Divide and Conquer: Variable Set Separation in Hybrid Systems Reachability Analysis*

Stefan Schupp Johanna Nellen Erika Ábrahám
RWTH Aachen University, Germany
{ stefan.schupp | johanna.nellen | abraham }@cs.rwth-aachen.de

In this paper we propose an improvement for flowpipe-construction-based reachability analysis techniques for hybrid systems. Such methods apply iterative successor computations to pave the reachable region of the state space by state sets in an over-approximative manner. As the computational costs steeply increase with the dimension, in this work we analyse the possibilities for improving scalability by dividing the search space in sub-spaces and execute reachability computations in the sub-spaces instead of the global space. We formalise such an algorithm and provide experimental evaluations to compare the efficiency as well as the precision of our sub-space search to the original search in the global space.

1 Introduction

The increasing usage of digital control for safety-critical dynamical systems has resulted in an increasing need for formal verification approaches for *hybrid systems*, i.e., for systems with mixed discrete-continuous behaviour, which are often modelled as *hybrid automata*. Due to intensive research, nowadays several approaches and tools exist for the *reachability analysis* of hybrid automata. As the reachability problem for hybrid automata is in general undecidable, most approaches compute an *over-approximation* of the set of states that are reachable in a given hybrid automaton model. Due to the over-approximation, these techniques can be used to prove the safety of system models, i.e., the fact that a given set of unsafe states is not reachable in the model, but they cannot be used to prove unsafety.

In this work we focus on *flowpipe-construction-based* reachability analysis techniques. These techniques use certain data types to *represent* state sets, whereas each representation has its strengths and weaknesses in terms of precision, memory requirements, and efficiency of certain operations on them which are needed for the reachability computations. The reachability analysis starts from an initial state set and iteratively over-approximates successors by further state sets. For a given set of states, the successors via a discrete computation step (*jump*) are over-approximated by a single set, the successors via time evolution (the so-called *flowpipe*) are covered in an over-approximative manner by a sequence of state sets.

Unfortunately, these successor computations often lead to either strong over-approximations or high computational costs. Though state-of-the-art tools like SpaceEx [7], Flow* [4], or HyPro [19] can already successfully verify a wide range of challenging applications, they still have problems to analyse large models with complex behaviour. Such models arise for example from applications, where a physical or chemical plant is controlled by a discrete controller. Our focus is on digital control by programs running on *programmable logic controllers* (*PLCs*). To build a formal model of such a system, the PLCs, the programs running on them, the dynamic plant behaviour, and the interactions between these

*This work was partially supported by the German Research Council (DFG) in the context of the HyPro project.

De Vink and Wiklicky (Eds.): QAPL 2017
EPTCS 250, 2017, pp. 1–14, doi:10.4204/EPTCS.250.1

components can be modelled by a hybrid automaton, to which available reachability analysis tools can be applied. For practically relevant systems, however, the size of the resulting composed models often exceeds the capabilities of state-of-the-art tools.

Whereas general techniques to increase the scalability of reachability analysis are hard to develop, for dedicated model types there might be some hot spots. Models of PLC-controlled plants have some specific properties we can exploit: Firstly, they possess a relevant number of *discrete variables*. Secondly, some actions are triggered by deadlines, modelled by the values of *clocks* along with corresponding thresholds. Thirdly, the evolution of some physical quantities might depend on the time *linearly*, others not. In standard reachability analysis, these model parts are handled uniquely. In this paper we propose to split the state space into several sub-spaces, between which the dependence is loose enough to execute successor computations independently. Though this procedure leads to additional over-approximation, the error can be reduced. Furthermore, we show on some experiments that this additional over-approximation is often minor and is well compensated by the reduced computational requirements.

We are aware of the work [5] that is closely related to the work described in this paper. The authors of [5] also use variable set separation and computations in sub-spaces, but with two main differences. On the one hand, the work [5] is more general as they allow also closer dependencies between the sub-spaces than we can support. On the other hand, their work is restricted to Taylor models, whereas our approach is applicable to any state set representation type.

Overview After providing some preliminaries in Section 2 and a description of our HyPro programming library in Section 3, in Section 4 we describe our method for the separation of variable dimensions and the modified reachability algorithm. We provide some experimental results in Section 5 before we conclude the paper in Section 6.

2 Preliminaries

Hybrid automata For a given set $X = \{x_1, \ldots, x_d\}$ of variables let $Pred_X$ be the set of all quantifier-free arithmetic predicates with free variables from X, using the standard syntax and semantics over the real domain. We use the notation $\dot{X} = \{\dot{x}_1, \ldots, \dot{x}_d\}$ to represent first derivatives and $X' = \{x'_1, \ldots, x'_d\}$ to represent the result of discrete resets of variable values. Sometimes we also see the variable space as the d-dimensional real space and use the vector notation $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$.

Definition 1 ([10]). *A* hybrid automaton $\mathcal{H} = (Loc, X, Flow, Inv, Edge, Init)$ *is a tuple specifying*

- *a finite set Loc of* locations *or* control modes*;*
- *a finite ordered set* $X = \{x_1, \ldots, x_d\}$ *of real-valued* variables*, where d is the* dimension *of* $\mathcal{H}$*;*
- *for each location its* flow *or* dynamics *by the function* $Flow : Loc \to Pred_{X \cup \dot{X}}$*;*
- *for each location an* invariant *by the function* $Inv : Loc \to Pred_X$*;*
- *a finite set* $Edge \subseteq Loc \times Pred_X \times Pred_{X \cup X'} \times Loc$ *of* discrete transitions *or* jumps*. For a jump* $(l_1, g, r, l_2) \in Edge$*,* l_1 *is its* source *location,* l_2 *is its* target *location, g specifies the jump's* guard*, and r its* reset *function;*
- *an* initial *predicate for each location by the function* $Init : Loc \to Pred_X$.

In this paper we consider only *autonomous linear* hybrid automata whose initial conditions, invariants and jump guards are linear and can be written in the form $Ax \leq b$ (where $x = (x_1, \ldots, x_d)$ are the

model variables, A is a $d \times d$ matrix and b a d-dimensional vector), whose jump resets are also linear and can be written as $x' = Ax$, and whose flows are defined by conjunctions of linear *ordinary differential equations* (*ODEs*), which can be written[1] as $\dot{x} = Ax$. Note that such automata allow only linear predicates, whose solutions are convex polytopes.

A *state* $(l,x) \in Loc \times \mathbb{R}^d$ of a hybrid automaton specifies the location $l \in Loc$ in which the control resides and the current values $x \in \mathbb{R}^d$ of the variables. For $p \subseteq \mathbb{R}^d$, by (l,p) we denote the state set $\{(l,x) \mid x \in p\}$. An *execution* $(l_0,x_0) \xrightarrow{t_0} (l_1,x_1) \xrightarrow{e_1} (l_2,x_2) \xrightarrow{t_2} \dots$ of a hybrid automaton starts in an initial state (l_0,x_0) such that x_0 satisfies $Init(l_0)$, and executes a sequence of alternating continuous and discrete steps. A continuous step $(l_i,x_i) \xrightarrow{t_i} (l_{i+1},x_{i+1})$ with $l_i = l_{i+1}$ models time evolution: starting from x_i, the variable values evolve according to the flow (ODEs) $Flow(l_i)$ of the current location for t_i time units, where the location's invariant must hold during the whole duration of the step. A discrete step $(l_i,x_i) \xrightarrow{e_i} (l_{i+1},x_{i+1})$ typically models controller execution: if the source of a jump e_i is l_i, the guard of e_i is satisfied by x_i, the reset predicate of e_i is satisfied by (x_i,x_{i+1}), and l_{i+1}'s invariant is true for x_{i+1} then the jump e_i can be taken, moving the control from l_i to l_{i+1} with resulting variable values x_{i+1}.

A state of a hybrid automaton $\mathscr{H}$ is called *reachable* if there is an execution leading to it. Given a set T of unsafe states, $\mathscr{H}$ is called *safe* if no state from T is reachable in $\mathscr{H}$.

Hybrid automata can be composed using *parallel composition*, which we do not define here formally. Intuitively, jumps in different components can be synchronised (using synchronisation labels) if they should take place simultaneously, whereas local computation steps can also be executed in isolation; time evolves simultaneously in all components.

Reachability analysis The *reachability problem* for hybrid automata is the problem to decide whether a given state (or any state from a given set) is reachable in a hybrid automaton. As the reachability problem for hybrid automata is in general undecidable, some approaches aim at computing an *over-approximation* of the set of reachable states of a given hybrid automaton. We focus on approaches based on *flowpipe construction*, which iteratively over-approximate the set of reachable states by the union of a set of state sets. To *represent* a state set, typically either a *geometric* or a *symbolic* representation is used. Geometric representations specify state sets by geometric objects like boxes, (convex) polytopes, zonotopes, or ellipsoids, whereas symbolic representations use, e.g., support functions or Taylor models. These representations might have major differences in the precision of the representation (the size of over-approximation), the memory requirements and the computational effort needed to apply operations like intersection, union, linear transformation, Minkowski sum or test for emptiness. For example, boxes perform well in terms of computational effort for set operations, but usually introduce a large over-approximation error. Thus the choice of the representation is a compromise between the advantages and disadvantages regarding these measures.

Before the reachability analysis starts, all predicates $\varphi \in Pred_X$ in the respective linear hybrid automaton (initial predicates, invariants, jump guards) as well as the unsafe state set need to be represented in some state set representation (usually the same representation for all predicates). Furthermore, the jump resets need to be formalised as linear transformations.

For a given state set p, flowpipe-construction-based approaches compute the successors of the states from p by first over-approximating the set of states reachable via time evolution (*flowpipe*) and afterwards the set of states reachable from the flowpipe as jump successors. Time evolution is usually restricted to a *time horizon* (either per location or for the whole execution), which is divided into smaller time steps. The states reachable from p via one time step are over-approximated by a state set p_1, for which again

[1] Note that ODEs of the form $\dot{x} = Ax + b$ can be also encoded without a b component on the cost of new variables with zero derivatives. A similar approach is possible for jump resets of the form $x' = Ax + b$.

the time successors p_2 via one time step are computed. This procedure is repeated until the time horizon is reached or the successor set gets empty (due to the violation of the current location's invariant). The union of the resulting state sets $p_1, \ldots, p_k$, which are called *flowpipe segments*, over-approximates the flowpipe. For each outgoing jump and each of the flowpipe segments the jump successors are computed, to which the above procedure is applied iteratively until a given upper bound (*jump depth*) on the number of jumps is reached or until a fixed point is detected. Thus the reachability computation results in a *search tree* with state sets as nodes. In order to reduce the computational effort, *clustering* and *aggregation* can be applied to over-approximate the successors of the flowpipe segments for a given jump by a fewer number of segments respectively by a single state set.

Programmable logic controllers *Programmable logic controllers* (*PLCs*) are digital controllers widely used in industrial applications, for instance in production chains. A PLC has input and output pins that are connected with the sensors and the actuators of a plant. Control programs running on a PLC specify the output of the PLC in dependence of its input. These control programs are executed in a cyclic manner. First, the PLC *reads* the current state of the sensors and the actuators of the plant and stores this information in input registers. Next all programs on the PLC *execute* in parallel to compute the next output values based on the last input, and store the results in some output registers. These computations might use local variables, stored in some local registers. In the last step of the cycle the PLC *writes* the computed output values to the output pins that are connected to the actuators of the plant. In contrast to some implementations that assure a cycle duration within a time interval, for simplicity in this work we assume a constant cycle time (however, our approach can be easily extended to interval durations).

To model a plant we introduce variable sets V_{cont}, V_{act}, and V_{sen} to represent the state of physical quantities, the actuators, respectively the sensors (see left of Figure 1). For the modelling of a controller we use sets V_{in}, V_{out}, and V_{loc} of variables to represent the PLC registers for input, output respectively local variables. Additionally, we need one variable (clock) per PLC to account for the PLC cycle time.

A schematic overview of the hybrid automaton we use to model PLC-controlled plants is shown in the right of Figure 1. We could model the system by specifying hybrid automata models for the plant, the PLC, and the programs running on the PLC, and compose them using label synchronisation to model synchronous events in the PLC cycle. However, these models allow heavy interleaving between continuous time evolution and discrete PLC computation steps, leading to models that pose a challenge for reachability analysis tools. Therefore, we make use of the fact that the PLC execution between reading the input and writing the output has no influence on the plant's state: we model the plant evolution and the concurrent cyclic PLC execution by toggling between a controller model and a plant model, assuming that all controller actions are executed instantaneously after the input is read, the plant evolves for the duration of the PLC cycle, and the output is written at the end of the cycle. We refer to [18] for more information on the modelling of PLC-controlled plants.

3 The HyPro Library

As mentioned before, there are several state set representations that can be used in flowpipe-construction-based reachability analysis algorithms. Hybrid systems reachability analysis tools like, e.g., CORA [1], FLOW* [4], HYCREATE [11], HYREACH [13], SOAPBOX [9], and SPACEEX [7] implement different techniques using different geometric or symbolic state set representations, each of them having individual strengths and weaknesses. For example, SPACEEX uses support functions, whereas FLOW* makes use of Taylor models.

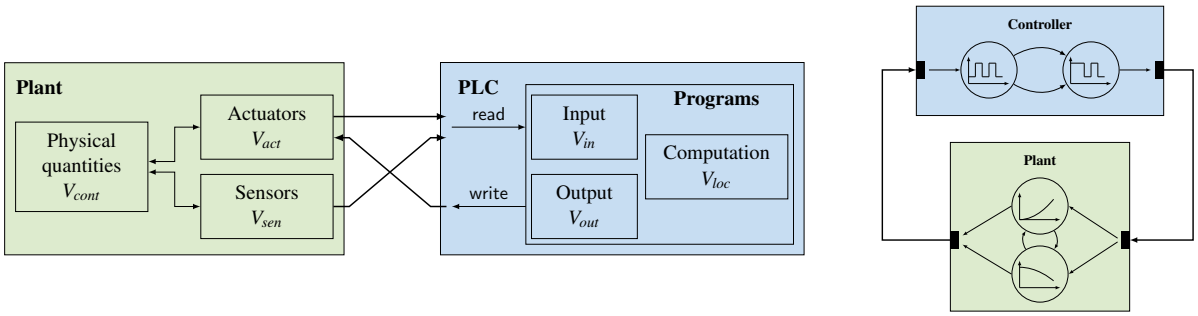

Figure 1: PLC controller: Interface between plant and controller (left) and cyclic execution model (right).

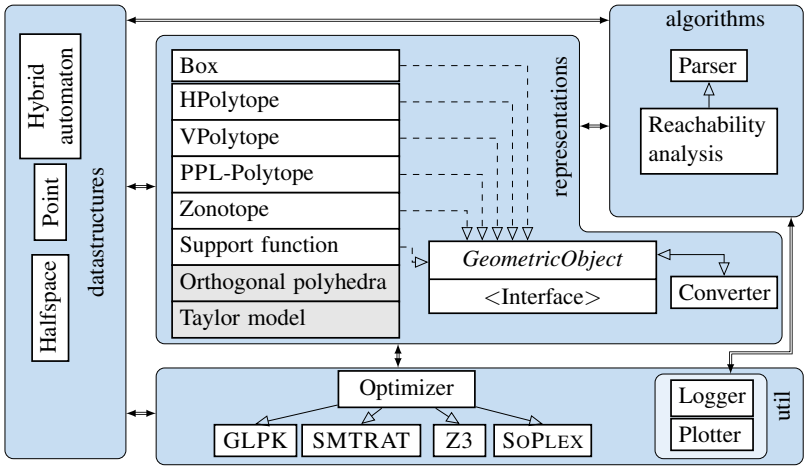

Figure 2: HyPro class structure [19].

The implementation of state set representations is tedious and time-consuming, and impedes the (even prototypical) implementation of new reachability analysis algorithms. To offer assistance for rapid implementation, we developed a free and open-source C++ programming library HyPro [19] (see Figure 2), which we will use in our experiments and which is published at `https://github.com/hypro/hypro`. HyPro contains implementations for several *state set representations* such as boxes [16], convex polytopes [21], zonotopes [8], support functions [14], orthogonal polyhedra [3], and Taylor models [4], different *operations* on them which are needed for the implementation of flowpipe-construction-based reachability analysis algorithms, and *conversions* between the different representations. *Reduction techniques* can be applied to reduce the representation sizes on the cost of additional over-approximation.

The implemented representations (with the exceptions of orthogonal polyhedra and Taylor models, depicted grey in Figure 2) share a *unified interface* to allow the usage of different representations within a single algorithm. This property is not only important for extensibility with new representations but also, e.g., for the implementation of counterexample-guided abstraction refinement (CEGAR) algorithms: the search can start with a low-precision but computationally cheap representation such as boxes, and it can be refined along paths that are detected to be potentially unsafe by switching to a high-precision but computationally more expensive representation.

Another important feature of HyPro is that it is templated in the *number type*, such that it can be instantiated both with exact as well as with inexact arithmetic. Linear solver backends such as GLPK

[15], SMTRAT [6], SoPlex [20], and Z3 [17], which are needed for the implementation of different operations and conversions, can be exchanged by the user by her tool of choice. The library is *thread-safe*, thus parallelisation can be exploited by the user. The efficient usage of the library is further eased by a *model parsing* module, a *plotting* engine, and various *debugging* tools.

In this work we illustrate the advantages of the HyPro library by proposing an algorithm to reduce the computational effort of the search on the cost of precision loss. Due to space restriction, we do not discuss refinement steps in this paper, but mention here that using HyPro the proposed method can be embedded into a CEGAR approach: if a potentially unsafe path is detected, more precise analysis can be used to check safety along those paths.

4 Reachability Analysis based on Variable Set Separation

Variable set separation and projective representation For practically relevant applications, the previously described modelling approach for PLC-controlled plants by hybrid automata leads to huge models, even if we exploit the mentioned reduction by restricted interleaving. The most serious problem is the high dimensionality: the variable set contains variables modelling the plant dynamics, the states of sensors and actuators, the input and output values of the PLC, the local variables used in program executions, and clocks for PLC cycle synchronisation. The high dimensionality leads to complex state set representations, causing heavy memory consumption and computationally expensive applications of state set operations during the reachability analysis.

To increase scalability and thus to allow the analysis of larger models, we start with some observations. Firstly, the variables of the PLC are *discrete* and thus their values do not change dynamically during time evolution but only upon taking a discrete transition in the controller part of the composed hybrid automaton. Furthermore, the states of actuators and sensors can be modelled by discrete variables, as actuator states change discretely (when writing the output) and the sensor values are relevant only at the beginning of each cycle (read plant state) as depicted in Figure 1. Thus only the physical quantities and the cycle clocks evolve continuously. Finally, computing flowpipes for clocks and other variables with constant derivatives can usually be done easier than for dynamics specified by general ODEs.

These observations gave us the idea to divide the variable set X into several disjoint subsets $X = X_1 \cup \ldots \cup X_n$ such that variables in the same subset X_i have some common properties relevant for reachability analysis. Once the variables are classified this way, we could try to modularise the reachability analysis computation by computing in the sub-spaces defined by the variable subsets, instead of computing in the global space. However, in order to compute reachability in the sub-spaces, the variables in different subsets must be independent in the sense that their evolutions do not directly influence each other. To be more formal, all predicates $\varphi \in Pred_X$ in the hybrid automaton definition must be decomposable to a conjunction $\varphi = \varphi_1 \wedge \ldots \varphi_n$ of predicates $\varphi_i \in Pred_{X_i}$ over the respective variable subsets X_i, and similarly for jump resets from $Pred_{X \cup X'}$ and flows from $Pred_{X \cup \dot{X}}$. If this condition holds then we call the subsets $X_1, \ldots, X_n$ themselves as well as variables from two different subsets *syntactically independent*.

Such a classification of the variable set X into syntactically independent subsets $X_1, \ldots, X_n$ allows us to represent (global) state sets $(l, p) \subseteq Loc \times \mathbb{R}^d$ by their projections $p \downarrow_{X_i} = p_i \subseteq \mathbb{R}^{|X_i|}$ to the sub-spaces; we call $(l, p_1, \ldots, p_n)$ the *projective representation* of (l, p) with respect to the variable separation $X_1, \ldots, X_n$. Note that the projective representation drops the connection between the sub-spaces and is therefore *over-approximative*, i.e., $p \subseteq p_1 \times \ldots \times p_n$ but in general $p \neq p_1 \times \ldots \times p_n$. One exception is the state set representation by *boxes*: the cross product of the projections of a box is the box itself, therefore the projective representation of boxes is exact.

Reachability computation based on variable set separation Given a separation of the variable set X into syntactically independent subsets $X_1,\ldots,X_n$ and projective representations based on this separation, we can over-approximate successors of a state set (l,p) by computing successors of its projective representation $(l,p_1,\ldots,p_n)$ in each sub-space modularly. As the computational effort for reachability analysis heavily increases with the dimension, this modular approach will help to reduce the running time. To explain why we need syntactical independence for sub-space computations, we first need a more formal description of how successor sets are computed:
(1) Reachability analysis computes for an initial set (l,p) the first flowpipe segment (l,Ω_0) that over-approximates all states reachable from p within a time interval $[0,\delta]$ in l as $\Omega_0 = (conv(p \cup e^{\delta A}p) \oplus \mathscr{B}) \cap Inv(l)$, where the flow in location l is $\dot{x} = Ax$, $e^{\delta A}$ is the matrix exponential for δA, $e^{\delta A}p$ are the states reachable from p at time point δ, $conv(\cdot)$ is the convex hull operator, $S_1 \oplus S_2 = \{a+b \mid a \in S_1 \wedge b \in S_2\}$ is the Minkowski sum of two sets, the bloating with the box $\mathscr{B}$ accounts for the non-linear behaviour between the time points 0 and δ, and $Inv(l)$ is the invariant for location l.
(2) Flowpipe segments (l,Ω_i) over-approximating the flowpipe within the time interval $[i\delta,(i+1)\delta]$ for $i > 0$ are computed by $\Omega_i = e^{\delta A}\Omega_{i-1} \cap Inv(l)$.
(3) For each jump e with source l, guard g, reset $x' = A'x$ and target location l', each flowpipe segment Ω_i is checked for possible successors along e by checking $(A'(\Omega_i \cap g)) \cap Inv(l')$ for emptiness. Non-empty successors are collected, possibly aggregated, and considered as initial state set(s) for location l'.

For each decomposition of X into syntactically independent variable sets $X_1,\ldots,X_n$, any flow $\dot{x} = Ax$ can be decomposed into $\wedge_{i=1}^n \dot{X}_i = A_i X_i$ (where we overload the notation X_i to also denote the sequence of variables in X_i). Similarly, $Inv(l) = \wedge_{i=1}^n Inv(l)_i$ with $Inv(l)_i \in Pred_{X_i}$. Furthermore, let $\mathscr{B}_i = \mathscr{B} \downarrow_{X_i}$ be the projection of $\mathscr{B}$ to X_i. Let $(l,p_0,\ldots,p_n)$ be the projective representation of a state set p,

$$\Omega_{0,i} = (conv(p_i \cup e^{\delta A_i} p_i) \oplus \mathscr{B}_i) \cap Inv(l)_i$$

and for each $j > 0$

$$\Omega_{j,i} = (e^{\delta A_i}\Omega_{j-1,i}) \cap Inv(l)_i\,.$$

Then $\Omega_j \subseteq \Omega_{j,1} \times \ldots \times \Omega_{j,n}$.

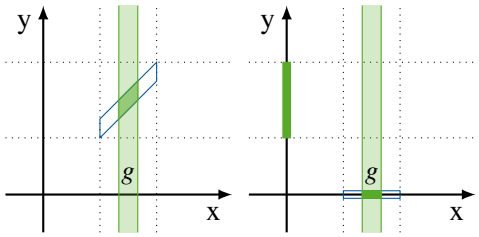

Figure 3: Intersection of a flowpipe segment with an invariant using global (left) and separated (right) variable sets.

The computations in the sub-spaces are precise as long as the initial set p is a box and the flowpipe resides inside the invariant, i.e., if p is a box and $\Omega_{m,i} \subseteq Inv(l)_i$ for all $1 \le m \le j$ and all $1 \le i \le n$ then $\Omega_j \downarrow_i = \Omega_{j,i}$.

However, syntactical independence does not imply semantical independence, as the different dimensions are usually still implicitly connected by the passage of time. If one of the projections runs out of a non-trivial invariant then the intersection with the (projection of the) invariant in a sub-space does not necessarily affect the computations in the other sub-spaces, thus the result might become over-approximative (see Figure 3). To increase precision, we can at least incorporate that if the projection of a flowpipe segment gets empty in one of the sub-spaces then the whole flowpipe segment gets empty: instead of $\Omega_{j,i}$ we use $\Omega'_{j,i}$ that is $\Omega_{j,i}$ if none of $\Omega_{j,k}$, $k = 1,\ldots,n$ is empty and the empty set otherwise.

For successors along jumps, the reachability computations in the sub-spaces work similarly. Also here, additional over-approximation might be introduced by intersections with guards and invariants in target locations, which we try to reduce by the above-described emptiness check.

As we use modular computations in sub-spaces, on the one hand our method speeds up reachability computations, but on the other hand it introduces additional over-approximations. Therefore, in our experiments we will thoroughly analyse the effect of our approach both to the running time as well as to the

over-approximation error. Besides the reduced computational effort, our method has further advantages. For example, state sets in the sub-spaces can be represented independently of each other, using different state set representations. We observed that the discrete variables can often be represented by boxes without serious over-approximation, whereas the plant dynamics requires a more precise representation, e.g. by support functions. Furthermore, for sub-spaces defined by clocks or by variables with constant derivatives one could use different, computationally less expensive techniques for computing flowpipes.

The algorithm Our reachability analysis algorithm based on variable set separation is presented in Algorithm 1. The input is a hybrid automaton H, a parameter δ that specifies the length of a time step in the flowpipe construction, a global time horizon T and a jump depth D that specify upper bounds on the total time duration respectively on the total number of jumps in the considered execution paths, and an aggregation flag that specifies whether aggregation should be applied to the successors of segments of a flowpipe along a jump. The algorithm outputs a set of flowpipe segments R, where the union of the segments in R is an over-approximation of the set of states that are reachable in H within the given time and jump bounds.

In a preprocessing step, we separate the variable set into three syntactically independent subsets of discrete variables, clocks, and the rest (lines 2-4). We have chosen this variable set separation as it seems to be practically helpful in our experiments, but other separation criteria could be defined, too. A state set representation can be chosen for each sub-space independently (line 5); for readability, in the algorithm we do not distinguish between state sets and their representations syntactically.

The invariant conditions for the initial states are checked for each initial set and each sub-space independently in the lines 8-13 and if the intersections of the initial sets with the invariants are non-empty in all sub-spaces then they are added to a set P of state sets, whose successors need to be determined. Additionally to the location and the projective representation of state sets, we attach to the state sets the time interval within which the represented states can be reached.

As long as P is not empty, we choose a state set $p \in P$. Before computing its flowpipe, we determine the set of jumps with p's location as source. As the values of discrete transitions do not change during time evolution, we can skip those jumps whose guard is violated by the initial values of the discrete variables, and store the remaining ones in the set E (line 16). The sets P^e will be used to collect non-empty successors from different flowpipe segments along the jumps $e \in E$.

Next we compute the segments of p's flowpipe as explained previously (lines 20-27). Instead of one single flowpipe of dimension d, our algorithm computes lower-dimensional flowpipes in the sub-spaces using a common time step size and a global time horizon (line 20) to be able to connect the flowpipe segments computed in the sub-spaces. The variable *isFirst*, initialised in line 18, is used to remember whether the flowpipe segment to be computed next is the first one (as the first one needs special handling).

Once the flowpipe segments are computed, their jump successors are determined and collected in the sets P^e for each outgoing jump $e \in E$ (lines 29-35), if aggregation is activated then they are aggregated (per jump, line 41), and finally added to P for further iterative successor computation. Note that the flowpipe as well as the jump successor computations in the sub-spaces are time-synchronised: if a successor set gets empty in one of the sub-spaces then the computation is stopped.

5 Experimental Results

All computations were carried out on an Intel Core i7 CPU with 8 cores and 16 GB RAM. We used the time step size $\delta = 0.01$ and unlimited jump depth. To be able to express a global time horizon, each model is equipped with a global clock.

Input: Hybrid system model $H = (Loc, X, Flow, Inv, Edge, Init)$, time step $\delta \in \mathbb{Q}_{\geq 0}$, global time horizon $T \in \mathbb{Q}_{\geq 0}$, jump depth $D \in \mathbb{N}_{\geq 0}$, aggregation flag $aggregation \in \{0,1\}$.

Output: An over-approximation $\{(l,x) | (l, p_{disc}, p_{clock}, p_{rest}, [t_1,t_2]) \in R \wedge x \in p_{disc} \times p_{clock} \times p_{rest}\}$ of the states reachable within the given bounds.

2 X_{disc} := maximal set of variables from X with derivatives 0 that are syntactically independent from $X \setminus X_{disc}$;
3 X_{clock} := maximal set of variables from X with derivatives 1 that are syntactically independent from $X \setminus X_{clock}$;
4 $X_{rest} := X \setminus (X_{disc} \cup X_{clock})$;
5 choose a representation type for each of X_{disc}, X_{clock}, and X_{rest};
6 bring each predicate φ in H to an equivalent form $\varphi_{disc} \wedge \varphi_{clock} \wedge \varphi_{rest}$, where each φ_i, $i \in \{disc, clock, rest\}$, is a predicate from $Pred_{X_i}$ resp. $Pred_{X_i \cup X_i'}$ (jump resets) resp. $Pred_{X_i \cup \dot{X}_i}$ (flows);
7 $P := \emptyset$; $R := \emptyset$;
8 **foreach** *location* $l \in Loc$ **do**
9 let $p_i := Init(l)_i \cap Inv(l)_i$ for each $i \in \{disc, clock, rest\}$;
10 **if** $p_{disc} \neq \emptyset \wedge p_{clock} \neq \emptyset \wedge p_{rest} \neq \emptyset$ **then**
11 add $(l, p_{disc}, p_{clock}, p_{rest}, [0,0])$ to P
12 **end**
13 **end**
14 **while** $P \neq \emptyset$ **do**
15 choose $p = (l, p_{disc}, p_{clock}, p_{rest}, [t_1,t_2]) \in P$ and remove p from P;
16 $E := \{(e, p^e_{disc}) \mid e = (l,g,r,l') \in Edge \wedge p^e_{disc} = jump(p_{disc}, g_{disc}, r_{disc}, Inv(l')_{disc}) \neq \emptyset\}$;
17 $isFirst := true$; **foreach** $(e, p^e_{disc}) \in E$ **do** $P^e := \emptyset$;
18 ;
19 **while** *true* **do**
 `// Compute flowpipe, considering also the invariant of the location`
20 **if** $t_1 < T$ **then**
21 $p_{clock} := flow(p_{clock}, Flow(l)_{clock}, Inv(l)_{clock}, \delta, isFirst)$; **if** $p_{clock} = \emptyset$ **then** break;
22 ;
23 $p_{rest} := flow(p_{rest}, Flow(l)_{rest}, Inv(l)_{rest}, \delta, isFirst)$; **if** $p_{rest} = \emptyset$ **then** break;
24 ;
25 $t_2 := t_2 + \delta$; **if** $\neg isFirst$ **then** $t_1 := t_1 + \delta$;
26 ;
27 **end**
28 $R := R \cup \{(l, p_{disc}, p_{clock}, p_{rest}, [t_1,t_2])\}$;
 `// Compute jump successors`
29 **foreach** $((l,g,r,l'), p^e_{disc}) \in E$ **do**
30 $p^e_{clock} := jump(p_{clock}, g_{clock}, r_{clock}, Inv(l')_{clock})$; **if** $p^e_{clock} = \emptyset$ **then** continue;
31 ;
32 $p^e_{rest} := jump(p_{rest}, g_{rest}, r_{rest}, Inv(l')_{rest})$; **if** $p^e_{clock} = \emptyset$ **then** continue;
33 ;
34 add $(l', p^e_{disc}, p^e_{clock}, p^e_{rest}, [t_1,t_2])$ to P^e;
35 **end**
36 $isFirst := false$; **if** $t_1 \geq T$ **then** break;
37 ;
38 **end**
39 **foreach** $e \in E$ **do**
40 **if** $P^e \neq \emptyset$ **then**
41 **if** *aggregation* **then** $P := P \cup \{aggregate(P^e)\}$;;
42 **else** $P := P \cup P^e$;
43 **end**
44 **end**
45 **end**
46 **return** R

Algorithm 1: Reachability analysis algorithm based on variable set separation for hybrid automata.

Table 1: Model sizes of the benchmarks.

Benchmark	Type	#variables			#modes		#jumps
		disc.	clocks	rest	controller	plant	
Leaking tank	original	0	0	12	8	3	34
	timed	0	2	10	8	3	34
	discrete	9	0	3	8	3	34
	timed & discrete	9	2	1	8	3	34
Two tanks	original	0	0	22	20	14	296
	timed	0	3	19	20	14	296
	discrete	17	0	5	20	14	296
	timed & discrete	17	3	2	20	14	296
Thermostat	original	0	0	8	6	2	18
	timed	0	2	6	6	2	18
	discrete	5	0	3	6	2	18
	timed & discrete	5	2	1	6	2	18

Benchmarks For our experiments we used three well-known benchmarks, which we slightly modified by adding model components for PLC controllers. Besides increasing the number of modes, these extensions add variables with discrete behaviour (i.e. with zero derivatives) to model the actuators and sensors of the plant and the input, output, and local variables of the controller. Furthermore, one clock variable is added for each introduced PLC controller to model the cycle time, and one discrete variable to store the controller mode. In our experiments we compare the analysis of the benchmarks without variable separation ("original") with variable-set-separation-based analysis separating only clocks ("timed"), only discrete variables ("discrete"), and both ("timed & discrete"). The sizes of the models are shown in Table 1. The modified versions of the benchmarks are accessible as part of our benchmark collection [2]. A binary of our implementation can be found at [12].

Leaking tank This benchmark models a water tank which leaks, i.e., it has a constant outflow. The tank can be refilled from an unlimited external resource with a constant inflow that is larger than the outflow. The PLC controller triggers refilling (by switching a pump on) if a sensor indicates a low water level ($h \leq 6$). If the water level is high ($h \geq 12$) the controller stops refilling (switches the pump off). Adding the controller introduces two controller input variables for low and high water levels, variables for the actuator (pump) state in the plant and the controller, and a variable to store the controller mode. Furthermore, a new clock is added to model the PLC cycle time. Besides the controller we also model a user which can manually switch the pump on and off as far as the water level allows it. In our implementation, the user constantly toggles between the pump states on and off. We analyse the system behaviour over a global time horizon of 40 seconds using a PLC cycle time of 2 seconds.

Two tanks This benchmark models the water levels of two water tanks in a closed system. Each tank has a constant inflow and a constant outflow. The tanks are connected via pipes, such that the amount of water outflow of the first tank is equal to the inflow of the second tank and vice versa. One pump per pipe allows to enable/disable the water flow. We add a controller to the two tank system that controls the pumps. A pump is switched off if the water level of the source tank is low ($h \leq 8$) or if the water level of the target tank is high ($h \geq 32$). Each time a pump is switched off by the controller, the other pump is switched on to balance the water levels in the tanks. The introduction of the controller adds variables to model sensing low and high water levels of both tanks and variables to model the actuator (pump) states in the plant and the controller. Moreover, we add a variable to store the controller mode and a new clock

Table 2: Benchmark results for different separation set-ups. Running times are in seconds, time-out (TO) was 20 minutes, in brackets we list the number of flowpipes computed.

			HyPro				SpaceEx
Benchmark	Rep.	Agg	original	timed	discrete	timed & discrete	original
Leaking tank	box	agg	2.70 (662)	2.08 (662)	1.06 (662)	1.13 (662)	3.67 (200)
	box	none	2.62 (662)	2.09 (662)	1.06 (662)	1.13 (662)	3.82 (200)
	sf	agg	TO (18)	TO (28)	161.12 (662)	37.03 (662)	448.3 (425)
	sf	none	TO (583)	1044.97 (662)	19.49 (662)	5.84 (662)	444.82 (425)
Two tanks	box	agg	4.39 (470)	2.60 (470)	0.97 (470)	1.15 (470)	5.49 (195)
	box	none	4.46 (470)	2.68 (470)	1.02 (470)	1.16 (470)	5.53 (195)
	sf	agg	TO (4)	TO (4)	900.11 (470)	329.80 (470)	TO (171)
	sf	none	TO (54)	TO (64)	35.04 (470)	14.64 (470)	TO (172)
Thermostat	box	agg	0.07 (95)	0.09 (95)	0.06 (95)	0.06 (95)	0.57 (95)
	box	none	0.11 (95)	0.09 (95)	0.06 (95)	0.06 (95)	0.57 (95)
	sf	agg	35.87 (95)	22.69 (95)	1.17 (95)	0.29 (95)	9.89 (84)
	sf	none	30.41 (95)	20.19 (95)	1.18 (95)	0.30 (95)	9.91 (84)

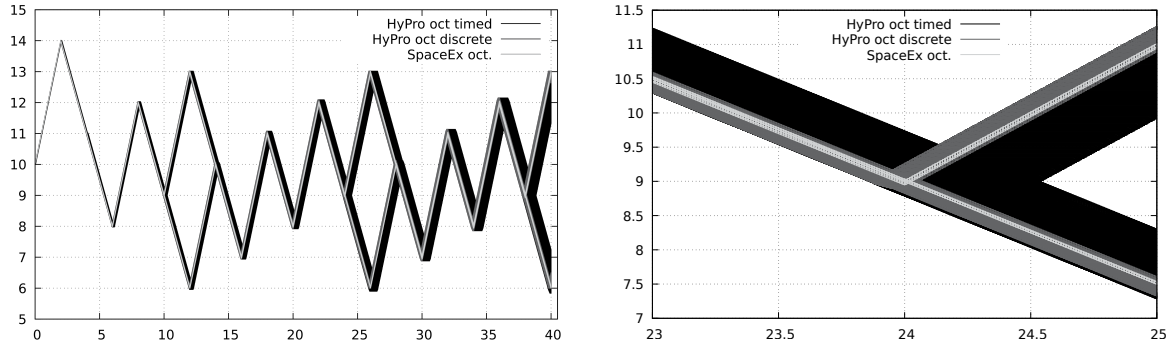

Figure 4: SpaceEx and HyPro results on the leaking tank benchmark with support function representation (using a regular octagonal (oct) template for evaluation), when HyPro separates either only clocks or only discrete variables.

to model the PLC cycle time. Again, we model a user which switches the pumps manually on or off as far as the water levels allow it. We implemented a user that toggles the state of each pump in each PLC cycle. The global time horizon and a PLC cycle time were set to 20 seconds respectively 1 second.

Thermostat In this benchmark a heater with a thermostat controller is modelled. Initially, the temperature is $T = 20°C$ and the heater is on. The controller keeps the temperature T between $16°C$ and $24°C$. The heater is switched off if the temperature rises above $23°C$ and it is switched on at a temperature below $18°C$. Adding a controller to the model introduces new variables for the low and high temperature sensors in the controller, a variable for the actuator (heater) state in the plant and the controller, and a variable to store the controller mode. Additionally, we introduce a new clock for the cycle time of the PLC. The global time horizon is 10 seconds and the PLC cycle time is 0.5 seconds.

Results We implemented our algorithm using the HyPro library and evaluated it on the above benchmarks. Table 2 shows results from our tool and SpaceEx version 0.9.8f. In our tool we used boxes and support functions (evaluated in 8 directions) to represent state sets, whereas in SpaceEx we used support functions with 4 and 8 directions, as SpaceEx does not support explicit box representations.

The HYPRO and SPACEEX results are not fully comparable because SPACEEX implements a fixed-point detection algorithm but HYPRO does not. The leaking tank benchmark as well as the two tank benchmark both cause branching in the execution paths which are merged later (see Figure 4). Our implementation does not recognise the merging of these paths and fully computes each branch independently. Thus HYPRO needs to compute a higher number of flowpipes (given in brackets behind the running times in Table 2) than SPACEEX. Another difference is that in HYPRO we varied the state set representation for the continuous sets between boxes and support functions (similarly to SPACEEX) but used boxes for the discrete and the clock variable sets in all settings.

Using variable separation clearly improves the running times, due to computations in lower-dimensional sub-spaces. However, we can also observe on the Figures 4 and 5, which show plots for the detected reachable regions for the leaking tank and the thermostat, that separating the clock variables (which measure the cycle time and the global time) introduces a slight over-approximation.

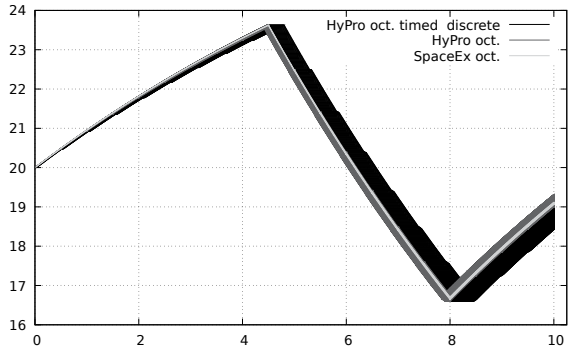

Figure 5: SPACEEX and HYPRO results on the thermostat benchmark with support function representation (using an octagonal (oct) template for evaluation), when HYPRO separates clocks as well as discrete variables.

The influence of the discrete variable separation is in general larger than the influence of a clock separation, probably because in our benchmarks the discrete variables outnumber the clocks. Nonetheless a separation of clocks already shows a speed-up of about 30%. As mentioned before, we used boxes as a state set representation for the set of discrete variables, which does not introduce any further over-approximation error, as the discrete variables themselves are all syntactically independent. We can observe that using boxes as a state set representation, our implementation outperforms SPACEEX (even when a lot more flowpipes are computed), which is expected, as boxes in general require less computational effort than support functions (evaluated in 4 directions) in reachability analysis.

In HYPRO, aggregation causes longer running times because in the current implementation aggregation is realised by a conversion of the single sets (which are to be aggregated) to polytopes, which is computationally expensive, especially in higher dimensions.

6 Conclusion

In this paper we presented an approach to reduce the computational effort in the reachability analysis of hybrid systems for certain applications. Our experimental results indicate that even state-of-the art reachability analysis tools struggle to analyse high-dimensional models with relatively simple dynamics, which are common in the application area of controlled plants.

In general controlled plants are composed of many single components such as the set of controllers or the physical quantities of the plant. A naive approach models each of these components and the full model is the result of a parallel composition of the single components. Even the relatively simple examples used in Section 5 yield large models which put state-of-the-art reachability analysis tools to their limits. To increase scalability, domain-specific knowledge helps to create more sophisticated and smaller models. For example knowing that PLC computation as well as the plant's behaviour do not

interfere during a PLC cycle already allows to prohibit arbitrary switching between the controller and the plant, which reduces the model complexity.

In contrast to common benchmarks for hybrid systems, our plant models exhibit a large number of discrete variables accounting for the controller's behaviour. Currently available tools do not distinguish between the different dynamics of variables, thus discrete variables usually are treated as continuous variables and unnecessarily increase the dimension of the state space.

Our approach allows to split variable sets according to their dynamics, which has a positive effect on the running times, as reachability analysis algorithms can be protected from working in high-dimensional spaces. We can observe that the distribution of variables to the different sets has a high influence on the computation time. We expect that splitting the set of continuous variables (if applicable) into multiple, independent sets with fewer variables each will result in the best results regarding computation time. Furthermore in applications with several PLC controllers, each control program operates independently, which allows to build separate variable sets for each controller. Depending on the dimension of the individual sets and the associated dynamics for the contained variables, utilizing individual state set representations can be beneficial. As state sets for independent discrete variables are always hyper-rectangles, using boxes instead of other, computationally more expensive state set representations has shown great improvements in terms of runtime. For higher dimensional state sets support functions can be expected to perform better than other state set representations.

In our application scenario, syntactically independent variable sets are directly given. In general, this is not necessarily the case for hybrid system models. Transforming the state space can help to identify independent variable sets and allows to apply the presented approach to systems where the independent variable sets are not obvious.

As to future work, we will improve our implementation by adding fixed-point detection and a more sophisticated implementation for state set aggregation. Second, we will embed the presented approach into a CEGAR framework to refine potentially unsafe paths. Finally, we also work on parallelisation approaches for flowpipe computations.

References

[1] Matthias Althoff & John M. Dolan (2014): *Online verification of automated road vehicles using reachability analysis*. *IEEE Transaction on Robotics* 30(4), pp. 903–918, doi:10.1109/TRO.2014.2312453.

[2] *Benchmarks of continuous and hybrid systems*. Available at `http://ths.rwth-aachen.de/research/projects/hypro/benchmarks-of-continuous-and-hybrid-systems/`.

[3] Olivier Bournez, Oded Maler & Amir Pnueli (1999): *Orthogonal polyhedra: Representation and computation*. In: *Proc. HSCC'99*, *LNCS* 1569, Springer, pp. 46–60, doi:10.1007/3-540-48983-5_8.

[4] Xin Chen, Erika Ábrahám & Sriram Sankaranarayanan (2013): *Flow*: An analyzer for non-linear hybrid systems*. In: *Proc. CAV'13*, *LNCS* 8044, Springer, pp. 258–263, doi:10.1007/978-3-642-39799-8_18.

[5] Xin Chen & Sriram Sankaranarayanan (2016): *Decomposed reachability analysis for nonlinear systems*. In: *Proc. RTSS'16*, IEEE Computer Society Press, pp. 13–24, doi:10.1109/RTSS.2016.011.

[6] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp & Erika Ábrahám (2015): *SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving*. In: *Proc. SAT'15*, *LNCS* 9340, Springer, pp. 360–368, doi:10.1007/978-3-319-24318-4_26.

[7] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable verification of hybrid systems*. In: *Proc. CAV'11*, *LNCS* 6806, Springer, pp. 379–395, doi:10.1007/978-3-642-22110-1_30.

[8] Antoine Girard (2005): *Reachability of uncertain linear systems using zonotopes*. In: *Proc. HSCC'05*, *LNCS* 3414, Springer, pp. 291–305, doi:10.1007/978-3-540-31954-2_19.

[9] Willem Hagemann, Eike Möhlmann & Astrid Rakow (2014): *Verifying a PI controller using SoapBox and Stabhyli: Experiences on establishing properties for a steering controller*. In: *Proc. ARCH'14*, *EPiC Series in Computer Science* 34, EasyChair, pp. 115–125.

[10] Thomas A. Henzinger (1996): *The theory of hybrid automata*. In: *Proc. LICS'96*, IEEE Computer Society Press, pp. 278–292, doi:10.1007/978-3-642-59615-5_13.

[11] *HyCreate*. Available at `http://stanleybak.com/projects/hycreate/hycreate.html`.

[12] *HyPro Project website*. Available at `http://ths.rwth-aachen.de/research/projects/hypro/`.

[13] HyReach. Available at `https://embedded.rwth-aachen.de/doku.php?id=en:tools:hyreach`.

[14] Colas Le Guernic & Antoine Girard (2010): *Reachability analysis of linear systems using support functions*. *Nonlinear Analysis: Hybrid Systems* 4(2), pp. 250–262, doi:10.1016/j.nahs.2009.03.002.

[15] Andrew Makhorin: *GNU Linear Programming Kit home page*. Available at `http://www.gnu.org/software/glpk/glpk.html`.

[16] Ramon E. Moore, Ralph Baker Kearfott & Michael J. Cloud (2009): *Introduction to interval analysis*. SIAM, doi:10.1137/1.9780898717716.

[17] Leonardo M. de Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *Proc. TACAS'08*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[18] Johanna Nellen (2016): *Analysis and synthesis of hybrid systems in engineering applications*. Ph.D. thesis, RWTH Aachen University, Aachen. Available at `https://publications.rwth-aachen.de/record/680323`.

[19] Stefan Schupp, Erika Abraham, Ibtissem Ben Makhlouf & Stefan Kowalewski (2017): *HyPro: A C++ library for state set representations for hybrid systems reachability analysis*. In: *Proc. NFM'17*, *LNCS* 10227, Springer, pp. 288–294, doi:10.1007/978-3-319-57288-8_20.

[20] Roland Wunderling (1996): *Paralleler und objektorientierter simplex-algorithmus*. Ph.D. thesis, Technische Universität Berlin.

[21] Günter M. Ziegler (1995): *Lectures on polytopes*. 152, Springer, doi:10.1007/978-1-4613-8431-1.