\| | \vec{v}, \vec{v}' \in S\}$. $B_\gamma(\vec{v}) = \{\vec{v}' | \|\vec{v}' - \vec{v}\| \leq \gamma\}$ denotes the γ-*hypercube* centered at $\vec{v}$. The *Minkowski sum* of two sets S and S' in a vector space is defined as $S \oplus S' = \{\vec{v} + \vec{v}' | \vec{v} \in S_1, \vec{v}' \in S'\}$. A continuous function $f : [0, a) \mapsto \mathbb{R}_{\geq 0}$ is a class $\mathcal{K}$ function if it is strictly increasing and $f(0) = 0$. It is said to belong to class $\mathcal{K}_\infty$ if it is defined on $\mathbb{R}_{\geq 0}$ and $f(s)$ goes to ∞ as s goes to ∞. A function $f : [0, a) \times \mathbb{R}_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$ is a class $\mathcal{KL}$ function if, for each fixed t, the function $f(\cdot, t)$ belongs to class $\mathcal{K}$; for each fixed s, the function $f(s, t)$ is decreasing and approaches to 0 as t goes to ∞. A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is *Lipschitz* if there exists a constant $c > 0$ such that $\|f(x) - f(x')\| \leq c\|x - x'\|$ for all x, x' in the domain of f. A matrix is called *Hurwitz* when all its eigenvalues have non-negative real parts.

In this chapter, we analyze a class of deterministic hybrid automata in which the continuous dynamics have piecewise constant bounded inputs. The following definition is a simplified version of the standard definition of input-output hybrid automaton (I/O HA) from [11]; the invariant, guard, and reset conditions only depend on states.

Definition 4. *A hybrid automaton with inputs is a tuple $A = (L, V, E, V_0, U, flow, inv, guard, reset)$, where:*

- ***Discrete state:*** *L is a finite set of discrete locations.*
- ***Continuous state:*** *$V = \{v_{(1)}, v_{(2)}, \ldots, v_{(n)}\}$ is a finite set of real-valued state variables. $\vec{v} = [v_{(1)}, v_{(2)}, \ldots, v_{(n)}]$ is their vector representation.*
- ***Discrete jumps:*** *$E \subseteq L \times L$ is the set of discrete jumps.*
- ***Initial state set:*** *$V_0 \subset \mathbb{R}^n$ is the nonempty bounded initial state set (initial set for short).*
- ***Continuous Input:*** *$U = \{u_{(1)}, u_{(2)}, \ldots, u_{(m)}\}$ is a finite set of real-valued input variables. Let $\vec{u}_{(i)}(t) : \mathbb{R}_{\geq 0} \mapsto \mathbb{R}$ be a piecewise continuous bounded function of t, that describes the value of $u_{(i)}$ changing over time. $\vec{u} = [u_{(1)}, u_{(2)}, \ldots, u_{(m)}]$ is the vector representation of input variables and $\vec{u}(t)$ is defined analogously.*

- ***Flow:*** *The flow function $flow$ assigns each location $l \in L$ a differential equation $flow(l) : \dot{\vec{v}} = f_l(\vec{v}, \vec{u})$ where $f_l : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n$ constrains the rates at which the variables change w.r.t. time.*
- ***Invariant:*** *The invariant function inv assigns each location $l \in L$ a predicate $inv(l) \subseteq \mathbb{R}^n$ over the domain of $\vec{v}$ that constrains the values of the variables within the location l.*
- ***Guard:*** *The guard function $guard$ assigns each discrete jump $e = (l, l') \in E$ a predicate $guard(e)$ over the domain of $\vec{v}$ such that the jump e is enabled at a state $\vec{v} \in inv(l)$ if and only if $\vec{v} \in guard(e)$.*
- ***Reset:*** *The reset function $reset$ assigns each discrete jump $e = (l, l') \in E$ a function $reset_e$ over the domain of $\vec{v}$ such that the jump e enabled at state $\vec{v} \in inv(l)$ is accompanied with a value reset: $\vec{v}' = reset_e(\vec{v}) \in inv(l')$. Define $reset_e(S) = \{\vec{v}' | \exists \vec{v} \in S, \vec{v}' = reset_e(\vec{v})\}$.*

In our study, we make the following assumptions:

- **Piecewise constant bounded input:** We assume that the input $u(t)$ for each initial state is a piecewise constant function, where the discrete changes occur only at the simulation sample times. In other words, $\forall t \geq 0, \vec{u}(t) \in B_{r_u}(\vec{u}_{\tilde{0}}(t))$, with $\vec{u}_{\tilde{0}}(t)$ is a piecewise constant function representing nominal input and $r_u \in \mathbb{R}_{\geq 0}$.
- **Linear guard predicate:** The guard predicates are linear, each representing a hyperplane in $\mathbb{R}^n$. This assumption is mainly for the ease of computation of the intersection between the polyhedral representation of the reachable set and the guards. The guard predicates can be taken to be more general semi-algebraic sets, if performing the set operations based on the the cylindrical algebraic decomposition [62].
- **Affine reset function:** The reset functions are affine. This again is more of computational simplicity, as affine maps preserve polyhedral sets.
- **Lipschitz continuity:** For each $l \in L$, the flow function f_l is Lipschitz in $\vec{v}$ and $\vec{u}$ with Lipschitz constant c_l.
- **Simulation time-step can be chosen smaller than dwell-time:** By dwell-time, we refer to the time spent while evolving within a single location. We assume the system possesses a

minimum dwell-time $\Delta > 0$ over the duration of simulation horizon, so that we can pick a simulation time-step $\delta < \Delta$ to avoid multiple discrete jumps in a single time-step.

Given a hybrid automaton A as defined in Definition 4 satisfying the above assumptions, we have the following definitions regarding its behavior and simulation.

Definition 5. *A trajectory of A, denoted by τ, is a bounded-time continuous evolution of the state inside a discrete location. Specifically, τ is a function, $\tau : [0, T] \mapsto inv(l)$, where $T \in \mathbb{R}_{>0}$ and $l \in L$. We denote the trajectory from state $\vec{v}_0 \in inv(l)$ with input signal $\vec{u}_0(t)$ as $\tau_{\vec{v}_0, \vec{u}_0(t)}$. $\tau_{\vec{v}_0, \vec{u}_0(t)}(t)$ is the solution of the flow differential equation $\dot{\vec{v}} = f_l(\vec{v}, \vec{u})$, with $f_l : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n$, initial state $\vec{v}_0$ and input signal $\vec{u}_0(t)$. The Lipschitz continuity assumption ensures the existence and uniqueness of the trajectory for all initial states and inputs.*

Definition 6. *A simulation trajectory of A, is a sequence of state-time pairs, obtained by simulation using numerical algorithms, that estimates the continuous variables at discrete time instances. For a location $l \in L$, initial state $\vec{v}_0 \in inv(l)$, input signal $\vec{u}_0(t)$, sample time-step $\delta > 0$, single time-step simulation error $\epsilon > 0$ and time bound T, a $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory is a finite sequence $(\vec{v}_0, 0)(\vec{v}_1, \delta) \ldots (\vec{v}_k, k\delta)$, where $k = \lceil T/\delta \rceil$, and*

- $\forall i \in \{0, \ldots, k\}, \vec{v}_i \in inv(l)$.
- $\forall i \in \{1, \ldots, k\}, \|\vec{v}_i - \tau_{\vec{v}_{i-1}, \vec{u}_0(t+(i-1)\delta)}(\delta)\| \leq \epsilon$.

Definition 7. *An execution of A, denoted by α, is a finite sequence of trajectories concatenated via discrete jumps. A finite sequence $\tau_0 e_1 \tau_1 \ldots e_k \tau_k$ is an execution of A if it satisfies the following*

- *Initial condition* $\tau_0(0) \in V_0$.
- $\forall i \in \{0, \ldots, k\}, \tau_i : [0, T_i] \mapsto inv(l_i)$, *is a trajectory.*
- $\forall i \in \{1, \ldots, k\}$, $e_i = (l_{i-1}, l_i) \in E$, $\tau_{i-1}(T_{i-1}) \in guard(e_i)$.
- $\forall i \in \{1, \ldots, k\}$, $\tau_i(0) = reset_{e_i}(\tau_{i-1}(T_{i-1}))$

A state $\vec{v}_* \in \mathbb{R}^n$ is reachable in bounded time T, if and only if there exist an execution $\alpha = \tau_0 e_1 \tau_1 \ldots e_k \tau_k$ such that $\tau_k(T_k) = \vec{v}_*$ and $\Sigma_{i=0}^{k} T_i \leq T$.

3.3 Reachable Tube Computation

3.3.1 Error Growth in Continuous Evolution

A tube constructed around a trajectory τ according to the discrepancy function of the location dynamics contains all the trajectories starting in a neighborhood of $\tau(0)$, and in which case τ is referred to as a reference trajectory. In a simulation-based bounded-horizon safety verification approach, we can use the simulation trajectory starting from $\tau(0)$ to approximate the reference trajectory τ, by accounting for the simulation error bound.

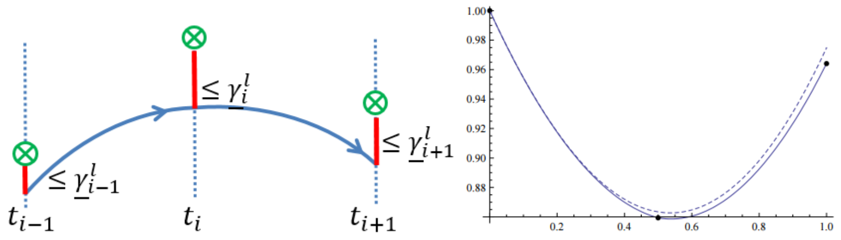
Consider a reference trajectory $\tau_{\vec{v}_0,\vec{u}_0(t)}$ over $[0,T]$ in l and a neighboring trajectory $\tau_{\vec{v}'_0,\vec{u}'_0(t)}$ also in l, where $\|\vec{v}'_0 - \vec{v}_0\| \leq r_v$ and $\|\vec{u}'_0(t) - \vec{u}_0(t)\| \leq r_u$ for all $t \in [0,T]$. Let $(\vec{v}_0, 0)\ldots(\vec{v}_k, k\delta)$ be the $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory, and $\{\gamma^l_i = \|\tau_{\vec{v}'_0,\vec{u}'_0(t)}(i\delta) - \vec{v}_k\|\}^k_{i=0}$ denote the collection of the ℓ^∞ distances between $\tau_{\vec{v}'_0,\vec{u}'_0(t)}$ (the actual trajectory) versus the simulation trajectory at the sample times. By definition, $\tau_{\vec{v}_0,\vec{u}_0(t)}$, $\tau_{\vec{v}'_0,\vec{u}'_0(t)}$ and the simulation trajectory follow the same continuous dynamics $flow(l)$. Then we have the following proposition [39], which can be derived from Lemma 1 in [54] but with extension to also include the inputs. For an unforced system (i.e., with zero input), Proposition 1 reduces to Lemma 1 in [54].

Proposition 1. *Consider the notation in the paragraph above. Then,*

$$\gamma^l_0 = \|\vec{v}'_0 - \vec{v}_0\| \leq r_v, \textit{ and } \forall i \in \{1, \ldots, i\}, \gamma^l_i \leq \gamma^l_{i-1} e^{c_l \delta} + r_u \delta (e^{c_l \delta} - 1) + \epsilon,$$

where c_l is the Lipschitz constant for $flow(l)$, δ is the simulation time-step, and ϵ is the single time-step simulation error.

The sequence $\{\gamma^l_i\}^k_{i=0}$ is restricted by the recursive inequalities given in Proposition 1. The unique sequence that satisfies the *equality* at each recursion gives the most precise upper bounds of the distances between $\tau_{\vec{v}'_0,\vec{u}'_0(t)}$ and the $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory at the sample times. This unique sequence, denoted by $\{\underline{\gamma}^l_i\}^k_{i=0}$, is useful since for most nonlinear dynamics, an analytical solution is not available. Figure 3.1(a) shows a fragment of $\tau_{\vec{v}'_0,\vec{u}'_0(t)}$ marked by the blue curve and a $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory with its simulation values marked by the green dots. The error at the i^{th} simulation time-step is bounded by $\underline{\gamma}^l_i$.


$\le \underline{\gamma}_{i-1}^l$
$\le \underline{\gamma}_i^l$
$\le \underline{\gamma}_{i+1}^l$
t_{i-1}
t_i
t_{i+1}
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.2
0.4
0.6
0.8
1.0

(a) (b)

Figure 3.1: (a) A two time-step fragment of $\tau_{\vec{v}_0',\vec{u}_0'(t)}$ and $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory, $\{\underline{\gamma}_i^l\}_{i=0}^k$ are computed by Proposition 1. (b) The first two time-steps of solving the equation $\dot{x} = t - 1/(1+x)$ with x(0)=1 and $\delta = 0.5$ numerically. The dashed curve shows the trajectory, the dots show the simulation values, while the solid curves show the parabolas that are used to compute the simulation. *Figure 3.1(b) is taken from [63].*

In the context of hybrid dynamics that allows state-triggered discrete jumps, it is not always possible to treat a sequence of state-time pairs obtained by a simulation engine as a reliable simulation trajectory. A simulation error may cause a discrete evolution to deviate, making the forward simulation unreliable. For this reason, we cannot use simulations over the entire time horizon to estimate the hybrid executions. To ensure that every simulation state-time pair is a reliable estimate of the actual behavior, the simulation is examined at the end of each time-step, to check whether the accumulated simulation error can cause a discrete location deviation. For this, we examine each error bound to see if any discrete jump can be triggered at the end of any time-step. The handling of the possible discrete jumps is described in Section 3.5, and integrated into the overall reachability over-approximation algorithm.

3.3.2 Reachable Set in a Single Time-Step

Consider the group of trajectories with initial discrete location $l \in L$, initial states in $B_{r_v}(\vec{v}_0)$, and input signals in $B_{r_u}(\vec{u}_0(t))$ over $[0, T]$. We construct a tube that contains all this group of trajectories around the $(\vec{v}_0, \vec{u}_0(t), \delta, \epsilon, T)$-simulation trajectory, $(\vec{v}_0, 0) \ldots (\vec{v}_k, k\delta)$, where recall that $k = \lceil T/\delta \rceil$ by Definition 6. The tube segments are computed for each time-step using Algorithm 3, which extends the one in [54] to also allow inputs.

Algorithm 3: Building tube segment during $[i\delta, (i+1)\delta]$.

```
  Input: $A, l, \vec{v}_0, \vec{u}_0(t), r_u, \delta, \underline{\gamma}_i^l, b \in \mathbb{R}_{>1}$.
1 $\vec{v}_i \leftarrow$ simu$(\vec{v}_0, \vec{u}_0(t), i)$, $\sigma \leftarrow \underline{\gamma}_i^l$;
2 do
3     $\sigma \leftarrow b \cdot \sigma$;
4     $d = \sup_{\vec{v} \in B_\sigma(\vec{v}_i), \vec{u} \in B_{r_u}(\vec{u}_0(i\delta))} \|f_l(\vec{v}, \vec{u})\|$;
5 while $\sigma - d\delta < \underline{\gamma}_i^l$;
6 $\overline{f} = \sup_{\vec{v} \in B_\sigma(\vec{v}_i), \vec{u} \in B_{r_u}(\vec{u}_0(i\delta))} f_l(\vec{v}, \vec{u})$;
7 $\underline{f} = \inf_{\vec{v} \in B_\sigma(\vec{v}_i), \vec{u} \in B_{r_u}(\vec{u}_0(i\delta))} f_l(\vec{v}, \vec{u})$;
8 $R^l_{[i,i+1]}(\vec{v}_0, \vec{u}_0(t)) = B_{\underline{\gamma}_i^l}(\vec{v}_i) \oplus \{t \cdot f | t \in [0, \delta], f \in [\underline{f}, \overline{f}]\}$;
  Output: $R^l_{[i,i+1]}(\vec{v}_0, \vec{u}_0(t))$
```

Input to Algorithm 3 includes a system model A, a particular location l where the continuous evolution occurs, simulation starting point $\vec{v}_0$, input signal $\vec{u}_0(t)$, input range parameter r_u, simulation time-step δ, the error bound $\underline{\gamma}_i^l$ computed using Propositional 1, and a constant gain factor $b > 1$. In line 1, simu$(\vec{v}_0, \vec{u}_0(t), i)$ returns the simulation value $\vec{v}_i$ at time $i\delta$ with initial value $\vec{v}_0$ and input signal $\vec{u}_0(t)$; σ is a constant that defines the σ-hypercube $B_\sigma(\vec{v}_i)$ (in Figure 3.2 it is the area marked by the dashed square) a conservative over-approximation of the one-step reachability of $B_{\underline{\gamma}_i^l}(\vec{v}_i)$ (in Figure 3.2 it is the area marked by the solid square centered at $\vec{v}_i$).

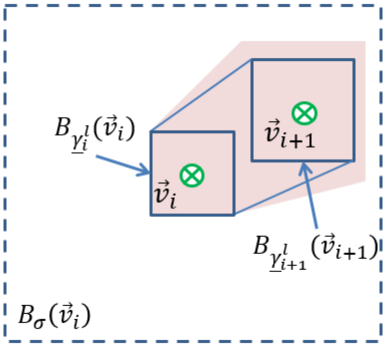

$B_{\underline{\gamma}_i^l}(\vec{v}_i)$
$\vec{v}_{i+1}$
$\vec{v}_i$
$B_{\underline{\gamma}_{i+1}^l}(\vec{v}_{i+1})$
$B_\sigma(\vec{v}_i)$

Figure 3.2: A 2-dimensional tube segment over the time-step $[i\delta, (i+1)\delta]$ (marked by the red shadowed area). $B_{\underline{\gamma}_i^l}(\vec{v}_i)$ is the over-approximation of the reachable set *at time instant* $i\delta$. Note $\{\underline{\gamma}_i^l\}_{i=0}^k$ only bound the trajectory deviations at the sample times, and so the convex hull of $B_{\underline{\gamma}_i^l}(\vec{v}_i)$ and $B_{\underline{\gamma}_{i+1}^l}(\vec{v}_{i+1})$ doesn't necessarily over-approximate the reachable set in between $i\delta$ and $(i+1)\delta$ due to the nonlinearity of the continuous dynamics. Hence the red shadowed area can be larger than the said convex hull.

The final choice of σ is obtained by repeatedly scaling itself by b starting with the initial value $\underline{\gamma}_i^l$ until the criteria in line 5 of Algorithm 3 is satisfied. The value of b is chosen by considering a trade-off between the precision versus the number of iterations. In practice, b is set between 1.1 and 2.5, and the default value is set to 2 in our implementation. [54] has proved that $c_l \cdot \delta < 1$, then this loop will indeed terminate. In lines 6-7, the element-wise maximum and minimum changing rates of $\vec{v}$ in $B_\sigma(\vec{v}_i)$, denoted by $\overline{f}$ and $\underline{f}$ respectively, are computed by solving an optimization problem over f_l. In line 4, d is computed analogously. Line 8 defines the tube segment as $R^l_{[i,i+1]}(\vec{v}_0, \vec{u}_0(t))$, which is the Minkowski sum of $B_{\underline{\gamma}_i^l}(\vec{v}_i)$ and its evolution in one time-step. The correctness of Algorithm 3 immediately follows the correctness of Proposition 1 (proved in [39]) and Algorithm 1 in [54]. (For the case of a discrete jump within a time-step, visit Section 3.5 below.)

3.4 Error Growth Control

From Proposition 1, the error bound based on the Lipschitz constant can be seen to grow exponentially. In this section, we introduce a dynamic partitioning scheme to alleviate the error growth problem, without having to introduce a discrepancy function that is more precise than the one based on Lipschitz constant. Note the motivation for doing so was discussed in introduction, namely, that only the latter is tractably computable. We also show that for an incremental (input-to-state) stable system, our approach provides a constant error bound in the limit, even though we continue to utilize Lipschitz continuity based error bound that is practically computable (in contrast to a more precise bound, based for example an incremental stability property, that may not even be available; our approach can certainly also use the precise bound when available).

Consider a location $l \in L$ of a hybrid automaton A with the flow dynamics

$$\dot{\vec{v}} = f_l(\vec{v}, \vec{u}), \tag{3.1}$$

with f_l Lipschitz in $\vec{v} \in \mathbb{R}^n$ and $\vec{u} \in \mathbb{R}^m$. Assuming no discrete jump, we propose Algorithm 4 to compute the reachable set over-approximation under continuous evolution within l starting within the r_v-hypercube $B_{r_v}(\vec{v}_0)$, under input signal $\vec{u}(t) \in B_{r_u}(\vec{u}_0(t))$, and over time bound T. We assume here for simplicity of illustration that there is no discrete evolution; the more general case is discussed in the next section. Algorithm 4 periodically repartitions the states reached at the end of a certain number of user-selected time-steps, still using the r_v-hypercube cells. This repartitioning "interrupts" the exponential growth of trajectory error bound of Proposition 1, and in fact leads to a bounded error in case of incremental (input-to-output) stable systems.

Algorithm 4: Multiple time-steps computation of the reachable set in location l starting from a single hypercube in bounded time with dynamic partitioning.

```
Input: $A, l, \vec{v}_0, \vec{u}_0(t), r_v, r_u, \{\underline{\gamma}_j\}_{j=0}^{m}, \delta, T, m \in \mathbb{R}_{>0}$.
$R^l \leftarrow B_{r_v}(\vec{v}_0), C^l \leftarrow \{\vec{v}_0\}$;
for $i = 0; i < \lceil T/\delta \rceil; i++$ do
    $R^l \leftarrow R^l \cup \bigcup_{\vec{v} \in C^l} R^l_{[i,i+1]}(\vec{v}, \vec{u}_0(t))$;
    if $(i > 0) \wedge (i\%m = 0)$ then
        $C^l \leftarrow$Partition$(\bigcup_{\vec{v} \in C^l} B_{\underline{\gamma}^l_m}(\vec{v}_m), r_v)$
    end
end
Output: $R^l$
```

Theorem 3. *Algorithm 4 is sound:* $\forall \vec{v} \in B_{r_v}(\vec{v}_0)$, $\forall t \in [0, T], \forall \vec{u}(t) \in B_{r_u}(\vec{u}_0(t))$, $\tau_{\vec{v},\vec{u}(t)}(t) \in R^l$.

Proof. In Algorithm 4, for $k = 0, 1, \ldots, \lfloor T/(m\delta) \rfloor$, C^l is unchanged in between $km\delta$ and $(k+1)m\delta$, and let it be denoted by C^l_k. From Propositional 1, $\bigcup_{\vec{v} \in C^l_k} B_{\underline{\gamma}^l_m}(\vec{v}_m)$ over-approximates the set of states reached from $\bigcup_{\vec{v} \in C^l_k} B_{r_v}(\vec{v})$ at the final time $m\delta$ of m time-steps, i.e., $\forall \vec{v} \in B_{r_v}(\vec{v}_0)$, $\forall \vec{u}(t) \in B_{r_u}(\vec{u}_0(t))$, $\forall k \in [0, \lfloor T/(m\delta) \rfloor - 1]$:

$$\tau_{\vec{v},\vec{u}(t)}(km\delta) \in \bigcup_{\vec{v} \in C^l_k} B_{r_v}(\vec{v}) \Longrightarrow \tau_{\vec{v},\vec{u}(t)}((k+1)m\delta) \in \bigcup_{\vec{v} \in C^l_k} B_{\underline{\gamma}^l_m}(\vec{v}_m) \subseteq \bigcup_{\vec{v} \in C_{k+1}} B_{r_v}(\vec{v}).$$

This is equivalent to saying that, $\tau_{\vec{v},\vec{u}(t)}(km\delta)$ is covered by C^l_k for any $k \in [0, \lfloor T/(m\delta) \rfloor]$. Consequently, there exists a tube segment within $[km\delta, (k+1)m\delta]$ that contains $\tau_{\vec{v},\vec{u}(t)}(t)$.

R^l is the union of all tube segments in $[0, \lceil T/\delta \rceil \cdot \delta]$. Thus, $\forall t \in [0, T], \tau_{\vec{v},\vec{u}(t)}(t) \in R^l$. □

For $k \in [1, \lfloor T/(m\delta) \rfloor]$, let $R^l_{km} = \bigcup_{\vec{v} \in C^l_{k-1}} B_{\underline{\gamma}^l_m}(\vec{v}_m)$ be the reachable set over-approximation at time-step km. Accordingly, let $e_k = D(R^l_{km})$ be the diameter of R^l_{km}. Figure 3.3 depicts the error growth over the first $2m$ time-steps. Algorithm 4 attains notably good results for the bounds $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ compared to what would be obtained by simply Algorithm 3, without the repartitioning introduced in Algorithm 4.

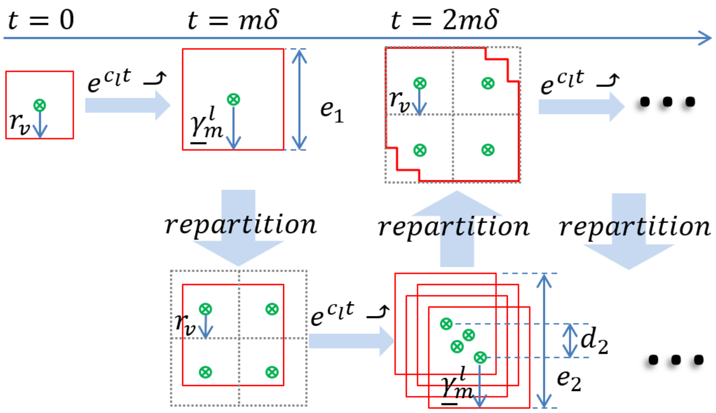


Figure 3.3: Error growth from an r_v-hypercube. Trajectory deviation bound between partitions grows exponentially, while distance among tubes (measured by the distance among simulation values at the sample times, such as d_2 in this figure) grows slower or even diminishes.

3.4.1 Effect of Stability Property on Error Growth

A nice feature of Algorithm 4 is that when the underlying dynamics converges over time, so does the error bound of over-approximation computed by the algorithm. We first introduce a notion of converging dynamics.

Definition 8. *Dynamics (3.1) is incremental input-to-state stable (δ-ISS for short) [64] if there exist a $\mathcal{KL}$ function β and a function $\gamma \in \mathcal{K}_\infty$ such that for any $t \geq 0$, any $\vec{v}, \vec{v}' \in \mathbb{R}^n$, and any pair of bounded input signals $\vec{u}(t), \vec{u}'(t)$, the following holds:*

$$\|\tau_{\vec{v},\vec{u}(t)}(t) - \tau_{\vec{v}',\vec{u}'(t)}(t)\| \leq \beta(\|\vec{v} - \vec{v}'\|, t) + \gamma(\sup_{0\leq\xi\leq t} \|\vec{u}(\xi) - \vec{u}'(\xi)\|).$$

The $\mathcal{KL}$ function $\beta(\|\vec{v} - \vec{v}'\|, t)$ is decreasing and goes to 0 as t goes to ∞, whereas the bounded input difference gives rise to a bounded trajectory deviation.

In Algorithm 4, partitions are updated at every m time-steps, and $e_k = D(R^l_{km})$ is the diameter of the reachable set over-approximation at time-step km. We establish the following Theorem:

Theorem 4. *For flow(l) that is δ-ISS, $\exists c \in \mathbb{Z}_+$ such that $\forall m \geq c$, the over-approximation error sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ has a constant bound under Algorithm 4.*

Proof. During the first m time-steps, the initial tube grows exponentially per Proposition 1. Hence, we have $e_1 = 2\underline{\gamma}^l_m$. When $k \geq 1$, at time-step km, the set of representative states of R^l_{km} is used as the simulation starting point set C^l_k. Since $\underline{\gamma}^l_m > \underline{\gamma}^l_0 = r_v$, we know $D(R^l_{km}) > r_v$. Hence the representative set of R^l_{km} (equivalently, C^l_k) contains at least two elements. Suppose τ_a, τ_b are two reference trajectories within $[km\delta, (k+1)m\delta]$ starting from $\vec{v}_{a0}, \vec{v}_{b0} \in C^l_k$. Due to partitioning,

$$\max_{\vec{v}_a, \vec{v}_b \in C^l_k} (\|\vec{v}_a - \vec{v}_b\|) \leq e_k.$$

By Definition 8, for some $\beta \in \mathcal{KL}$ and $\gamma \in \mathcal{K}_\infty$,

$$\|\tau_a(m\delta) - \tau_b(m\delta)\| \leq \beta(\|\vec{v}_a - \vec{v}_b\|, m\delta) + \gamma(r_u).$$

Let d_{k+1} denote the maximum ℓ^∞ distance between any two simulation values at time $(k+1)m\delta$ *before* partition (see for example d_2 in Figure 3.3). Due to the simulation error, there exists deviation between a reference trajectory and the corresponding simulation trajectory. Let $\underline{\gamma}^l_\epsilon$ denote the bound on their deviation at time-step m. Since the two start at the same state, $\underline{\gamma}^l_\epsilon$ can be computed by Proposition 1, using a base value 0 and m recursive equalities. Combining all factors, we have:

$$d_{k+1} \leq \beta(e_k, m\delta) + \gamma(r_u) + 2\underline{\gamma}^l_\epsilon.$$

Since β is decreasing and goes to 0 as t goes to ∞, we can pick some constant $q \in (0,1)$, $\exists c \in \mathbb{Z}_+$ such that $\forall m \geq c$, $\beta(e_k, m\delta) \leq q e_k$. Then, $d_{k+1} \leq q e_k + \gamma(r_u) + 2\underline{\gamma}^l_\epsilon$.

By definition, $e_{k+1} = d_{k+1} + 2\underline{\gamma}^l_m$. Thus,

$$e_{k+1} \leq q e_k + \gamma(r_u) + 2\underline{\gamma}^l_\epsilon + 2\underline{\gamma}^l_m. \tag{3.2}$$

Recursively applying (3.2) from $k = 1$ (with $e_1 = \underline{\gamma}^l_m$), we have:

$$e_{k+1} \leq \left(\gamma(r_u) + 2\underline{\gamma}^l_m + 2\underline{\gamma}^l_\epsilon\right) \sum_{i=0}^{k} q^i - q^k\left(\gamma(r_u) + 2\underline{\gamma}^l_\epsilon\right),$$

the right hand side of which contains a sum of the geometric series with sum equal to $\frac{1}{1-q}$, and a negative term that approaches to 0 as k approaches ∞. So we have that

$$\lim_{k \to \infty} e_k = \frac{\gamma(r_u) + 2\underline{\gamma}^l_m + 2\underline{\gamma}^l_\epsilon}{1-q}, \tag{3.3}$$

that serves as a constant bound for the errors $\{e_k\}_{k=1}^{\lfloor T/(m\delta) \rfloor}$. □

Next we consider the special case of the unforced version of dynamics (3.1) with $\vec{u}(t) = 0$, namely,

$$\dot{\vec{v}} = f_l(\vec{v}, 0), \tag{3.4}$$

where f_l is locally Lipschitz in $\vec{v} \in \mathbb{R}^n$. Then similar to Definition 8, we have:

Definition 9. *Dynamics* (3.4) *is incremental globally asymptotically stable (δ-GAS for short) or incremental stable [64, 65] if there exists a function $\beta \in \mathcal{KL}$ so that for any $t \geq 0$ and any $\vec{v}, \vec{v}' \in \mathbb{R}^n$ the following holds:*

$$\|\tau_{\vec{v},0}(t) - \tau_{\vec{v}',0}(t)\| \leq \beta(\|\vec{v} - \vec{v}'\|, t).$$

By comparing Definition 9 with Definition 8, and replacing $\gamma(r_u)$ with 0 all throughout the proof of Theorem 3, we can obtain the following corollary:

Corollary 4.1. *For unforced flow(l) that is δ-GAS, $\exists c \in \mathbb{Z}_+, \forall m \geq c$ such that the over-approximation error sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta) \rfloor}$ has a constant bound under Algorithm 4, given by:*

$$\lim_{k \to \infty} e_k = \frac{2(\underline{\gamma}_m^l + \underline{\gamma}_\epsilon^l)}{1 - q}. \tag{3.5}$$

Now consider the special case of a stable LTI system. For such a system $\dot{\vec{v}} = A\vec{v} + B\vec{u}$ with *Hurwitz matrix A*, we can easily identify the associated $\mathcal{KL}$ function β and $\mathcal{K}_\infty$ function γ by taking advantage of its explicit solution:

$$\vec{v}(t) = e^{At}\vec{v}(0) + \int_0^t e^{A(t-\xi)} B\vec{u}(\xi) d\xi,$$

and use the bound $\|e^{At}\| \leq ke^{-\lambda t}, \forall t \geq 0$ for some $k, \lambda > 0$. Therefore, for any $\vec{v}, \vec{v}' \in \mathbb{R}^n$ and any pair of bounded input signals $\vec{u}(t), \vec{u}'(t)$,

$$\begin{aligned}
\|\tau_{\vec{v},\vec{u}(t)}(t) - \tau_{\vec{v}',\vec{u}'(t)}(t)\| &= \|e^{At}(\vec{v} - \vec{v}')\| + \int_0^t e^{A(t-\xi)} \|B(\vec{u}(\xi) - \vec{u}'(\xi))\| d\xi \\
&\leq ke^{-\lambda t}\|\vec{v} - \vec{v}'\| + \int_0^t ke^{-\lambda(t-\xi)} \|B(\vec{u}(\xi) - \vec{u}'(\xi))\| d\xi \\
&\leq ke^{-\lambda t}\|\vec{v} - \vec{v}'\| + \frac{k\|B\|}{\lambda} \sup_{0 \leq \xi \leq t} \|\vec{u}(\xi) - \vec{u}'(\xi)\|.
\end{aligned}$$

This not only shows the δ-ISS property of the above system, but also an exponential decay of the deviation of the unforced trajectories. Consequently, we can have the following corollary:

Corollary 4.2. *For an LTI dynamics $\dot{\vec{v}} = A\vec{v} + B\vec{u}$ with A Hurwitz, $\exists c \in \mathbb{Z}_+$ such that $\forall m \geq c$, the over-approximation error sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ has a constant bound under Algorithm 4.*

Remark 3. *Once a constant bound on the over-approximation error sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ is guaranteed, then the number of simulation trajectories is bounded. So consider replacing input $\{\vec{v}_0\}$ with a non-singleton representative set of initial states in Algorithm 4. Then we will have $e_1 \leq qD(V_0) + \gamma(r_u) + 2\underline{\gamma}^l_\epsilon + 2\underline{\gamma}^l_m$. Using this as the initial value for the sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$, we can have*

$$e_k \leq \big(\gamma(r_u) + 2\underline{\gamma}^l_m + 2\underline{\gamma}^l_\epsilon\big)\sum_{i=0}^{k} q^i + q^k D(V_0). \tag{3.6}$$

On the right hand side of (3.6), the second term decays as long as $D(V_0)$ is bounded, regardless of its size. Thus, the sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ converges to the same value as in (3.3), regardless of the diameter of the bounded set V_0, as summarized in the following corollary.

Corollary 4.3. *For dynamics of δ-ISS or δ-GAS property, $\exists c \in \mathbb{Z}_+$ such that $\forall m \geq c$, the over-approximation error sequence $\{e_k\}_{k=1}^{\lfloor T/(m\delta)\rfloor}$ converges to the same value as in (3.3) under Algorithm 4 for any bounded initial set V_0.*

Remark 4. *In (3.5), $\underline{\gamma}^l_m$, $\underline{\gamma}^l_\epsilon$, and hence e_k, can be made arbitrarily small, by choosing sufficiently small partition and simulation parameters r_v, ϵ, and δ. While the constant error bound is achieved regardless of the choices of these parameters, improving those can help to get a smaller constant bound. On the other hand, while smaller m is, smaller $\underline{\gamma}^l_m$ and $\underline{\gamma}^l_\epsilon$ become, yet a small m may not guarantee the trajectory convergence. This is because for a small m, the condition $q < 1$ in (3.2) may not hold. In implementation, m is given a default value by rule-of-thumb if not user-specified.*

Remark 5. *In (3.3), a sufficiently small r_u also makes $\gamma(r_u)$ sufficiently small. In this chapter we focus on the partitions of the state set, we have fixed r_u as for our algorithms for simplicity. Theoretically, a bounded range of inputs can also be partitioned into smaller subranges, with the assumption of piecewise constant inputs remaining intact. Then for each input subrange, we can compute as in Algorithm 4, and perform union of the over-approximation state sets thus computed.*

3.4.2 Benefits from Dynamic Partitions

The main advantage of the dynamic partitioning scheme adopted by Algorithm 4 is that it keeps the error growth under control while using only a Lipschitz-based discrepancy function, that can be practically computed. (In contrast, there is no low complexity technique to obtain a more precise discrepancy function, but if it is available, our Algorithm 4 can certainly also use it.) Theorem 4 and its corollaries in section 3.4 show that Algorithm 4 achieves constant error bound in case of stable dynamics, without requiring the β and γ functions of Definitions 8 and 9. Thus Algorithm 4 provides an approach for simulation-based reachability analysis, even when the specialized discrepancy functions are not provided.

A consequence of dynamic partitioning is that, the number of simulation trajectories in Algorithm 4 is dynamic and decided in run-time, running only as many simulations as necessary, and this number can even reduce. This number is related to the size of the error, and if that decreases (as for example in case of a convergent system), then the number of simulation runs also decreases. Corollary 4.3 suggests that, the error sequence approaches a constant value implying that the number of simulation trajectories become constant, and in fact may be smaller than the initial number. Thus, the analysis can even speed up and eventually settle down.

3.5 The Overall Algorithm

Algorithm 4 is extended to obtain the overall Algorithm 5 for the reachability over-approximation of a hybrid automaton A in a bounded time T, with partition parameter r_v, simulation time-step δ, the repartition time-step period number m, piecewise bounded input signal $\vec{u}(t) \in B_{r_u}(\vec{u}_0(t))$ with its nominal piecewise constant input signal $\vec{u}_0(t)$, and simulation time-step δ. W.l.o.g., we assume the initial discrete location to be l_0 and the initial continuous states $V_0 \subseteq inv(l_0)$.

In line 1, R^l is the overall reachable set over-approximation within discrete mode l that is forward simulated as in Algorithm 3; L^{re} is the over-approximation of the reachable location set; C^l is the set of representative states in R^l. The loop over lines 2-28 expands Algorithm 4 to account for concurrent continuous evolution in multiple locations, as well as the state-triggered discrete

Algorithm 5: Multi-step computation of the bounded-time reachable location/state set over-approximation for general hybrid automaton.

```
Input: $A = (L, V, E, V_0, U, flow, inv, guard, reset)$, $l_0, r_v, r_u, \vec{u}_0(t), T, m, \delta$.
$\forall l \in L \setminus l_0, R^l \leftarrow \emptyset$; $R^{l_0} \leftarrow V_0$; $L^{re} \leftarrow \{l_0\}$;$\forall l \in L \setminus l_0, C^l \leftarrow \emptyset$; $C^{l_0} \leftarrow \text{Partition}(V_0, r_v)$;
for $i = 0; i < \lceil T/\delta \rceil; i++$ do
    foreach $l \in L^{re}$ do
        $R_{[i,i+1]}(C^l, \vec{u}_0(t)) = \bigcup_{\vec{v} \in C^l} R_{[i,i+1]}(\vec{v}, \vec{u}_0(t))$;
        $R^l \leftarrow R^l \cup \Big(\text{inv}(l) \cap R_{[i,i+1]}(C^l, \vec{u}_0(t))\Big)$;
    end
    foreach $\vec{v} \in C^l$ do
        if $R^l_i(\vec{v}, \vec{u}_0(t)) \cap inv(l) = \emptyset$ then
            $C^l \leftarrow C^l \setminus \{\vec{v}\}$;
        end
    end
    foreach $l' \in \{l^* \mid (l, l^*) \in E\}$ do
        $\text{Face}_{(l,l')} = \text{reset}_{(l,l')}(R_{[i,i+1]}(C^l, \vec{u}_0(t)) \cap \text{guard}((l, l')))$;
        $\text{Face}_{(l,l')} \leftarrow \text{Face}_{(l,l')} \setminus \bigcup_{\vec{v} \in C^{l'}} B_{r_v}(\vec{v})$;
        if $Face_{(l,l')} \neq \emptyset$ then
            $L^{re} \leftarrow L^{re} \cup \{l'\}$;
            $C^{l'}_{new} = \texttt{Partition}(\text{Face}_{(l,l')}, r_v)$;
            if $R_{[0,1]}(C^{l'}_{new}, \vec{u}_0(t)) \setminus inv(l') \neq \emptyset$ then
                return Zeno alarm;
            end
            $R^{l'} \leftarrow R^{l'} \cup R_{[0,1]}(C^{l'}_{new}, \vec{u}_0(t))$;
            $C^{l'} \leftarrow C^{l'} \cup C^{l'}_{new}$;
        end
    end
    if $(i > 0) \wedge (i\%m = 0)$ then
        $C^l \leftarrow \texttt{Partition}(\bigcup_{\vec{v} \in C^l} R^l_i(\vec{v}, \vec{u}_0(t)), r_v)$;
    end
end
Output: $L^{re}, \{R^l\}_{l \in L^{re}}$
```

jumps in the context of hybrid automaton. The inner loop lines 3-27 compute the single time-step continuous and discrete reachability over-approximation from each current reachable location. In line 4, $R_{[i,i+1]}(C^l, \vec{u}_0(t))$ denotes the reachable set over-approximation from the states covered by C^l and under input signal $\vec{u}(t) \in B_{r_u}(\vec{u}_0(t))$ defined over time interval $[i\delta, (i+1)\delta]$. It is the union of the tube segments propagating from C^l computed individually by Algorithm 3. Line 5 updates R^l, adding the newly reached set over-approximation within $inv(l)$. Lines 6-10 check each simulation starting point $\vec{v} \in C^l$, and remove it from C^l if its reachable set over-approximation at time-step i, denoted $R^l_i(\vec{v}, \vec{u}_0(t))$ (see Remark 6 below), has exited $inv(l)$.

Remark 6. *Suppose a representative state $\vec{v} \in C^l$ is created at time-step j, then at time-step $j \leq i \leq j+m$, we have $R^l_i(\vec{v}, \vec{u}_0(t)) = B_{\underline{\gamma}^l_{i-j}}(\vec{v}_{i-j})$. In Algorithm 4, all representative states in l are created via partitions. But in Algorithm 5, they can also be generated via discrete jumps from other locations as in lines 16-21. Thus the computation of $R^l_i(\vec{v}, \vec{u}_0(t))$ may vary accordingly.*

Lines 11-23 detect and handle all possible outgoing discrete jumps from l. For each possible discrete jump (l, l') as shown in Figure 3.4, line 12 defines its *entry face*, denoted by $Face_{(l,l')}$, as the set of states reached upon reset from the states that trigger $guard((l, l'))$.

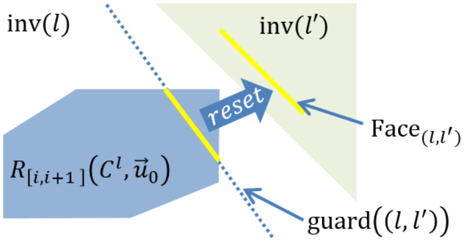


Figure 3.4: Discrete jump and value reset. Blue shadowed area is the union of all tube segments of $[i\delta, (i+1)\delta]$.

Line 13 refines $Face_{(l,l')}$ to remove the states already represented by $C^{l'}$. In lines 14-22, a one-step reachability computation within l' is performed starting from the nonempty $Face_{(l,l')}$, to over-approximate the post-jump continuous evolution in l' for the remainder sample time (so its sum with the pre-jump evolution time equals δ). Specifically, line 15 updates the reachable location set due to a jump; line 16 generates new representative set $C^{l'}_{new}$ by partitioning the reachability over-approximation in l'; lines 17-19 check if the one-step reachability over-approximation stays within $inv(l')$ and returns a "Zeno alarm" otherwise (see Remark 7); line 20 updates $R^{l'}$ with the one-step reachable set over-approximation and line 21 adds $C^{l'}_{new}$ into existing $C^{l'}$.

Remark 7. *The one-step reachable set over-approximation computed in line 16 may trigger a discrete jump from l' to some l'', which requires another one-step over-approximation in l'' , and so on, all in one single sample time-step. Under our assumption of minimum dwell-time and the choice of a sample time-step smaller than the dwell-time, we expect this situation to not arise. But as a precaution, in this case, we let the algorithm terminate and report a "Zeno alarm" message.*

Lines 23-25 periodically generate partitions, as in Algorithm 4, using r_v-hypercube cells to cover the over-approximation set at the current final time. Algorithm 5 returns the reachable location set over-approximation L^{re} and the reachable state set over-approximation R^l for each $l \in L^{re}$. Both L^{re} and $\{R^l\}_{l\in L^{re}}$ are updated at each time-step until time bound T is reached. We now establish the correctness of Algorithm 5 as follows.

Theorem 5. *Algorithm 5 is correct. That is, in the case of no Zeno alarm, if some $\vec{v}_* \in inv(l_*)$ is reachable from some $\vec{v}_0 \in V_0 \subseteq inv(l_0)$ under the input $\vec{u}_0(t) \in B_{r_u}(\vec{u}_0(t))$ over $[0, T]$, then $l_* \in L^{re}$ and $\vec{v}_* \in R^{l_*}$.*

Proof. If $l_* = l_0$ and $\vec{v}_*$ is reachable from $\vec{v}_0 \in V_0 \subseteq inv(l_0)$ without any discrete jump, then $l_0 \in L^{re}$ by default and it follows from Theorem 4 that $\vec{v}_* \in R^{l_*}$.

Otherwise, consider the execution α from $\vec{v}_0 \in V_0$ to $\vec{v}_* \in inv(l_*)$ within $[0, T_*]$. Then α has a finite length, and let $\alpha = \tau_0 e_1 \tau_1 \dots e_k \tau_k$, where $e_i = (l_{i-1}, l_i)$ is a discrete jump, and τ_i with local time variable $t_i \in [0, T_i]$ is a continuous trajectory in l_i. By Definition 7, we have $\tau_0(0) = \vec{v}_0$, $\tau_k(T_k) = \vec{v}_*$ and $l_k = l_*$.

Regarding the discrete jumps, by Definition 7, we have:

$$T_* = \Sigma_{j=0}^{k} T_j \leq T, \tag{3.7}$$

and, for $0 \leq i \leq k$,

$$\tau_{i+1}(0) = reset_{(l_i,l_{i+1})}(\tau_i(T_i)). \tag{3.8}$$

Let $R^{l_i}_{\leq j}$ denote the value of R^{l_i} after the j^{th} iteration of lines 2-27 in Algorithm 5. It's easy to see that:

$$\forall j \in [0, \lceil T/\delta \rceil - 1], R^{l_i}_{\leq j} \subseteq R^{l_i}_{\leq j+1}, \text{ and } \forall j \in [0, \lceil T/\delta \rceil], R^{l_i}_{j} \subseteq R^{l_i}_{\leq j}.$$

Next, we prove $\forall t_i \in [0, T_i]$, $\tau_i(t_i) \in R^{l_i}_{\leq \lceil T_*/\delta \rceil}$ for $0 \leq i \leq k$ by induction.

For base step $i = 0$, $\tau_0(0) \in R^{l_0}_{\leq 0}$ holds by the initial condition. From Theorem 3, we have $\forall t_0 \in [0, T_0]$, $\tau_0(t_0) \in R^{l_0}_{\leq \lceil T_0/\delta \rceil}$.

For the inductive step, assume $l_i \in L^{re}$ and $\forall t_i \in [0, T_i]$, $\tau_i(t_i) \in R^{l_i}_{\leq \lceil \Sigma^i_{j=0} T_j/\delta \rceil}$. Then $\tau_i(T_i) \in R^{l_i}_{\leq \lceil \Sigma^i_{j=0} T_j/\delta \rceil}$. It implies that the discrete transition (l^i, l^{i+1}) was executed at some time-step $s \leq \lceil \Sigma^i_{j=0} T_j/\delta \rceil$. Thus from (3.8) we have,

$$\tau_{i+1}(0) = reset_{(l_i, l_{i+1})}(\tau_i(T_i)) \in Face(l_i, l_{i+1}).$$

$Face(l_i, l_{i+1})$ is covered by either the existing or the new representative set at time-step s. Hence,

$$\forall t_{i+1} \in [0, \delta], \tau_{i+1}(t_{i+1}) \in R^{l_{i+1}}_{\leq s}, \tag{3.9}$$

owing to correctness of Algorithm 3 by Theorem 3. From (3.9) we know $\tau_{i+1}(\delta) \in R^{l_{i+1}}_{\leq s}$. Then from Theorem 3, we have:

$$\forall t_{i+1} \in [\delta, T_{i+1}], \tau_{i+1}(t_{i+1}) \in R^{l_{i+1}}_{\leq s + \lceil (T_{i+1} - \delta)/\delta \rceil}. \tag{3.10}$$

From $s \leq \lceil \Sigma^i_{j=0} T_j/\delta \rceil$, we have:

$$s + \lceil (T_{j+1} - \delta)/\delta \rceil \leq \lceil \Sigma^i_{j=0} T_j/\delta \rceil + \lceil T_{i+1}/\delta \rceil - 1 \leq \lceil \Sigma^{i+1}_{j=0} T_j/\delta \rceil. \tag{3.11}$$

Combining (3.9)-(3.11), we can conclude:

$$\forall t_{i+1} \in [0, T_{i+1}], \tau_{i+1}(t_{i+1}) \in R^{l_{i+1}}_{\leq \lceil \Sigma^{i+1}_{j=0} T_j/\delta \rceil}.$$

Together with the base step and inductive step, we have:

$$\vec{v}_* = \tau_k(T_k) \in R^{l_k}_{\leq \lceil \Sigma^k_{j=0} T_k/\delta \rceil} = R^{l_*}_{\leq \lceil T_*/\delta \rceil},$$

with $T_* = \Sigma^k_{j=0} T_j$ from (3.7), and $l_k = l_*$ by definition. Since $T_* \leq T$, we have $\vec{v}_* \in R^{l_*}_{\leq \lceil T/\delta \rceil} = R^{l_*}$.

Finally, in lines 15-16, Algorithm 5 adds any reachable location l to L^{re} the first time l is reached. Thus we can also conclude that by the time-step $i = \lceil T_*/\delta \rceil$, $l_* \in L^{re}$. □

Algorithm 5 introduces extra over-approximation error when handling discrete jumps. Specifically, the maximum error from partitioning the entry face after reset is r_v, and then there is the error from one-step over-approximation. Both can be made arbitrarily small by the choice of partition/simulation parameters. For safety verification against an unsafe zone S_{unsafe}, we can simply extend Algorithm 5 by inserting a line for checking the nonemptiness of $S_{\text{unsafe}} \cap \bigcup_{l \in L^{re}} R^l$ after line 27. Empty intersection indicates system safety. Otherwise, the system may still be safe, and to ascertain this, we may increase the granularity of partition by replacing r_v with $r_v/2$ and re-executing Algorithm 5. Like all other algorithms for hybrid system safety verification, the iteration doesn't guarantee termination due to the inherent undecidability of the problem.

Remark 8. *If the safety verification is the only goal, the partition function can be further enhanced for computational efficiency. An enhanced function Partition(S, r_v) returns the set of representative states that cover only the boundary of S (or its convex hull). This is because any trajectory that starts from the interior of S has to traverse through the boundary before it reaches any exterior portion. Boundary partition gives representative set of size $\mathcal{O}(D(S))$, compared to $\mathcal{O}((D(S))^2)$ for a bounded region S.*

3.6 Implementation and Experimental Results

3.6.1 Implementation and Architecture

We developed a prototype tool, *Hybrid System Step Simulation Verifier* ($\texttt{HS}^3\texttt{V}$), that implements our algorithms using C# for the bounded-time reachability over-approximation for general hybrid systems. Figure 3.5 shows its architecture. It contains seven main modules including a core step procedure engine of four modules. Program interacts with three open source license libraries to facilitate data processing. Modules and their functionalities are list as follows:

- **Model Parser:** This module accepts input files (in .txt or .xml format), parsing the syntactical lines in each input file, and passes the data containing the description information of the model and safety specification to the Verifier's step procedure engine.

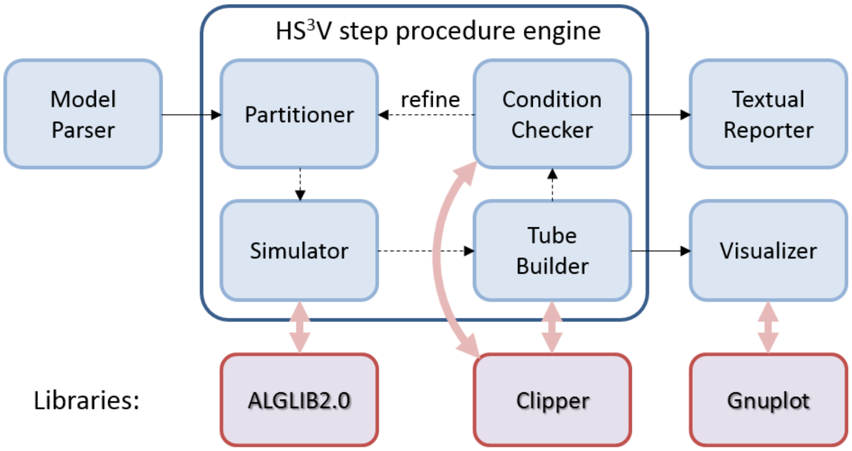


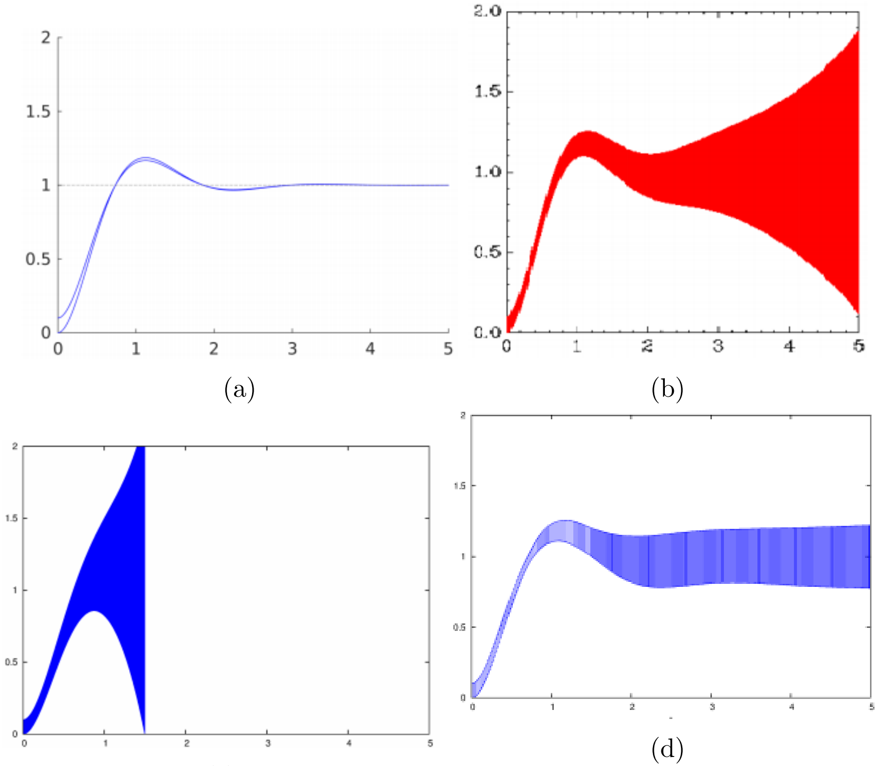
Figure 3.5: Architecture of HS^{3}V.

- **Partitioner:** This module generates the representative state set of a bounded region S using cell radius parameter γ. The state space is partitioned into uniform γ-hypercubes. Then the representative set, i.e. the center points of the γ-hypercubes overlapping S, is of size $\mathcal{O}((D(S))^2)$. Further improvement is done by selecting the center points of γ-hypercubes that cover only the boundary of S (or its convex hull) to get the representative set of size $\mathcal{O}(D(S))$.
- **Simulator:** This module generates simulation trajectories whose simulation values are computed using ALGLIB 2.0 [66].
- **Tube Builder:** This module builds tube segments around simulation values by Algorithm 3, using Clipper [67] for polygon operations.
- **Visualizer:** This module plots the reachable tubes (and other optional data) using Gnuplot [68].
- **Condition Checker:** This module checks the current reachable set over-approximation against guard conditions and unsafe set, also using Clipper. When the safety specification is violated, partition parameter is refined to restart the verification process.
- **Textual Reporter:** This module generates a textual report regarding the safety satisfaction after the step procedure has been repeated till the given time bound.

3.6.2 Experimental Results

Example 3. *(Double-Integrator System [32]). Consider a double-integrator, such as a point moving along a 1-d line, controlled through its acceleration. The dynamics are $\dot{x} = v$, $\dot{v} = a$, where the acceleration a is set periodically by a PD controller with gains $P = 10$ and $D = 3$. The controller_update function periodically assigns $a := P * (1 - x) + D * -v$. The period of the control task is $T = 0.005$ seconds. The system has a fixed point of $x = 1$.*

We study the system's position response over 5 seconds, with initial state $x \in [0, 0.1]$ and $v = 0$. Results of simulation and over-approximation using three existing tools are shown in Figure 3.6.



(a)

(b)

(c)

(d)

Figure 3.6: (a) Matlab Simulation output. (b) SpaceEx output. (c) Flow* output with original model. (d) Flow* output with Hyst translated model. *All plots in Figure 3.6 are taken from [32] without modification.*

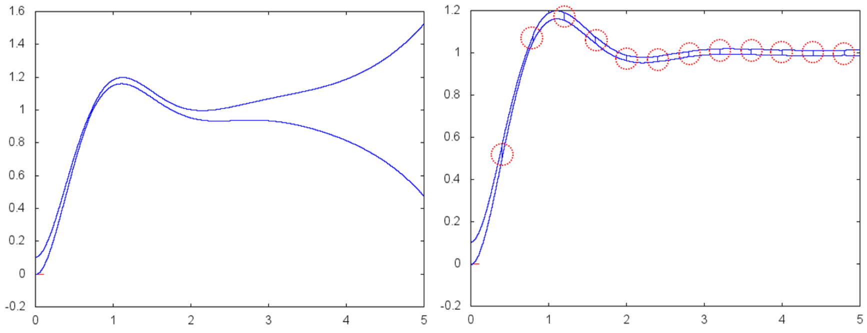
Although the simulation may suggest that x stabilizes at 1, both SpaceEx and Flow* give divergent reachability due to the errors introduced in frequent discrete transitions. SpaceEx uses a support function representation of the reachable set, which is not efficient in performing intersection and deciding containment – operations often used in determining if a guard is triggered or an safety condition is violated. Thus, `SpaceEx` has to introduce polyhedral representation into the verification algorithm, causing extra over-approximating translations at the times of the discrete transitions. `Flow*` uses Taylor Model representation and suffers from similar issues. `Flow*` gives a better result on `Hyst` [31, 32] translated model. `Hyst` over-approximates the original frequently switching continuously-controlled system with a continuously-controlled system with additional bounded non-deterministic input, resulting in the so-called *continuization*, which eliminates a large number of discrete transitions, thereby eliminating error growth caused by set transformations. Similar approach was also seen in [69].

For our tool, HS^{3}V, we set the simulation time-step $\delta = 0.0025s$, the state partition parameter $r_v = 0.003$. Let (x_0, v_0) be one representative point of the initial zone. The reference trajectory evolves according to

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a, \tag{3.12}$$

with $x(0) = x_0, v(0) = v_0$ and piecewise constant input signal $a(t) = (10 - 10x(i\delta) - 3v(i\delta))$ over $[2i\delta, 2(i+1)\delta)$ for $i \in \mathbb{Z}_{\geq 0}$. Tube segments for the first 0.005s (2 time-steps) can be built with the initial state set $B_{r_v}((x_0, v_0))$, in which $a(t)$ ranges over $B_{13r_v}(10 - 10x_0 - 3v_0)$. At $t = 0.005s$, forward simulation continues from a current representative value (x_2, v_2) with input range updated to $B_{13\underline{\gamma}_2}(10 - 10x_2 - 3v_2)$. In general, at the i^{th} control update, input value for simulation is updated to $10 - 10x_{2i} - 3v_{2i}$ and input range is updated to $B_{13\underline{\gamma}_{2i}}(10 - 10x_{2i} - 3v_{2i})$.

Figure 3.7 compares the reachability results for variable x before and after applying dynamic repartitioning. Figure 3.7(a) gives good result within 3s, after which the error starts to grow rapidly. This agrees with the exponential growth with Lipschitz constant 1 (see equation (3.12)). Figure 3.7(b) using dynamic repartitioning with parameter $m = 160$ ($\underline{\gamma}_m \approx 1.5\gamma_0$), shows much improved result converging to a constant deviation comparable to the simulation result.



(a) (b)

Figure 3.7: (a) HS^{3}V output with exponential error growth without dynamic repartitioning. (b) HS^{3}V output with dynamic repartitioning happening inside dashed circles.

Example 4. *(Brusselator system). It is a nonlinear model for a type of auto-catalytic reaction. The system's behavior is captured by the differential equations* $\dot{x} = 1+x^2y-2.5x$ *and* $\dot{y} = 1.5x-x^2y$.

Suppose $\mu = 1$ and initial set $x \in [1.25, 1.3] \wedge y \in [2.25, 2.3]$. We again use a simulation time-step of $0.01s$ and the state partition parameter γ of 0.001. The reachability results computed before (red) and after (blue) applying dynamic repartitioning are shown in Figure 3.8. The later outperforms the former over time that accuracy is maintained: $[0, 7]s$ compared to $[0, 1]s$.

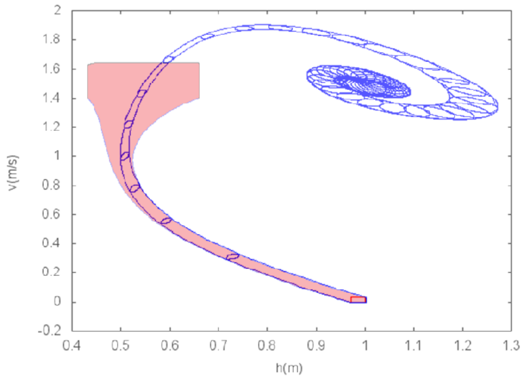


Figure 3.8: HS^{3}V output for Brusselator system.

Example 5. *(Bouncing ball) This is a classic example of a hybrid system. The continuous dynamics is given by* $\dot{v} = -g$ *and* $\dot{h} = v$ *where* g *is the acceleration due to gravity,* h *is the height of the ball and* v *is the velocity. The hybrid aspect of the model stems from modeling the collision of the ball with the ground as a partially elastic collision that causes energy loss. Accordingly, the* *bouncing ball displays a jump $v^+ = -cv^-$, where $c \in [0,1]$ is a constant, at the transition guard condition, $h = 0$.*

The bouncing ball may exhibit Zeno behavior since each time the ball bounces, it loses energy, making the subsequent jumps closer in time. From Theorem 5, the correctness of the analysis result can be guaranteed until a Zeno alarm is triggered. Let the constant of elastic collisions $c = 0.75$. Suppose the initial zone is $h \in [10, 10.1] \wedge v = 0$, the simulation time-step $\delta = 0.01s$, and the state partition parameter $\gamma = 0.003$. Figure 3.9 shows the reachability results, comparing before (red) and after (blue) applying dynamic repartitioning. The later maintains accuracy of over-approximation for over $[0, 20]s$. Both reachability over-approximation show correct and fully automatic discrete transitions with the guard and the reset corresponding to the collision event.

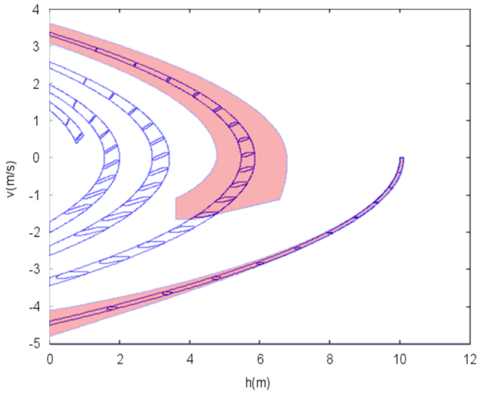


Figure 3.9: HS^{3}V output for bouncing ball system.

3.6.3 Performance

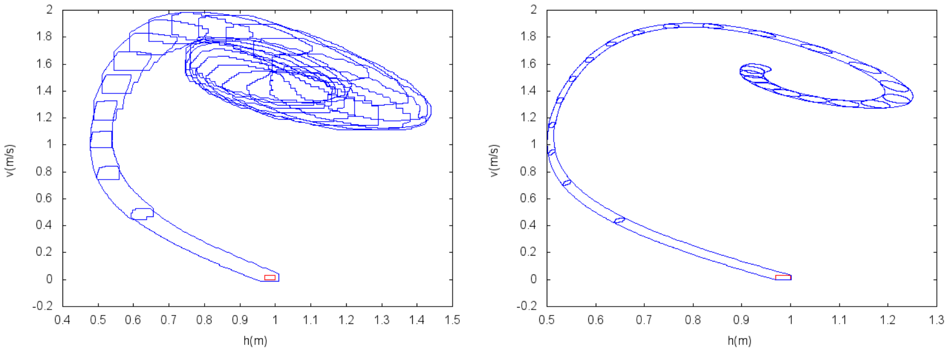
All the experiments in this section were performed on a computer with 8G memory and Intel Core @2.30GHz processor. Operating system used was Windows 7 x64. Table 3.1 gives performance of HS^{3}V on the benchmarks with dynamic repartitioning. Each benchmark is experimented with three parameter settings. For each benchmark, the setting used by the figure in the previous subsection is marked by *. Since the number of simulation time-steps is not fixed in our algorithms, we only present the maximum number of simulation branches during program execution for simplicity.

From the table, one can see that the execution time of the program is roughly propositional to the number of time-steps and the (maximum) number of simulation trajectories. Figure 3.10 shows the trade-off between the execution time versus the precision, via the Brusselator benchmark. By allowing finer granularity of partition, precision improves, while computation time goes up. The error bound of precision can be calculated using the bounds provided in the chapter.

Table 3.1: Table of Performance on benchmarks.

Set.	Benchmark	T(sec)	stepNo.	m(step)	simNo.	time(sec)
1	DIS	5	1000	300	22	4.93
2	DIS	5	1000	80	22	9.44
3	DIS*	5	2000	160	64	13.01
4	Bruss	10	333	55	63	5.19
5	Bruss	10	333	15	64	7.62
6	Bruss*	10	1000	40	96	23.37
7	VDP	10	333	55	246	10.37
8	VDP	10	333	15	295	25.84
9	VDP*	10	1000	40	278	29.41
10	B.Ball	10	333	55	191	8.25
11	B.Ball	10	333	15	226	18.62
12	B.Ball*	10	1000	40	238	38.25

Set.: experimental setting index,T: time bound, stepNo.: number of total time-steps, m: partition period, simNo: maximum simulation trajectories during program execution, time: program execution time.



(a)

(b)

Figure 3.10: (a) With Setting 5 in Table 3.1, HS^{3}V execution time is 7.62s. (b) With Setting 6 in Table 3.1, HS^{3}V execution time is 23.37s.

CHAPTER 4. “ReLIC: REDUCED LOGIC INFERENCE FOR COMPOSITION” FOR QUANTIFIER ELIMINATION BASED COMPOSITIONAL REASONING AND VERIFICATION

4.1 Introduction

From compositional perspective, distributed cyber-physical systems (CPSs) integrate physical dynamical components with computing hardware and software, interconnected over an embedded communication network. For example, Unmanned Systems Autonomy Services (UxAS) software system [70] consists of a collection of modular services that interact via a common message passing architecture. It provides a framework to construct and deploy software services that are used to enable autonomous capabilities by flexibly implementing autonomy algorithms on-board Unmanned Aerial Vehicles (UAV)s [71]. Successful missions engaging autonomous actions of the UAVs require highly dependable design. Within the model-based verification paradigm, compositional reasoning is employed for scalability, by utilizing composition of component properties to establish the properties of a system made out of those components. The component-based compositional design paradigm emphasizes the separation of concerns with respect to the system design through a modular and reuse based paradigm for defining, implementing, and composing components into systems. In this chapter, we establish that “quantifier elimination” provides a foundation for compositional reasoning, and also can be used to aid formal verification and model-checking.

Quantifier elimination (QE) is a powerful technique for gaining insight, through simplification, into problems involving logic expressions in various theories. QE is essentially a projection problem where a formula is projected to a lower dimension over only its free variables. For example, over the field of reals, $\exists x(y > x^2)$ is equivalent to $y > 0$ (since x^2 can only be non-negative), in which the quantified variable x has been eliminated or projected out. A theory admits quantifier elimination if for every formula in this theory, there is an equivalent quantifier-free formula. It has been proved that the real closed field and an extension of Presburger arithmetic (a linear theory of integers) both admit QE [72, 73]. This makes the QE applicable to many real world applications.

While Tarski [72] showed that the first-order logic over the real closed field admits quantifier elimination, it was Collins who in 1975, introduced the first implementable quantifier elimination procedure, based on cylindrical algebraic decomposition (CAD) [74]. Over the past few decades, the QE techniques and tools have undergone further enrichment, and the efforts made along the way have contributed to newer additions. In particular, specialized procedures for restricted problem classes led to newer, more advanced QE procedures, documented in tools such as `Mathematica`, `Redlog`, and `Qecad`. For example, `Redlog` [4], implements *virtual substitution* [75, 76] and *partial CAD* [77] algorithms, that work for formulae where the degrees of the quantified variables are small.

Many problems in systems and controls can be formulated as formulae in the first-order logic of real closed field. QE was applied in [78] to solve nonlinear continuous control system design with simple properties. In [79, 80], QE was used to compute exact reachable sets for linear systems with certain eigen-structures and semi-algebraic initial sets, and this method was generalized in [81] to handle linear systems with almost arbitrary eigen-structures. Further, [82, 83] extended the application of real QE to formal verification and synthesis of continuous and switched dynamical systems. QE solvers are also used as back-end reasoning engines in the bounded model-checking based algorithm in [84] and in the theorem prover for hybrid systems `KeYmaera` [85].

In this chapter, we establish that QE can provide a foundation for compositional reasoning, that are techniques being developed to cope with state-space explosion in concurrent systems [86, 87, 88, 89]. Essentially the strategy of divide-and-conquer is being employed where one first establishes the properties of the system components, and then uses those to establish the global properties of a complex system. Initially, during component development phase, each component is annotated with an assume-guarantee style contract. Supposing a system is composed of N components, the contract formula of the i^{th} component can be expressed as $A_i \Rightarrow G_i$ where A_i (the "assumption"), G_i (it's "guarantee") are both expressed by formulae over the set of component variables. Then the set of all the system behaviors is constrained by the conjunction of all the components' contracts $\bigwedge_{i=1}^{N}(A_i \Rightarrow$

G_i). Under these contracts, we show that *the strongest system property*, that can be claimed that the system satisfies, can be obtained by existentially quantifying the system's internal variables in the conjunct of $\bigwedge_{i=1}^{N}(A_i \Rightarrow G_i)$ and the constraints resulting from the connectivity relation among the components. Thus we establish that QE serves as a foundation for property/contract composition. Now to check whether a system satisfies a postulated property, we only need to check if the postulated property is implied by the aforementioned strongest system property. This in itself can be cast as a QE problem.

Another important contribution of our work is the extension of QE-based property composition to the case of time-dependent properties, which can depend on a (finite) history of input/output variables. We show that the composed property may involve a longer history, but no more than the cumulative histories of all its components. Accordingly, we introduce the notion of property order, deduction of system order, and the composition of given properties along with their time-shifted replicas to infer the strongest system property. We have implemented our QE-based compositional verification approach in a prototype tool, `ReLIC` (Reduced Logic Inference for Composition), based on the integration of `Redlog` with `AGREE` [2, 86, 90]—the former supports QE, while the latter is a compositional analyzer for a system and its components described in the modeling framework of AADL [1]. Our integration uses only the front-end of `AGREE` for specifying system architecture/connectivity, components, and their properties in AADL and `AGREE` annex, and reporting the result of composition to the user.

In addition to using QE for compositional reasoning, we also show that the problem of satisfiability checking used in formal verification and model-checking can be reduced to one of QE. Specifically, we consider the verification scheme based on k-induction [3, 91, 92, 93], implemented for example as `JKind` [5], that is used to verify invariant properties of programs written in the language Lustre [94]. Under this scheme, to prove a transition system satisfies some invariant property, one needs to prove the base case and the inductive case for some k. For each step, the verification of the base (or inductive) case can be reduced to an instance of an SMT (Satisfiability Modulo Theory) problem, namely checking the satisfiability of a first-order logic formula. Thus SMT-solving is integral to k-induction based verification. QE can offer alternatives for SMT-solving since checking the satisfiability of a formula $\phi(x_1, \ldots, x_n)$ in n-variables is equivalent to checking that the existentially quantified formula $\exists x_1 \ldots \exists x_n \phi(x_1, \ldots, x_n)$ evaluates to $true$ or $false$. Thus with regards to satisfiability, the capability of SMT solvers and QE solvers overlap, and can vary depending on the algorithms they employ and the theories they support. The accumulated experimental data provided in [95] has shown that Redlog along with other QE tools can offer advantage over SMT solvers like Z3, iSAT, cvc3, specially for non-linear arithmetic, in terms of the execution time, and the range of problems those can solve. To provide alternative options of back-end solvers to model-checkers, we have implemented the integration of Redlog with JKind, so SMT-solving can also be performed based on quantifier-elimination. This thereby enhances JKind's ability of checking properties that may involve nonlinearity. A related application is the generation of property-directed invariants by using QE in a k-induction-based framework [96].

In summary, the key contributions of the presented work are:

- Establish quantifier elimination as a foundation for property composition.
- Introduce the notion of strongest system property that can be inferred from the given component properties and their connectivity relation, and provide a QE-based derivation approach.
- Extend the QE-based property composition formalism to time-dependent properties involving temporal behavior.
- Implementation of the above in a new tool called ReLIC, that integrates a quantifier elimination tool Redlog with another tool AGREE that supports AADL specification of system architecture and component properties.
- Establish QE has an alternative choice for SMT-solving to be used by model-checkers.
- Implement within ReLIC the integration of QE solver Redlog with the model-checker JKind.
- Demonstrate the working of our implementations on simple illustrative examples.

The rest of the chapter is organized as follows. Section 4.2 describes the integration of Redlog with JKind to provide the model-checker with an additional solver option. Section 4.3 describes our framework for QE-based property composition, along with the developed prototype tool ReLIC for time-independent/temporal property composition. Section 4.4 provides the extension of QE-based property composition to compose time-dependent properties. Each of these sections provides illustrative examples of the said implementations.

4.2 QE support for Verification: Integration of `Redlog` with `JKind`

4.2.1 Preliminary

One approach for formal verification is bounded or k-induction model-checking. In this section we demonstrate how QE can be used to aid k-induction based verification. The tool JKind [5] supports k-induction proofs for transition systems described in Lustre [90]. It is based on a precursor tool `Kind` [92], to make it platform independent and easily integratable into Java-based tools.

Before describing the tools, we briefly recall the formulation of k-induction [3, 92]. Consider a transition system S specified in some logic $\mathcal{L}$, by an initial state condition $I(x)$ and transition relation $T(x, x')$, where x, x' are state variable vectors. Let $x(i)$ denote the variable at the i^{th} time step. Then a state property $\phi(x)$ is *invariant* for S, i.e., satisfied by every reachable state of S, if the following base and inductive conditions hold in $\mathcal{L}$ for some $k \in \mathbb{Z}_{\geq 1}$ and for any $n \in \mathbb{Z}_{\geq 0}$:

- $I(x(0)) \wedge \bigwedge_{i=0}^{k-2} T\big(x(i), x(i+1)\big) \Rightarrow \bigwedge_{i=0}^{k-1} \phi(x(i));$
- $\bigwedge_{i=n}^{n+k-1} T\big(x(i), x(i+1)\big) \wedge \bigwedge_{i=n}^{n+k-1} \phi(x(i)) \Rightarrow \phi(x(n+k)).$

The first condition checks the base case that $\phi(x)$ is satisfied at each step from an initial state of S for k steps. Any violation of this condition yields a concrete counterexample that falsifies the property ϕ. The second condition describes the inductive case, which checks that if ϕ holds at each state along a k-step trace, then ϕ also holds at the state reached in $k+1$ steps. A counterexample trace for the inductive step does not necessarily yield a concrete counterexample because it may start from an unreachable state of S. On the other hand, in order to prove the invariance of ϕ over all states, the base and inductive cases must be *true* for some k. Hence a normal way of a k-induction proof starts with $k = 0$, and increments k as necessary to rule out any spurious counterexamples generated in an induction case. Induction based verification approach doesn't guarantee termination for the general case (the problem in general is undecidable), so an a priori bound on k is specified to prevent entering into an indefinite loop.

The model-checker JKind supports bounded model-checking using k-induction proofs on transition systems described in Lustre [94]. Figure 4.1 illustrates the general architecture of JKind. It takes, as input, a Lustre based description, containing system model and properties to be checked, and spawns three processes, respectively for base case, inductive case, and invariant generation case. The invariant generation case tries to prove some candidate invariants from pre-defined templates, that could be used to facilitate the base or induction case proofs by strengthening their hypotheses. Each process interacts with its own copy of an SMT solver at the back-end. All three processes are coordinated under a director, exchanging messages asynchronously. The SMT results are interpreted to produce the verification output.

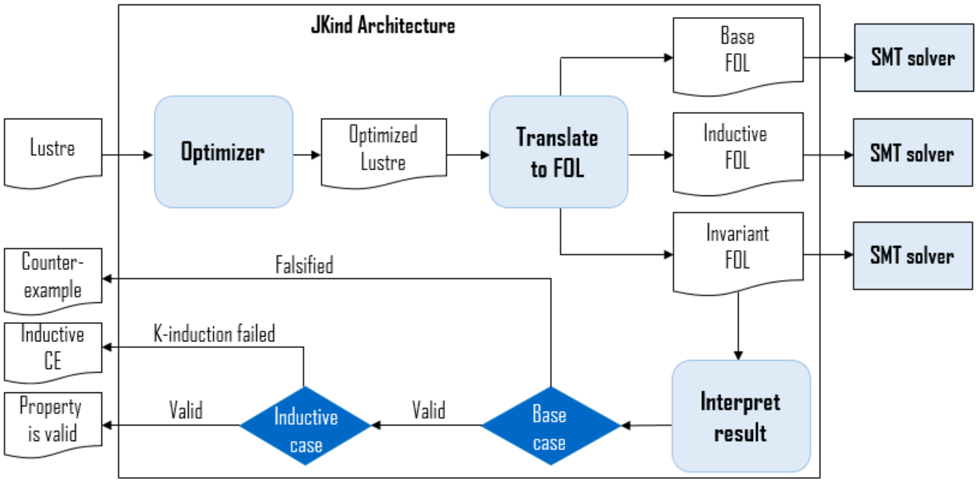


Figure 4.1: JKind architecture.

4.2.2 Reduction of SMT instance to QE instance

An SMT instance is a formula in a first-order logic, and the problem is to check whether such a formula is satisfiable. A formula $\phi(x_1, \ldots x_n)$, with $x_1 \ldots x_m$ as un-quantified free variables, is satisfiable if and only if there *exists* an assignment of the free variables that makes the formula evaluate to *true*. The same can be expressed as a QE instance, $\exists x_1 \ldots \exists x_m \phi(x_1, \ldots x_n)$, and now since all the variables in $\phi(x_1, \ldots x_n)$ are quantified, the equivalent quantifier-free formula that a QE process returns is either *true* or *false*. In the case of former, a satisfiable assignment of $(x_1 \ldots x_n)$ is also returned.

State-of-art SMT solvers such as Z3 that are commonly used in the research community, support linear/non-linear arithmetic in mixed integer/real domain and various data structures such as lists, arrays, bit vectors etc. Redlog, On the other hand, each first-order formula in Redlog must exclusively contain atoms from one particular Redlog-supported domain, which determines the choice of admissible functions and relations with specified semantics. Redlog-supported domains include non-linear real arithmetic (Tarski Algebra), Presburger arithmetic, parametric quantified Boolean formulae, and others.

We have implemented (i) a translator from SMT-Lib 2.0 input format to Redlog input format, named S2RTool, using the front-end parser generated by Antlr v4 [97], and (ii) a back-end interpreter written in Java. Also, we have reprogrammed the JKind director in Figure 4.1 to redirect its output .smt2 files meant for SMT to S2RTool, along each of its three processes, as shown in Figure 4.2. The dotted arrows denote the redirected data flow after integration of JKind with Redlog, whereas the solid arrows denote the original flow. The S2RTool's translated output is received by Redlog for QE-based SMT-solving; its results are processed by our interpreter before forwarding those to JKind for a possible next round of induction iteration. A simple comparison example of Z3 vs. Redlog inputs is shown in Figure 4.3.

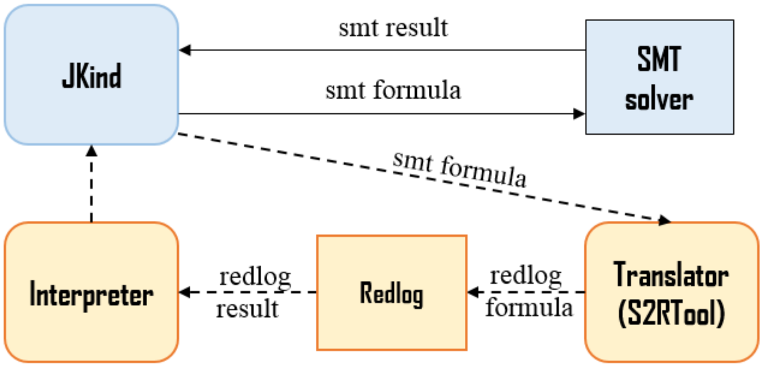
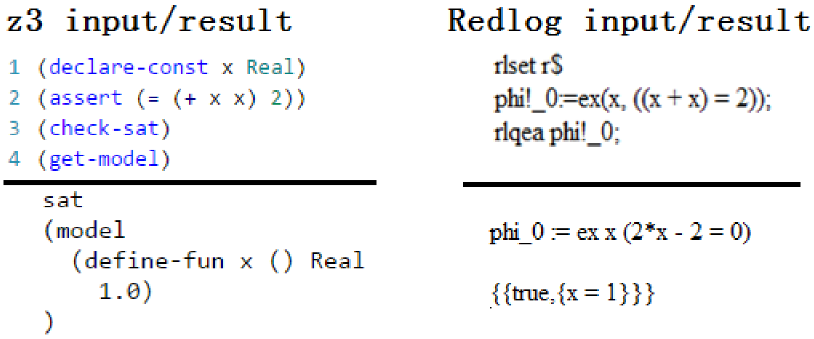


Figure 4.2: Data redirection in Redlog-integrated JKind.



z3 input/result

```
(declare-const x Real)
(assert (= (+ x x) 2))
(check-sat)
(get-model)
```

```
sat
(model
  (define-fun x () Real
    1.0)
)
```

Redlog input/result

```
rlset r$
phi!_0:=ex(x, ((x + x) = 2));
rlqea phi!_0;
```

```
phi_0 := ex x (2*x - 2 = 0)

{{true,{x = 1}}}
```

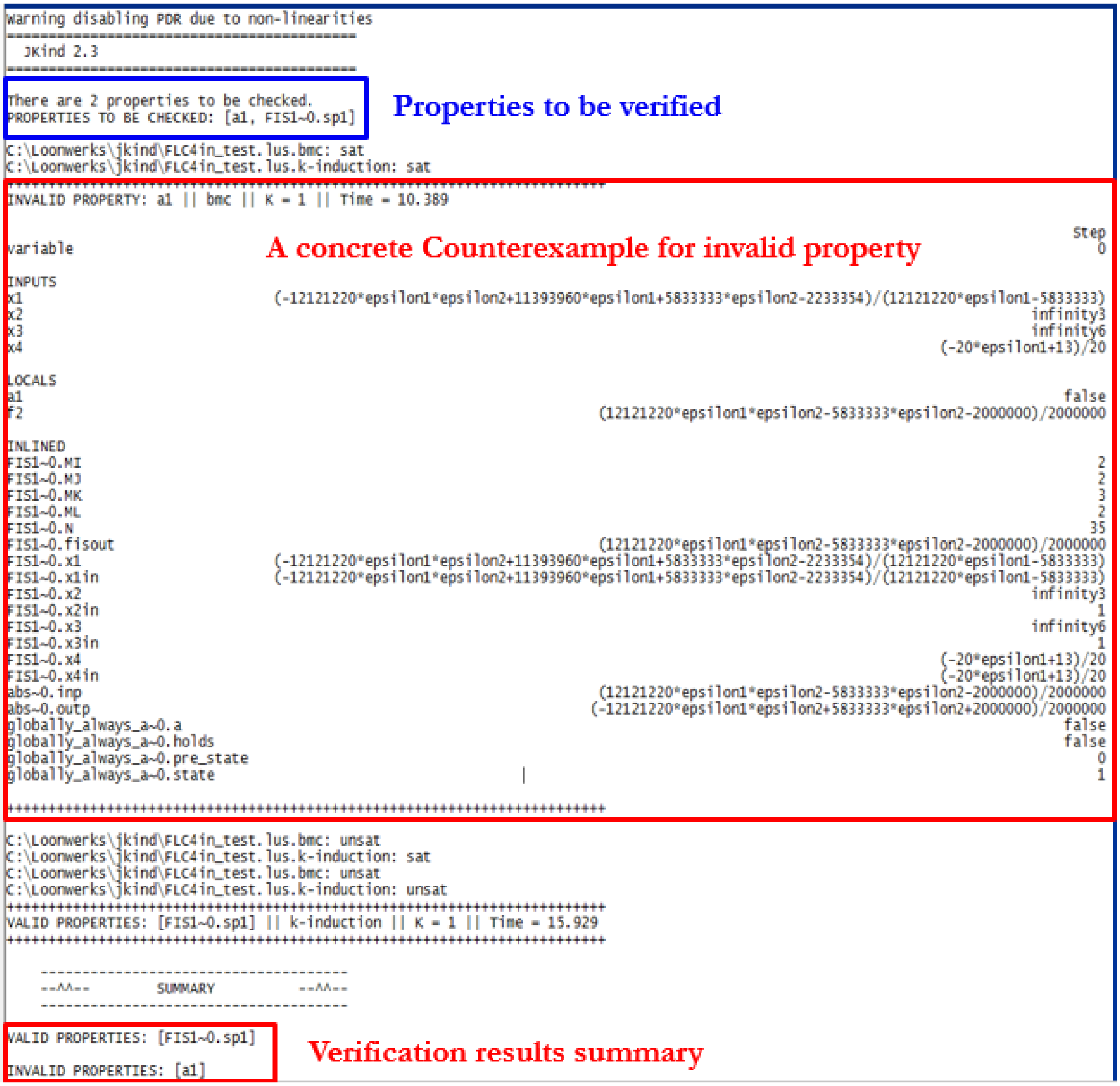
Figure 4.3: `Z3` vs. `Redlog`.

Remark 9. *Note while* `Redlog` *does not directly support mixed real-integer variables, the case of mixed real/integer variables can still be handled as follows. If the integer variables are in bounded ranges, then, in* `Redlog`*, the problem can be mapped to real domain as follows: for each integer variable x, we append the formula $\bigvee_{i=1}^{n}(x = v_i)$, where $\{v_1, \ldots, v_n\}$ is the set of possible integer values of x. For the case of general mixed real/integer variables, where an integer variable may not be finitely enumerated, we solve an over-approximation problem with all the variables interpreted in the real domain. In this case, a "false" result of the over-approximation implies a "false" result of the original problem. On the other hand, in a "true" case, if the assignment of x happens to be an integer, this particular assignment is accepted for the original problem. Otherwise if the assignment of x is not an integer, we simply report "unknown".*

4.2.3 Experimental result

We have successfully employed the `Redlog`-integrated `JKind` on many Lustre programs including a fuzzy logic model involving non-linear computations. The fuzzy logic model contains 54 different modes, and where a main program selects the correct mode depending on the conditions satisfied by 4 real-valued inputs. Each mode invokes a call to a corresponding sub-program that computes a specified 4^{th}-order non-linear polynomial to arrive at the result for the only output variable. The input/state/output variables are defined in real domain except the mode selection variable N is an integer that varies from 1 to 54. For example, if the input satisfies the condition for the mode $N = 1$, the following 4^{th}-order non-linear polynomial computes the output: $(-2.22222) * x1in +$

$$(-2.00000)*x2in+(-4.00000)*x3in+(10.00000)*x4in+(8.88889)*x1in*x2in+(7.40741)*x1in*x3in+(59.25926)*x1in*x4in+(12.00000)*x2in*x3in+(32.00000)*x2in*x4in+(40.00000)*x3in*x4in+(-59.25926)*x1in*x2in*x3in+(-177.77778)*x1in*x2in*x4in+(-74.07407)*x1in*x3in*x4in+(-240.00000)*x2in*x3in*x4in+(888.88889)*x1in*x2in*x3in*x4in+(1.00000).$$



```
warning disabling PDR due to non-linearities
==========================================
  JKind 2.3
==========================================

There are 2 properties to be checked.
PROPERTIES TO BE CHECKED: [a1, FIS1~0.sp1]
C:\Loonwerks\jkind\FLC4in_test.lus.bmc: sat
C:\Loonwerks\jkind\FLC4in_test.lus.k-induction: sat
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
INVALID PROPERTY: a1 || bmc || K = 1 || Time = 10.389

                                                                                                                                Step
variable                                                                                                                           0

INPUTS
x1                              (-12121220*epsilon1*epsilon2+11393960*epsilon1+5833333*epsilon2-2233354)/(12121220*epsilon1-5833333)
x2                                                                                                                         infinity3
x3                                                                                                                         infinity6
x4                                                                                                              (-20*epsilon1+13)/20

LOCALS
a1                                                                                                                             false
f2                                                                     (12121220*epsilon1*epsilon2-5833333*epsilon2-2000000)/2000000

INLINED
FIS1~0.MI                                                                                                                          2
FIS1~0.MJ                                                                                                                          2
FIS1~0.MK                                                                                                                          3
FIS1~0.ML                                                                                                                          2
FIS1~0.N                                                                                                                          35
FIS1~0.fisout                                                          (12121220*epsilon1*epsilon2-5833333*epsilon2-2000000)/2000000
FIS1~0.x1                       (-12121220*epsilon1*epsilon2+11393960*epsilon1+5833333*epsilon2-2233354)/(12121220*epsilon1-5833333)
FIS1~0.x1in                     (-12121220*epsilon1*epsilon2+11393960*epsilon1+5833333*epsilon2-2233354)/(12121220*epsilon1-5833333)
FIS1~0.x2                                                                                                                  infinity3
FIS1~0.x2in                                                                                                                        1
FIS1~0.x3                                                                                                                  infinity6
FIS1~0.x3in                                                                                                                        1
FIS1~0.x4                                                                                                       (-20*epsilon1+13)/20
FIS1~0.x4in                                                                                                     (-20*epsilon1+13)/20
abs~0.inp                                                              (12121220*epsilon1*epsilon2-5833333*epsilon2-2000000)/2000000
abs~0.outp                                                            (-12121220*epsilon1*epsilon2+5833333*epsilon2+2000000)/2000000
globally_always_a~0.a                                                                                                          false
globally_always_a~0.holds                                                                                                      false
globally_always_a~0.pre_state                                                                                                      0
globally_always_a~0.state                                                                                                          1

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
C:\Loonwerks\jkind\FLC4in_test.lus.bmc: unsat
C:\Loonwerks\jkind\FLC4in_test.lus.k-induction: sat
C:\Loonwerks\jkind\FLC4in_test.lus.bmc: unsat
C:\Loonwerks\jkind\FLC4in_test.lus.k-induction: unsat
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
VALID PROPERTIES: [FIS1~0.sp1] || k-induction || K = 1 || Time = 15.929
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


    --------------------------------------
    --^^--          SUMMARY          --^^--
    --------------------------------------

VALID PROPERTIES: [FIS1~0.sp1]

INVALID PROPERTIES: [a1]
```

Properties to be verified

A concrete Counterexample for invalid property

Verification results summary

Figure 4.4: Verification of a fuzzy logic model using QE-integrated `JKind`.

The Lustre program of the fuzzy logic model contains two properties to be verified, including checking whether the fuzzy logic output remains bounded by 1 in magnitude. SMT-based `JKind` is unable to resolve this problem since `Z3` and other alternative SMT solvers fail to terminate. In contrast, our QE-integrated `JKind` proves one property as valid and the second property as invalid, reporting a concrete counterexample in less then 16 seconds on a standard laptop, during which each query to `Redlog` spends less than 1 second. The log of QE-integrated `JKind` execution is shown in Figure 4.4, where a concrete counterexample found to violate the boundedness property of the output stated above is reported.

4.3 `ReLIC` for time-independent property composition

A modular approach to establishing system correctness involves the so-called, assume-guarantee compositional paradigm [86, 87, 88, 89] within which, a module (component/system) contract is specified by a pair (A, G), where A and G are first-order logic formulae: G describes the guaranteed behavior of the module while A describes the assumed behavior of the environment with which the module interacts. The contract itself expresses the requirement or property $(A \Rightarrow G)$. One aspect of compositional verification aims to derive the system contract from the contracts of its components together with their interactions through shared variables or event-synchronizations, that does not reference any of the component models.

Example 6. *For illustration, consider a simple case where two components connect in a cascade composition, where their input/output properties are described by the formulae* $\big((u_1 \geq 0) \Rightarrow (y_1 \geq 0)\big)$ *and* $\big((u_2 \geq 0) \Rightarrow (y_2 \geq 0)\big)$ *respectively, where* u *and* y *denote input and output variables. Also, owing to the cascade configuration,* $u_2 = y_1$. *Then it is easy to see that the cascaded system satisfies the property* $\big((u_1 \geq 0) \Rightarrow (y_2 \geq 0)\big)$.

Note this derivation, does not require the internal details of the components. [90] provides a compositional reasoning approach that for a system with N components requires $N+1$ verification steps to establish or refute a postulated system contract from the given component contracts: one verification step for each component and one for the system as a whole. The component verification steps establish that the assumptions of each component are implied by the system-level assumptions and the guarantees of all the upstream components. The system-level verification step checks that the system guarantees follow from the system assumptions and the guarantees of the components. A tool that supports compositional reasoning for components, their contracts, and system architecture described in AADL [1] is AGREE. In this tool, while the architecture is described in AADL, the properties in assume-guarantee style are specified within the AGREE annex. AGREE uses JKind as its back-end model-checker for checking the above $N+1$ conditions, and it itself exists as a plug-in tool within the open-source Eclipse-based platform OSATE2 [98] that supports AADL v2. The architecture of AGREE within OSATE2 is shown in Figure 4.5.

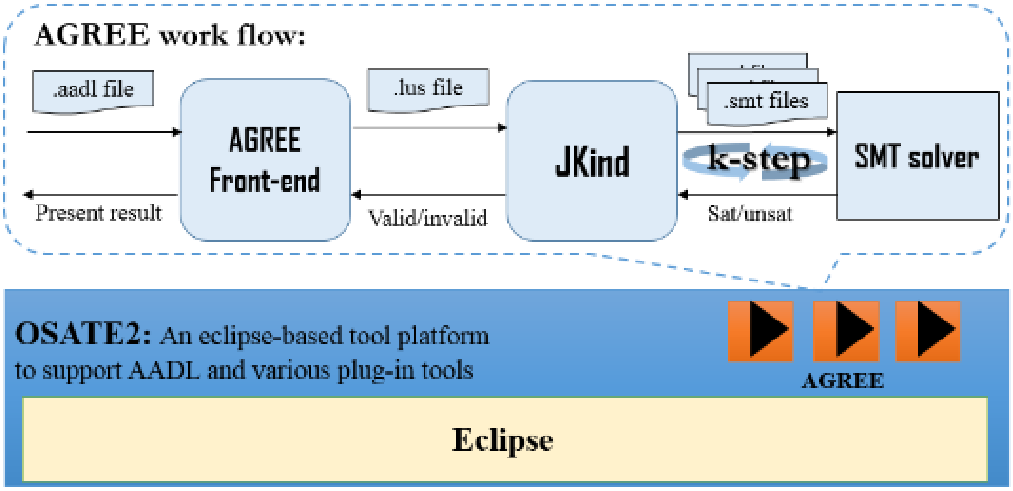


Figure 4.5: Architecture of AGREE plug-in within OSATE2 platform.

4.3.1 Proposed QE-based compositional verification

Our QE-based compositional reasoning approach is based upon the “strongest system property”, derived from the given component-level properties. To introduce this notion, we first introduce some notation. Consider a system S composed of N components. Let $X := \{x_1, \ldots, x_n\}$ be the set of all the variables in S, $X_{\text{int}} := \{x_1, \ldots, x_m\} \subseteq X (m \leq n)$, be the set of internal variables (namely, the internal variables of components themselves along with the component inputs/outputs internalized within the system), $X_{\text{sys}} := X \setminus X_{\text{int}} = \{x_{m+1}, \ldots, x_n\}$ be the set of external variables (namely, the inputs and outputs of S), and $C := \{(x_p, x_q) \mid x_p$ and x_q are variables of connected ports in $S\}$ be the set of connectivity relation among component variables. Suppose the i^{th} component's property is described by a contract (A_i, G_i) in a certain first-order logic. We next define the *strongest system property* and present a result that provides a method to derive it.

Definition 10. *The strongest system property is the system property that implies any other system properties established upon the given component properties and their connectivity relation.*

Theorem 6. *The strongest system property, established upon the component contracts and connectivity relation of system S described above is given by,*

$$\exists x_1 \ldots \exists x_m \Big(\bigwedge_{i=1}^{N} (A_i \Rightarrow G_i) \wedge \bigwedge_{(x_p,x_q)\in C} (x_p = x_q) \Big). \tag{4.1}$$

Proof. We prove this theorem by proving that indeed (4.1) is a system property, and that it implies any other system property.

First, let V denote the valid signal values over X, and V_{sys} denote the valid signal values over X_{sys}. By definition, V_{sys} is the projection of V from n to $n-m$ dimensions. For any system-level signal value $\vec{v}_{\text{sys}} = (v_{m+1} \ldots v_n)^{\mathsf{T}} \in V_{\text{sys}}$, where $(\cdot)^{\mathsf{T}}$ denotes the transpose operation, it must correspond to a component-level signal value $\vec{v} = (v_1 \ldots v_n)^{\mathsf{T}} \in V$. Meanwhile, $\vec{v} \in V$ if and only if:

$$\Big(\bigwedge_{i=1}^{N} (A_i \Rightarrow G_i) \wedge \bigwedge_{(x_p,x_q)\in C} (x_p = x_q) \Big)\Big|_{\vec{x}=\vec{v}} \equiv \mathit{true}, \tag{4.2}$$

where (4.2) simply states that a valid signal value must comply with the components contracts and the connectivity relation. Therefore when $\vec{x}_{\text{sys}} = \vec{v}_{\text{sys}}$, we have $\vec{v}_{\text{int}} = (v_1 \ldots v_m)^{\mathsf{T}}$ that makes the following formula *true*:

$$\Big(\bigwedge_{i=1}^{N} (A_i \Rightarrow G_i) \wedge \bigwedge_{(x_p,x_q)\in C} (x_p = x_q) \Big)\Big|_{\vec{x}_{\text{sys}}=\vec{v}_{\text{sys}}}.$$

As a result, we have that any valid signal value, $\vec{v}_{\text{sys}}$ satisfies:

$$\exists x_1 \ldots \exists x_m \Big(\bigwedge_{i=1}^{N} (A_i \Rightarrow G_i) \wedge \bigwedge_{(x_p,x_q)\in C} (x_p = x_q) \Big). \tag{4.3}$$

(4.3) suggests that any system-level signal value satisfies (4.1), in another word, (4.1) is indeed a system property.

Secondly, to show that (4.1) is also the strongest system property, assume ϕ is some given system property of S. For any $\vec{v}_{\text{sys}}$ that does not satisfies ϕ, $\vec{v}_{\text{sys}} \notin V_{\text{sys}}$ by definition. We need to show that $\vec{v}_{\text{sys}} \not\models$ (4.1). If this is $false$, then we can find $v_1, \ldots, v_m$ that together with $\vec{v}_{\text{sys}}$ forms a valid signal value in V. Since V_{sys} is the projection on $x_{m+1}, \ldots, x_n$, we can conclude that $\vec{v}_{\text{sys}} \in V_{\text{sys}}$, thereby arriving at a contradiction. In summary, any signal value that violates ϕ, also violates (4.1), alternatively speaking, (4.1) $\Rightarrow \phi$, as desired. □

Remark 10. *Through (4.1) in Theorem 6, we have shown that property composition, in a component-based compositional framework, is essentially a QE problem. Based on this insight, we have put forth a two-step QE-based compositional verification procedure. The first step is to generate the strongest system property, through a QE process of (4.1), applied to component contracts and connectivity relation. The strongest system property upon QE, denoted ϕ_{sys}, contains only the system-level input/output variables. The second step is to check if ϕ_{sys} implies any postulated system property ϕ_{postl} that also contains only system-level input/output variables. Note we can employ yet another QE process $\forall x_{m+1} \ldots \forall x_n(\phi_{sys} \Rightarrow \phi_{postl})$ to reduce the checking the implication $\phi_{sys} \Rightarrow \phi_{postl}$ to "true" or "false".*

4.3.2 Implementation and experimental result

We have implemented a prototype tool `ReLIC` (Reduced Logic Inference for Composition) employing the above strategy of Remark 10, integrating `AGREE` (for system and component specification in AADL) and `Redlog` (for QE). Within `ReLIC`, the AADL architecture description and the contracts specified in the `AGREE` annex are abstracted and formulated into a QE problem in the `Redlog` input format. `Redlog` acts as a back-end solver and interacts with `AGREE` as shown in Figure 4.6. The figure also illustrates that the data flow of our QE-based compositional verification completely bypasses `JKind`, removing the multi-step proof required under the `AGREE`'s model-checking scheme. More importantly, our QE-based approach is able to automatically infer the strongest system property, given the component properties. This feature is missing from the current compositional reasoning tools.

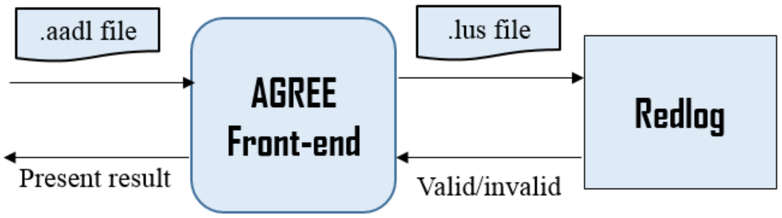


Figure 4.6: Data flow of compositional verification through `Redlog`.

Figure 4.7 shows an illustrative model example taken from [2], that we denote as system S. It is composed of three components, whose architecture (components and connectivity relation) is specified in an .aadl file. The assume-guarantee style component contracts for the components A, B, C, and postulated contract for system S are as listed below:

- $Contract_A$: $(In_A < 20) \Rightarrow (Out_A < 2 \times In_A)$;
- $Contract_B$: $(In_B < 20) \Rightarrow (Out_B < In_B + 15)$;
- $Contract_C$: $true \Rightarrow (Out_C = In_C1 + In_C2)$;
- $Contract_S$: $(In_S < 10) \Rightarrow (Out_S < 50)$.

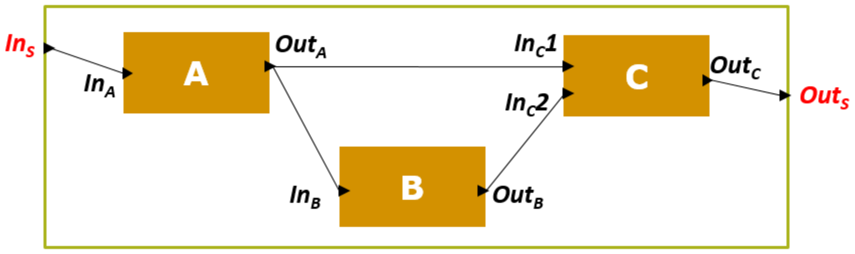
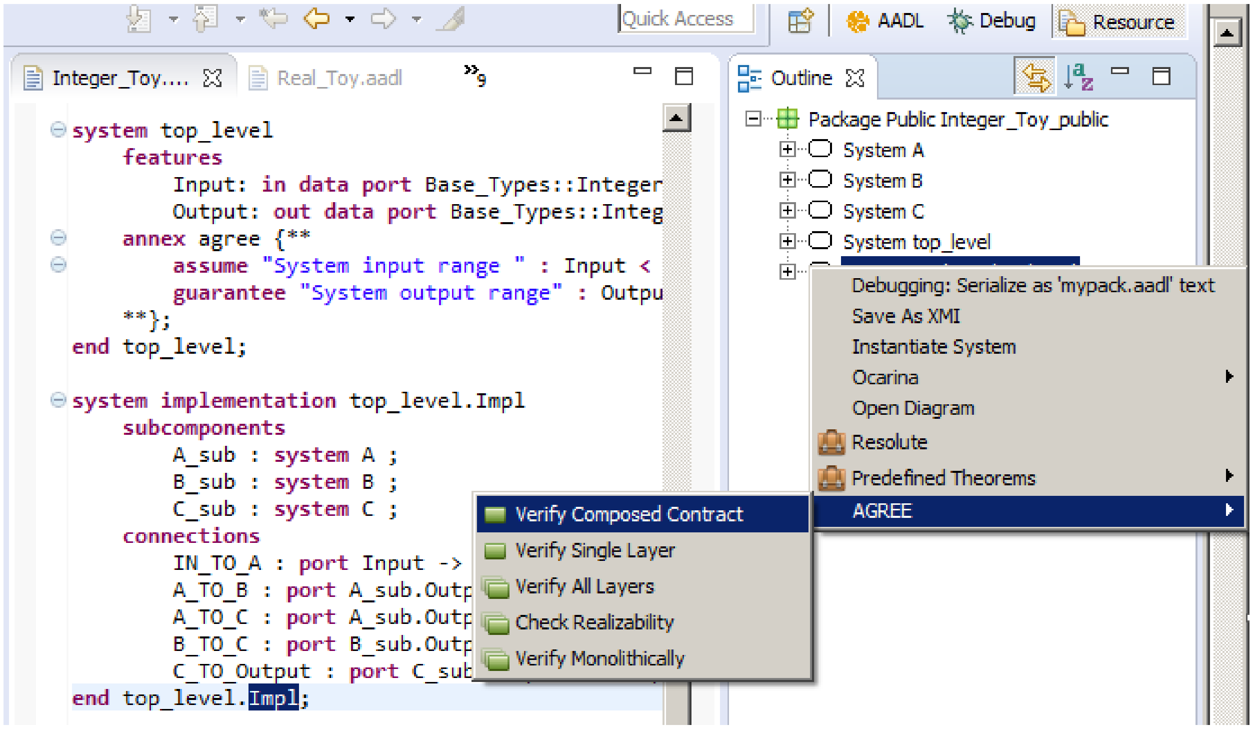


Figure 4.7: An example model architecture, modified from [2].

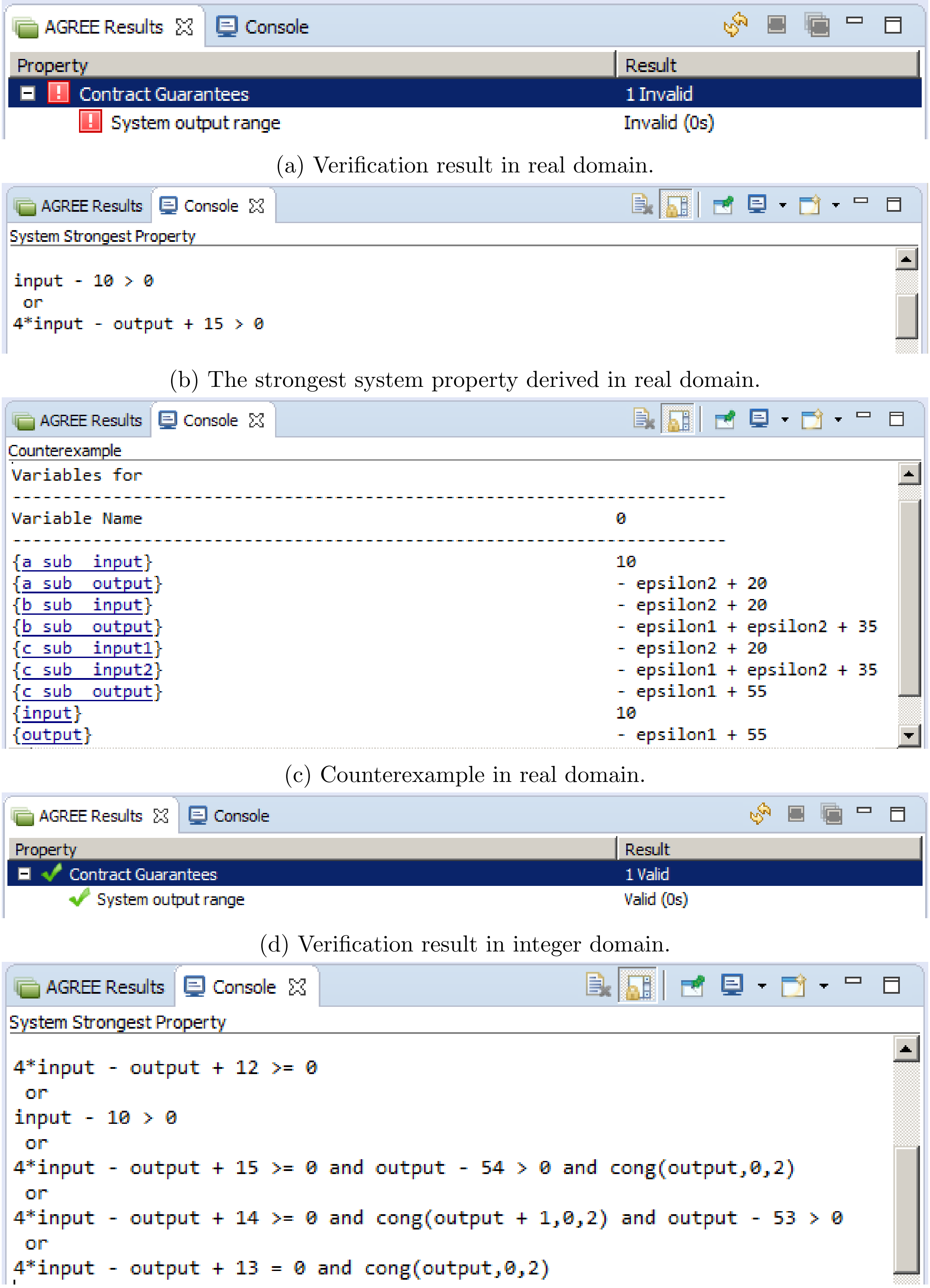
Using the AGREE front-end, `ReLIC` can be executed by a newly added `AGREE` menu button "Verify Composed Contract" as in Figure 4.8. Figure 4.9(a) shows the verification result of the example interpreted in real domain, where the derived strongest system property $(In_S \leq 10) \Rightarrow (Out_S < 4 \times In_S + 15)$ is output to the OSATE2 console as shown in Figure 4.9(b). This is seen to not imply the postulated $Contract_S$, e.g., when (In_S, Out_S) is assigned the values $(9, 50)$. A counterexample reported by Redlog is shown in Figure 4.9(c). Note that the Redlog counterexample answer can contain constants named *infinity* or *epsilon*, both indexed by a number: All *infinity*'s are positive and infinite, and all *epsilon*'s are positive and infinitesimal with respect to the underlying field.



```
system top_level
    features
        Input: in data port Base_Types::Integer
        Output: out data port Base_Types::Integ
    annex agree {**
        assume "System input range " : Input <
        guarantee "System output range" : Outpu
    **};
end top_level;

system implementation top_level.Impl
    subcomponents
        A_sub : system A ;
        B_sub : system B ;
        C_sub : system C ;
    connections
        IN_TO_A : port Input ->
        A_TO_B : port A_sub.Outp
        A_TO_C : port A_sub.Outp
        B_TO_C : port B_sub.Outp
        C_TO_Output : port C_sub
end top_level.Impl;
```

Figure 4.8: Execution of ReLIC via AGREE command in OSATE2.

In contrast, in the integer domain, the strongest system property implies the postulated system property, and the "Verify Composed Contract" returns *true* as in Figure 4.9(d). The ReLIC derived strongest system property is shown in the Figure 4.9(e), in which the syntax $cong(p_1, p_2, p_3)$ is a Redlog representation of congruences with the non-parametric modulus given by the third argument. The strongest system property in the integer domain $(In_S \leq 10) \Rightarrow (Out_s \leq 4 \times In_s + 12)$ is more stringent. This is easily shown to imply the postulated $Contract_S$. In both instances (of real vs. integer domain), the entire verification process is performed within 1 second on a standard laptop.



Property	Result
⊟ ! Contract Guarantees	1 Invalid
! System output range	Invalid (0s)

(a) Verification result in real domain.

System Strongest Property

```
input - 10 > 0
 or
4*input - output + 15 > 0
```

(b) The strongest system property derived in real domain.

Counterexample

```
Variables for
-----------------------------------------------------------------------
Variable Name                                                     0
-----------------------------------------------------------------------
{a sub  input}                                                    10
{a sub  output}                                                   - epsilon2 + 20
{b sub  input}                                                    - epsilon2 + 20
{b sub  output}                                                   - epsilon1 + epsilon2 + 35
{c sub  input1}                                                   - epsilon2 + 20
{c sub  input2}                                                   - epsilon1 + epsilon2 + 35
{c sub  output}                                                   - epsilon1 + 55
{input}                                                           10
{output}                                                          - epsilon1 + 55
```

(c) Counterexample in real domain.

Property	Result
⊟ ✓ Contract Guarantees	1 Valid
✓ System output range	Valid (0s)

(d) Verification result in integer domain.

System Strongest Property

```
4*input - output + 12 >= 0
 or
input - 10 > 0
 or
4*input - output + 15 >= 0 and output - 54 > 0 and cong(output,0,2)
 or
4*input - output + 14 >= 0 and cong(output + 1,0,2) and output - 53 > 0
 or
4*input - output + 13 = 0 and cong(output,0,2)
```

(e) The strongest system property derived in integer domain.

Figure 4.9: `ReLIC` verification results on the illustrative example.

4.4 ReLIC for time-dependent property composition

Complex systems often exhibit time-dependent features through components such as PID controller, counter, or state-machine. In such cases, a component property can be a constraint over its input/internal/output variables at different time-steps. The extension of QE-based property composition to the time-dependent scenarios is not obvious; in fact (4.1) cannot be used as is.

Example 7. *In order to see the difficulty encountered in case of time-dependent property composition, consider a simple example that consists of* cascade *of two identical components with input u and output x for the first system, and input x and output y for the second system, and with properties $x > \mathbf{pre}(u)$ and $y > \mathbf{pre}(x)$ respectively, in which $\mathbf{pre}(\cdot)$ denotes the previous value function, whereas the cascade connectivity is implied by the common variable x. Then one can see that the strongest system property is simply the formula, $y > \mathbf{pre}(\mathbf{pre}(u))$. Note this final formula includes the term $\mathbf{pre}(\mathbf{pre}(u))$ that does not appear in the given component property formulae, and so a standard quantifier elimination as in (4.1) cannot be employed to obtain the above final formula. Strikingly, if we shift each component property by one time step, and compose the component properties and their time shifted replicas with the internal variable x, $\mathbf{pre}(x)$, and $\mathbf{pre}(\mathbf{pre}(x))$ existentially quantified: $\exists x \exists \mathbf{pre}(x) \exists \mathbf{pre}(\mathbf{pre}(x))\big((x > \mathbf{pre}(u)) \wedge (y > \mathbf{pre}(x)) \wedge \big(\mathbf{pre}(x) > \mathbf{pre}(\mathbf{pre}(u))\big) \wedge \big(\mathbf{pre}(y) > \mathbf{pre}(\mathbf{pre}(x))\big)\big)$, then upon quantifier elimination, we do get the desired composed property: $y > \mathbf{pre}(\mathbf{pre}(u))$.*

4.4.1 Approach to time-dependent property composition

Based on the above simple example, it is clear that the component properties may need to be time-shifted to match the possibly higher order time-shifts needed for the governing equations/inequations of the composed system. To formalize the amount of time-shifts required, we first introduce the notion of component/system order. For a time-dependent component/system, each internal and output value is governed by the values of itself and of component/system inputs over a finite history, which can be formulated as a set of difference equations/inequations over the component/system variables.

Definition 11. *We define component/system order as the difference of maximum and minimum time-shifts present in its governing difference equations/inequations.*

Remark 11. *Note that the order as defined above concerns all the component/system relations, and it is possible that the composition of the equations/inequations results in a lower order input-output property upon simplification. For example, a system with input u, internal variable x, and output y can have properties $x = u - \mathbf{pre}(x)$ and $y = x + \mathbf{pre}(x)$. Then its order by our definition is at least 1, but the simplified input-output property $y = x + \mathbf{pre}(x) = (u - \mathbf{pre}(x)) + \mathbf{pre}(x) = u$ is of zero order.*

Consider a component/system with a set of variables $X := \{x_1, \ldots, x_n\}$ and set of internal variables $X_{\text{int}} := \{x_1, \ldots, x_m\} \subseteq X$, with $m \leq n$. Let $x(k)$ denote the variable at the k^{th} time step with step 0 being the initial step, and for $s, t \in \mathbb{Z}_{\geq 0}, s \leq t : X([s, t]) := \{x(k) | x \in X, k \in [s, t]\}$ be the variables over the time interval $[s, t]$. If the order of this component/system is less or equal to M, then its behavior can be summarized by the set of constraints over any interval $[k, k + M]$ of $M + 1$ consecutive time steps for all $k \in \mathbb{Z}_{\geq 0}$:

$$\bigwedge \text{all constraints over } X([k, k + M]).$$

Then the formula for its input-output property is given by the existential quantification of all the internal variables over all the $M + 1$ time steps:

$$\exists x_1(k) \ldots \exists x_1(k + M) \ldots \exists x_m(k) \ldots \exists x_m(k + M) \Big(\bigwedge \text{all constraints over } X([k, k + M]) \Big),$$

which can be written in a short form as:

$$\exists X_{\text{int}}([k, k + M]) \Big(\bigwedge \text{all constraints over } X([k, k + M]) \Big). \tag{4.4}$$

The existentially quantified variables can be eliminated by applying QE on (4.4) to obtain a simplified property formula that may be of a lower order than M. As illustrated in the beginning of Section 4.4, additional time-shifts of component properties may be required before those are composed. In order to decide the number of time-shifts needed, the order M_{sys} of the composed system must be estimated. This is presented in the following theorem which states that an upper bound for the system order is the sum of all its component orders.

Theorem 7. *Given a system S composed of N multi-input-multi-output (MIMO) [99] components, if the i^{th} component is of order M_i, then $\sum_{i=1}^{N} M_i$ is an upper bound for the order M_{sys} of S.*

Proof. Without loss of generality, pick any two components of S with orders M_1 and M_2 respectively. Each component possesses a set of properties specified as nonlinear difference equations/inequations in the general form of $f(\cdot) \sim 0$, where $\sim \in \{>, \geq, =\}$. For an inequation, we can introduce a slack variable $u_f \sim 0$ such that the given difference *inequation* can equivalently be written as the conjunction of a difference *equation* $f(\cdot) - u_f = 0$ and an extra linear constraint $u_f \sim 0$. Note this additional constraint is of zero order and hence can not alter the overall order.

For the set of difference equations of component i with N_i inputs (including slack variables) and O_i outputs, $i = 1, 2$, it is known that there exists an equivalent state-space representation [99], and also the number of the state variables equals the order of the component. The general form of such a state-space representation for component i is:

$$\mathbf{x_i}(k+1) = \mathbf{f_i}\big(\mathbf{x_i}(k), \mathbf{u_i}(k)\big),$$
$$\mathbf{y_i}(k) = \mathbf{g_i}\big(\mathbf{x_i}(k), \mathbf{u_i}(k)\big),$$

where vectors $\mathbf{u_i}$ (size: $N_i \times 1$), $\mathbf{x_i}$ (size: $M_i \times 1$), and $\mathbf{y_i}$ $(size : O_i \times 1)$ are respectively the input, state, and output variable vectors of component i. (The constraints on the added slack variables also exist, but as noted, do not involve time-shifts and so do not alter the component order.) $\mathbf{f_i}(\cdot)$ (size: $M_i \times 1$) and $\mathbf{g_i}(\cdot)$ (size: $O_i \times 1$) are vectors of functions.

One can simply stack the two state space representations into a single one:

$$\begin{bmatrix} \mathbf{x_1}(k+1) \\ \mathbf{x_2}(k+1) \end{bmatrix} = \begin{bmatrix} \mathbf{f_1}\big(\mathbf{x_1}(k), \mathbf{u_1}(k)\big) \\ \mathbf{f_2}\big(\mathbf{x_2}(k), \mathbf{u_2}(k)\big) \end{bmatrix},$$
$$\begin{bmatrix} \mathbf{y_1}(k) \\ \mathbf{y_2}(k) \end{bmatrix} = \begin{bmatrix} \mathbf{g_1}\big(\mathbf{x_1}(k), \mathbf{u_1}(k)\big) \\ \mathbf{g_2}\big(\mathbf{x_2}(k), \mathbf{u_2}(k)\big) \end{bmatrix}.$$

It is then clear that the number of states of these, so far unconnected components, equals the sum of the numbers of states of the two individual components. We claim that when the components are connected, the state size does not grow, from which the desired result stated in the theorem can be obtained. A general connectivity relation between two components can be formulated as $\mathbf{u_c} = \mathbf{y_c}$ where $\mathbf{u_c}$ is a vector of inputs from the union of the two component inputs, and $\mathbf{y_c}$ is a vector of outputs from the union of two component outputs. We note that one input can only connect to one output whereas one output may connect to multiple inputs. Since the overall connected system can be obtained by iteratively adding one connection at a time, it suffices to show that adding a single connection does not introduce an additional state variable. For compactness of notation, let the state-space representation of the combined system be:

$$\mathbf{x}(k+1) = \mathbf{f}\big(\mathbf{x}(k), \mathbf{u}(k)\big),$$
$$\mathbf{y}(k) = \mathbf{g}\big(\mathbf{x}(k), \mathbf{u}(k)\big),$$

where the state set has $M_1 + M_2$ variables. Suppose a single connection (u_p, y_q) is introduced within the system, where $u_p \in \mathbf{u}$, $y_q \in \mathbf{y}$, and u_p, y_q are from different components. Let $\mathbf{u}'$ (resp. $\mathbf{y}'$) be the vector after removing u_p (resp. y_q) from $\mathbf{u}$ (resp. $\mathbf{y}$). Then, we have $u_p(k) = y_q(k) = g_q\big(\mathbf{x}(k), \mathbf{u}(k)\big) = g_q\big(\mathbf{x}(k), \mathbf{u}'(k)\big)$ with $g_q \in \mathbf{g}$, and so the state-space representation can be written as:

$$\mathbf{x}(k+1) = \mathbf{f}\Big(\mathbf{x}(k), \mathbf{u}'(k), g_q\big(\mathbf{x}(k), \mathbf{u}'(k)\big)\Big),$$
$$\mathbf{y}(k) = \mathbf{g}\Big(\mathbf{x}(k), \mathbf{u}'(k), g_q\big(\mathbf{x}(k), \mathbf{u}'(k)\big)\Big).$$

Then it is easily seen that the updated state-space representation possesses input variables from $\mathbf{u}'$ (u_p becomes internal), output variables from $\mathbf{y}$ (or $\mathbf{y} \setminus \{y_q\}$ if y_q also becomes internal), while the state-space remains $\mathbf{x}$. (Further simplification and a state-space reduction may be possible.) It follows that the system order of the connected system remains upper bounded by $M_1 + M_2$. The result of the theorem then follows by iteratively connecting more components to the composition.

□

Note any component may be described by a *set* of properties over its own input, internal, and output variables (see for example the component CNTRL of the vehicle example in Figure 4.11), and can be treated as a set of sub-components, each associated with a single property, and with their connectivity relation implied by their common variables. Hence we have the following corollary:

Corollary 7.1. *The order of a component is bounded by the sum of all its property orders. The system order of S in Theorem 7 is bounded by the sum of all the individual property orders of all its components.*

After the composed system order bound M_{sys} is determined, the i^{th} individual property of the components can be replicated $M_{\text{sys}} - M_i$ times, where M_i is the order of the i^{th} individual property, and the j^{th} replica shifted j time-steps ($j = 1, \ldots, M_{\text{sys}} - M_i$) to obtain the set of all constraints over $X([k, k + M_{\text{sys}}])$. These replicated and time-shifted properties of all the components can then be composed to deduce the strongest system property:

$$\exists X_{\text{int}}([k, k + M_{\text{sys}}])\Big(\bigwedge \text{all constraints over } X([k, k + M_{\text{sys}}])\Big), \tag{4.5}$$

where as before, the existential quantification is w.r.t. the internal variables $X_{\text{int}}([k, k + M_{\text{sys}}])$ of the system.

Remark 12. *When the actual system order M_{sys} is less than the upper bound $\sum_{i=1}^{N} M_i$ provided by Theorem 7, the result of the QE in (4.5) will contain the conjunction of $\sum_{i=1}^{N} M_i - M_{sys}$ redundant expressions that are the time-shifted replicas of some other conjuncts within the same expression.*

Example 8. *Consider three properties $z = y + x$, $y = \mathbf{pre}(u)$, and $x = \mathbf{pre}(w)$, where x and y are internal variables. Then, per Theorem 7, the system order upper bound is the sum of the three subsystem orders: $0 + 1 + 1 = 2$. On the other hand, it is easy to see that the composed property over the external variables z, u, and w is $z = y + x = \mathbf{pre}(u) + \mathbf{pre}(w)$, which is only of order 1.*

The computation of the composed property using our approach requires shifting each component by its order difference with the upper bound ($2, 1, 1$ *resp.), and then combining those using (4.5):*

$$\begin{aligned}&\exists x(k)\exists x(k+1)\exists x(k+2)\exists y(k)\exists y(k+1)\exists y(k+2)\\&\Big(\big(z(k)=y(k)+x(k)\big)\wedge\big(z(k+1)=y(k+1)+x(k+1)\big)\wedge\big(z(k+2)=y(k+2)+x(k+2)\big)\\&\quad\wedge\big(y(k+1)=u(k)\big)\wedge\big(y(k+2)=u(k+1)\big)\wedge\big(x(k+1)=w(k)\big)\wedge\big(x(k+2)=w(k+1)\big)\Big).\end{aligned}$$

Upon quantifier elimination we obtain:

$$\big(z(k+1)=u(k)+w(k)\big)\wedge\big(z(k+2)=u(k+1)+w(k+1)\big),$$

in which $z(k+2)=u(k+1)+w(k+1)$ *is a one-step shifted replica of* $z(k+1)=u(k)+w(k)$*, hence redundant. By expressing the QE result in conjunctive normal form, the redundant expressions can be readily identified as the time-shifted replicas of some other expressions, and eliminated.*

Following a similar approach as described above, we can also obtain the system-level initial condition by composing the component-level initial conditions. For a system of order M_{sys}, initial conditions over the first M_{sys} steps $(0,\ldots,M_{\text{sys}}-1)$ suffice. So the system-level initial condition can be obtained using:

$$\exists X_{\text{int}}([0,M_{\text{sys}}-1])\Big(\bigwedge \text{all constraints over } X([0,M_{\text{sys}}-1])\Big). \tag{4.6}$$

Example 9. *Consider the same example given at the beginning of this section with the component properties and initial conditions:* $y>\mathbf{pre}(x), y(0)>0$ *and* $x>\mathbf{pre}(u), u(0)>1$*. Then in this case, (4.6) is encoded as:*

$$\exists y([0,1])\Big(\big(z(0)>0\big)\wedge\big(z(1)>y(0)\big)\wedge\big(y(0)>1\big)\wedge\big(y(1)>x(0)\big)\Big),$$

which is equivalent to $\big(z(0)>0\big)\wedge\big(z(1)>1\big)$.

Once the system-level property and initial condition are encoded using (4.5) and (4.6) respectively, the verification of a postulated system property can be done using an induction-based proof.

4.4.2 Implementation and experimental result

The overall approach for time-dependent/temporal property composition and checking the correctness of a postulated property is summarized in Algorithm 6, whose steps can be understood as follows:

Algorithm 6: Quantifier elimination-based compositional verification for time-dependent properties.

```
    Input: System $S$, set of all variables $X := \{x_1, \dots, x_n\}$, set of internal variables
           $X_{\text{int}} := \{x_1, \dots, x_m\}(m \leq n)$, set of connectivity relations $C$, set of component
           properties $\Phi_{\text{comp}}$, set of component initial conditions $I_{\text{comp}}$, postulated system
           property $\phi_{\text{postl}}$ over system inputs and outputs $X \setminus X_{\text{int}}$.
 1  $\Phi_{\text{all}} \leftarrow \Phi_{\text{comp}} \cup \{(x_p^k = x_q^k) | (x_p, x_q) \in C\}$;
 2  $M_{\text{sys}} \leftarrow \Sigma_{\phi \in \Phi_{\text{comp}}} \text{Order}(\phi)$;
 3  $\Phi_{\text{all}}([k, k + M_{\text{sys}}]) \leftarrow \{\text{Shift}(\phi, i) | \phi \in \Phi_{\text{all}}, i \in [0, M_{\text{sys}} - \text{Order}(\phi)]\}$;
 4  $f_{\text{sys}} \leftarrow \text{Compose}\big(X([k, k + M_{\text{sys}}]), X_{\text{int}}([k, k + M_{\text{sys}}]), \Phi_{\text{all}}([k, k + M_{\text{sys}}])\big)$;
 5  $\phi_{\text{sys}} \leftarrow \text{Redlog}(f_{\text{sys}})$;
 6  $I_{\text{sys}} \leftarrow \mathit{true}$;
 7  if $I_{comp} \neq \emptyset$ then
 8      $I_{\text{all}}([0, M_{\text{sys}} - 1]) \leftarrow I_{\text{comp}} \cup \Phi_{\text{all}}([0, M_{\text{sys}} - 1])$;
 9      $f_{\text{sys}} \leftarrow \text{Compose}\big(X([0, M_{\text{sys}} - 1]), X_{\text{int}}([0, M_{\text{sys}} - 1]), I_{\text{all}}([0, M_{\text{sys}} - 1])\big)$;
10      $I_{\text{sys}} \leftarrow \text{Redlog}(f_{\text{sys}})$;
11  end
12  $(\mathit{result}, \mathit{ce}) \leftarrow k\text{-induction}(\phi_{\text{sys}}, I_{\text{sys}}, \phi_{\text{postl}}, K)$;
    Output: $\mathit{result}, \mathit{ce}$
```

Line 1 collects the set of all constraints Φ_{all} over variables X as the union of component properties and the constraints from their connectivity relation. Line 2 assigns the system order M_{sys} with its upper bound computed according to Theorem 7 and its corollary. In Line 3, all the constraints over $X([k, k + M_{\text{sys}}])$, denoted $\Phi_{\text{all}}([k, k + M_{\text{sys}}])$, are collected to comprise the set of all the necessary time-shifted replicas of properties in Φ_{all}. The system-level composed property f_{sys} is obtained in line 4 over $X([k, k + M_{\text{sys}}])$, by internalizing $X_{\text{int}}([k, k + M_{\text{sys}}])$ in $\Phi_{\text{all}}([k, k + M_{\text{sys}}])$ as in (4.5). The quantifier eliminator tool `Redlog` then performs QE on f_{sys} to obtain the quantifier-free formula ϕ_{sys} in line 5. Line 6 initializes the system-level initial condition I_{sys} to *true* (which is the default value when there are no component initial conditions). If the component initial condition set is non-empty, lines 8-10 are used to derive the quantifier-free system-level initial condition I_{sys} based on (4.6). In line 12, k-induction based model-checking by QE-integrated `JKind` checks whether the strongest system property ϕ_{sys} implies the postulated system property ϕ_{postl}, and to obtain the verification result $result$ which could be $true$, $false$, or $unknown$. A counterexample ce is generated if $result$ is $false$. Note this inductive proof of the postulated system property is carried out entirely at the system-level, without having to go back to the component-level, as it is desired of a compositional reasoning approach.

We extended our prototype tool `ReLIC` to compose finite-order time-dependent properties (contracts). The extended architecture of `ReLIC` is shown in Figure 4.10. It utilizes the front-end infrastructure of the `AGREE` tool for capturing and parsing the input AADL models and also its output environment for presenting the results to the console. The "Property Composer" module computes the upper bound for the system-level order based on Theorem 7 and its corollary and computes the strongest system-level property (contract). This is then used by the "Induction Verifier" module to perform induction-based system-level verification of a postulated system property. Both these modules utilize `Redlog` as a back-end solver.

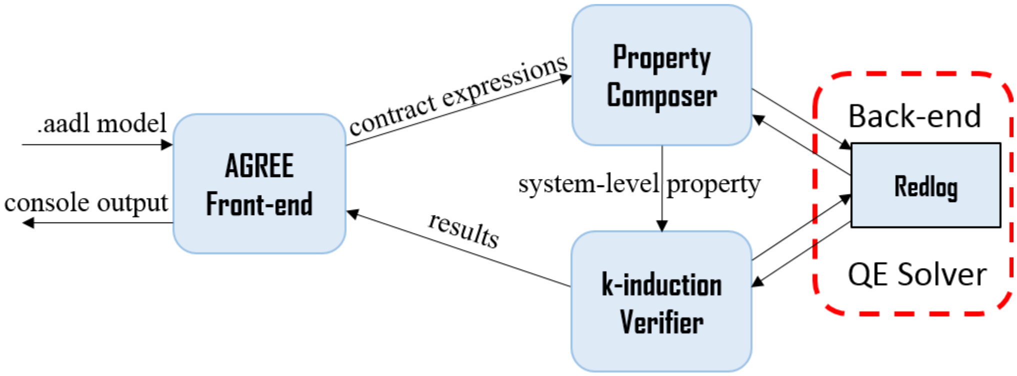


Figure 4.10: Architecture of ReLIC.

To demonstrate our tool, we tested it on a vehicle model as shown in Figure 4.11(a). The vehicle consists of two components, namely, a PID control component "Speed_Control" (CNTRL) and a vehicle throttle "Throttle" (THROT), within a feedback configuration. The dynamics of "Speed_Control" is specified by a set of difference equations involving also certain state variables

(Figure 4.11(b)), whereas the dynamics of "Throttle" is specified by a difference equation over only its input and output variables (Figure 4.11(c)). There are three first-order expressions in total:

$$e\_int = \mathbf{prev}(e, 0.0) + e,$$

$$e\_dot = \mathbf{prev}(e, 0.0) - e, \text{ and}$$

$$Actual.val = \mathbf{prev}(Actual.val, 0.0) + 0.1 * Actuator\_Input,$$

where $\mathbf{prev}(\cdot)$ is the extended delay operator $\mathbf{pre}(\cdot)$ with the second argument denoting the initial value. The postulated system property is

$$const\_tar\_speed \Rightarrow (Actual\_Speed.val < 1.0),$$

where $const\_tar\_speed$ is a Boolean variable whose truth indicates system target speed is set to a constant value 1.0 (see its defining expression in Figure 4.12). The `ReLIC` inferred strongest system property (based on (4.5) and (4.6)) contains the system-level difference equation as well as the initial condition over system input $Target\_Speed.val$ and system output $Actual\_Speed.val$. While the upper bound for the composed system order is 3, it turns out that the system input-output property order is only 1 due to the inherent parallelism among the components, as also computed by our approach (see the encoded expressions and the console output in Figure 4.12):

$$51 * actual\_speed - 49 * pre\_\_actual\_speed - pre\_\_target\_speed - target\_speed = 0,$$

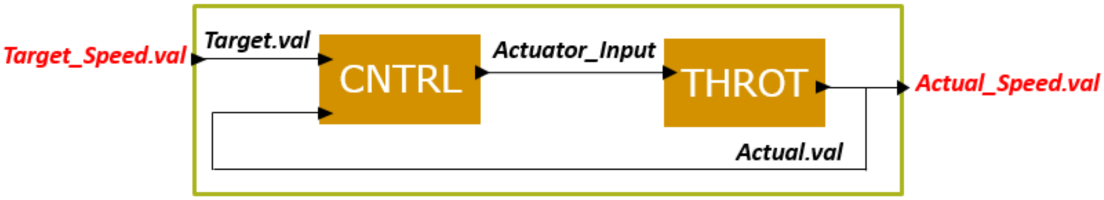
with the initial condition:

$$51 * actual\_speed - target\_speed = 0$$

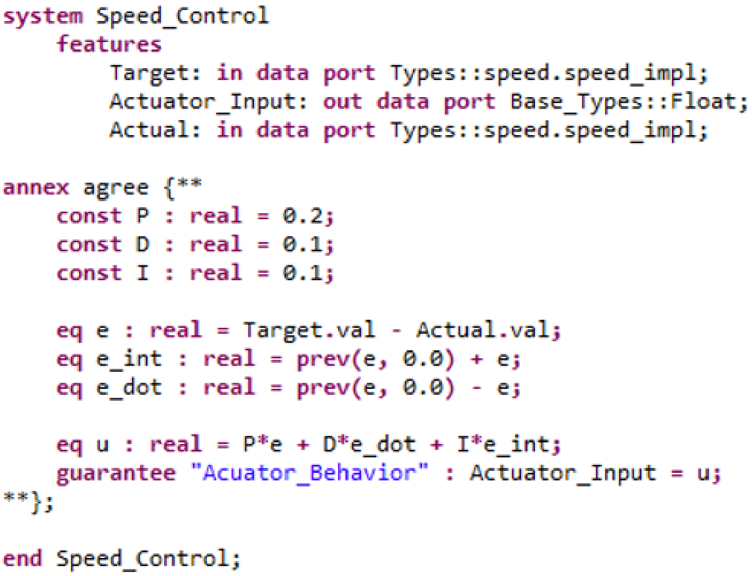
These are easily shown to imply the postulated system property using induction for $k = 1$, where the base step and the inductive step are:

$$(51 * actual\_speed - target\_speed = 0) \Rightarrow \big((target\_speed = 1) \Rightarrow (actual\_speed < 1)\big), \text{ and}$$

$$\begin{aligned}\big((51 * actual\_speed - 49 * pre\_\_actual\_speed - pre\_\_target\_speed - target\_speed = 0)\\ \wedge (pre\_\_target\_speed = 1) \wedge (pre\_\_actual\_speed < 1)\big)\\ \Rightarrow \big((target\_speed = pre\_\_target\_speed) \Rightarrow (actual\_speed < 1)\big).\end{aligned}$$

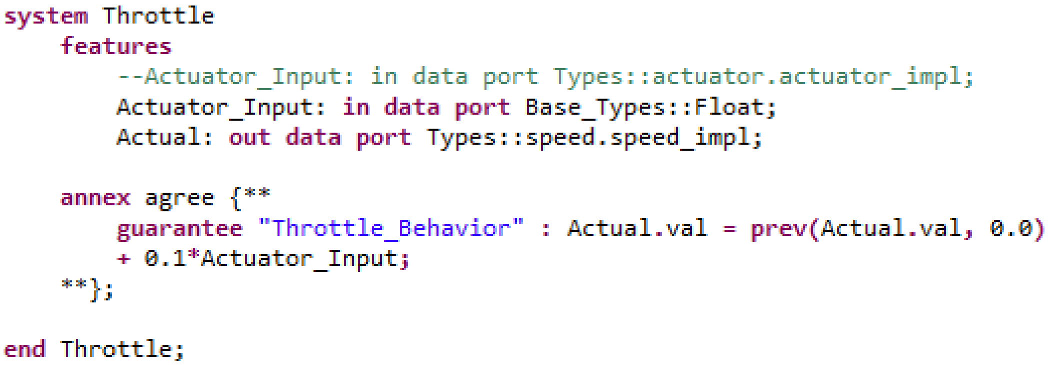


(a) A vehicle model and its components dynamics.



```
system Speed_Control
    features
        Target: in data port Types::speed.speed_impl;
        Actuator_Input: out data port Base_Types::Float;
        Actual: in data port Types::speed.speed_impl;

annex agree {**
    const P : real = 0.2;
    const D : real = 0.1;
    const I : real = 0.1;

    eq e : real = Target.val - Actual.val;
    eq e_int : real = prev(e, 0.0) + e;
    eq e_dot : real = prev(e, 0.0) - e;

    eq u : real = P*e + D*e_dot + I*e_int;
    guarantee "Acuator_Behavior" : Actuator_Input = u;
**};

end Speed_Control;
```

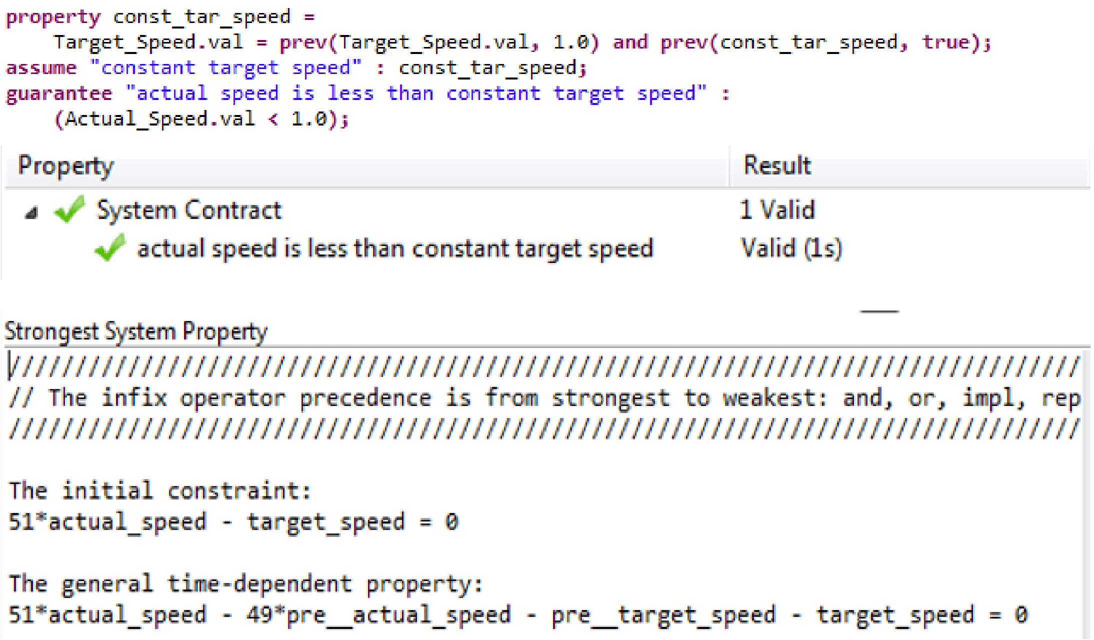
(b) Specification of CNTRL component.



```
system Throttle
    features
        --Actuator_Input: in data port Types::actuator.actuator_impl;
        Actuator_Input: in data port Base_Types::Float;
        Actual: out data port Types::speed.speed_impl;

    annex agree {**
        guarantee "Throttle_Behavior" : Actual.val = prev(Actual.val, 0.0)
        + 0.1*Actuator_Input;
    **};

end Throttle;
```

(c) Specification of THROT component.

Figure 4.11: A vehicle model, modified from [2].



```
property const_tar_speed =
    Target_Speed.val = prev(Target_Speed.val, 1.0) and prev(const_tar_speed, true);
assume "constant target speed" : const_tar_speed;
guarantee "actual speed is less than constant target speed" :
    (Actual_Speed.val < 1.0);
```

Property	Result
◢ ✓ System Contract	1 Valid
✓ actual speed is less than constant target speed	Valid (1s)

Strongest System Property

```
///////////////////////////////////////////////////////////////////////////////
// The infix operator precedence is from strongest to weakest: and, or, impl, rep
///////////////////////////////////////////////////////////////////////////////

The initial constraint:
51*actual_speed - target_speed = 0

The general time-dependent property:
51*actual_speed - 49*pre__actual_speed - pre__target_speed - target_speed = 0
```

Figure 4.12: ReLIC verification output on the vehicle model.

CHAPTER 5. CONCLUSION

Our work on the formal verification of the model-based cyber-physical systems involves research over different but related subjects within the compositional verification paradigm. We started from approaches applied at the component-level to tackle the problems raised by models with complex transition behaviors and/or nonlinear continuous dynamics. Once the component-level properties are established, compositional reasoning technique can take over to the next level, i.e., establishing the system-level properties without looking again into redundant component-level implementation details. The derived system property further can be used to prove postulated system property to complete the verification process if the former is more stringent. At each of the three stages of our work, we developed a prototype tool that implemented our verification algorithms and demonstrated the capability and efficiency of our approaches.

In Chapter 2, we presented a counterexample fragment based specification relaxation (CEFSR) approach for the safety verification of linear hybrid automata. While preserving the discrete-time behaviors, a linear hybrid automaton is translated to an equivalent linear transition system with the same discrete transition graph. The abstract model of the linear transition system, i.e., its discrete transition graph, is model checked against the safety specification to find a counterexample in each iteration. Feasibility analysis of a counterexample in the concrete model is conducted by solving the set of constraints collected along the counterexample path in the concrete model and compacted to find an unsatisfiable core. We also explained the way to encode the unsatisfiable core of a spurious counterexample and to use it to relax the current specification. The relaxed specification eliminates all counterexamples possessing the same unsatisfiable core in a single iteration. The above approach is implemented in our prototype verifier `LhaVrf` , integrated with the state-of-the-art symbolic model checker `NuSMV` and SMT solver `Z3`. The verifier accepts a set of input files containing the constituent linear hybrid automata written in an easy-to-specify textual syntax. In case of termination, the verifier outputs a concrete counterexample that reaches an unsafe location, or reports that safety is satisfied. Since the reachability problem for LHAs is undecidable, there is no a priori guarantee of termination (as is the case with any hybrid system verifier). The scalability of the tool is demonstrated by applying it to an instance of Fischers mutual exclusion protocol with 10 processes (containing approximately 1 million discrete locations and 13 million transition edges).

In Chapter 3, we presented an on-the-fly dynamic repartitioning based scheme, integrated into a simulation-based approach for over-approximating of the bounded-time reachability of hybrid systems. The dynamic, on-the-fly partitioning not only helps limit the over-approximation error, but also extends the applicability to general hybrid systems. Prior simulation-based approach did not possess such generality because the occurrence of new discrete transitions in the states reached within the simulation error bound. As a result, the usage of prior simulation-based verification approaches has remained limited to either continuous dynamical systems (with no discrete jumps), or to switched systems subject to time-driven switching with predefined switching signals, or to hybrid systems with strict assumptions on state continuity, and location consistency among simulation and execution traces. Our development shows that the computation of the reachable set is guaranteed to be reliable in each of the simulation time-steps. This is then used to generate the reliable simulation seeds of the next time-steps. Thereby, the reliability of simulation trajectory in over-approximating the reachability is maintained. Meanwhile, our algorithm performs state repartition at each discrete transition, so that it can handle general hybrid systems with guard/reset predicates, thereby allowing state-triggered discrete transitions. Our approach also contributed to error growth control. In order to alleviate the exponential rate of error growth of the tubes, our approach periodically partitions the reachable set, which converges for convergent dynamics. As a result, the number of running simulations is dynamically decided in run-time. It may even decrease compared to the initial number for a convergent dynamics. These are not guaranteed for many of the existing tools as shown by our example. For certain type of convergent systems, we provided convergent bounds on trajectory deviation, even for using simple to compute Lipschitz-based discrepancy functions. As a result, for systems that exhibit convergent behaviors, system safety over infinite horizon may be also assured as witnessed in Figure 3.10. Another novel feature of our approach is that it supports systems with certain types of inputs (i.e., open systems). Finally, we also presented the implementation of our prototype tool (HS^{3}V) which employs the proposed algorithms, and experiment results reporting improvements on a variety of benchmarks.

Compositional reasoning is central to scaling model-based approaches for establishing correctness. In Chapter 4, we showed that the foundational principle underlying the compositional reasoning is Quantifier Elimination (QE). Within our compositional verification framework, QE was used to derive the strongest system property from the given components properties and their connectivity relation. This simplifies the compositional verification into two fixed steps, independent of the number of components, and also reduces the number of variables needed in the proof of the postulated system properties, thus improving the efficiency of compositional reasoning. We further extended our property composition framework to support the composition of time-dependent properties. This is a foundational step towards compositional reasoning of systems with "memory". The extension developed a new procedure to determine the system order, given the component orders, which we implemented in our prototype tool ReLIC that uses AGREE at the front-end for model input and result output, and Redlog at the back-end for performing QE. Within this tool, we also implemented the QE procedure to infer the system-level property that incorporates time-dependence. The proof of a postulated system-level property involves induction, which is also supported in ReLIC. Aside from property composition using QE, we showed that certain formal verification steps can also be viewed as QE problems. Thereby, the QE tools provide options to also extend the existing verification tools (like the compositional reasoning tools). In order to take advantage of QE's ability of solving the satisfiability problem over first-order logic, we integrated the QE tool Redlog as a back-end solver, in parallel with the existing SMT solvers, for the k-induction based model-checker JKind, enhancing the later's capability to model-check nonlinear properties. Our QE-integrated JKind was able to resolve a fuzzy logic problem involving non-linear computation efficiently, whereas the SMT-integrated version was unable to terminate, showing the extended capability for model-checking provided by QE.

BIBLIOGRAPHY

[1] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[2] A. Gacek, J. Backes, W. Mike, C. Darren, and L. Jing, "Agree users guide, v0.9," 2017.

[3] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*, pp. 127–144, Springer, 2000.

[4] "Redlog." http://www.redlog.eu.

[5] A. Gacek, "Jkind-a java implementation of the kind model checker," *Retrieved June*, vol. 15, p. 2016, 2015.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical computer science*, vol. 138, no. 1, pp. 3–34, 1995.

[7] T. A. Henzinger, "The theory of hybrid automata," in *Verification of Digital and Hybrid Systems*, pp. 265–292, Springer, 2000.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*, pp. 154–169, Springer, 2000.

[9] S. Jiang, "Abstraction based reachability analysis of concurrent linear hybrid automata," tech. rep., GM Research Report ECI-293, 2006.

[10] S. Jiang, “Reachability analysis of linear hybrid automata by using counterexample fragment based abstraction refinement,” in *American Control Conference, 2007. ACC’07*, pp. 4172–4177, IEEE, 2007.

[11] R. Kumar, C. Zhou, and S. Jiang, “Safety and transition-structure preserving abstraction of hybrid systems with inputs/outputs,” in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 206–211, IEEE, 2008.

[12] H. Ren, J. Huang, S. Jiang, and R. Kumar, “A new abstraction-refinement based verifier for modular linear hybrid automata and its implementation,” in *Networking, Sensing and Control (ICNSC), 2014 IEEE 11th International Conference on*, pp. 30–35, IEEE, 2014.

[13] “z3.” https://github.com/Z3Prover/z3.

[14] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, “Hytech: A model checker for hybrid systems,” *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997.

[15] R. Alur, “Formal verification of hybrid systems,” in *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pp. 273–278, IEEE, 2011.

[16] G. Frehse, S. K. Jha, and B. H. Krogh, “A counterexample-guided approach to parameter synthesis for linear hybrid automata,” in *International Workshop on Hybrid Systems: Computation and Control*, pp. 187–200, Springer, 2008.

[17] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Hycomp: An smt-based model checker for hybrid systems,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 52–67, Springer, 2015.

[18] E. Asarin, T. Dang, and O. Maler, “The d/dt tool for verification of hybrid systems,” in *International Conference on Computer Aided Verification*, pp. 365–370, Springer, 2002.

[19] T. Dang and O. Maler, “Reachability analysis via face lifting,” in *International Workshop on Hybrid Systems: Computation and Control*, pp. 96–109, Springer, 1998.

[20] A. Chutinan and B. H. Krogh, “Computational techniques for hybrid system verification,” *IEEE transactions on automatic control*, vol. 48, no. 1, pp. 64–75, 2003.

[21] P. S. Duggirala, M. Potok, S. Mitra, and M. Viswanathan, “C2e2: a tool for verifying annotated hybrid systems,” in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pp. 307–308, ACM, 2015.

[22] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, “C2e2: a verification tool for stateflow models,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 68–82, Springer, 2015.

[23] C. Fan and S. Mitra, “Bounded verification with on-the-fly discrepancy computation,” in *International Symposium on Automated Technology for Verification and Analysis*, pp. 446–463, Springer, 2015.

[24] C. Fan, J. Kapinski, X. Jin, and S. Mitra, “Locally optimal reach set over-approximation for nonlinear systems,” in *Proceedings of the 13th International Conference on Embedded Software*, p. 6, ACM, 2016.

[25] C. Le Guernic and A. Girard, “Reachability analysis of hybrid systems using support functions,” in *International Conference on Computer Aided Verification*, pp. 540–554, Springer, 2009.

[26] C. Le Guernic and A. Girard, “Reachability analysis of linear systems using support functions,” *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 250–262, 2010.

[27] 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 *International Conference on Computer Aided Verification*, pp. 379–395, Springer, 2011.

[28] S. Minopoli and G. Frehse, “Sl2sx translator: from simulink to spaceex models,” in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pp. 93–98, ACM, 2016.

[29] M. Althoff and G. Frehse, “Combining zonotopes and support functions for efficient reachability analysis of linear systems,” in *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pp. 7439–7446, IEEE, 2016.

[30] G. Frehse, S. Bogomolov, M. Greitschus, T. Strump, and A. Podelski, “Eliminating spurious transitions in reachability with support functions,” in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pp. 149–158, ACM, 2015.

[31] S. Bak, S. Bogomolov, and T. T. Johnson, “Hyst: a source transformation and translation tool for hybrid automaton models,” in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pp. 128–133, ACM, 2015.

[32] S. Bak and T. T. Johnson, “Periodically-scheduled controller analysis using hybrid systems reachability and continuization,” in *Real-Time Systems Symposium, 2015 IEEE*, pp. 195–205, IEEE, 2015.

[33] M. Berz and K. Makino, “Verified integration of odes and flows using differential algebraic methods on high-order taylor models,” *Reliable Computing*, vol. 4, no. 4, pp. 361–369, 1998.

[34] K. Makino and M. Berz, “Rigorous integration of flows and odes using taylor models,” in *Proceedings of the 2009 conference on Symbolic numeric computation*, pp. 79–84, ACM, 2009.

[35] X. Chen, E. Abraham, and S. Sankaranarayanan, “Taylor model flowpipe construction for non-linear hybrid systems,” in *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pp. 183–192, IEEE, 2012.

[36] X. Chen, E. Ábrahám, and S. Sankaranarayanan, “Flow*: An analyzer for non-linear hybrid systems,” in *International Conference on Computer Aided Verification*, pp. 258–263, Springer, 2013.

[37] G. Frehse, R. Kateja, and C. Le Guernic, "Flowpipe approximation and clustering in space-time," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pp. 203–212, ACM, 2013.

[38] J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Arechiga, "Simulation-guided lyapunov analysis for hybrid dynamical systems," in *Proceedings of the 17th international conference on Hybrid systems: computation and control*, pp. 133–142, ACM, 2014.

[39] H. Ren and R. Kumar, "Step simulation/overapproximation-based verification of nonlinear deterministic hybrid system with inputs," *IFAC-PapersOnLine*, vol. 48, no. 27, pp. 21–26, 2015.

[40] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT press, 1999.

[41] M. Kacprzak, A. Lomuscio, A. Niewiadomski, W. Penczek, F. Raimondi, and M. Szreter, "Comparing bdd and sat based techniques for model checking chaum's dining cryptographers protocol," *Fundamenta Informaticae*, vol. 72, no. 1-3, pp. 215–234, 2006.

[42] S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke, "Reachability for linear hybrid automata using iterative relaxation abstraction," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 287–300, Springer, 2007.

[43] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," *Formal Methods in System Design*, vol. 46, no. 2, pp. 105–134, 2015.

[44] G. Frehse, "Phaver: algorithmic verification of hybrid systems past hytech," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 3, pp. 263–279, 2008.

[45] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer, "How to model and prove hybrid systems with keymaera: a tutorial on safety," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 1, pp. 67–91, 2016.

[46] T. T. Johnson and S. Mitra, “Anonymized reachability of hybrid automata networks,” in *International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 130–145, Springer, 2014.

[47] S. N. Krishna and A. Trivedi, “Hybrid automata for formal modeling and verification of cyber-physical systems,” *arXiv preprint arXiv:1503.04928*, 2015.

[48] P. S. Duggirala, C. Fan, M. Potok, B. Qi, S. Mitra, M. Viswanathan, S. Bak, S. Bogomolov, T. T. Johnson, L. V. Nguyen, *et al.*, “Tutorial: software tools for hybrid systems verification, transformation, and synthesis: C2e2, hyst, and tulip,” in *Control Applications (CCA), 2016 IEEE Conference on*, pp. 1024–1029, IEEE, 2016.

[49] G. Morbé, F. Pigorsch, and C. Scholl, “Fully symbolic model checking for timed automata,” in *International Conference on Computer Aided Verification*, pp. 616–632, Springer, 2011.

[50] E. Asarin, O. Bournez, T. Dang, and O. Maler, “Approximate reachability analysis of piecewise-linear dynamical systems,” in *International Workshop on Hybrid Systems: Computation and Control*, pp. 20–31, Springer, 2000.

[51] A. Donzé and O. Maler, “Systematic simulation using sensitivity analysis,” in *International Workshop on Hybrid Systems: Computation and Control*, pp. 174–189, Springer, 2007.

[52] E. M. Clarke and P. Zuliani, “Statistical model checking for cyber-physical systems,” in *International Symposium on Automated Technology for Verification and Analysis*, pp. 1–12, Springer, 2011.

[53] P. S. Duggirala, S. Mitra, and M. Viswanathan, “Verification of annotated models from executions,” in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, p. 26, IEEE Press, 2013.

[54] Z. Huang and S. Mitra, “Computing bounded reach sets from sampled simulation traces,” in *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pp. 291–294, ACM, 2012.

[55] Z. Huang and S. Mitra, "Proofs from simulations and modular annotations," in *Proceedings of the 17th international conference on Hybrid systems: computation and control*, pp. 183–192, ACM, 2014.

[56] M. Althoff, "An introduction to cora 2015.," in *ARCH@ CPSWeek*, pp. 120–151, 2015.

[57] M. Althoff and D. Grebenyuk, "Implementation of interval arithmetic in cora 2016.," in *ARCH@ CPSWeek*, pp. 91–105, 2016.

[58] M. Zamani, G. Pola, M. Mazo, and P. Tabuada, "Symbolic models for nonlinear control systems without stability assumptions," *IEEE Transactions on Automatic Control*, vol. 57, no. 7, pp. 1804–1809, 2012.

[59] J. Liu and N. Ozay, "Finite abstractions with robustness margins for temporal logic-based control synthesis," *Nonlinear Analysis: Hybrid Systems*, vol. 22, pp. 1–15, 2016.

[60] A. Girard, G. Pola, and P. Tabuada, "Approximately bisimilar symbolic models for incrementally stable switched systems," *IEEE Transactions on Automatic Control*, vol. 55, no. 1, pp. 116–126, 2010.

[61] G. Wood and B. Zhang, "Estimation of the lipschitz constant of a function," *Journal of Global Optimization*, vol. 8, no. 1, pp. 91–103, 1996.

[62] A. Strzeboński, "Computation with semialgebraic sets represented by cylindrical algebraic formulas," in *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pp. 61–68, ACM, 2010.

[63] K. Mørken, "Numerical solution of differential equations." http://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h10/kompendiet/kap13.pdf, 2010.

[64] D. Angeli, "A lyapunov approach to incremental stability properties," *IEEE Transactions on Automatic Control*, vol. 47, no. 3, pp. 410–421, 2002.

[65] B. S. Rüffer, N. van de Wouw, and M. Mueller, "From convergent dynamics to incremental stability," in *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pp. 2958–2963, IEEE, 2012.

[66] "Alglib 2.0." http://www.alglib.net.

[67] "Clipper." http://www.angusj.com/delphi/clipper.php.

[68] "Gnuplot." http://www.gnuplot.info.

[69] M. Althoff, S. Yaldiz, A. Rajhans, X. Li, B. H. Krogh, and L. Pileggi, "Formal verification of phase-locked loops using reachability analysis and continuization," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pp. 659–666, IEEE, 2011.

[70] S. Rasmussen, K. Kalyanam, and D. Kingston, "Field experiment of a fully autonomous multiple uav/ugs intruder detection and monitoring system," in *Unmanned Aircraft Systems (ICUAS), 2016 International Conference on*, pp. 1293–1302, IEEE, 2016.

[71] D. Kingston, S. Rasmussen, and L. Humphrey, "Automated uav tasks for search and surveillance," in *Control Applications (CCA), 2016 IEEE Conference on*, pp. 1–8, IEEE, 2016.

[72] A. Tarski, "A decision method for elementary algebra and geometry," in *Quantifier elimination and cylindrical algebraic decomposition*, pp. 24–84, Springer, 1998.

[73] H. Enderton and H. B. Enderton, *A mathematical introduction to logic*. Academic press, 2001.

[74] G. Collins, "Quantifier elimination in the elementary theory of real closed fields by cylindrical algebraic decomposition, vol. 33," *Lecture Notes in Computer Science*, pp. 134–183.

[75] V. Weispfenning, "The complexity of linear problems in fields," *Journal of symbolic computation*, vol. 5, no. 1-2, pp. 3–27, 1988.

[76] V. Weispfenning, "The complexity of almost linear diophantine problems," *Journal of Symbolic Computation*, vol. 10, no. 5, pp. 395–403, 1990.

[77] G. E. Collins and H. Hong, "Partial cylindrical algebraic decomposition for quantifier elimination," in *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pp. 174–200, Springer, 1998.

[78] M. Jirstrand, "Nonlinear control system design by quantifier elimination," *Journal of Symbolic Computation*, vol. 24, no. 2, pp. 137–152, 1997.

[79] G. Lafferriere, G. J. Pappas, and S. Yovine, "Symbolic reachability computation for families of linear vector fields," *J. of Symbolic Computation*, vol. 11, pp. 1–23, 2001.

[80] H. Anai and V. Weispfenning, "Reach set computations using real quantifier elimination," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 63–76, Springer, 2001.

[81] A. Tiwari, "Approximate reachability for linear systems," in *HSCC*, vol. 2623, pp. 514–525, Springer, 2003.

[82] A. Taly and A. Tiwari, "Switching logic synthesis for reachability," in *Proceedings of the tenth ACM international conference on Embedded software*, pp. 19–28, ACM, 2010.

[83] T. Sturm and A. Tiwari, "Verification and synthesis using real quantifier elimination," in *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, pp. 329–336, ACM, 2011.

[84] M. Kwiatkowska, A. Mereacre, N. Paoletti, and A. Patanè, "Synthesising robust and optimal parameters for cardiac pacemakers using symbolic and evolutionary computation techniques," in *International Workshop on Hybrid Systems Biology*, pp. 119–140, Springer, 2015.

[85] A. Platzer and J.-D. Quesel, "Keymaera: A hybrid theorem prover for hybrid systems (system description)," in *International Joint Conference on Automated Reasoning*, pp. 171–178, Springer, 2008.

[86] D. Stewart, M. W. Whalen, D. Cofer, and M. P. Heimdahl, “Architectural modeling and analysis for safety engineering,” in *International Symposium on Model-Based Safety and Assessment*, pp. 97–111, Springer, 2017.

[87] T. A. Henzinger, M. Minea, and V. Prabhu, “Assume-guarantee reasoning for hierarchical hybrid systems,” in *HSCC*, vol. 2034, pp. 275–290, Springer, 2001.

[88] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. Pasareanu, A. Podelski, and T. Strump, “Assume-guarantee abstraction refinement meets hybrid systems,” in *Haifa verification conference*, pp. 116–131, Springer, 2014.

[89] H. Mehrpouyan, D. Giannakopoulou, G. Brat, I. Y. Tumer, and C. Hoyle, “Complex engineered systems design verification based on assume-guarantee reasoning,” *Systems Engineering*, vol. 19, no. 6, pp. 461–476, 2016.

[90] D. Cofer, A. Gacek, S. Miller, M. Whalen, B. LaValley, and L. Sha, “Compositional verification of architectural models,” *NASA Formal Methods*, pp. 126–140, 2012.

[91] N. Eén and N. Sörensson, “Temporal induction by incremental sat solving,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003.

[92] T. Kahsai and C. Tinelli, “Pkind: A parallel k-induction based model checker,” *arXiv preprint arXiv:1111.0372*, 2011.

[93] E. Ghassabani, M. Whalen, A. Gacek, and R. Collins, “Efficient generation of all minimal inductive validity cores,” FMCAD, 2017.

[94] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data flow programming language lustre,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[95] D. Jovanović and L. De Moura, “Solving non-linear arithmetic,” *ACM Communications in Computer Algebra*, vol. 46, no. 3/4, pp. 104–105, 2013.

[96] A. Champion, R. Delmas, and M. Dierkes, "Generating property-directed potential invariants by quantifier elimination in a k-induction-based framework," *Science of Computer Programming*, vol. 103, pp. 71–87, 2015.

[97] "Antlr v4." `http://www.antlr.org`.

[98] "Osate 2.2.0." `http://osate.github.io`.

[99] M. S. Fadali and A. Visioli, *Digital control engineering: analysis and design*, ch. 7. State-Space Representation. Academic Press, 2012.