ParaPlan: A Tool for Parallel Reachability Analysis of Planar Polygonal Differential Inclusion Systems

Andrei Sandler
School of Computer Science
University of Hertfordshire
United Kingdom
a.sandler@herts.ac.uk

Olga Tveretina
School of Computer Science
University of Hertfordshire
United Kingdom
o.tveretina@herts.ac.uk

Abstract. We present the ParaPlan tool which provides the reachability analysis of planar hybrid systems defined by differential inclusions (SPDI). It uses the parallelized and optimized version of the algorithm underlying the SPeeDI tool [2]. The performance comparison demonstrates the speed-up of up to 83 times with respect to the sequential implementation on various benchmarks. Some of the benchmarks we used are randomly generated with the novel approach based on the partitioning of the plane with Voronoi diagrams.

1 Introduction

A hybrid system is a dynamic system that exhibits both continuous and discrete behaviour. Examples of such systems come from robotics, avionics, air traffic management and automated highway management. Most of the hybrid systems are safety critical and errors can have serious consequences. Formally, verifying safety properties of hybrid systems consists of building a set of reachable states and checking if this set intersects with a set of unsafe states. Therefore one of the most fundamental problems in the analysis of hybrid systems is the reachability problem.

The reachability problem is only decidable for special classes of hybrid systems [11]. Currently, a number of tools for analysing the reachability problem are available, including dReach [12], Flow* [6], KeYmaera [15] and HSolver [16].

The focus in developing tools for the reachability analysis is now mainly on improving the performance of sequential algorithms, because the sequential algorithms do not always provide the required computational efficiency. Approaches for parallelisation are still uncommon and the benefits of parallel execution on multi-core platforms are not well understood [19]. The main motivation for our work is to understand further computational benefits of parallelization of the reachability analysis.

In this paper we consider a decidable class of hybrid systems, called Planar Polygonal Differential Inclusions (SPDIs), which naturally arises from the analysis of hybrid systems with two continuous variables. SPDIs are defined by giving a finite partitioning of the plane into convex polygonal sets, together with a differential inclusion associated with each region P and defined by a couple of vectors l_P and r_P. An algorithm for solving the reachability problem for SPDIs has been introduced in [4]. It abstracts trajectory segments into so-called signatures (sequences of edges and simple cycles) and then even further into types of signatures (signatures which do not take into account the number of times each simple cycle is iterated).

In [2, 18] the authors present the SPeeDI toolkit which is a collection of utilities to manipulate and reason automatically about SPDIs. The tool is implemented in Haskell and also provides trace generation on top of the reachability analysis, but it was never benchmarked or optimized.

P. Bouyer, A. Orlandini & P. San Pietro (Eds.): 8th Symposium on Games, Automata, Logics and Formal Verification (GandALF'17)
EPTCS 256, 2017, pp. 283–296, doi:10.4204/EPTCS.256.20

The decidability result for SPDIs has also been extended to generalized SPDIs in [14]. Those are SPDIs not satisfying the goodness assumption (the dynamics of a region of the SPDI do not allow a trajectory to traverse an edge in opposite directions).

There is a generalized version of the SPeeDI tool, namely GSPeeDI [10], written in Python. It computes all simple cycles using the algorithm of Tarjan [20]. Obviously, the number of simple cycles is the bottleneck determining when the problem becomes infeasible.

The choice of SPDIs has been triggered by two factors: on the one hand this class of hybrid systems is decidable and on the other hand it is powerful enough to exhibit relatively complex behaviour. Moreover, SPDIs cannot be straightforwardly verified by the existing tools due to non-determinism expressed by differential inclusions.

Contribution. Our contribution is twofold. First, we present the ParaPlan tool for PARAllel analysis of PLANar differential inclusion systems which implements the optimized and parallelized version of the sequential algorithm underlying the SPDI tool. Second, we describe the novel approach for random generation of benchmarks using Voronoi diagrams. ParaPlan is available online at [17]. It has been tested on a series of benchmarks, including those from [18] and random benchmarks generated using our approach. Absolute testing time and relative speed-up against original algorithm is measured.

Related work. To the best of our knowledge, the results on parallelization of the reachability problem for hybrid systems were reported only in [13] and [9].

In the earlier paper [13], although purely theoretical, the authors introduce a compositional algorithm for splitting the reachability task into several independent tasks in the strongly connected regions of an SPDI. We decided not to implement this algorithm because the only case it can speed up calculations is when multiple tasks are solved consequently on the same SPDI, and pre-calculations take as much time as it is needed to solve one reachability task with the original algorithm.

In [9] two parallel state-space-exploration algorithms have been proposed for the reachability analysis of general hybrid systems, which are implemented in the XSpeed model checker. The first algorithm uses the parallel, breadth-first-search algorithm of the SPIN model checker. The second algorithm improves load balancing. Their approach is to parallelize BFS algorithm and divide calculations inside a discrete state into 'atomic' tasks for better load balancing between threads. In case of the ParaPlan tool the state space is divided into atomic tasks (dynamic flow on the region's edges) by design.

Although there are many tools available for hybrid systems analysis, their comparative evaluation is problematic as they do not support the same model classes. Moreover, it is impossible to use those tools on SPDI directly, because it is not allowed to use differential inclusions inside a discrete state.

Outline. In Section 2 we formally describe the class of two-dimensional non-deterministic hybrid systems studied in this paper, namely SPDIs. In Section 3 we recall the original approach for computing reachable states for SPDIs introduced in [5, 3] and in Section 4 we present our optimisation of the algorithm and its parallel version. In Section 5 we describe the novel approach for generating random benchmarks. Performance evaluation can be found in Section 6. Section 7 contains concluding remarks.

2 Polygonal Differential Inclusions

The notion of an SPDI is a generalization of Piecewise-constant Derivative Systems (PCD) studied in [1]. The new characteristic of SPDIs with respect to PCDs is non-determinism. Informally, an SPDI consists of a partition of a plane subset into convex polygonal regions, together with a differential inclusion associated with each region [4]. That is, the class of SPDI systems can be represented as non-deterministic linear hybrid automata with continuous trajectories which derivative in every point inside any convex

region is bounded by a given angle.

Now we will define an SPDI formally, and we will use $\angle_r^l$ to denote the angle defined by two non-zero vectors $\vec{l}$ and $\vec{r}$.

Definition 1 *We define an SPDI as a hybrid automaton* $H = (Q, D, X, f, E, Init, G, R)$*, where*

- $Q = \{q_1, q_2 \ldots, q_n\}$ *with* $n > 0$ *is a finite set of discrete states (regions);*
- $X = \mathbb{R}^2$ *is a set of continuous states;*
- $f(\cdot,\cdot) : Q \times X \to X$ *is a vector field bounded in every region* q_i *by* $\angle_{r_i}^{l_i}$*;*
- *A domain function* $D(\cdot) : Q \to P(X)$ *defines regions as a set of convex (possibly infinite) polygons, forming a convex polygon partitioning of* $\mathbb{R}^2$*;*
- *Init is a set of edge intervals;*
- *E is a set of edges between regions, formed by all polygon boundaries;*
- *A guard condition* $G(\cdot) : E \to P(X)$ *is a linear guard condition, defined by the edges of the partitioning;*
- *A reset map* $R(\cdot,\cdot) : E \times X \to P(X)$ *is an identity function.*

Figure 1 illustrates the SPDI randomly generated using the method described in Section 5.1 and a trajectory segment.

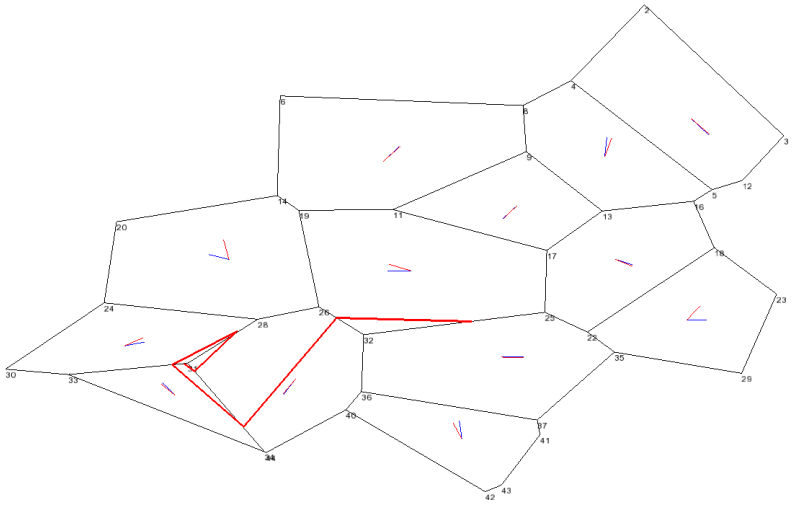


Figure 1: The SPDI with 13 regions and a trajectory segment

The edge-to-edge reachability problem for SPDI has been proved to be decidable ([5]) and could be stated as follows:

Given two edges e_0 *and* e_f*, does there exist* $x_0 \in e_0$ *and* $x_f \in e_f$ *such that there is a trajectory segment starting at* x_0 *and ending at* x_f*?*

This task could be interpreted as following: if the whole model represents the dynamics of a real-world system, a trajectory represents one possible evolution of the system, and there are some unsafe states, then the existence of a trajectory starting in an initial set of states and ending in an unsafe state proves such system to be unsafe to use.

3 Sequential Algorithm

In this section we recall the original approach for computing reachable states, which is introduced in [5, 3] and based on the characterization of the qualitative behaviours of trajectories.

In general, there are infinitely many trajectories from the starting set S to the final set F, but they all are determined by the angles associated with the regions q_i. Therefore, for edges $e, e' \in q_i$ and an interval $(s,s') \subseteq e$, there is an interval $(f,f') \subseteq e'$, such that every trajectory starting in (s,s') will intersect with (f,f'). Hence, we can calculate the reachable states just by examining the edge intervals that the trajectories traverse. If a trajectory crosses some intervals successively on the edges $e_1, e_2, \ldots, e_n$, then the sequence $\sigma = (e_1, e_2, \ldots, e_n)$ is called the *edge signature*.

For computing the successive interval images, it is convenient to introduce a one-dimensional coordinate system on each edge e, with zero (0) denoting one chosen vertex v_0 of e and one (1) denoting the other vertex v_1. Now each point between the vertices of each edge has the coordinate $v_\lambda = \lambda v_0 + (1-\lambda)v_1$ with $0 < \lambda < 1$. Then, a series of successor functions on edges of the SPDI is defined.

- The successor $Succ_{\mathbf{c}}(x)$ of a point x on an edge e under a dynamics, defined by a single vector $\mathbf{c}$ is an image x' on the edge e' of the same region, where the point x will be projected along $\mathbf{c}$.
- The successor $Succ_{(\mathbf{c}_1,\mathbf{c}_2)}(x_1,x_2)$ of an interval (x_1,x_2) on edge e in region r with dynamics, defined by $\angle_{c_2}^{c_1}$ is an interval (x'_1,x'_2) on e', where

$$x'_1 = min(1, Succ_{\mathbf{c}_1}(x_1), Succ_{\mathbf{c}_2}(x_1))$$

$$x'_2 = max(0, Succ_{\mathbf{c}_1}(x_2), Succ_{\mathbf{c}_2}(x_2))$$

  If $x'_1 > x'_2$, then the successor is the empty set. In other words, the successor of an interval is the interval reachable under the region's dynamics.
- The successor $Succ_\sigma(x_{e_1,1}, x_{e_1,2})$ of an interval $(x_{e_1,1}, x_{e_1,2})$ on edge e_1 along the edge signature $\sigma = (e_1, e_2, \ldots, e_n)$, is a result of applying $Succ_{(\mathbf{c}_{e_i,1},\mathbf{c}_{e_i,2})}(x_{e_i,1}, x_{e_i,2})$ consequently to $e_1, e_2, \ldots, e_{n-1}$. Roughly speaking, the successor of an interval along σ is the set of points on e_n reachable from the points on e_1 through $e_2, e_3, \ldots, e_{n-1}$.

The edge signature of a trajectory can possibly contain simple cycles, but nested cycles are not permitted. In general, an edge signature has the following form:

$$r_1 s_1^{k_1} r_2 s_2^{k_2} \ldots r_n s_n^{k_n} r_{n+1}$$

where r_i denotes the path between cycles, and $s_i^{k_i}$ denotes the cycle s_i repeated k_i times.

Still, an SPDI can have infinitely many edge signatures, because in some trajectories there are cycles s_i that could be repeated any k_i times. In order to compactify all edge signatures into a finite set, it is generalized again using signature types.

The *signature type* of an edge signature is the following sequence:

$$r_1 s_1 r_2 s_2 \ldots r_n s_n r_{n+1}$$

The following theorem defines a set of edge signatures, which is only needed to be examined for finding the trajectory from S to F.

Theorem 2 (Asarin, Schneider, Yovine, [5]) *Only those signature types having disjoint paths r_i and unique (as sets of edges) cycles s_i, could correspond to a trajectory starting in initial set S and ending in final set F. It is easy to see that there are only finite number of such signature types on any given SPDI.*

Having fixed a signature type and starting intervals on the first edge of the signature type, one can algorithmically calculate the successor function along the edges of the signature type and check if the final set could be reached. Cycles are treated in a special way described in [5]. Following from the existence of the deciding algorithm, the reachability problem on SPDI is decidable.

4 Optimization of Sequential Algorithm

The algorithm underlying the SPeeDI tool constructs all feasible types of signatures. If even the time for analysing each signature type is not significant, it computes also the signature types which cannot be realised in any trajectory. The example of such behaviour is given in Figure 2.

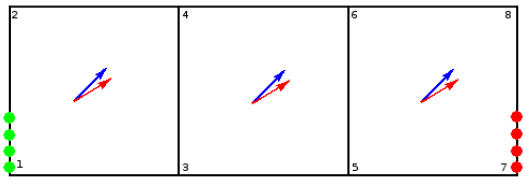


Figure 2: The red interval is not reachable from the green interval,
but the corresponding edge signature is feasible

In our approach we explore feasible signatures via DFS and simultaneously compute reachable states. This way we look up less or equal number of signatures, because every explored signature would have a trajectory realization (because of reachability). During the edge exploration we check the conditions of Theorem 2 and do not take into consideration those edges that form nested or non-unique cycles.

For effective analysis of cycles in signatures we use the technique described in [5]. There are only five different types of cycle behaviour, and the exit sets of points in four cases out of five could be effectively calculated without iterating the cycle. The last cycle type requires iteration, but it is proved to be finite.

The main algorithm for signature types discovering is listed below. To solve the whole reachability task, one simply need to explore all the signature types from every starting edge:

Algorithm 1 Solving the reachability problem

```
function SolveReachabilityTask(spdi, reachTask)
    for startEdge in reachTask.StartEdges do
        if DFSSignaturesExploration(startEdge, spdi, reachTask) return SUCCESS
    return FAILURE
```

In the main DFS function we use the fact (Theorem 2) that all already visited edges must belong to the last discovered path r_i, otherwise there will be a nested cycle.

Algorithm 2 Main depth-first search function for exploring the signatures of trajectories

```
function SIGNATURESEXPLORATION(currentEdge, borders, spdi, reachTask)
    if final state on currentEdge is reached  return SUCCESS
    if currentEdge is visited
        iterate back to restore cycle and current path r_i
        if cycle is visited OR ends not in current path r_i
            return FAILURE                          ▷ only simple cycles allowed
        mark new cycle as visited
        reachable ← TESTCYCLEANDGETFINALIMAGES(cycle, borders, spdi, reachTask)
        if final state is reached in reachableStates  return SUCCESS
        for image in reachable do
            if image is not valid  return FAILURE
            for all possible nextEdge, connected to currentEdge do
                nextImage ← SUCCINT(image, currentEdge, nextEdge)
                if nextImage is valid
                    return SIGNATURESEXPLORATION(nextEdge, nextImage, spdi, reachTask)
    else
        mark currentEdge visited
        add currentEdge to current path r_i
        for all possible nextEdge, connected to currentEdge do
            nextImage ← SUCCINT(borders, currentEdge, nextEdge)
            if nextImage is valid
                return SIGNATURESEXPLORATION(nextEdge, nextImage, spdi, reachTask)
     return FAILURE
```

Auxiliary function `TestCycleAndGetFinalImages` determines the type of cycle (one of {`STAY`, `EXIT-LEFT`, `EXIT-RIGHT`, `EXIT-BOTH`, `DIE`}, see [5]), and effectively calculates the set of intervals of the first cycle edge which will be visited during cycle iteration.

4.1 Parallelization

We further improve the sequential algorithm by parallelizing the DFS-like exploration of signature types. We do it by assigning sub-trees of DFS to different threads and loading them again by new sub-trees when they are finished with previous tasks. When the algorithm iterates over all possible next edges in the signature type, except the last, it will check if there is a free thread which could be loaded with the DFS sub-tree starting with this edge. Mutexes are used to eliminate the possible race condition.

During the computation, some sub-tasks may finish earlier than others. In this case the remaining sub-tasks will be divided to load the free threads again, so the CPU utilisation will be full all the time.

Data is partly shared between threads (such as SPDI representation and reachability task), but partly it is copied when creating new threads, because it cannot be stored globally. For example, visited edges and cycles are different at almost all times in each two different threads threads, therefore storing it in one data structure will give no gain in memory or time consumption.

Algorithm 3 Sequential algorithm parallelization

```
global FreeThreads = NumberOfThreads
function SOLVEREACHTASK(spdi, reachTask)
    for e in reachTask.StartEdgeParts do
        if FreeThreads > 0 AND e is not the last
            FreeThreads ← FreeThreads - 1
            CREATETHREAD(SignaturesExploration, e.edge, e.borders, spdi, reachTask)
        else
            SIGNATURESEXPLORATION(startEdge, e.edge, e.borders, spdi, reachTask)
    for T in threads do
        JOINTHREAD(T)
        FreeThreads ← FreeThreads + 1
```

5 Benchmarks

We investigated several sources ([18], [7], [8]) where we looked for SPDI examples suitable to run benchmarks on. In the following list we present benchmarks we have managed to use in our experiments.

- SPDI generated by MSPDI library [18]. This is a Perl library for generating SPDI files from 2-dimensional ordinary differential equations. Three examples we used include pendulum equations, spiral ODE with one focal point and the following non-linear system:

$$\begin{cases} \dot{x} = y \\ \dot{y} = -0.5y - 2x - x^2 \end{cases}$$

- The model example from [18]
- Randomly generated SPDIs (see 5.1)

5.1 Random SPDI Generation

Here we introduce the algorithm for random SPDI generation. The motivation behind it is such that randomly generated examples are usually much more complex to solve and relatively easy to obtain. The algorithm is based on convex polygonal partitioning of a plane using Voronoi diagrams. It places N random points on a $[0;1000]^2$ square of $\mathbb{R}^2$ plane and creates a Voronoi diagram with this points as regions centers. The resulting partitioning is guaranteed to be convex based on the properties of Voronoi diagram (see Figure 1, where SPDI is randomly generated).

An edge of the region is called the *output edge* if there is at least one trajectory that goes out of this region through this edge. Each edge in an SPDI could be an output edge for at most one region (we do not consider SPDIs with Zeno behaviour).

To define angles on a given partition, we assign a sequence of edges to be output edges for each region and test that the oriented angle between the pre-leftmost and the post-rightmost edges is positive (see Figure 3). If the output edges for all regions are correctly defined, we assign two vectors to each region by taking it randomly between the pre-leftmost and post-rightmost edge vectors.

Output edges and vectors are assigned to each region by Algorithm 4. Resulting SPDI is represented in the format proposed in [18].

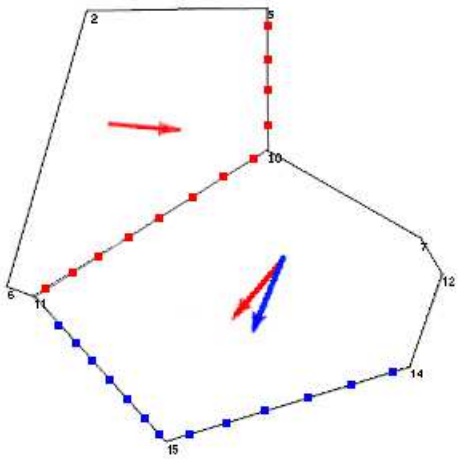


Figure 3: Red dots denote the output edges for the upper region, and blue dots denote the output edges for the lower region. The oriented angles between edges (6-11) and (2-5) for the upper region, and between edges (10-11) and (12-14) for the lower region, are positive.

Algorithm 4 Constructing random differential inclusion

```
Randomly iterate by all regions
for region R in all regions, shuffled do
    find all starting edges E                    ▷ free edge with non-free previous neighbour
    if E is empty
        if no free edges in R
            return FAILURE
        else
            E ← {random edge from R}             ▷ all edges are free, no edge with non-free neighbour
    for e ∈ E do
        try to construct an output set for R starting with e
        if no output set is obtained
            return FAILURE
        else
            assign two random vectors between pre-leftmost and post-rightmost edges to R
```

For benchmarking we implemented the random task generator which produces random reachability tasks for a given SPDI. It can also generate fixed sequences of random tasks, which is achieved by fixing a random seed. A reachability task is generated as a random set of start and final edge intervals. As the formats of reachability tasks in SPeeDI and ParaPlan differ, the generator produces the same set of tasks in both formats.

6 Experiments

We set up two series of experiments, one for comparing ParaPlan and SPeeDI, and the other for measuring the profit from paralellization. All our experiments were conducted on a 64-bit Linux computer with 8 core Intel Core-i7-4790T (2.7 GHz, 8MB cache) processor and 16 GB RAM. ParaPlan tool is implemented in C++ using Pthreads library for paralellization. Full code of the tool could be found online (see [17]), as well as the SPDI benchmarks.

6.1 Comparison of ParaPlan and SPeeDI

We compared ParaPlan and SPeeDI using the model SPDI example from [18]. For this purpose we ran a series of 100 and 1000 different reachability tasks. The results are presented in Table 1.

Table 1: Comparison of ParaPlan and SPeeDI on 100 and 1000 tasks

Tool	100 tasks	1000 tasks
ParaPlan	0m 1.151s	0m 9.804s
SPeeDI	1m 16.857s	18m 40.828s

ParaPlan outperforms SPeeDI by 75-100 times on this example. Partly this result is achieved because our code is written in C++, while the original code is in Haskell, and partly due to the optimisation of the algorithm. We observed that in approximately 13% cases the answers produced by the tools were different and we manually found several tasks on which SPeeDI produced an incorrect answer.

6.2 Comparison of Parallel and Optimised Sequential Algorithms

Second experiment was aimed at revealing whether there is any profit that could be gained from parallelization. We generated a new series of reachability tasks for different SPDIs and compared the average time it took the ParaPlan tool to process all the tasks on different number of threads. The reachability tasks series for each SPDI was of length 100 and contained $1 \leqslant s \leqslant 10$ starting edges and $1 \leqslant f \leqslant 10$ final edges, so that every combination of s and f is presented in tests.

We divided all SPDIs onto two groups - the group of "heavy tests", where the average computational time on one thread was higher than 0.1 second, and the "light tests" group, where this time was less than 0.1 second. The group of heavy tests consists of SPDIs of various size, obtained from spiral ODE using MSPDI library. Light tests group was formed of example SPDI and randomly generated SPDI consisting of 100 regions. Other benchmarks were solved too fast to rely on the measured time and were not included in the final results table.

Each group of tests was divided in two, depending on the reachability of the final set. It is done on purpose, because the algorithm terminates as soon as it reaches any point of final set, and in case of reachable tasks it happens much earlier than the whole DFS tree is looked up, which leads to much more considerable speed-up.

We did not take into consideration those tests which did not finish in 5 seconds on at least one number of threads. In all other tests time is clipped in $[0,5]$ seconds interval. Hence, the speed-up results we obtain is a lower bound for the real speed-up. We also did not perform an extreme test to determine how many regions in SPDI our tool can process, because it mainly depends on the complexity of the dynamics and not on the number of regions.

In the following subsections we present the results of our testing. Each subsection contains two tables, one for the mean value of absolute time of testing, and the other for the relative speed-up observed. The data is presented also on two graphs in each subsection.

6.2.1 Heavy Tests, Unreachable States

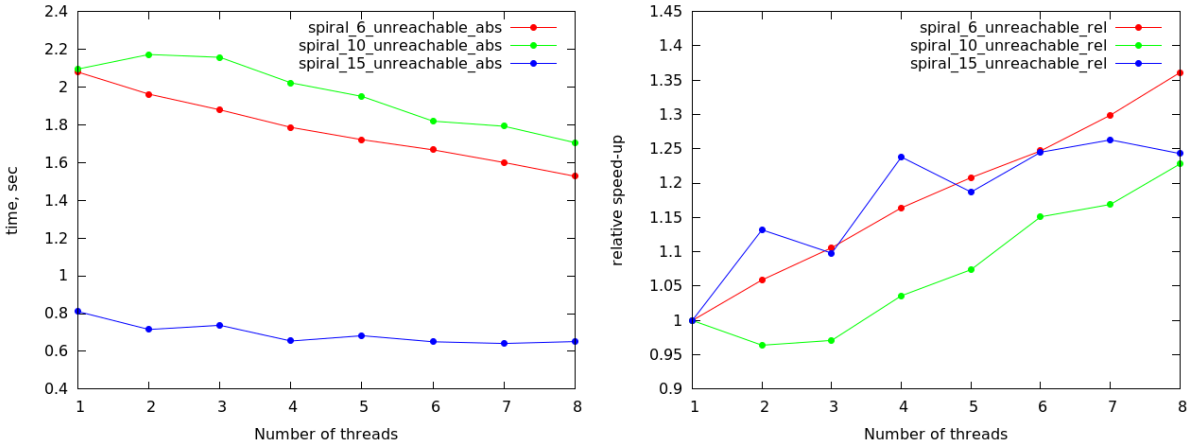
This is the hardest test for our tool because the whole DFS tree is needed to be looked up, and the SPDI is rather complex. However, we observe the growing speed-up of about 1.4 times at its peak.

Table 2: Absolute testing time, mean value

Number of threads	1	2	3	4	5	6	7	8
spiral_6	2.079	1.962	1.879	1.786	1.721	1.667	1.600	1.527
spiral_10	2.095	2.172	2.157	2.022	1.950	1.819	1.792	1.705
spiral_15	0.811	0.716	0.738	0.655	0.683	0.651	0.642	0.652

Table 3: Relative speed-up

Number of threads	1	2	3	4	5	6	7	8
spiral_6	1.000	1.059	1.106	1.164	1.208	1.247	1.299	1.361
spiral_10	1.000	0.964	0.971	1.036	1.074	1.151	1.169	1.228
spiral_15	1.000	1.132	1.098	1.238	1.187	1.245	1.263	1.243



(a) Absolute testing time, mean value

(b) Relative speed-up

Figure 4: Heavy tests, unreachable states

6.2.2 Heavy Tests, Reachable States

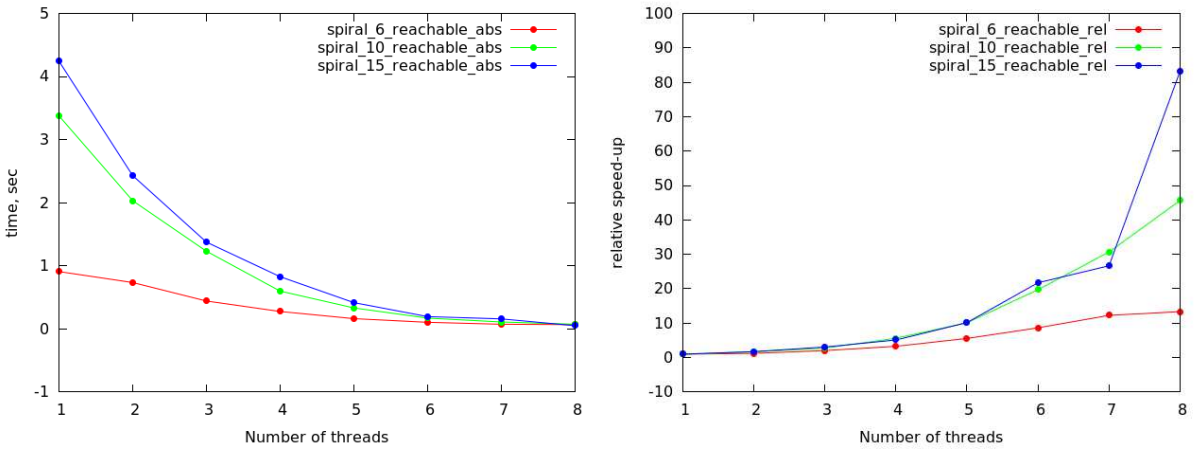
Here we observe much more significant acceleration (up to 83 times faster), because often the algorithm finishes long before the whole DFS tree is done.

Table 4: Absolute testing time, mean value

Number of threads	1	2	3	4	5	6	7	8
spiral_6	0.909	0.735	0.442	0.276	0.163	0.105	0.074	0.068
spiral_10	3.378	2.033	1.232	0.599	0.333	0.171	0.110	0.074
spiral_15	4.245	2.431	1.376	0.828	0.417	0.195	0.159	0.051

Table 5: Relative speed-up

Number of threads	1	2	3	4	5	6	7	8
spiral_6	1.000	1.236	2.056	3.293	5.576	8.657	12.28	13.36
spiral_10	1.000	1.661	2.741	5.639	10.14	19.75	30.70	45.64
spiral_15	1.000	1.746	3.085	5.126	10.17	21.76	26.69	83.23



(a) Absolute testing time, mean value

(b) Relative speed-up

Figure 5: Heavy tests, reachable states

6.2.3 Light Tests, Unreachable States

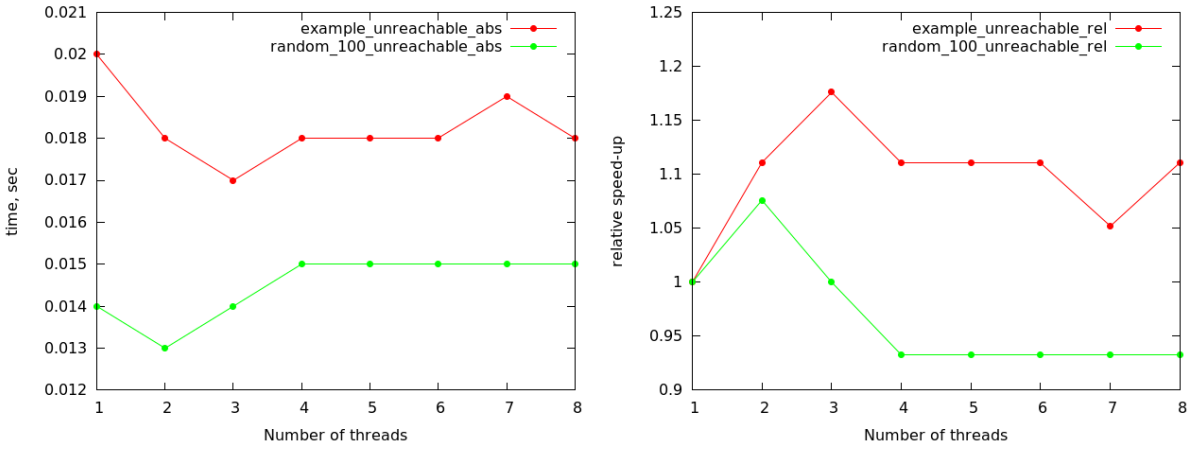
On light tests there is practically no effect of paralellization, as the tasks finish very fast. There is even a little slowdown on random SPDI because of the threads overhead. We also observe that against our expectations random SPDIs happened to be relatively easy to solve for our tool.

Table 6: Absolute testing time, mean value

Number of threads	1	2	3	4	5	6	7	8
example	0.020	0.018	0.017	0.018	0.018	0.018	0.019	0.018
random_100	0.014	0.013	0.014	0.015	0.015	0.015	0.015	0.015

Table 7: Relative speed-up

Number of threads	1	2	3	4	5	6	7	8
example	1.0	1.111	1.176	1.111	1.111	1.111	1.052	1.111
random_100	1.0	1.076	1.0	0.933	0.933	0.933	0.933	0.933



(a) Absolute testing time, mean value

(b) Relative speed-up

Figure 6: Light tests, unreachable states

6.2.4 Light Tests, Reachable States

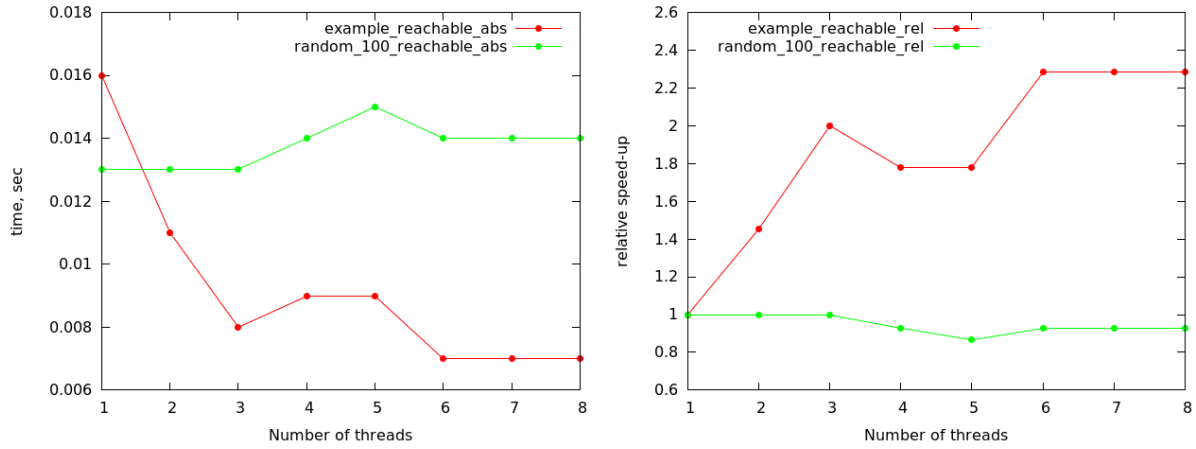
Here we see little to none effect of parallelization, and again a little slowdown on random SPDI, and a slight speed-up on the model example, which stops on the 6 threads.

Table 8: Absolute testing time, mean value

Number of threads	1	2	3	4	5	6	7	8
example	0.016	0.011	0.008	0.009	0.009	0.007	0.007	0.007
random_100	0.013	0.013	0.013	0.014	0.015	0.014	0.014	0.014

Table 9: Relative speed-up

Number of threads	1	2	3	4	5	6	7	8
example	1.0	1.454	2.0	1.777	1.777	2.285	2.285	2.285
random_100	1.0	1.0	1.0	0.928	0.866	0.928	0.928	0.928



(a) Absolute testing time, mean value

(b) Relative speed-up

Figure 7: Light tests, reachable states

7 Conclusion and Future Work

We presented the paralellized algorithm for reachability analysis of 2-dimensional systems of polygonal differential inclusions. The algorithm is the optimized version of the original algorithm from [5]. It was experimentally demonstrated that a speed-up could be gained via parallelization, which depends on the complexity of an SPDI itself and on whether or not the final states are reachable. We also presented the algorithm for generating random SPDIs which could be useful for future research in this area. Possible future work may be focused on representing SPDIs as general hybrid automata and comparing the results of solving reachability tasks on SPDIs using ParaPlan and the existing model checkers for hybrid systems.

References

[1] E. Asarin, O. Maler & A. Pnueli (1995): *Reachability Analysis of Dynamical Systems Having Piecewise-Constant Derivatives*. *Theoretical Comp. Science* 138(1), pp. 35–65, doi:10.1016/0304-3975(94)00228-B.

[2] E. Asarin, G. J. Pace, G. Schneider & S. Yovine (2002): *SPeeDI - A Verification Tool for Polygonal Hybrid Systems*. In: *14th International Conference on Computer Aided Verification (CAV)*, pp. 354–358, doi:10.1007/3-540-45657-0_28.

[3] E. Asarin, G. J. Pace, G. Schneider & S. Yovine (2008): *Algorithmic analysis of polygonal hybrid systems, Part II: Phase portrait and tools*. *Theoretical Computer Science* 390(1), pp. 1–26, doi:10.1016/j.tcs.2007.09.025.

[4] E. Asarin, G. Schneider & S. Yovine (2001): *On the Decidability of the Reachability Problem for Planar Differential Inclusions*. In: *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, pp. 89–104, doi:10.1007/3-540-45351-2_11.

[5] E. Asarin, G. Schneider & S. Yovine (2007): *Algorithmic analysis of polygonal hybrid systems, part I: Reachability*. *Theoretical Computer Science* 379(1-2), pp. 231–265, doi:10.1016/j.tcs.2007.03.055.

[6] X. Chen, E. Ábrahám & S. Sankaranarayanan (2013): *Flow*: An Analyzer for Non-linear Hybrid Systems*. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 258–263, doi:10.1007/978-3-642-39799-8_18.

[7] X. Chen, S. Schupp, I. B. Makhlouf, E. Ábrahám, G. Frehse & S. Kowalewski (2015): *A Benchmark Suite for Hybrid Systems Reachability Analysis*. In: *NASA Formal Methods - 7th International Symposium, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pp. 408–414, doi:10.1007/978-3-319-17524-9_29.

[8] A. Fehnker & F. Ivančić (2004): *Benchmarks for hybrid systems verification*. In: *International Workshop on Hybrid Systems: Computation and Control*, Springer, pp. 326–341, doi:10.1007/978-3-540-24743-2_22.

[9] A. Gurung, A. Deka, E. Bartocci, S. Bogomolov, R. Grosu & R. Ray (2016): *Parallel reachability analysis for hybrid systems*. In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*, pp. 12–22, doi:10.1109/MEMCOD.2016.7797741.

[10] H. A. Hansen & G. Schneider (2009): *GSPeeDi–a Verification Tool for Generalized Polygonal Hybrid Systems*. In: *International Colloquium on Theoretical Aspects of Computing*, Springer, pp. 343–348, doi:10.1007/978-3-642-03466-4_23.

[11] T. A. Henzinger, P. W. Kopke, A. Puri & P. Varaiya (1998): *What's Decidable About Hybrid Automata?* *Journal of Computer and System Sciences* 57(1), pp. 94–124, doi:10.1006/jcss.1998.1581.

[12] S. Kong, S. Gao, W. Chen & E. M. Clarke (2015): *dReach: δ-Reachability Analysis for Hybrid Systems*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 200–205, doi:10.1007/978-3-662-46681-0_15.

[13] G. Pace & G. Schneider (2006): *A compositional algorithm for parallel model checking of polygonal hybrid systems*. In: *International Colloquium on Theoretical Aspects of Computing*, Springer, pp. 168–182, doi:10.1007/11921240_12.

[14] G. J. Pace & G. Schneider (2008): *Relaxing Goodness Is Still Good*. In: *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, pp. 274–289, doi:10.1007/978-3-540-85762-4_19.

[15] A. Platzer & J.-D. Quesel (2008): *KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description)*. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pp. 171–178, doi:10.1007/978-3-540-71070-7_15.

[16] S. Ratschan & Z. She (2005): *Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement*. In: *Hybrid Systems: Computation and Control, 8th International Workshop (HSCC)*, pp. 573–589, doi:10.1007/978-3-540-31954-2_37.

[17] A. Sandler & O. Tveretina: *ParaPlan Tool*. Available at `https://github.com/asandler/ParaPlan`.

[18] G. Schneider & G. Pace (2006): *SPeeDI+: A 2 Dimensional Hybrid System Model Checker*. http://www.cs.um.edu.mt/∼svrg/Tools/SPeeDI/index.html. [Online; accessed 27-January-2017].

[19] S. Schupp, E. Ábrahám, X. Chen, I. B. Makhlouf, G. Frehse, S. Sankaranarayanan & S. Kowalewski (2015): *Current challenges in the verification of hybrid systems*. In: *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, Springer, pp. 8–24, doi:10.1007/978-3-319-25141-7_2.

[20] R. E. Tarjan (1973): *Enumeration of the Elementary Circuits of a Directed Graph*. *SIAM J. Comput.* 2(3), pp. 211–216, doi:10.1137/0202017.