Verification Condition Generation for Hybrid Systems

Xian Li and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—Verification condition generators (VCGs) can reduce overall correctness statements about sequential programs to verification conditions (VCs) that can then be proved independently by automatic theorem provers like SMT solvers. SMT solvers became not only more powerful in recent years in that they can now solve much bigger problems than before, they can now also solve problems of less restricted logics, for example, by covering non-linear arithmetic as required by some hybrid systems. However, there is so far still no VCG procedure that could generate VCs of hybrid programs for these SMT solvers.

We therefore propose in this paper a first VCG procedure for hybrid systems that is based on induction proofs on the strongly connected components (SCCs) of the underlying state transition diagrams. Given the right invariants for a safety property, the VCs can be automatically generated for the considered hybrid system. The validity of the VCs is then independently proved by SMT solvers and implies the correctness of the considered safety property.

I. INTRODUCTION

The behavior of a hybrid system [1–4] consists of both discrete and continuous transitions which are often the result of considering a discrete reactive system in a continuous (physical) environment. Within a discrete state, differential equations determine the values of the variables until some release condition becomes true to change the discrete state. A discrete transition performs then a sequence of assignments to the variables and determines a new set of differential equations and a new release condition.

Hybrid systems play an important role in model-based design which is common in engineering disciplines: Here, a model of the system to be built is made, simulated, analyzed, and optimized before actually building the system – preferably with the help of computer-aided design tools. To this end, hybrid automata [1, 5, 6] are often used as underlying semantic model for hybrid systems that is supported by many languages and tools [7, 8]. While this semantic model is widely accepted, it is less clear how to define comfortable modeling languages that lend themselves well for both the discrete and the continuous facets of the hybrid system, and that offer a realistic kind of concurrency which can be used for compositional reasoning [9, 10]. The semantics of these languages is often difficult to describe and it has to deal with special phenomena like unwanted Zeno behaviors [11, 12].

For analyzing hybrid systems, simulation is the first technique to be chosen, but already here, tools are challenged by numeric accuracy problems, in particular, by the event detection (often also called the zero-crossing problem) [13, 14] to make sure that no trigger event for a discrete transition is missed between two steps of the simulator. As a new solution, non-standard analysis has been recently suggested to deal with these problems [15].

Since simulation can only deal with a fixed input trace, formal verification of hybrid systems is mandatory in many application areas to guarantee the absence of errors (however, see also [16]). Unfortunately, even very simple verification problems like deciding the reachability of states can only be decided for severely restricted hybrid automata like timed automata [17] and initialized rectangular automata [18] (see also [1, 5, 6, 19]). Classic tools like HyTech1 [20–23] and PHAVer2 [10, 24] therefore approximate the state sets by polyhedra where each polyhedron is represented as a conjunction of inequalities. This works fine for linear hybrid automata [6, 23] since polyhedra are closed under their state transitions. Polyhedra are also used (by means of flowpipe approximations) by tools like Checkmate3 [25], SpaceEX4 [26], and d/dt5 [27, 28] that even allow linear differential equations in the hybrid automata.

The complexity of operations on polyhedra grows however exponentially in the number of variables which often limits the applicability of these tools. Hence, abstraction techniques frequently have to be used for the verification of hybrid systems. An abstraction is thereby a simplified (maybe even finite) model whose correctness implies the correctness of the original system. HyTech already used phase portrait approximations where differential equations were replaced by lower and upper bounds of derivatives [20–23]. Also predicate abstractions have been used for hybrid systems [29, 30] where the abstractions to boolean predicates are often developed by counterexamples (CEGAR). Finally, abstractions were also obtained by computing simulation relations: For timed automata and o-minimal hybrid systems, one can even compute the greatest simulation relation and use its quotient as discrete abstract system [31–33].

For reachability and safety verification problems of hybrid systems, the main line of research concentrated so far on

1http://embedded.eecs.berkeley.edu/research/hytech
2http://verimag.imag.fr/~frehse/phaver_web/
3http://users.ece.cmu.edu/~krugl/checkmate
4http://spaceex.imag.fr
5http://verimag.imag.fr/~tdang/ddt.html
model-checking procedures that follow a brute-force approach to compute or approximate the reachable states. Many of the available optimizations of model-checking discrete systems have also been adapted for hybrid systems like symbolic SAT/SMT-based [34–36], interpolation-based [37], IC3-based [38], and many abstraction refinement techniques [39–41].

A general alternative to model-checking are deductive techniques [42–44] that are based on automated or interactive theorem provers like HOL [45], Isabelle [46], PVS [47], ACL2 [48], and many others. The user interacts with these systems by setting up proof goals and applying proof rules until a proof is finally obtained. The mentioned systems are general theorem provers⁶ and support undecidable and very expressive higher order logics. These theorem provers were used for the verification of many kinds of systems, in particular, many kinds of software systems [50].

For software verification, the most popular techniques are based on Hoare calculus [51–53] where program statements \( S \) are enclosed in pre- and postconditions to form proof goals \( \{ \varphi \} S \{ \psi \} \) which state that if \( \varphi \) holds at starting time of \( S \) and if \( S \) will terminate, then \( \psi \) holds at termination time of \( S \). The Hoare calculus then provides for every statement of the considered programming language a decomposition rule to reduce a proof goal for that statement to proof goals using only its sub-statements. In particular, loops are thereby reduced by Hoare’s famous invariant rule: Given a loop statement \( \text{while}(\sigma) S \) with its invariant \( I \), one can deduce from a proof of \( \{ \sigma \land I \} S \{ I \} \) that \( \{ I \} \text{while}(\sigma) S \{ \neg \sigma \land I \} \) holds. Since invariants, as well as pre- and postconditions usually have to be provided by the user, the approach is in general considered to be interactive.

Verification condition generators (VCGs) [54–56] are used to aggressively apply all rules of the Hoare calculus to a program whose statements are annotated with pre- and postconditions so that the validity of the obtained VCs implies the correctness of the given proof goal \( \{ \varphi \} S \{ \psi \} \). In principle, it is sufficient to only provide the right loop invariants, since intermediate assertions can be computed automatically by weakest preconditions. VCGs can work very fast, since they just make a linear pass over the program text, and the obtained formulas can then be checked one after the other. This way, the overall verification problem is split into two independent parts: first, generating the VCs, and second proving them which can be done by different tools and also in parallel.

As already explained, many hybrid systems are used in safety-critical applications, and therefore, their formal verification is of high interest. In particular, there has been considerable progress on SMT solvers that can deal with arithmetic formulas like MathSAT [57, 58], CVC [59, 60], ICS [61], iSAT [62, 63], HySAT [34–36, 64], ABSOLVER [65], BACH [66], Z3 [67], MetiTarski [68], HybridSAL [69], and on special techniques for non-linear arithmetics [70, 71]. These tools are already used for the verification of (linear) hybrid systems, mainly employed by bounded-model checking where safety properties are unrolled for some finite number of discrete transitions. However, there is so far no VCG procedure to generate proof goals of correctness statements about hybrid programs, mainly because of the lack of modeling languages for hybrid systems with structured programming constructs.

In this paper, we therefore propose such a VCG procedure for hybrid programs and therefore open a new branch for the verification of hybrid systems. To this end, we consider our Esterel-like language Quartz [72] with its recent extension to hybrid systems [73]. After first experimenting with the extension of Hoare calculi to synchronous languages [74], we found that the abstraction to logical time as done by synchronous languages introduces irreducible state machines (which is also the case for any other state-based languages). It is however well-known that sequential programs with structured statements yield reducible flow graphs [75, 76]. Hence, any state-based reasoning cannot follow a decomposition along structured programming language statements, which makes Hoare calculus not applicable. However, VCG is not limited to Hoare calculus: Indeed, early work done by Floyd [77] did not consider structured programs, and used inductive assertions instead of loop invariants.

Hence, we employ Floyd’s induction-based approach for the generation of verification conditions of hybrid Quartz programs. Our VCG procedure consists of two steps, where the first step consists of computing for a hybrid Quartz program its extended finite state machine (EFSM) according to the operational semantics of the language (see Fig. 3 as an example). This EFSM has one node for every discrete control flow state of the program, and every node is labeled with a set of guarded actions that encode the dataflow of that node, i.e., assignments to both discrete and continuous variables. Transitions between nodes are labeled with trigger conditions, and every transition corresponds with one macro step of the synchronous reactive program. In the second step, the user has to provide inductive assertions (invariants) for each safety property and each strongly connected component (SCCs) of the generated EFSM. Our VCG procedure applies then the induction rules and generates this way proof goals that correspond with induction steps and bases.

Induction-based approaches without VCG have already been applied to the verification of hybrid systems: A similar idea has been presented by barrier certificates [78] which ask for a function that maps safe and unsafe states to non-negative and negative real numbers, respectively, such that the transitions do not make the function’s value negative. The method is therefore in the spirit of Lyapunov functions for proving the stability of ordinary differential equations. An improved induction approach has been presented by Platzer’s differential induction [79–81] that does not require to solve the differential equations and uses the directional derivative instead. Our method focuses however on the decomposition of the proof goals (and can be combined with differential induction). We do also not consider the generation of inductive invariants like [82] in this paper and assume that these are given by the user.

⁶There are however also specialized theorem provers for hybrid systems like KeYmaera [3, 49].
The outline of the rest of the paper is as follows: in the next section, we briefly describe our modeling language Quartz, its extension to hybrid systems, and the representation of Quartz programs by EFSMs. Section III introduces the automated VCG approach based on inductive proofs over the SCCs of the EFSM. Experimental results are described in Section IV. The paper will be concluded in Section V.

II. THE QUARTZ LANGUAGE AND EFSMs

A. The Quartz Language

Quartz [72] is a programming and modeling language that has been originally derived from the Esterel language [83]. The execution of a Quartz program is defined by so-called micro and macro steps, where macro steps are defined by pause statements in the program. A macro step consists of reading new inputs, and executing the code starting from the active pause statements to the next reached pause statements as the reaction of the program to the inputs. Due to parallel statements \((S_1 ∥ S_2)\), more than one pause statement may be active at a time. Each pause statement introduces a control-flow location that is given a name by the compiler or the programmer.

The flow statement \(\text{flow}\{S_1;...;S_n\}\text{until}(\sigma)\) generalizes a pause statement in that it extends the discrete transition by a continuous transition where the continuous variables behave according to the flow assignments \(S_1;...;S_n\). These are either assignments to a variable \(x \leftarrow \tau\) or to its derivation \(\text{drv}(x) \leftarrow \tau\). In contrast to the discrete transitions, continuous transitions require physical time and terminate as soon as the condition \(\sigma\) becomes true.

The continuous transition of the macro step starts with the variable environment determined by the discrete (immediate) assignments as initial values. To distinguish between the initial and current values of variables on a continuous transition, a new operator \(\text{cont}\{x\}\) is introduced: \(x\) refers to the initial value of \(x\) and \(\text{cont}(x)\) refers to its current value on the continuous evolution.

Due to space limitations, we cannot give a complete overview of the language here, and refer to [72, 73] for more details. Instead, we just list some of the statements used in this paper and give an idea of their meaning:

- \(x = \tau\) and \(\text{next}(x) = \tau\) (immed./delayed assignments)
- \(\text{assume}(\varphi), \text{assert}(\varphi)\) (assumptions and assertions)
- \(\ell : \text{pause}\) (start/end of macro step)
- \(S_1;S_2\) (sequences)
- \(S_1 ∥ S_2\) (synchronous concurrency)
- \(\text{loop } S\) (iteration)
- \(\text{flow }\{S_1;...;S_n\}\text{until}(\sigma)\) (flow statement)
- \(M(\{\text{params}\})\) (module call)

The operational semantics of Quartz is defined by two sets of SOS (structural operational semantics) rules [72]: SOS reaction rules define computation of the discrete outputs within a macro step, and SOS transition rules determine the movement of the control flow from the current to the next macro step.

B. EFSMs of Quartz Programs

Our Averest system\(^7\) provides algorithms that translate a Quartz program to a set of guarded actions \(G\) [72]. Guarded actions are pairs \((\gamma, \alpha)\) consisting of a trigger condition \(\gamma\) and an action \(\alpha\) and express that \(\alpha\) is executed whenever \(\gamma\) holds. Actions are thereby immediate \(x = \tau\) and delayed assignments \(\text{next}(x) = \tau\) (for discrete transitions), flow assignments \(x \leftarrow \tau\) and \(\text{drv}(x) \leftarrow \tau\) (for continuous transitions), assumptions \(\text{assume}(\varphi)\), assertions \(\text{assert}(\varphi)\), and release conditions \(\text{release}(\varphi)\) (to set up verification goals) where \(\gamma, \tau, \varphi\) are program expressions.

Applying SOS rules, we can also directly generate extended finite state machines (EFSMs) for Quartz programs. Such EFSMs have one node for every reachable control flow state of the program, and every node is labeled with a set of guarded actions that encode the dataflow of that node. For every node \(s_i\) of a EFSM, we consider two sets of guarded actions: \(D_d(s_i)\) and \(D_c(s_i)\) that encode the discrete and continuous guarded actions, respectively. For any pair of nodes \((s_i, s_{i+1})\), there is moreover a path condition \(\varphi(s_i, s_{i+1})\) that must hold to take the transition from \(s_i\) to \(s_{i+1}\). Note that every transition corresponds to one macro step of the synchronous reactive program.

EFSMs are frequently used as a convenient formalism to describe programs with potentially infinite data types. Additionally, this description is useful for deductive verification in that one can decompose the proof goal with respect to the reachable control flow states.

C. Symbolic Transition Relation of the EFSMs

The EFSM of a Quartz program has finitely many nodes and transitions, but describes an infinite transition system that has discrete and continuous transitions occurring in pairs. States of this transition system are variable environments \(\xi : \mathcal{V} → \mathbb{R}\) that map variables to values. To describe the continuous states along continuous transitions, we consider however variable environments \(\chi : \mathcal{V} → (\mathbb{R} → \mathbb{R})\) that map variables to functions over time. A run \(\pi\) through this transition system is described by an infinite sequence of triples \((\xi_i, \chi_i, t_i)\) for \(i \in \mathbb{N}\), where

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

Since discrete states of the transition system are associated with variable environments \(\xi : \mathcal{V} → \mathbb{R}\), we define

- \([x]_\xi := \xi(x)\) for variables \(x\)

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

- \([x]_{\chi,t} := (\chi(x), t)\) for variables \(x\)

\(^7\)http://www.averest.org
The SafePath default assignment in every macro step (compilers can add environment (input variables). In the following, we assume of the program. To express the VCs in a readable way, we make use when needed).

fication conditions (VCs) for a safety property and a Quartz

uous transitions in refers to the starting node

we have to consider three variable environments: (1) \( \xi_i \) which refers to the starting node \( s_i \), (2) \( \chi_i \) which describes the continuous transitions in \( s_i \), and (3) \( \xi_{i+1} \) which refers to the target node \( s_{i+1} \). We then define \( \{(s_i, s_{i+1})\}_{i \in \mathbb{N}} \xi_i, \chi_i, \xi_{i+1} \) accordingly, so that whenever it holds, the triple \( (\xi_i, \chi_i, \xi_{i+1}) \) describes a discrete transition followed by the cumulated continuous transitions symbolically represented in \((s_i, s_{i+1})\). To that end, \( \chi_i \) must map the variables to the solution of the corresponding ordinary differential equation system (ODE) imposed by the continuous transition.

### III. VCG using SCC-Invariants

In the following, we will explain how to generate the verification conditions (VCs) for a safety property and a Quartz program. To express the VCs in a readable way, we make use of the SafePath predicate as explained below.

#### A. The SafePath Predicate

In every macro step, the values of variables are either determined by assignments (for local/output variables) or by the environment (input variables). In the following, we assume that for every local/output variable there is exactly one enabled assignment in every macro step (compilers can add default actions when needed).

\( \psi_i \) is a continuous invariant for node \( s_i \), if and only if the formula \( \text{cont}_\psi(\psi_i) \) is valid, where the predicate \( \text{cont}_\psi \) demands that \( \psi_i \) must hold during the continuous phase in the node \( s_i \). Because of the delayed assignments, \( D_d(s_i) \) has occurrences of the variables \( \vec{x} \) and also of \( \text{next}(\vec{x}) \), while \( D_c(s_i), \varphi(s_i, s_{i+1}), \) and \( \psi(s_i) \) have only occurrences of the variables \( x \). Finally, we write \( [D_d(s_i)]x, [D_c(s_i)]x \), [\( \varphi(s_i, s_{i+1}) \)]x, and [\( \psi_i \)]x for the formulas that are obtained by replacing all occurrences of variables \( \vec{x} \) and \( \text{next}(\vec{x}) \) by new variables \( \vec{x}_i \) and \( \vec{x}_{i+1} \) in the formulas \( D_d(s_i), D_c(s_i), \varphi(s_i, s_{i+1}), \) and \( \psi_i \) respectively.

For every path \( s_0, \ldots, s_{n+1} \) of the EFSM starting from the discrete state of node \( s_0 \) and ending in the discrete state of node \( s_{n+1} \), we define the following abbreviation:

\[
\text{SafePath}((s_0, \ldots, s_{n+1}), (\psi_0, \ldots, \psi_n), \alpha, \beta, \gamma) \iff \left( \bigwedge_{i=0}^{n+1} [D_d(s_i)]x, [D_c(s_i)]x, \varphi(s_i, s_{i+1}), [\psi_i]x \right) \land \left( \bigwedge_{i=0}^{n} \text{cont}_\psi([D_c(s_i)]x) \right) \land \left( \bigwedge_{i=0}^{n} \varphi(s_i, s_{i+1})x \right) \land \left( \bigwedge_{i=0}^{n} [\psi_0]x \right) \land \left( \bigwedge_{i=1}^{n} \text{cont}_\psi([\psi_i]x) \right) \land \left( \bigwedge_{i=1}^{n} [\beta]x \right) \land \left( \gamma \right) x
\]

The SafePath predicate formalizes the following two statements:

- The variables \( \vec{x}_0, \ldots, \vec{x}_{n+1} \) have values that are possible on a path from the discrete state of node \( s_0 \) to the discrete state of node \( s_{n+1} \), i.e., all discrete actions \( [D_d(s_i)]x, [D_c(s_i)]x, \varphi(s_i, s_{i+1})x, [\psi_i]x, [\beta]x, [\gamma]x \) and continuous actions \( [D_c(s_i)]x \) of node \( s_i \) are respected as well as all transition conditions \( [\varphi(s_i, s_{i+1})]x \).

- If the given formulas \( \alpha, \beta, \gamma \) hold in the discrete state and the continuous phase of node \( s_0 \), respectively, then (1) for each node \( s_i \) between the head and tail nodes \( s_0 \) and \( s_{n+1} \), formula \( \psi_i \) holds during the continuous phase of the node \( s_i \) and formula \( \beta \) holds in the discrete state \( s_i \); and (2) formula \( \gamma \) holds finally in the discrete state of node \( s_{n+1} \).

The SafePath predicate is comparable with a Hoare calculus triple \( (\alpha)S(\gamma) \) which intuitively says that the post-condition \( \gamma \) must hold after executing the program \( S \) that satisfies the pre-condition \( \alpha \). It is also comparable with dynamic logic expressions \( \alpha \rightarrow [S] \gamma \), but additionally expresses properties that must hold during the execution.

#### B. Paths without Cycles

If the corresponding EFSM contains only paths without cycles, then the following simple approach can be used to generate VCs that ensure the validity of a safety property \( \Phi \): For every path with \( s_0, \ldots, s_{n+1} \) of the EFSM starting in the discrete state of the root node \( s_0 \) and ending in the discrete state of the terminal node \( s_{n+1} \), we construct the following VC using continuous invariants \( \Psi_i \) for the states \( s_i \):

\[
\Phi \land \text{cont}_\psi([\psi_0]x) \land \bigwedge_{i=0}^{n} \Psi_i \rightarrow \Phi \land \text{SafePath}((s_0, \ldots, s_{n+1}), (\psi_0, \ldots, \psi_n), \gamma_0, \Phi, \gamma_{n+1})
\]

where we have to prove that \( \Psi_i \rightarrow \Phi \) for the continuous phase of each node \( s_i \). \( \gamma_0 \) is either the safety property \( \Phi \) or the continuous invariant \( \Psi_0 \) of the node \( s_0 \) where the path comes from. Similarly, \( \gamma_{n+1} \) is either the safety property \( \Phi \) or the continuous invariant \( \Psi_{n+1} \) of the node \( s_{n+1} \) where the path ends.

#### C. Paths with Cycles

If the corresponding EFSM contains paths with cycles, we can first use the method of the previous section to check its paths without cycles. However, for the remaining paths with cycles, the above approach cannot be used. Instead, we generate VCs that set up proof goals for an induction proof (which is in the spirit of Floyd’s inductive assertions [77]).

For the entire proof, we call a SCC a trivial SCC if it contains only a single state without a self-loop, while all the other SCCs are called nontrivial. The root SCC only contains the root state \( s_0 \), and it is provably always a trivial SCC for the EFSMs generated for Quartz programs. The terminal SCCs are those nontrivial SCCs without outgoing edges. For example, in Fig. 1, the states in the same dashed rectangle form a nontrivial SCC where \( s_{10} \) is the only terminal SCC. The remaining states \( s_0, s_5 \) and \( s_8 \) are trivial SCCs and \( s_{10} \) is the root SCC.

For each nontrivial SCC \( C_i \) and its invariant \( I_i \), we have to generate a VC for the induction base and another one for the
induction step. The induction base of SCC $C_i$ is established as the following: 
\[ \phi_i \iff \bigwedge_{k=0}^{i} (\Psi_i \rightarrow \Phi) \wedge \]
\[ \bigwedge_{(s_k, s_{k+1}) \in \text{Paths}(C_i)} \text{SafePath}((s_k, s_{k+1}), (\Psi_{k+1} \wedge \text{true}, \Psi_{k+1})) \]

where Paths($C_i$) denotes all the finite acyclic paths from other nontrivial SCCs or the root SCC to SCC $C_i$. If node $s_k$ belongs to SCC $C_k$, then $\Psi_k$ is either the SCC-invariant $\mathcal{I}_k$ or the continuous invariant $\Psi_k$ of the node where the path comes from, or the safety property $\Phi$ in case of the root SCC. $\mathcal{I}_i$ is either the SCC-invariant $\mathcal{I}_i$ or the continuous invariant $\Psi_i$ of the node where the path ends.

The induction step of SCC $C_i$ is established by the following VC: 
\[ \phi_i' \iff \bigwedge_{k=0}^{i} (\Psi_i \rightarrow \mathcal{I}_i) \wedge \]
\[ \bigwedge_{(s_k, s_{k+1}) \in \text{Paths}(in C_i)} \text{SafePath}((s_k, s_{k+1}), (\Psi_{k+1} \wedge \text{true}, \Psi_{k+1})) \]

where Paths(in $C_i$) denotes all the single transitions from the discrete state of one node in $C_i$ to the discrete state of one of its successor nodes in $C_i$. Each path has only two nodes, so $\beta = \text{true}$ in the above formula. $\mathcal{I}_i$ is either the SCC-invariant $\mathcal{I}_i$ or the continuous invariant $\Psi_i$ of node $s_k$. Similarly, $\mathcal{I}_{k+1}$ is either the SCC-invariant $\mathcal{I}_i$ or the continuous invariant $\Psi_{k+1}$ of node $s_{k+1}$.

Taking the EFSM in Fig. 1 as an example, to set up the induction base for $C_3$, we have to consider all paths from other nontrivial SCCs to $C_3$, i.e., path $s_4, s_8, s_9$, as well as all finite paths from the root SCC to $C_3$ that do not traverse any other nontrivial SCCs, i.e., path $s_0, s_9$ and path $s_0, s_8, s_9$.

For the EFSM with $n$ nontrivial SCCs, the overall proof goal is the following:
\[ \Phi_{\mathcal{E}_0} \wedge \text{conty}([\Psi_{0}]_{\mathcal{E}}) \wedge \bigwedge_{i=1}^{n} (\phi_i \wedge \phi_i') \wedge \bigwedge_{i=1}^{n} (\mathcal{I}_i \rightarrow \Phi) \]

where we have to prove (in addition to the base case and the inductive step) that (1) the safety property $\Phi$ and the continuous invariant $\Psi_0$ hold in the discrete state and the continuous phase of the root SCC $s_0$, respectively, and (2) $\mathcal{I}_i \rightarrow \Phi$ for each nontrivial SCC invariant $\mathcal{I}_i$. The overall proof goal may yield several VCs, depending on how many transitions and paths exist.

D. Correctness of the Approach

The overall task is to prove that a given safety property $\Phi$ holds for all computations paths. Consider therefore any (arbitrary) computation path $\pi$ of the EFSM where $\pi(t)$ is the node of the path that is reached at step $t \in \mathbb{N}$. We can split the path into a finite set of possibly empty subsequences $\pi = \pi_0 \pi_1 \ldots \pi_n$ where both the start and end states of $\pi_i$ are the discrete states of some node. $\pi_0$ is the finite path between the root SCC and a nontrivial SCC, the finite paths $\pi_{2i}$ with nonzero even indices are paths between two nontrivial SCCs, the finite paths $\pi_{2i+1}$ with odd indices are paths inside a single nontrivial SCC, and $\pi_n$ is either an infinite suffix that is completely contained in
an SCC (if \( n \) is odd) or (if \( n \) is even) it is a finite path ending in a terminal SCC of the EFSM.

To see that \( \Phi \) holds on every node of the path \( \pi \), it is sufficient to observe the following:

- For paths between the root SCC \( s_0 \) and an arbitrary node \( s_k \in C_k \), we have proven by our approach that
  1. SafePath(\( \pi_0, (\Psi_0, \ldots, \Psi_k, \gamma_0, \Phi, \gamma_k) \)),
  2. \( \Phi x_0 \land \cont

- For a path \( \pi_{2i} \) between two arbitrary nodes \( s_k \in C_k \) and \( s_k \in C_k \), where both \( C_k \) and \( C_k \) are non-trivial SCCs, we have proven by our approach that
  1. SafePath(\( \pi_{2i}, (\Psi_k, \ldots, \gamma), \gamma_k, \Phi, \gamma_k) \)),
  2. \( I_k \rightarrow \Phi \land \cont \rightarrow \Phi \land \cont \rightarrow \Phi \) which means if \( I_k \) is the invariant of SCC \( C_k \), then \( \Phi \) holds on every state of the path \( \pi_{2i} \).

- For a path \( \pi_{2i+1} \) between two arbitrary nodes \( s_k \) and \( s_{k+1} \) inside an arbitrary nontrivial SCC \( C_k \), we have proven by our approach that
  1. SafePath(\( \pi_{2i+1}, (\Psi_k, \ldots, \gamma), \gamma_k, true, \gamma_k+1) \)),
  2. \( I_k \rightarrow \Phi \land \cont \rightarrow \Phi \land \cont \rightarrow \Phi \) holds. Hence, \( \Phi \) holds on every state of the path \( \pi_{2i+1} \).

Note that the latter argumentation is also valid if \( \pi_{2i+1} \) is an infinite suffix of the considered computation path \( \pi \). Note further that the completeness of our approach follows by induction over the SCCs: if we consider any path \( \pi \), it can be partitioned into sequences that correspond to the SCCs.

IV. Experiments

We illustrate our procedure with a parameterized water tank system in this section.

A. Hybrid Quartz Program WaterTank with its EFSM

As shown in Fig. 2, the water tank regulates water level \( y \) by filling or emptying the water tank. The initial water level is \( y = 5 \). \( inV \) and \( outV \) are parameters that describe the water level changes, and the module can either set \( \text{drv}(y) \leftarrow \text{inV} \) or \( \text{drv}(y) \leftarrow \text{outV} \) to fill or empty the water tank. The reaction of the system is delayed, so that both the filling and emptying procedures are extended by \( \text{delta} \)-unit time. Both \( \text{delta} \) and \( y \) are reals.

Note that the following formulas \( \Psi_{fill}, \Psi_{stop}, \Psi_{drain}, \Psi_{start} \) hold in mode \( fill, stop, drain, \) and \( start \), respectively.

\[ \begin{align*}
\Psi_{fill} & :\quad (y \leq 10) \\
\Psi_{stop} & :\quad (10 \leq y) \land (y \leq 10 + \text{inV} \ast \text{delta}) \\
\Psi_{drain} & :\quad (5 \leq y) \\
\Psi_{start} & :\quad (5 - \text{outV} \ast \text{delta} \leq y) \land (y \leq 5)
\end{align*} \]

The corresponding EFSM is shown in Fig. 3. It has 10 nodes in total, one trivial SCC (the root node \( s_0 \)), and one nontrivial SCC that contains the remaining 9 nodes.

B. VC Generation Using Induction over SCCs

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

\[ \Phi :\quad (5 - \text{outV} \ast \text{delta} \leq y) \land (y \leq 10 + \text{inV} \ast \text{delta}) \]

using the following continuous invariants for each node \( s_i \):

\[ \begin{align*}
\Psi_0 & :\quad \Psi_{fill} \land (5 \leq y) \\
\Psi_1 & :\quad \Psi_{fill} \land (5 - \text{outV} \ast \text{delta} \leq y) \\
\Psi_2 & :\quad ((st = stop) \rightarrow \Psi_{stop}) \land ((st = drain) \rightarrow \Psi_{drain} \land (y \leq 10 + \text{inV} \ast \text{delta})) \land ((st = start) \rightarrow \Psi_{start}) \\
\Psi_3 & :\quad \Psi_{stop} \\
\Psi_4 & :\quad ((st = drain) \rightarrow \Psi_{drain}) \land ((st = start) \rightarrow \Psi_{start}) \\
\Psi_5 & :\quad \Psi_{start} \\
\Psi_6 & :\quad 5 - \text{outV} \ast \text{delta} = y \\
\Psi_7 & :\quad ((st = stop) \rightarrow \Psi_{stop}) \land ((st = start) \rightarrow \Psi_{start}) \land ((st = drain) \rightarrow \Psi_{drain} \land (y \leq 10 + \text{inV} \ast \text{delta})) \land ((st = fill) \rightarrow (\Psi_{fill} \land (5 - \text{outV} \ast \text{delta} \leq y)) \\
\Psi_8 & :\quad \Psi_{drain} \land (y \leq 10 + \text{inV} \ast \text{delta}) \\
\Psi_9 & :\quad 5 = y
\end{align*} \]

It is easy to prove \( \Psi_i \rightarrow \Phi \) for each node \( s_i \).

We use the safety property \( \Phi \) as the SCC invariant \( I_1 \). As explained in the previous section, our tool automatically sets up the following induction base and induction step of \( I_1 \) for the only nontrivial SCC \( C_1 \) to prove the validity of \( \Phi \):

- Induction Base of \( I_1 \) for \( C_1 \):
  \[ \bigwedge_{i=0}^{5} (\Psi_i \rightarrow \Phi) \land \bigwedge_{i=1}^{5} \text{SafePath}(s_0, s_i, (\Psi_0), \Phi, true, I_1) \]

Fig. 2: Quartz Program WaterTank
In total, we therefore obtain $3 \cdot \text{Induction Step of actions:}$

$s_0$ of $C_0$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 2) \land (\text{st} = 3)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_2$ of $C_1$

actions:

$\neg (\text{st} = 1) \land (\text{st} = 2) \land (\text{st} = 3)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_3$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_5$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_6$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_7$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_8$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

$s_9$ of $C_1$

actions:

$\neg (\text{st} = 0) \land (\text{st} = 1) \land (\text{st} = 2)$

$\Rightarrow \text{der}(y) \leftarrow \text{int2real}(0 - \text{nat2int}(\text{ouV}))$

Each SafePath condition is thereby a single VC. We need one more VC to ensure that the safety property $\Phi$ holds also in the discrete state and the continuous phase of the root SCC $s_0$. In total, we therefore obtain 24 VCs.

To simplify the verification goals, all ODEs in the VCs are solved with the aid of a computer algebra tool like Mathematica. Hence, each flow statement $(\text{drv}(x) \leftarrow \tau)$ is replaced by the solutions of the corresponding ODE system. For example, $(\text{drv}(y) \leftarrow \text{inV})$ is replaced by $(0 \leq t) \land (y = y_0 + \text{inV} \cdot t)$, where $t$ is a new introduced clock variable that is set to 0.0 at the initial time of the corresponding continuous phase, and $y_0$ is the value of $y$ when $t = 0.0$.

To give some idea on how the VCs look like after this replacement, we give the following example that describes SafePath($s_8$, $s_9$, $\Psi_8$, $I_1$, $true$, $I_1$):

$\forall X_R, X_Z, (0 \leq t) \land (5 \leq y) \land (y = y_0 - \text{outV} \cdot t) \land (0 \leq \text{delta}) \land (0 \leq \text{inV}) \land (0 \leq \text{outV}) \land (5 - \text{outV} \cdot \text{delta} \leq y_0 \leq 10 + \text{inV} \cdot \text{delta}) \land (5 - \text{outV} \cdot \text{delta} \leq y \leq 10 + \text{inV} \cdot \text{delta})$ where $X_R = \{t, t', y, y_0, \text{delta}\}$ and $X_Z = \{\text{inV}, \text{outV}\}$. Similar to the above formula, we notice that most of the

Fig. 3: The EFSM of the Quartz Program WaterTank
24 VCs are ∀-quantified mixed-integer non-linear arithmetic formulas.

C. Proving VCs with SMT Solvers

To prove the generated VCs, we used CVC4\(^8\), Z3\(^9\), and iSAT\(^10\). According to our experiments, CVC4’s non-linear real and non-linear integer arithmetic support is currently not sufficient for our needs. Therefore, we focused on Z3 and iSAT to prove the VCs. As shown in Fig. 4, it takes generally longer to generate a verification result using Z3 than using iSAT.

Note that neither Z3 nor iSAT can prove the validity of all the VCs, as shown in Table 1. The VCs that Z3 fails to prove are disjoint from the ones that iSAT could not verify. Therefore, we conclude that the safety property holds for the water tank system, since the combination of the two tools can prove the validity of all the VCs. It is therefore helpful to use more than one backend solver to prove the VCs.

### TABLE I: Experimental Results

<table>
<thead>
<tr>
<th>VC No.</th>
<th>Z3</th>
<th>iSAT</th>
<th>VC No.</th>
<th>Z3</th>
<th>iSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unknown</td>
<td>valid</td>
<td>13</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>2</td>
<td>valid</td>
<td>14</td>
<td>valid</td>
<td>15</td>
<td>valid</td>
</tr>
<tr>
<td>3</td>
<td>valid</td>
<td>16</td>
<td>Unknown</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>4</td>
<td>unknown</td>
<td>valid</td>
<td>17</td>
<td>Unknown</td>
<td>valid</td>
</tr>
<tr>
<td>5</td>
<td>valid</td>
<td>18</td>
<td>valid</td>
<td>19</td>
<td>valid</td>
</tr>
<tr>
<td>6</td>
<td>valid</td>
<td>20</td>
<td>valid</td>
<td>21</td>
<td>valid</td>
</tr>
<tr>
<td>7</td>
<td>valid</td>
<td>22</td>
<td>valid</td>
<td>23</td>
<td>valid</td>
</tr>
<tr>
<td>8</td>
<td>valid</td>
<td>24</td>
<td>Unknown</td>
<td>valid</td>
<td>valid</td>
</tr>
</tbody>
</table>

All experiments were performed on a machine with 3.2 GHz Intel Core i5-3470 processor, 8.1 GB RAM, and 64-bit Debian GNU/Linux 7.8 (wheezy).

V. Conclusion

We presented an automated approach towards deductive verification of hybrid systems that is based on induction over the SCCs of the EFSM. Given the right SCC-invariants and continuous invariants for a safety property, our procedure can automatically generate a set of verification conditions (VCs) that can then be checked by means of third-party SAT/SMT solvers. Our approach is not limited to Quartz programs and can be applied to any kind of hybrid systems with a state-based semantics.

### References


Fig. 4: Execution Time
