A CEGAR Tool for the Reachability Analysis of PLC-Controlled Plants using Hybrid Automata *
Technical Report

Johanna Nellen, Erika Ábrahám, and Benedikt Wolters

RWTH Aachen University, Aachen, Germany
{johanna.nellen,abraham}@cs.rwth-aachen.de
benedikt.wolters@rwth-aachen.de

Abstract. In this paper we address the safety analysis of chemical plants controlled by programmable logic controllers (PLCs). We consider sequential function charts (SFCs) for the programming of the PLCs, extended with the specification of the dynamic plant behavior. The resulting hybrid SFC models can be transformed to hybrid automata, opening the way to the application of advanced techniques for their reachability analysis. However, the hybrid automata models are often too large to be analyzed. To keep the size of the models moderate, we propose a counterexample-guided abstraction refinement (CEGAR) approach, which starts with the purely discrete SFC model of the controller and extends it with those parts of the dynamic behavior, which are relevant for proving or disproving safety. Our algorithm can deal with urgent locations and transitions, and non-convex invariants. We integrated the CEGAR approach in the analysis tool SpaceEx and present an example.

Keywords: hybrid systems, reachability analysis, CEGAR, verification

1 Introduction

In automation, *programmable logic controllers* (*PLCs*) are widely used to control the behavior of plants. The industry standard IEC 61131-3 [25] specifies several languages for programming PLCs, among others the graphical language of *sequential function charts* (*SFCs*).

Since PLC-controlled plants are often safety-critical, SFC *verification* has been extensively studied [19]. There are several approaches which consider either a SFC in isolation or the combination of a SFC with a model of the plant [21,5]. The latter approaches usually define a timed or hybrid automaton that specifies the SFC, and a hybrid automaton that specifies the plant. The composition of these two models gives a hybrid automaton model of the controlled plant. Theoretically, this composed model can be analyzed using existing tools for hybrid

* This work was partly supported by the German Research Foundation (DFG) as part of the Research Training Group “AlgoSyn” (GRK 1298) and the DFG research project “HyPro” (AB 461/4-1).

automata reachability analysis. In practice, however, the composed models are often too large to be handled by state-of-the-art tools.

In this paper we present a *counterexample-guided abstraction refinement* (*CEGAR*) [12] approach to reduce the verification effort. Instead of hybrid automata, we use *conditional ordinary differential equations* (*ODEs*) to specify the plant dynamics. A conditional ODE specifies the evolution of a physical quantity over time under some assumptions about the current control mode. For example, the dynamic change of the water level in a tank can be given as the sum of the flows through the pipes that fill and empty the tank. This sum may vary depending on valves being open or closed, pumps being switched on or off, and connected tanks being empty or not. Modeling the plant dynamics with conditional ODEs is natural and intuitive, and it supports a wider set of modeling techniques (e. g., effort-flow modeling).

Our goal is to consider only safety-relevant parts of the complex system dynamics in the verification process. Starting from a purely discrete model of the SFC control program, we apply reachability analysis to check the model for safety. When a counterexample is found, we refine our model stepwise by adding some pieces of information about the dynamics along the counterexample path.

The main advantage of our method is that it does not restart the reachability analysis after each refinement step but *the refinement is embedded into the reachability analysis procedure* in order to prevent the algorithm from re-checking the same model behavior repeatedly.

Related work Originating from discrete automata, several approaches have been presented where CEGAR is used for hybrid automata [10,11,2]. The work [15] extends the research on CEGAR for hybrid automata by restricting the analysis to fragments of counterexamples. Other works [34,26] are restricted to the class of rectangular or linear hybrid automata. Linear programming for the abstraction refinement is used in [26]. However, none of the above approaches exploits the special properties of hybrid models for plant control.

In [13] a CEGAR verification for PLC programs using timed automata is presented. Starting with the coarsest abstraction, the model is refined with variables and clocks. However, this work does not consider the dynamic plant behavior.

A CEGAR approach on step-discrete hybrid models is presented in [35], where system models are verified by learning reasons for spurious counterexamples and excluding them from further search. However, this method was designed for control-dominated models with little dynamic behavior.

A CEGAR-based abstraction technique for the safety verification of PLC-controlled plants is presented in [14]. Given a hybrid automaton model of the controlled plant, the method abstracts away from parts of the continuous dynamics. However, instead of refining the dynamics in the hybrid model to exclude spurious counterexamples, their method adds information about enabled and disabled transitions.

Several tools for the *reachability analysis* of hybrid automata have been developed [22,17,24,30,3,20,27,4,33]. We chose to integrate our approach into a tool that is based on *flowpipe-computation*. Prominent candidates are SpaceEx [16]

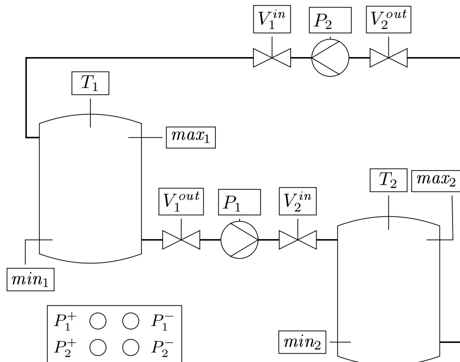

Fig. 1: An example plant and its operator panel

and FLOW* [9]. For the implementation of our CEGAR algorithm we must be able to generate a (presumed) *counterexample* if an unsafe state is reachable. Moreover, our modeling approach uses *urgent locations* in which time cannot elapse, and *urgent transitions* whose enabledness forces to leave the current location. Although FLOW* provides presumed counterexamples, we decided to integrate our method into SPACEEX. The reason is that SPACEEX provides different algorithms for the reachability analysis. Important for us is also the recently published PHAVER scenario [29] that supports urgent transitions and non-convex invariants for a simpler class of hybrid automata. Furthermore, in [8] an extension of SPACEEX for hybrid automata is presented where the search is guided by a cost function. This enables a more flexible way of searching the state space compared to a breath- or depth-first-search.

This paper is an extension of [32]. We made the approach of [32] more efficient by introducing urgent locations for hybrid automata, defining dedicated methods to handle urgent locations, urgent transitions and non-convex invariants in the reachability analysis, and provide an implementation of the proposed methodology. The tool and a technical report containing further details can be accessed from `http://ths.rwth-aachen.de/research/tools/spaceex-with-cegar/`.

Outline After some preliminaries in Section 2, we describe our modeling approach in Section 3. In the main Section 4 we present our CEGAR-based verification method. The integration of our CEGAR-based method into the reachability analysis algorithm, some details on the implementation, and an example are discussed in Section 5. We conclude the paper in Section 8.

2 Preliminaries

2.1 Plants

A simple example of a *chemical plant* is depicted in Figure 1. It consists of two cylindrical tanks T_1 and T_2, with equal diameters, that are connected by

pipes. The variables h_1 and h_2 denote the water level in the tanks T_1 and T_2, respectively. Each tank T_i is equipped with two sensors min_i and max_i, at height $0 < L < U$, that detect low and high water levels, respectively.

The plant is equipped with two pumps P_1 and P_2 which can pump water when the adjacent valves are open. P_1 pumps water from T_1 to T_2, decreasing h_1 and increasing h_2 by k_1 per time unit. P_2 pumps water through a second pipeline in the other direction, causing a height increase of k_2 per time unit for h_1 and a height decrease of k_2 per time unit for h_2.

We overload the meaning of P_i and use it also to represent the state (on: $P_i = 1$ or off: $P_i = 0$) of pump i.

The pumps are manually controlled by the operator panel which allows to switch the pumps on (P_i^+) or off (P_i^-). The control receives this input along with the sensor values, and computes some output values, which are sent to the environment and cause actuators to be controlled accordingly, by turning the pumps on or off. The pumps are coupled with the adjacent valves, which will automatically be opened or closed, respectively.

We want the control program to prevent the tanks from running dry: If the water level of the source tank is below the lower sensor the pump is switched off and the connected valves are closed automatically. For simplicity, in the following we neglect the valves and assume that the tanks are big enough not to overflow.

The *state* of a plant, described by a function assigning values to the physical quantities, evolves continuously over time. The plant specification defines a set of *initial* states. The dynamics of the evolution is specified by a set of *conditional ordinary differential equations* (*ODEs*), one conditional ODE for each continuous physical quantity (plant variable). Conditional ODEs are pairs of a condition and an ODE. The conditions are closed linear predicates over both the plant's physical quantities and the controller's variables; an ODE specifies the dynamics in those states that satisfy its condition. We require the conditions to be convex polytopes, which overlap only at their boundaries. In cases, where none of the conditional ODEs apply, we assume chaotic (arbitrary) behavior.

Example 1. For the example plant, assume $k_1 \geq k_2$ and let $\varphi_{1\to 2} \equiv P_1 \wedge h_1 \geq 0$ denote that pump P_1 is on and the first tank is not empty; the meaning of $\varphi_{2\to 1} \equiv P_2 \wedge h_2 \geq 0$ is analogous. We define the following conditional ODE system for h_1:

$$(c_1, \mathrm{ODE}_1^{h_1}) = (\ \ \varphi_{1\to 2} \wedge \ \ \varphi_{2\to 1}, \quad \dot{h}_1 = k_2 - k_1) \tag{1}$$

$$(c_2, \mathrm{ODE}_2^{h_1}) = (\ \ \varphi_{1\to 2} \wedge \neg P_2, \quad \dot{h}_1 = -k_1) \tag{2}$$

$$(c_3, \mathrm{ODE}_3^{h_1}) = (\neg P_1 \wedge \ \ \varphi_{2\to 1}, \quad \dot{h}_1 = k_2) \tag{3}$$

$$(c_4, \mathrm{ODE}_4^{h_1}) = (\neg \varphi_{1\to 2}, \quad \dot{h}_1 = 0) \tag{4}$$

The conditional ODEs for h_2 are analogous.

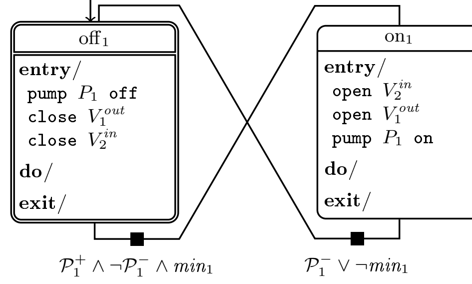

Fig. 2: SFC for pump P_1

2.2 Sequential Function Charts

To specify controllers we use *sequential function charts* (*SFCs*) as given by the industry norm IEC 61131-3 [25], with a formal semantics as specified in [31] that is based on [7,6,28] with slight adaptations to a certain PLC application.

Example 2. Figure 2 shows a possible control program for our example plant. We specify only the control of P_1, which runs in parallel with an analogous SFC for the second pump.

A SFC has a finite set *Var* of typed *variables*, classified into input, output and local variables. A *state* $\sigma \in \Sigma$ of a SFC is a function that assigns to each variable $v \in Var$ a value from its domain. By $Pred_{Var}$ we denote the set of linear predicates over *Var*, evaluating to true or false in a given state.

The control is specified using a finite set of *steps* and *guarded transitions* between them, connecting the bottom of a source step with the top of a target step. A distinguished initial step is *active* at start. A transition is *enabled* if its source step is active and its transition guard from $Pred_{Var}$ is true in the current state; taking an enabled transition moves the activation from its source to its target step. Apart from transitions that connect single steps, also parallel branching can be specified by defining sets of source/target steps.

A partial order on the transitions defines *priorities* for concurrently enabled transitions that have a common source step. For each step, the enabled transition with the highest priority is taken. Transitions are *urgent*, i. e., a step is active only as long as no outgoing transition is enabled.

Each step contains a set of prioritized *action blocks* specifying the actions that are performed during the step's activation period. An action block $b = (q, a)$ is a tuple with an *action qualifier* q and an *action* a. The set of all action blocks using actions from the set *Act* is denoted by B_{Act}.

The action qualifier $q \in \{\texttt{entry}, \texttt{do}, \texttt{exit}\}$[1] specifies when the corresponding action is performed. When control enters a step, its `entry` and `do` actions are ex-

[1] In the IEC standard, the qualifiers $P1, N$ and $P0$ are used instead of `entry`, `do` and `exit`. The remaining qualifiers of the industry standard are not considered in this paper.

ecuted once. As long as the step is active, its `do` actions are executed repeatedly. The `exit` actions are executed upon deactivation.

An action a is either a variable assignment or a SFC. Executing an assignment changes the value of a variable, executing a SFC means activating it and thus performing the actions of the active step.

The execution of a SFC on a programmable logic controller performs the following steps in a cyclic way:

1. Get the input data from the environment and update the values of the input variables accordingly.
2. Collect the transitions to be taken and execute them.
3. Determine the actions to be performed and execute them in priority order.
4. Send the output data (the values of the output variables) to the environment.

Between two PLC cycles there is a time delay δ, which we assume to be equal for all cycles (however, our approach could be easily extended to varying cycle times). Items 1. and 4. of the PLC cycle implement the communication with the environment, e. g. with plant sensors and actuators, whereas 2. and 3. execute the control.

2.3 Hybrid Automata

A popular modeling language for systems with mixed discrete-continuous behavior are *hybrid automata*. A set of real-valued *variables* describe the system state. Additionally, a set of *locations* specify different control modes. The change of the current control mode is modeled by *guarded transitions* between locations. Additionally, transitions can also change variable values, and can be urgent. Time can evolve only in non-urgent locations; the values of the variables change continuously with the evolution of time. During this evolution (especially when entering the location), the location's *invariant* must not be violated.

Definition 1 (Hybrid Automaton [1]). *A* hybrid automaton *(HA) is a tuple* $\mathrm{HA} = (Loc, Lab, Var, Edge, Act, Inv, Init, Urg)$ *where*

- *Loc is a finite set of* locations*;*
- *Lab is a finite set of* labels*;*
- *Var is a finite set of real-valued* variables. *A* state $\nu \in V$, $\nu : Var \to \mathbb{R}$ *assigns a value to each variable. A* configuration $s \in Loc \times V$ *is a location-valuation pair;*
- $Edge \subseteq Loc \times Lab \times Pred_{Var} \times (V \to V) \times Loc$ *is a finite set of* transitions, *where the function from* $V \to V$ *is linear;*
- *Act is a function assigning a set of time-invariant* activities $f : \mathbb{R}_{\geq 0} \to V$ *to each location, i. e., for all* $l \in Loc$, $f \in Act(l)$ *implies* $(f + t) \in Act(l)$ *where* $(f + t)(t') = f(t + t')$ *for all* $t, t' \in \mathbb{R}_{\geq 0}$*;*
- $Inv : Loc \to 2^V$ *is a function assigning an* invariant *to each location;*
- $Init \subseteq Loc \times V$ *is a set of* initial configurations *such that* $\nu \in Inv(l)$ *for each* $(l, \nu) \in Init$*;*

– $Urg : (Loc \cup Edge) \to \mathbb{B}$ *(with* $\mathbb{B} = \{0, 1\}$*) a function defining those locations and transitions to be* urgent*, whose function value is* 1*.*

The activity sets are usually given in form of an *ordinary differential equation* (*ODE*) system, whose solutions build the activity set. Furthermore, it is standard to require the invariants, guards and initial sets to define convex polyhedral sets: if they are not linear, they can be over-approximated[2] by a linear set; if they are not convex, they can be expressed as a finite union of convex sets (corresponding to the replacement of a transition with non-convex guard by several transitions with convex guards, and similarly for initial sets and location invariants). In the following we syntactically allow linear non-convex conditions, where we use such a transformation to eliminate them from the models.

Example 3. An example HA (without invariants) is shown in Figure 4. A star $*$ in a location indicates that the location is urgent. Similarly, transitions labeled with a star $*$ are urgent.

The semantics distinguishes between *discrete steps (jumps)* and *time steps (flows)*. A jump follows a transition $e = (l, \alpha, g, h, l')$, transforming the current configuration (l, ν) to $(l', \nu') = (l', h(\nu))$. This transition, which has a synchronization label α (used for parallel composition), must be *enabled*, i. e., the guard g is true in ν and $Inv(l')$ is true in ν'. Time steps model time elapse; from a state ν, the values of the continuous variables evolve according to an activity $f \in Act(l)$ with $f(0) = \nu$ in the current location l. Time cannot elapse in *urgent locations* l, identified by $Urg(l) = 1$, but an outgoing transition must be taken immediately after entering the location. Control can stay in a non-urgent location as long as the location's invariant is satisfied. Furthermore, if an *urgent transition* e, identified by $Urg(e) = 1$, is enabled, time cannot further elapse in the location and an outgoing transition must be taken. For the formal semantics and the parallel composition of HA we refer to [1]. The parallel composition of a set of locations *Loc* yields an urgent location, if *Loc* contains at least one urgent location. Analogously, the parallel composition of a set of transitions *Trans* is an urgent transition, if there is at least one urgent transition in *Trans*.

Though the *reachability problem* for HA is in general undecidable [23], there exist several approaches to compute an *over-approximation* of the set of reachable states. Many of these approaches use geometric objects (e. g. convex polytopes, zonotopes, ellipsoids, etc.) or symbolic representations (e. g. support functions or Taylor models) for the over-approximative representation of state sets. The efficiency of certain operations (i. e. intersection, union, linear transformation, projection and Minkowski sum) on such objects determines the efficiency of their employment in the reachability analysis of HA.

The basic idea of the reachability analysis is as follows: Given an initial location l_0, a set P_0 of initial states (in some representation), a step size $\tau \in \mathbb{R}_{>0}$ and a time bound $T = n\tau$ ($n \in \mathbb{N}_{>0}$), first the so-called *flow pipe*, i. e., the set of states reachable from P_0 within time T in l_0, is computed. To reduce the

[2] For over-approximative reachability analysis; otherwise under-approximated.

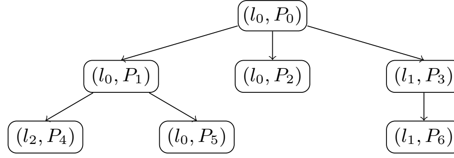

Fig. 3: Example search tree

approximation error, this is done by computing a sequence $P_1, \ldots, P_n$ of *flow pipe segments*, where for each $0 < i \leq n$ the set P_i over-approximates the set of states reachable from P_0 in time $[(i{-}1)\tau, i\tau]$ according to the dynamics in l_0. The intersection of these sets with the invariant of l_0 gives us the time successors of P_0 within time T. Finally, we also need to compute for each of the flow pipe segments (intersected with the invariant) all possible jump successors. This latter computation involves the intersection of the flow pipe segments with the transition guards, state set transformations computing the transitions' effects, and an intersection computation with the target location's invariant; special attention has to be payed to urgent locations and transitions.

The whole computation of flow pipe segments and jump successors is applied in later iterations to each of the above-computed successor sets (for termination usually both the maximal time delay in a location and the number of jumps along paths are bounded). Thus the reachability analysis computes a *search tree*, where each node is a pair of a location and a state set, whose children are its time and jump successors (see Figure 3).

Different heuristics can be applied to determine the node whose children will be computed next. Nodes, whose children still need to be computed, are marked to be *non-completed*, the others *completed*. When applying a fixed-point check, only those nodes which are not included in other nodes are marked as non-completed.

In our approach, we use the SpaceEx tool [16], which is available as a standalone tool with a web interface as well as a command-line tool that provides the analysis core and is easy to integrate into other projects. To increase efficiency, SpaceEx can compute the composition of HA on-the-fly during the analysis.

2.4 CEGAR

Reachability analysis for HA can be used to prove that no states from a given "unsafe" set are reachable from a set of initial configurations. For complex models, however, the analysis might take unacceptably long time. In such cases, *abstraction* can be used to reduce the complexity of the model at the cost of over-approximating the system behavior. If the abstraction can be proven to be safe then also the concrete model is safe. If the abstraction is too coarse to satisfy the required safety property, it can be *refined* by re-adding more detailed information about the system behavior. This iterative approach is continued until either the refinement level is fine enough to prove the specification correct or the

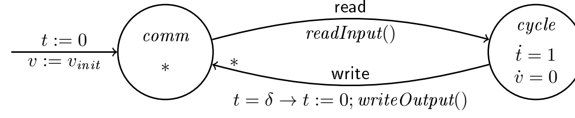

Fig. 4: Hybrid automaton for PLC cycle synchronization and the user input. At the beginning of each cycle, the input variables (including the user input) are read. At the end of each cycle, the output variables are written.

model is fully concretized. In *counterexample-guided abstraction refinement* (*CEGAR*), the refinement step is guided by a counterexample path, leading from the initial configuration to an unsafe one in the abstraction (i. e., one or more states on the abstract counterexample path get refined with additional information).

3 Modeling Approach

SFC-controlled plants can be modeled by a HA, built by the composition of several HA for the different system components [31]: One HA is used to synchronize on the PLC cycle time and model the user input (see Figure 4). The control is modeled by one HA for each SFC running in parallel (see Figure 5). The last automaton models the plant dynamics according to a given conditional ODE system (see Figure 7). The parallel composition of these automata gives us a model for the controlled plant.

In the models, we use a vector v^{dyn} of variables for the physical quantities in the plant dynamics. A vector v^{sen} of variables and expressions represents the input for the SFC, containing control panel requests, actuator states and sensor values. The input, local and output variables of the SFCs are v^{in}, v^{loc} and v^{out}.

Example 4. For our example plant, we will use the following encodings:

- $v^{dyn} = (v_1^{dyn}, v_2^{dyn})$ with $v_i^{dyn} = h_i$ for the water height in the tanks;
- $v^{sen} = (v_1^{sen}, v_2^{sen})$ for the input of the SFC with $v_i^{sen} = (*, *, P_i, h_i \geq L, h_i \geq U)$, where the first two entries $*$ encode arbitrary control panel requests P_i^+ and P_i^-, P_i is the state of pump i (0 or 1, encoding off or on) and the values of the sensors min_i and max_i;
- $v^{in} = (v_1^{in}, v_2^{in})$ with $v_i^{in} = (\mathcal{P}_i^+, \mathcal{P}_i^-, \mathcal{P}_i, m_i, M_i)$ for SFC input variables receiving the values of v^{sen} from above with the control panel requests, the actuator state, and sensor values;
- $v^{loc} = ()$, i. e. there are no local SFC variables;
- $v^{out} = (v_1^{out}, v_2^{out})$ with $v^{out} = (\mathcal{P}_i^{on}, \mathcal{P}_i^{off})$ for the output variables of the SFC, that control the actuators of the plant. When both commands are active for pump i, i. e. $\mathcal{P}_i^{on} = \mathcal{P}_i^{off} = 1$, then pump i it will be switched off. Otherwise, $\mathcal{P}_i^{on} = 1$ will cause pump i to be switched on and $\mathcal{P}_i^{off} = 1$ will lead to switching pump i off.

PLC Cycle Synchronization. SFCs running parallel on a PLC synchronize on reading the input and writing the output. Before each cycle the input is read, where *readInput*() stores the current memory image v^{in} to v^{sen}. The values of v^{sen} are accessible for all parallel running components and will not change for the duration of the current cycle. After a constant cycle time δ, the output is written (e. g. commands that control the actuators of the plant are immediately executed). We model this behavior using the HA shown in Figure 4. We use a clock t with initial value 0 to measure the cycle time. The initial location *comm* is urgent, represented by a star $*$, thus the outgoing transition to location *cycle* will be taken immediately. The transition from *cycle* to *comm* is urgent, again represented by a star $*$, forcing the writing to happen at the end of each cycle. The synchronization labels read and write force all parallel running HA that share those labels to synchronize on these transitions. While time elapses in location *cycle*, the SFCs perform their active actions and the dynamic behavior of the environment evolves according to the specified differential equations. The ODE $\dot{v} = 0$ expresses that the derivative of *all* involved *discrete* variables appearing in v^{sen}, v^{in}, v^{loc} or v^{out} is zero. (For simplicity, here we specify the derivative 0 for all discrete variables in the PLC synchronizer model; in our implementation the SFC variables are handled in the corresponding SFC models.)

Example 5. For the tank example, we allow arbitrary (type-correct) user input, where we use $*$ to represent a non-deterministically chosen value. (Note, that this $*$ has a different meaning than the one used for urgency.) Reading the input *readInput*() executes $\mathcal{P}_i^+ := *$, $\mathcal{P}_i^- := *$, $\mathcal{P}_i := P_i$, $m_i := (h_i \geq L)$ and $M_i := (h_i \geq U)$. Writing the output *writeOutput*() updates $P_i := (P_i \vee \mathcal{P}_i^{on}) \wedge \neg \mathcal{P}_i^{off}$.

HA for SFC. In the HA model of a SFC (see Figure 5), for each step s of the SFC there is a corresponding location pair in the HA: the location s^{in} is entered upon the activation of the step s and it is left for location s when the input is read. The execution of the actions is modeled to happen at the beginning of the PLC cycle by defining location s to be urgent. The outgoing transitions of s represent the cycle execution: If s remains activated then its `do` actions else its `exit` actions and both the `entry` and the `do` actions of the next step are executed in their priority order. The location s_0^{in} that corresponds to the initial step s_0 defines the initial location of the HA.

Example 6. The hybrid automaton model for the SFC for pump P_1 in Figure 2 is modeled by the hybrid automaton depicted in Figure 6.

Plant Dynamics. Assume that the plant's dynamics is described by sets of conditional ODEs, one set for each involved physical quantity. We define a HA for each such quantity (see Figure 7); their composition gives us a model for the plant. The HA for a quantity contains one location for each of its conditional ODEs and one for chaotic (arbitrary) behavior. The conditions specify the locations' invariants, the ODEs the activities; the chaotic location has the invariant

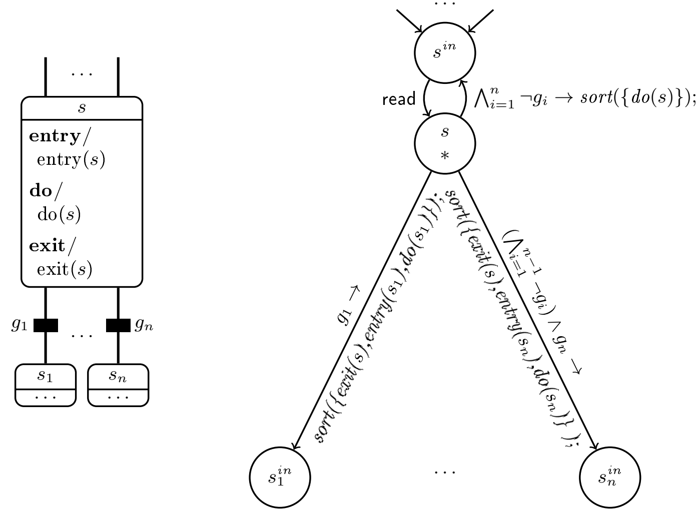

Fig. 5: Hybrid automaton for an SFC. The actions are sorted according to a specified priority order.

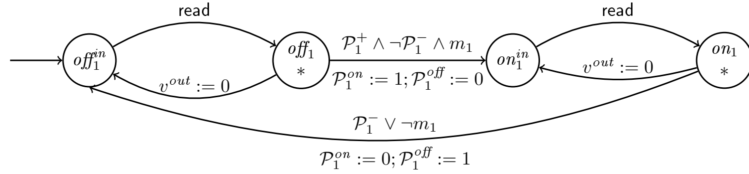

Fig. 6: Hybrid automaton model of the SFC for pump P_1

true. Each pair of locations, whose invariants have a non-empty intersection, is connected by a transition. To assure that chaotic behavior is specified only for undefined cases, we define all transitions leaving the chaotic location to be urgent. Note that a transition is enabled only if the target location's invariant is not violated.

Example 7. The plant dynamics of the tank example is modeled by the hybrid automaton in Figure 8. Note that, since the conditions cover the whole state space, time will not evolve in the chaotic location $(l, 5)$.

Parallel Composition. Due to the parallel composition, the models can be very large. Though some simple techniques can be used to reduce the size, the remaining model might still be too large to be analyzed. E. g. we can remove from the model all locations with *false* invariants, transitions between locations whose invariants do not intersect, and non-initial locations without any incoming transitions.

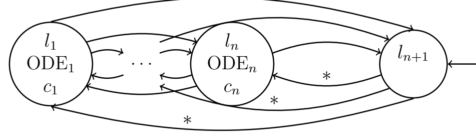

Fig. 7: Hybrid automaton for the plant dynamics using the conditional ODEs $\{(c_1, ODE_1), \ldots (c_n, ODE_n)\}$

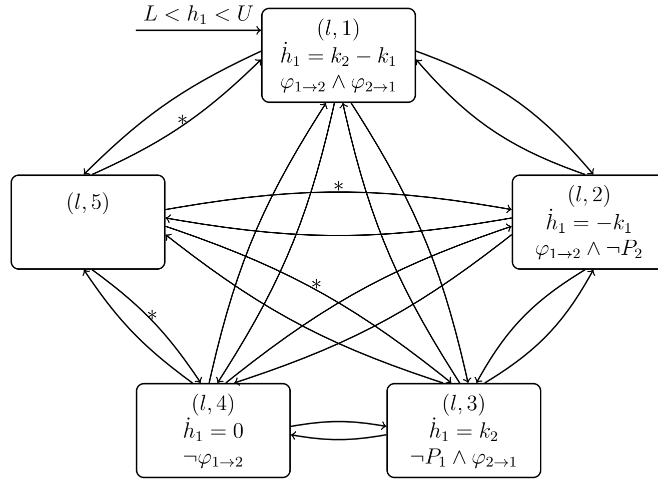

Fig. 8: Hybrid automaton model of the plant dynamics for tank T_1 with $k_1 \geq k_2$

4 CEGAR-Based Verification

In this chapter we explain our CEGAR approach for the verification of SFC-controlled plants. Besides this special application, our method could be easily adapted to other kinds of hybrid systems.

One of the main barriers in the application of CEGAR in the reachability analysis of hybrid systems is the complete re-start of the analysis after each refinement. To overcome this problem, we propose an *embedding of the CEGAR approach into the HA reachability analysis algorithm*: our algorithm refines the model on-the-fly during analysis and thus reduces the necessity to re-compute parts of the search tree that are not affected by the refinement. Besides this advantage, our method also supports the handling of *urgent locations* and *urgent transitions*, which is not supported by most of the HA analysis tools. Last but not least, our algorithm can be used to extend the functionalities of currently available tools to generate (at least presumed) *abstract counterexamples*.

4.1 Model Refinement

The basis for a CEGAR approach is the generation of a counterexample and its usage to refine the model. Therefore, first we explain the mechanism for this (explicit) model refinement before we describe how we embed the refinement into the reachability algorithm to avoid restarts.

Abstraction. Intuitively, the abstraction of the HA model of a SFC-controlled plant consists of removing information about the plant dynamics and assuming chaotic behavior instead. Initially, the whole plant dynamics is assumed to be chaotic; the refinement steps add back more and more information. The idea is that the behavior is refined only along such paths, on which the controller's correctness depends on the plant dynamics. Therefore, the abstraction level for the physical quantities (plant variables) of the plant will depend on the controller's configuration.

The abstraction level is determined by a function *active* that assigns to each location-variable pair (l, x) a subset of the conditional ODEs for variable x. The meaning of this function is as follows: Let H be the HA composed from the PLC-cycle synchronizer and the SFC model *without* the plant dynamics. Let l be a location of H, x a dynamic variable in the plant model and let $\mathit{active}(l, x) = \{(c_1, \mathrm{ODE}_1), \ldots, (c_n, \mathrm{ODE}_n)\}$. Then the global model of the controlled plant will define x to evolve according to ODE_i if c_i is valid and chaotically if none of the conditions $c_1, \ldots, c_n$ holds. A refinement step *extends* a subset of the sets $\mathit{active}(l, x)$ by adding new conditional ODEs to some variables in some locations.

Counterexample-Guided Refinement. The refinement is counterexample-guided. Since the reachability analysis is over-approximative, we generate *presumed* counterexamples only, i. e., paths that might lead from an initial configuration to an unsafe one but might also be spurious. For the refinement, we choose the first presumed counterexample that is detected during the analysis using a breadth-first search, i. e. we find *shorter* presumed counterexamples first. However, other heuristics are possible, too.

A counterexample is a property-violating path in the HA model. For our purpose, we do not need any concrete path, we only need to identify the sequence of nodes in the search tree from the root to a node (l, P) where P has a non-empty intersection with the unsafe set. If we wanted to use some other refinement heuristics that requires more information, we could *annotate* the search tree nodes with additional bookkeeping about the computation history (e. g., discrete transitions taken or time durations spent in a location).

We refine the abstraction by extending the specification of the (initially chaotic) plant dynamics with some conditional ODEs from the concrete model, which determines the plant dynamics along a presumed counterexample path. Our refinement heuristics computes a set of tuples $(l, x, (c, \mathrm{ODE}))$, where l is a location of the model composed from the synchronizer and the SFC models *without* the plant model, x is a continuous variable of the plant, and $(c, \mathrm{ODE}) \notin \mathit{active}(l, x)$ a conditional ODE for x from the plant dynamics that was not yet considered in location l.

Possible heuristics for choosing the locations are to refine the first, the last, or all locations of the presumed counterexample. The chosen location(s) can be refined for each variable by any of its conditional ODEs that are applicable but not yet active. Applicable means that for the considered search tree node (l, P) the ODE's condition intersects with P. If no such refinements are possible any more then the counterexample path is *fully refined* and the algorithm terminates with the result that the model is possibly unsafe.

Building the Model at a Given Level of Abstraction. Let again H be the HA composed from all HA models *without* the plant dynamics. Let $x_1, \ldots, x_n$ be the continuous plant variables and let *active* be a function that assigns to each location l of H and to each continuous plant variable x_i a subset $\mathit{active}(l, x_i) = \{(c_{i,1}, \mathrm{ODE}_{i,1}), \ldots, (c_{i,k_i}, \mathrm{ODE}_{i,k_i})\}$ of the conditional ODEs for x_i. We build the global HA model H' for the controlled plant, induced by the given *active* function, as follows:

- The locations of H' are tuples $\hat{l} = (l, l_1, \ldots, l_n)$ with l a location of H and $1 \leq l_i \leq k_i + 1$ for each $1 \leq i \leq n$. For $1 \leq i \leq n$, l_i gives the index of the conditional ODE for variable x_i and $l_i = k_i + 1$ denotes chaotic behavior for x_i. We set $\mathit{Urg}'(\hat{l}) = \mathit{Urg}(l)$.
- The variable set is the union of the variable set of H and the variable set of the plant.
- For each transition $e = (l, \alpha, g, f, l')$ in H, the automaton H' has a transition $e' = (\hat{l}, \alpha, g, f, \hat{l}')$ with $\mathit{Urg}'(e') = \mathit{Urg}(e)$ for all locations $\hat{l}$ and $\hat{l}'$ of H' whose first components are l and l', respectively. Additionally, all locations $\hat{l} = (l, l_1, \ldots, l_n)$ and $\hat{l}' = (l, l'_1, \ldots, l'_n)$ of H' with identical first components are connected; these transitions have no guards and no effect; they are urgent iff $l'_j = k_j + 1$ implies $l_j = k_j + 1$ for all $1 \leq j \leq n$ (all chaotic variables in $\hat{l}'$ are also chaotic in $\hat{l}$).
- The activities in location $\hat{l} = (l, l_1, \ldots, l_n)$ are the solutions of the differential equations $\{\mathrm{ODE}_{i,l_i} \mid 1 \leq i \leq n, l_i \leq k_i\}$ extended with the ODEs of H in l.
- The invariant of a location $(l, l_1, \ldots, l_n)$ in H' is the conjunction of the invariant of l in H and the conditions c_{i,l_i} for each $1 \leq i \leq n$ with $l_i \leq k_i$.
- The initial configurations of H' are those configurations $((l, l_1, \ldots, l_n), \nu)$ for which l and ν projected to the variable set of H is initial in H, and ν projected to the plant variable set is an initial state of the plant.

Dealing with Urgency. The hybrid automaton H resulting from a refinement contains *urgent locations* and *urgent transitions*. However, the available tools SpaceEx and Flow* for the reachability analysis of hybrid automata do not support urgency. Though a prototype implementation of PHAVer [29] supports urgent transitions, it is designed for a restricted class of models with polyhedral derivatives. To solve this problem, we make adaptations to the reachability analysis algorithm and apply some model transformations as follows.

Firstly, we adapt the reachability analysis algorithm such that no time successors are computed in urgent locations.

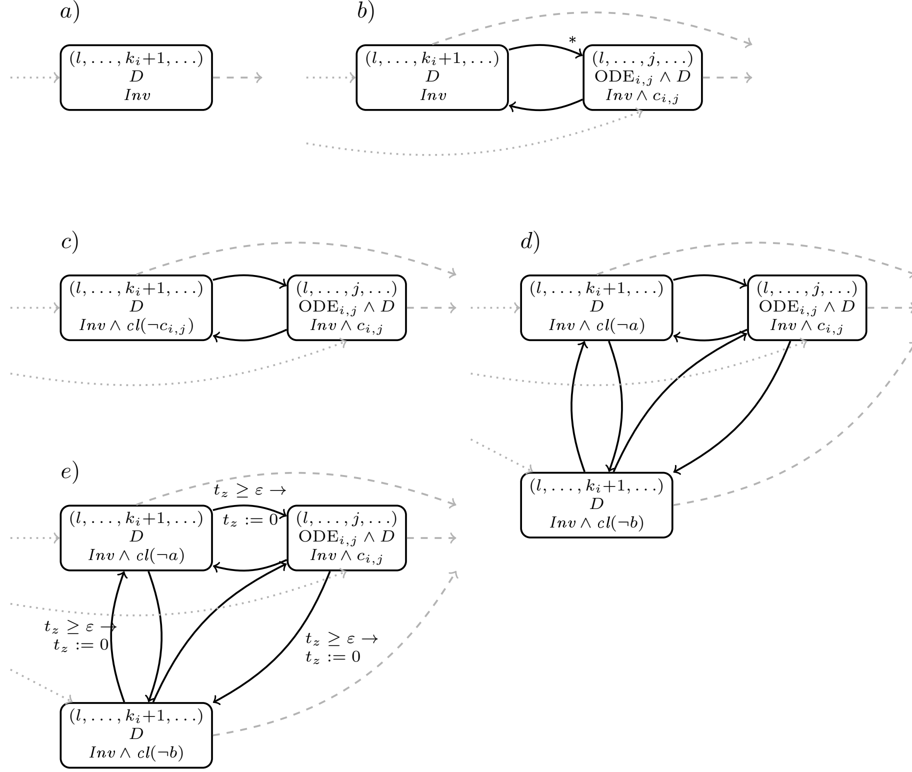

Fig. 9: a) Location $\hat{l}$ before the refinement; b) Refinement using $(l, (c_{i,j}, \mathrm{ODE}_{i,j}))$; c) Modeling the urgency (over-approximated); d) Modeling non-convex invariants (here: $\neg c_{i,j} = \neg(a \wedge b)$); e) Zeno path exclusion

Secondly, for the urgent transition in the PLC synchronizer model (see Figure 4), we remove its urgency and set the time horizon T in the reachability analysis to δ, i. e., we restrict the time evolution in location *cycle* to δ.

Thirdly, for the remaining urgent transitions in the plant dynamics, we use model transformations to eliminate them: We replace urgent transitions by non-urgent transitions and extend the source locations' invariants by additional conjunctive terms as follows.

Remember that $x_1, \ldots, x_n$ are the plant variables and let $active(l, x_i) = \{(c_{i,1}, \mathrm{ODE}_{i,1}), \ldots, (c_{i,k_i}, \mathrm{ODE}_{i,k_i})\}$ be the active conditional ODEs for x_i in l. Let $cl(\cdot)$ denote the closure of a set.

Each urgent transition $e = ((l, l_1, \ldots, l_n), \alpha, g, f, (l, l'_1, \ldots, l'_n))$ in the plant model is made non-urgent. Additionally, for each variable x_i which is chaotic in the source location ($l_i = k_i{+}1$) but not chaotic in the target location ($l'_i \leq k_i$), we conjugate the invariant of the source location with the negated condition of the

ODE for x_i in the target location. Thus the new invariant is $Inv((l, l_1, \ldots, l_n)) \wedge \bigwedge_{1 \leq i \leq n,\, l_i = k_i + 1,\, l'_i \leq k_i} cl(\neg c_{i,l'_i})$. The resulting automaton is shown in Figure 9c.

Note that the elimination of urgent transitions is *over-approximative*, since in the transformed model we can still stay in a chaotic location after the condition of an outgoing urgent transition became true. However, in a chaotic location the dynamics will not enter the inner part (without its boundary) of any active ODE condition.

Dealing with Non-Convex Invariants. The above transformation of urgent transitions to non-urgent ones introduces non-convex invariants unless the conditions of the conditional ODEs are half spaces. Since state-of-the-art tools do not support non-convex invariants, we again use a transformation step to eliminate them.

The non-convex invariants can be represented as finite unions of convex sets $NC = C_1 \cup \ldots \cup C_k$. Thus for each location l with a non-convex invariant NC we compute such a union. This can be obtained by computing the *disjunctive normal form* $NC = c_1 \vee \ldots \vee c_k$, where each *clause* c_i is a conjunction of convex constraints.

The original location l is replaced by a set of locations $l_1, \ldots, l_k$ with invariants $c_1, \ldots, c_k$. The sets of incoming/outgoing transitions and the dynamics of l are copied for each location $l_1, \ldots, l_k$. To allow mode switching between the new locations, we add a transition between each pair of different locations from $l_1, \ldots, l_k$ with *true* guard and without effect (see Figure 9d).

Dealing with Zeno Paths. The construction to resolve non-convex invariants allows paths with infinitely many mode switches in zero time. This is called *Zeno* behavior which should be avoided since both the running time and the over-approximation might increase drastically.

One possibility to avoid these Zeno behaviors is to force a minimal time elapse ε in each cycle of a location set introduced for the encoding of a non-convex invariant. To do so, we can introduce a fresh clock t_z and modify at least one transition $e = (l, \alpha, g, h, l')$ in each cycle by an annotated variant $(l, \alpha, g \wedge t_z \geq \varepsilon,\ h \wedge t_z{:=}0,\ l')$. Additionally, we add the differential equation $\dot{t_z} = 1$ to the source location of the annotated transition. The result of this transformation is shown in Figure 9e. Note that the above transformation eliminates Zeno paths, but it leads to an *under-approximation* of the original behavior.

Another possibility avoiding the introduction of a new variable is to modify the reachability analysis algorithm such that the first flow pipe segment in the source location of such transitions $e = (l, \alpha, g, h, l')$ computes time successors for ϵ and from this first segment no jump successors are computed along e. If the model is safe, we complete the reachability analysis also for those, previously neglected jump successors, in order to re-establish the over-approximation.

CEGAR *Iterations.* For the initial abstraction and after each refinement we start a reachability analysis procedure on the model at the current level of abstraction. The refinement is iterated until 1) the reachability analysis terminates

without reaching an unsafe state, i. e. the model is correct, or 2) a fully refined path from an initial state to an unsafe state is found. In the case of 2), the unsafe behavior might result from the over-approximative computation, thus the analysis returns that the model is *possibly* unsafe.

5 Integrating CEGAR into the Reachability Analysis

5.1 Adapting the Reachability Analysis Algorithm

Restarting the complete model analysis in each refinement iteration leads to a re-computation of the whole search tree, including those parts that are not affected by the refinement step. To prevent such restarts, we do the model refinement on-the-fly during the reachability analysis and backtrack in the search tree only as far as needed to remove affected parts.

For this computation we need some additional bookkeeping: During the generation of the search tree, we label all time successor nodes (l, P) with the set V of all plant variables, for which chaotic behavior was assumed in the flow pipe computation, resulting in a node (l, P, V). In the initial tree, all time successor nodes are labeled with the whole set of plant variables. Discrete successors are labeled with the empty set.

We start with the fully abstract model, i. e., with the composition H of the synchronizer and the SFC models, extended with the variables of the plant. Note that initially we do not add any information about the plant dynamics, i. e., we allow chaotic behavior for the plant.

We apply a reachability analysis to this initial model. If it is proven to be safe, we are done. Otherwise, we identify a path in the search tree that leads to an unsafe set and extend the *active* function to $active'$ as previously described.

However, instead of re-starting the analysis on the explicit model induced by the extended $active'$ function, we proceed as follows:

- *Backtracking:* When a refinable counterexample is found, we backtrack in the tree for each pair of location l and variable x with $active'(l, x) \setminus active(l, x) \neq \emptyset$. We delete all nodes (l, P, V) with $x \in V$, i. e. those nodes whose configuration contains location l and for which x was assumed to be chaotic in the flowpipe construction. We mark the parents of deleted nodes as not completed.
- *Model Refinement:* After the backtracking, we refine the automaton model that is used for the analysis on-the-fly, by replacing the location(s) with chaotic behavior in newly refined variables x by the locations that result from the refinement. After this modification, we can apply the unchanged analysis algorithm on the parents of refined nodes to update the search tree.
- *Reachability Computation:* According to a heuristics, we iteratively apply Algorithm 1 to non-completed nodes in the search tree, until we detect an unsafe set (in which case a new refinement iteration starts) or all nodes are completed (in which case the model is safe).

Algorithm 1: SuccessorComputation

```
   input :
1    SearchTree tree;
2    Node n_0 = (l_0, P_0, V_0) in tree;          // non-completed node;
3    Set V of variables that are chaotic in l_0;
4    Set C of the ODEs of l_0;
5    Time horizon T = mτ;
6  if l_0 is not urgent then
       /* compute the flow pipe segments                              */
7      for i = 1 to m do
8          P_i := flow(P_{i-1}, C, τ);
9          n_i :=tree.addChild(n_{i-1}, (l_0, P_i, V));
       /* compute jump successors                                     */
10     for i = 1 to m do
11         foreach transition e = (l_0, α, g, h, l) do
12             P := h(P_i ∩ g) ∩ Inv(l);
13             tree.addChild(n_i, (l, P, ∅));
14     for i = 0 to m do
15         mark n_i completed;
16 if l_0 is urgent then
       /* compute jump successors                                     */
17     foreach transition e = (l_0, α, g, h, l) do
18         P := h(P_0 ∩ g) ∩ Inv(l);
19         tree.addChild(n_0, (l, P, ∅));
20 mark n_0 completed;
```

In the following we explain how Algorithm 1 generates the successors of a tree node (l_0, P_0, V_0).

First the algorithm computes the time successors if the location is not urgent (lines 7-9): The set of states $\mathit{flow}(P_i, C, \tau)$ reachable from the node n_i under dynamics C within time τ is computed for all flow pipe segments within the time horizon, and added as a child of n_i, with the set V of the chaotic variables in l_0 attached to it. Note that, though we use a fixed step size τ, it could also be adjusted dynamically.

Next, the successor nodes of each flow pipe segment n_i are computed that are reachable via a discrete transition (lines 10-13). For each transition $e = (l_0, \alpha, g, f, l)$, the set of states P is computed that is reachable via transition e when starting in P_i (line 12). The successor $(l, P, \emptyset)$ is inserted into the search tree as a child of n_i; it is labeled with the empty set of variables since no chaotic behavior was involved (line 13).

Finally, since all possible successors of all n_i are added to the search tree, they are marked as completed (lines 14-15). Optionally, we can also mark all new nodes, whose state sets are included in the state set of another node with the same location component, as completed. Since the inclusion check is expensive, it is not done for each new node, but in a heuristic manner.

If the node n_0 is urgent, only the jump successors are computed (lines 16-19).

In either case n_0 is marked as completed since all possible successor state have been computed (line 20).

5.2 Implementation

We integrated the proposed CEGAR-approach into the analysis tool SpaceEx. Some implementation details are discussed in the following paragraphs.

Some SpaceEx *Implementation Details.* The SpaceEx tool computes the set of reachable states in so-called *iterations*. In each iteration, a state set is chosen, for which both the time elapse and the jump successors are computed. The *waiting list* of states that are reachable but have not yet been analyzed, is initialized with the set of initial states. At the beginning of an iteration, the front element w of the waiting list is picked for exploration. First the time elapse successors T of w are computed and stored in the set of reachable states (*passed list*). Afterwards, the jump successors J are computed for each $s \in T$. These states are added to the waiting list, as they are non-completed.

In SpaceEx, the *passed* and the *waiting list* are implemented as *FIFO* (first in, first out) queues, i. e., elements are added at the end of the list and taken from the front.

When either the waiting list is empty (i. e., a fixed-point is reached) or the specified number of iterations is completed, the analysis stops. The reachable states are the union of the state sets in the passed and the waiting list. If *bad states* are specified, the intersection of the reachable and the bad states is computed.

Search Tree. An important modification we had to make in SpaceEx is the way of storing reachable state sets discovered during the analysis. Whereas SpaceEx stores those sets in a queue, our algorithm relies on a search tree. Thus we added a corresponding tree data structure. We distinguish between *jump successor* and *time successor* nodes which we represent graphically in Figure 10 by filled rectangles and filled circles, respectively. The set of initial states are the children of the distinguished root node, which is represented by an empty circle. Each node can have several jump and at most one time successor nodes as children. For a faster access, each jump successor node stores a set of pointers to the next jump successors in its subtree (dashed arrows in Figure 10).

To indicate whether all successors of a node have been computed, we introduce a *completed* flag. In each iteration, we determine a non-completed tree node. If its location is non-urgent, we compute its time successors and the jump successors of all time successors. Afterwards, the chosen node and all computed time successors are marked as completed. For urgent locations, only the jump successors are computed. We use *breadth-first search* (BFS) to find the next non-completed tree node. The iterative search stops if either all tree nodes are completed or the state set of a node intersects with a given set of bad states. The latter is followed by a refinement and the deletion of a part of the search

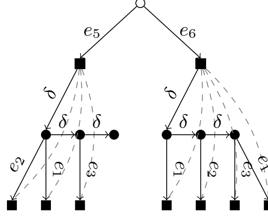

Fig. 10: The search tree (empty circle: distinguished root; filled rectangle: jump successor node; filled circle: time successor node; dashed connections: shortcut to the next jump successors)

tree (note, that the parents of deleted nodes are marked as non-completed). After this backtracking we start the next iteration with a new BFS from the root node. If the BFS gives us a node which already has some children then this node was previously completed and some its children were deleted by backtracking. In this case we check for each successor whether it is already included as a child of the node before adding it to the tree.

Refinement relies on counterexample paths, i. e. on paths from an initial to a bad state. To support more information about counterexample paths, we annotate the nodes in the search tree as follows. Each jump successor node contains a reference to the transition that led to it, and each time successor node stores the time horizon that corresponds to its time interval in the flowpipe computation.

Urgent Locations. Our implementation supports urgent locations, for which no time successors are computed.

Bad States. In SpaceEx, a set of bad (*forbidden*) states can be specified by the user. After termination of the reachability analysis algorithm, the intersection of the reachable states with the forbidden ones are computed and returned as output information.

In our implementation we stop the reachability computation once a reachable bad state is found. Therefore, we perform the intersection check for each node directly after it has been added to the tree. This allows us to perform a refinement as soon as a reachable bad state is detected.

Refinement. When a counterexample is detected, a heuristics chooses a (set of) location(s) and corresponding conditional ODEs for the refinement. We extend the set of active ODEs and refine the hybrid automaton model on-the-fly. Afterwards, the analysis automatically uses the new automaton model and the backtracked search tree to continue the reachability analysis.

Backtracking. When the model refinement is completed, we delete all nodes (and their subtrees) whose location was refined. The parents of deleted nodes

are marked as non-completed. This triggers that their successors will be re-computed. Since we use a BFS search for non-completed nodes, first the successors of such nodes will be computed, before considering other nodes.

Refinement Heuristics. We implemented a command line interface that allows us to choose the set of locations and corresponding conditional ODEs for the refinement manually whenever a counterexample was detected. This provides us with the most flexibility since any strategy can be applied. We plan to investigate several heuristics and to implement automated strategies for the promising heuristics.

Analysis Output. In case a counterexample path is detected that is fully refined, we abort the analysis and output the counterexample path. It can be used to identify the part of the model that leads to a violation of the specified property. Otherwise, the analysis is continued until either a fixed-point is found or the number of maximal allowed iterations was computed.

5.3 Example

We use the 2-tank example from Section 2 to illustrate how the implementation works. Up to now, we used the PHAVer scenario of SpaceEx for the reachability analysis since it computes exact results for our model and is much faster than a reachability analysis using the LGG scenario. However, we integrated our approach into the LGG scenario to be able to verify more complex examples in the future.

We first present our integrated CEGAR method on a two tank model with a single pump. Afterwards, we show the results for the presented two tank model. All model files and the SpaceEx version with the integrated CEGAR method are available for download at `http://ths.rwth-aachen.de/research/tools/spaceex-with-cegar/`.

Model with a Single Pump. First, we model a system of the two tanks without the second pump P_2. Initially, the pump P_1 is switched off, i. e., we start in the urgent location $(\text{off}_1^{in}, \text{comm})$. The initial variable valuation is:

Constants: $k_1 = 1$, $\delta = 3$, $L = 2$, $U = 30$
Pump state: $P_1 = 0$
Continuous vars: $h_1 = 7$, $h_2 = 5$, $t = 0$

We assume that the user input is $P_1^+ = 1$, $P_1^- = 0$ in the beginning. We want to prove that the water level of tank T_1 will never fall below 2, i. e. we define the set of unsafe states as φ_1 with $\varphi_1 := h_1 \leq 2$.

- The first counterexample is detected by the analysis for an initial user input $P_1^+ = 1$ and $P_1^- = 0$, along the location sequence $(\text{off}_1^{in}, \text{comm})$, $(\text{off}_1, \text{cycle})$, $(\text{on}_1^{in}, \text{cycle})$, where h_1 behaves chaotic. Thus, the unsafe states are reachable and we refine the location $(\text{on}_1^{in}, \text{cycle})$ with the first conditional ODE of h_1, which reduces to $(P_1 \wedge h_1 \geq 0, \quad \dot{h}_1 = 1)$.

– When the analysis reaches the location $(\text{on}_1^{in}, \text{cycle})$ via the previously described path, the behavior for h_1 is specified, i. e. after a cycle time of three time units, the value of h_1 has decreased from seven to four. Afterwards, the input reading is synchronized in location $(\text{on}_1^{in}, \text{comm})$. We reach again the location $(\text{on}_1, \text{cycle})$. For chaotic user input, both location $(\text{off}_1^{in}, \text{cycle})$ and $(\text{on}_1^{in}, \text{cycle})$ are reachable. If $(\text{off}_1^{in}, \text{cycle})$ is analyzed first, a counterexample is found which can be resolved using the conditional ODE $(\neg P_1, \quad \dot{h}_1 = 0)$. However, if location $(\text{on}_1^{in}, \text{comm})$ is processed, time can elapse, which yields $h_1 = 1$ at the end of the second PLC cycle. A water level below $L = 2$ violates our property. Since the counterexample is fully refined, the analysis is aborted since the model is incorrect.

The Two Tank Model. Let us now consider both pumps, which are switched off initially in location $(\text{off}_1^{in}, \text{off}_2^{in}, \text{comm})$. The initial variable valuation is:

Constants: $k_1 = 5$, $k_2 = 3$, $\delta = 1$, $L = 1$, $U = 30$
Continuous vars: $h_1 = 5$, $h_2 = 5$, $P_1 = 0$, $P_2 = 0$, $t = 0$

We want to check that the water level of the tanks is always above L, i. e. we define the set of unsafe states as $\varphi_1 \wedge \varphi_2$ with $\varphi_1 := h_1 \leq L$ and $\varphi_2 := h_2 \leq L$.

– The first detected counterexample is $(\text{off}_1^{in}, \text{off}_2^{in}, \text{comm})$, $(\text{off}_1, \text{off}_2, \text{cycle})$, $(\text{on}_1^{in}, \text{on}_2^{in}, \text{cycle})$ for the initial user input $P_1^+ = 1$, $P_1^- = 0$, $P_2^+ = 1$, and $P_2^- = 0$. We refine the last location on the path using $(c_1, \text{ODE}_1^{h_1}) = \varphi_{1\to 2} \wedge \varphi_{2\to 1}, \quad \dot{h}_1 = k_2 - k_1)$ and $(c_1, \text{ODE}_1^{h_2}) = \varphi_{1\to 2} \wedge \varphi_{2\to 1}, \quad \dot{h}_2 = k_1 - k_2)$.
– Now, time can elapse in location $(\text{on}_1^{in}, \text{on}_2^{in}, \text{cycle})$ and the values of h_1 and h_2 are decreased or increased according to the differential equations. After the first PLC cycle, we have $h_1 = 3$ and $h_2 = 7$.
– Depending on the user input, the locations each pump might be switched on or off, thus there are four jumps to different locations possible. Depending on the order in which they are analyzed, several refinements are neccessary before the case that both pumps stay switched on is processed.
  - When the first three locations are processed, counterexamples are detected which can be resolved using those conditional ODEs whose conditions are enabled.
  - With a user input $P_1^+ = P_1^- = P_2^+ = P_2^- = 0$ for the second PLC cycle, both pumps stay switched on and time can elapse again. Thus, at the end of the second cycle, we have $h_1 = 1$ and $h_2 = 9$. Since the value of h_1 is again below L, we have detected the same counterexample than in the smaller model with only a single pump. Note that it depends on the pumping capacity k_i of the pumps and on the initial values of h_i, which tank can dry out first.

The models can be corrected by lifting the position of the lower sensors in the tanks, i. e. for a new sensor position $L' > L + 2\delta k_1$ the models are safe.

6 User Manual

6.1 Getting Started

The Linux binaries of our tool and a benchmark set are available for download on our website. The content of the Tarball-archive is listed in Table 2.

http://ths.rwth-aachen.de/research/tools/spaceex-with-cegar/

The binary distributable comes in a Tarball archive, which can be extracted using:

```
tar xzf spaceex_cegar.tar.gz
```

The provided Linux binaries have been compiled on a 64bit Ubuntu 12.04 machine. To run the tool via commandline, switch to the location of the executable and run:

```
./spaceex_64bit/spaceex_cegar --config [path to config
    file] -m [path to model file] --dynamics [path to
    dynamics file]
```

We provide executable bash scripts for our benchmarks in the `benchmark` folder. I. e. the CEGAR analysis is automatically started with the correct parameters by the script. Switch to the location of the bash script, ensure that the script file is marked as executable by running:

```
chmod +x run_{instance}
```

The benchmark can be executed by running:

```
./run_{instance}
```

The benchmarks that we provide are listed in Table 1.

Table 1: The benchmarks that we provide.

Benchmark	Manual Control	Script Name
benchmarks/two_tank_example_single_pump	none chaotic	run_without_user run_with_chaotic_user
benchmarks/two_tank_example	none chaotic	run_without_user run_with_chaotic_user

Table 2: The content of the downloadable tar-archive.

File/Folder	Description
`LICENCE.txt`	license file
`README.txt`	readme file
`spaceex_64bit/spaceex_cegar`	bash script to run the tool
`spaceex_64bit/bin/sspaceex_cegar`	executable of the tool
`spaceex_64bit/lib`	folder with precompiled libraries
`benchmarks/two_tank_example_single_pump/chaotic_user`	folder with the model, config, and dynamics file for the two tank example with a single pump with chaotic user input
`benchmarks/two_tank_example_single_pump/no_user/`	folder with the model, config, and dynamics file for the two tank example with a single pump without user
`benchmarks/two_tank_example_single_pump/run_with_chaotic_user`	bash script to run the two tank example with a single pump with chaotic user input
`benchmarks/two_tank_example_single_pump/run_without_user`	bash script to run the two tank example with a single pump without user input
`benchmarks/two_tank_example/chaotic_user`	folder with the model, config, and dynamics file for the two tank example with chaotic user input
`benchmarks/two_tank_example/no_user/`	folder with the model, config, and dynamics file for the two tank example without user
`benchmarks/two_tank_example/run_with_chaotic_user`	bash script to run the two tank example with chaotic user input
`benchmarks/two_tank_example/run_without_user`	bash script to run the two tank example without user input

6.2 Input Format

Our tool needs three separate input files: The *model file* describes the analyzed system. We did not alter the SpaceEx syntax. However, we only support single automata instead of networks. The *configuration file* describes the analysis parameters for SpaceEx (including initial and forbidden states) and in the *dynamics file* a set of conditional ODEs is given, that specifies the dynamics of the system. All files should be provided as commandline arguments when the tool is started. In case no dynamics file is given, our tool asks for it when the first counterexample was detected.

Model File. A SpaceEx model file describes a network of automata. The input/output behavior between the automata is given by a set of network automata. In general, we adhere to the SpaceEx syntax for which a detailed manual can be found at the SpaceEx homepage [18].
However, we do not support the on-the-fly composition of a network of automata. Thus, we have to restrict the SpaceEx model file to specify a single automaton and a single network automaton that defines a single instance of the specified automaton.
All locations whose name includes `URGENT` are handled as urgent locations. Although time cannot elapse in such a location, it is necessary to provide differential equations for all continuous variables. Otherwise, SpaceEx assumes chaotic behavior as soon as the location is entered.

```
<!-- Example for an urgent location -->
<location id="1" name="loc_URGENT">
  <!-- An invariant might be specified here-->
  <invariant/>
  <!-- specify the dynamics for all continuous
     variables -->
  <flow>var1' == 0 &amp; var2' == 0</flow>
</location>
```

For the refinement and the zeno avoidance, the following variables, labels, locations and transitions are needed additionally. These elements should form a connected component that is not reachable from any other location, i.e. they are neglected during the analysis. However, we need them to add locations and transitions during a refinement step.

```
<!-- The following variables and transition labels
   are needed for the refinement and the zeno
   avoidance -->
<param dynamics="any" name="t" controlled="true"
   local="true" type="real" d2="1" d1="1"/>
<param dynamics="any" name="zeno_t" controlled="true"
    local="true" type="real" d2="1" d1="1"/>
<param name="copy_transition" local="true" type="
   label"/>
```

```
<!-- This location is needed for the refinement -->
<location name="refinement_loc" id="1000">
  <!-- Insert flow equations with right-hand side '0'
      for each continuous variable -->
  <flow>[default flows] & t' == 1 & zeno_t' == 0</
     flow>
</location>
<!-- A transition with guard true and without
   assignments is needed for the refinement -->
<transition source="1000" target="1000">
  <label>copy_transition</label>
</transition>
<!-- The following copy transition is needed for the
   under-approximative zeno avoidance -->
<transition source="1000" target="1000">
  <label>copy_transition</label>
  <guard>zeno_t >= epsilon</guard>
  <assignment>zeno_t = 0</assignment>
</transition>
```

Configuration File. A configuration file specifies the analysis parameters, a set of initial and a set of forbidden states. We list some parameters in Table 3 but refer to the SpaceEx homepage [18] for more detailed information.
Our implementation depends on fixed values for the configuration parameters that are given in Table 4.

Dynamics File. The dynamics of the model are specified in a `.xml` file using conditional ordinary differential equations. We restrict the ODEs to be linear since the LGG scenario of SpaceEx does not support non-linear ODEs. The doctype definition of the dynamics file is given in Figure 11.
The root element of a dynamics file is the `condODEsys` tag that specifies a set of

```
<!ELEMENT condODEsys (condODE)+>
<!ATTLIST condODEsys refersTo CDATA #REQUIRED>
<!ELEMENT condODE (cond,(equation)+)>
<!ELEMENT cond (#PCDATA)>
<!ELEMENT equation (#PCDATA)>
```

Fig. 11: Doctype definition of the dynamics file.

conditional ODEs. The attribute `refersTo` specifies the system model to which the conditional ODE file belongs, i.e. it should match the analyzed system that is specified in the model file and marked as the analyzed system in the config file.

Table 3: Some SpaceEx configuration parameters.

Name	Command	Description
System	`system = tanks`	The analyzed system must be a network component.
Initial States	`initially ="(loc(aut)== loc1 & system.var1 == 1)"`	Initial location and variable constraints.
Forbidden States	`forbidden = "h1 <= min1"`	Location and variable constraints. If no forbidden states are specified, a reachability analysis is performed, i.e. no counterexamples will be detected.
Sampling Time	`sampling-time = 0.1`	Discretization step of the time horizon.
Time Horizon	`time-horizon = 4`	Maximal time elapse in each iteration.
Iterations	`iter-max = -1`	Maximal number of analysis iterations. The value *-1* starts an analysis that runs until a fixed point is found.
Relative Error	`rel-err = 1.0e-12`	The relative error.
Absolute Error	`abs-err = 1.0e-13`	The absolute error.

Table 4: The SpaceEx configuration parameters that are fixed for our tool.

Name	Command	Description
Scenario	`scenario = supp`	Currently, we only support the LGG Support Function Scenario.
Representation	`directions = box`	The state set representation.
Clustering	`clustering = 0`	Currently, the clustering must be set to 0.
Set-Aggregation	`set-aggregation = "none"`	Currently, the set-aggregation must be set to 0.

```
<!-- A conditional ODE system contains a set of
    conditional ODEs -->
<condODEsys refersTo="system" />
```

A conditional ODE contains a condition and a non-empty set of linear ODEs.

```
<!-- A conditional ODE consists of a condition and a
    non-empty set of differential equations -->
<condODE />
```

A condition is a closed linear predicate over the physical quantities of the plant and the controller's variables. It is given by a single constraint or a conjunction of constraints. The comparison operators $==, \geq, \leq$ are supported. Note, that strict operators as well as negations and disjunctions are not supported, since SpaceEx cannot handle non-convex constraints in invariants and guards. Boolean variables can be assigned with `var == 1` or `var` (*true*) or with `var == 0` or `NOT var` (*false*). An example condition is `a >= 1.3 AND a == c AND NOT b`.

```
<!-- A condition is a conjuction of constraints -->
</cond>
```

A linear ODE is specified using the following notation: `c' == 3.1a + b`. The left-hand side contains the primed continuous variable for which the dynamics is given. The right-hand side contains a linear equation over the discrete variables of the controller and the continuous variables that model the plant dynamics.

```
<!-- An equation specifies a linear differential
    equation -->
<equation/>
```

7 Usage

After the analysis is started each time a counterexample is detected, it is printed on the console. The locations visited along the path are given together with the information, of this location visited as a jump successor (`j`) and if time elapse has been computed (`t`).

```
Counterexample path:
{off1_in_off2_in_control1_comm_URGENT}, j, t
{off1_URGENT_off2_URGENT_control2_cycle1}, j, t
{on1_in_on2_in_control2_cycle$$0}, j, t
{on1_in_on2_in_control1_comm_URGENT}, j, t
{on1_on2_URGENT_control2_cycle}, j, t
{on1_in_on2_in_control2_cycle$$0}, j, t
```

If the dynamics file location was not passed as an initial configuration parameter, it has to be specified when the first counterexample is found.

```
Enter name of conditional ODE XML file:
[condODEs.xml]
```

When a counterexample is detected, you can decide to refine (enter 'r') or to continue the analysis (enter 'c'). This enables you to apply a set of refinements before you continue the analysis.

```
What's next? (R)efine / (C)ontinue:
```

When `(R)refine` is selected, you have to choose a location that has to be refined. Note, that fully refined locations and urgent locations are not listed. To continue, enter the corresponding number of the location that should be refined, i.e. to refine location *loc2*, enter '2'.

```
Choose a location to refine :
1) loc1
2) loc2
```

Next, you have to choose a conditional ODE that is used to refine the location. This is done analogously. Note, that only those conditional ODE whose condition intersects with the invariant of the chosen location and that have not yet been used to refine this location are shown.

Afterwards, the refinement status of the chosen location is updated:

```
Location: off_1_in_control2_cycle
1:[X], 2:[-], 3:[ ]
```

Table 5 explains the syntax of the refinement status.

When `(C)ontinue` was selected, the selected refinement is performed and after-

Table 5: The Refinement status for a given location.

Notation	Description
`i:[X]`	indicates, that the i-th conditional ode is marked (or has previously been used) for refinement
`i:[-]`	indicates, that the i-th conditional ode is not allowed for the refinement of the chosen location
`i:[ ]`	indicates, that the i-th conditional ode is allowed for refinement, but has not been chosen so far

wards, the analysis continues. Note, that the analysis is aborted, if no pair of location and conditional ODE was marked for refinement.

When a location `loc` is refined, several copies of the location are created. The non-chaotic location instances are named `loc$$i$`, where i indicates the ith copy of the original location. Copies with chaotic behavior are named `loc$$ineg_j` where j indicates the jth chaotic location copy.

7.1 Analysis Result

The analysis is finished if either a fixed point is computed, the maximal number of iterations is reached or a fully refined counterexample is detected. In the latter case, it is printed along with the following information: `Fully refined counterexample detected`

At the end of the analysis, a short report is given which includes the following information:

Message	Description
`Fully refined counterexample detected after 8 iterations.`	A fully refined counterexample was detected
`Iteration 8 done after 0s`	The number of computed iterations
`Computing reachable states done after 9.158s`	The overall computation time
`Performed max. number of iterations (5)without finding fixpoint.`	A fixed point is computed
`Forbidden states are reachable.`	The reachable states intersect with the forbidden states

8 Conclusion

In this paper we proposed a CEGAR-based framework for the analysis of SFC-controlled chemical plants that can handle urgent locations and transitions, and non-convex invariants. We described its implementation in SpaceEx and presented a small example. As future work, we plan to analyze a larger case study with parallel acting controllers. Although a complex system will affect the running time of the analysis, we expect, that our CEGAR approach will not cause too much overhead since the analysis is interrupted if a counterexample is detected. Especially when the checked property depends only on a part of the dynamic plant behavior, we can benefit from the CEGAR approach since it suffices to analyze an abstraction instead of the concrete model. Moreover, we will cover further optimizations of the presented algorithm.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
2. Alur, R., Dang, T., Ivancic, F.: Counter-example guided predicate abstraction of hybrid systems. In: Proc. of TACAS'13. LNCS, vol. 2619, pp. 208–223. Springer (2003)
3. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Proc. of CAV'02. LNCS, vol. 2404, pp. 746–770. Springer (2002)

4. Balluchi, A., Casagrande, A., Collins, P., Ferrari, A., Villa, T., Sangiovanni-Vincentelli, A.L.: Ariadne: A framework for reachability analysis of hybrid automata. In: Proc. of MTNS'06 (2006)
5. Baresi, L., Carmeli, S., Monti, A., Pezzè, M.: PLC programming languages: A formal approach. In: Proc. of Automation '98. ANIPLA (1998)
6. Bauer, N.: Formale Analyse von Sequential Function Charts. Ph.D. thesis, Universität Dortmund (2004)
7. Bauer, N., Huuck, R., Lukoschus, B., Engell, S.: A unifying semantics for sequential function charts. In: In the Final Report of the SoftSpez DFG Priority Program. LNCS, vol. 3147, pp. 400–418. Springer (2004)
8. Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T.T., Ladan, H., Podelski, A., Wehrle, M.: Abstraction-based guided search for hybrid systems. In: Proc. of SPIN'13. LNCS, vol. 7976, pp. 117–134. Springer (2013)
9. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. of CAV'13. LNCS, vol. 8044, pp. 258–263. Springer (2013)
10. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. Journal of Foundations of Computer Science 14(04), 583–604 (2003)
11. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Proc. of TACAS'03. LNCS, vol. 2619, pp. 192–207. Springer (2003)
12. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. of CAV'00. LNCS, vol. 1855, pp. 154–169. Springer (2000)
13. Dierks, H., Kupferschmid, S., Larsen, K.: Automatic abstraction refinement for timed automata. In: Proc. of FORMATS'07. LNCS, vol. 4763, pp. 114–129. Springer (2007)
14. Engell, S., Lohmann, S., Stursberg, O.: Verification of embedded supervisory controllers considering hybrid plant dynamics. Int. Journal of Software Engineering and Knowledge Engineering 15(2), 307–312 (2005)
15. Fehnker, A., Clarke, E., Jha, S., Krogh, B.: Refining abstractions of hybrid systems using counterexample fragments. In: Proc. of HSCC'05. LNCS, vol. 3414, pp. 242–257. Springer (2005)
16. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. of CAV'11. LNCS, vol. 6806, pp. 379–395. Springer (2011)
17. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. Int. Journal on Software Tools for Technology Transfer 10, 263–279 (2008)
18. Frehse, G.: An Introduction to SpaceEx v0.8. Verimag, France (2010), http://spaceex.imag.fr/documentation/user-documentation/introduction-spaceex-27
19. Frey, G., Litz, L.: Formal methods in PLC programming. In: Proc. of SMC'00. vol. 4, pp. 2431–2436. IEEEXplore (2000)
20. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. TAC'07' 52(5), 782–798 (2007)
21. Hassapis, G., Kotini, I., Doulgeri, Z.: Validation of a SFC software specification by using hybrid automata. In: Proc. of INCOM'98. pp. 65–70. Pergamon (1998)
22. Henzinger, T.A., Ho, P., Wong-Toi, H.: Hytech: A model checker for hybrid systems. Int. Journal on Software Tools for Technology Transfer 1(1-2), 110–122 (1997)
23. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences 57(1), 94–124 (1998)

24. Herceg, M., Kvasnica, M., Jones, C., Morari, M.: Multi-Parametric Toolbox 3.0. In: Proc. of the ECC'13. pp. 502–510. Zürich, Switzerland (2013)
25. Int. Electrotechnical Commission: Programmable Controllers, Part 3: Programming Languages, 61131-3 (2003)
26. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Proc. of HSCC'07. pp. 287–300. LNCS, Springer (2007)
27. Kurzhanskiy, A., Varaiya, P.: Ellipsoidal toolbox. Tech. rep., EECS, UC Berkeley (2006)
28. Lukoschus, B.: Compositional Verification of Industrial Control Systems - Methods and Case Studies. Ph.D. thesis, Christian-Albrechts-Universität zu Kiel (2005)
29. Minopoli, S., Frehse, G.: Non-convex invariants and urgency conditions on linear hybrid automata. In: Legay, A., Bozga, M. (eds.) Formal Modeling and Analysis of Timed Systems. LNCS, vol. 8711, pp. 176–190. Springer International Publishing (2014)
30. Mitchell, I., Tomlin, C.: Level set methods for computation in hybrid systems. In: Proc. of HSCC'00. LNCS, vol. 1790, pp. 310–323. Springer (2000)
31. Nellen, J., Ábrahám, E.: Hybrid sequential function charts. In: Proc. of MBMV'12. pp. 109–120. Verlag Dr. Kovac (2012)
32. Nellen, J., Ábrahám, E.: A CEGAR approach for the reachability analysis of PLC-controlled chemical plants. In: Proc. of FMi'14 (2014)
33. Platzer, A., Quesel, J.D.: Keymaera: A hybrid theorem prover for hybrid systems. In: Proc. of IJCAR'08. LNCS, vol. 5195, pp. 171–178. Springer (2008)
34. Prabhakar, P., Duggirala, P., Mitra, S., Viswanathan, M.: Hybrid-automata-based CEGAR for rectangular hybrid systems. In: Proc. of VMCAI'13. LNCS, vol. 7737, pp. 48–67. Springer (2013)
35. Segelken, M.: Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models. In: Proc. of CAV'07. LNCS, vol. 4590, pp. 433–448. Springer (2007)