Spread the Work: Multi-threaded Safety Analysis for Hybrid Systems

Stefan Schupp and Erika Ábrahám(✉)

Theory of Hybrid Systems, RWTH Aachen University, Aachen, Germany
{stefan.schupp,abraham}@cs.rwth-aachen.de

Abstract. We consider a method for the bounded safety analysis of hybrid systems, whose continuous behaviour is intertwined with discrete execution steps. The method computes a tree of state sets, which together over-approximate reachability by bounded-length executions. If none of the state sets intersects with a given set of unsafe states then we have proven bounded safety. Otherwise, we iteratively repeat parts of the computations with locally refined search parameters, in order to reduce the over-approximation error.

In this paper we present a parallelization technique for the above method. We identify independent computations that can be carried out by different threads/processes concurrently, and examine how to achieve work-balance between the threads at low communication cost. Furthermore, we discuss how to assure mutually exclusive node access during refinement computations, without high synchronization costs. We evaluate our proposed solutions experimentally on some benchmarks.

1 Introduction

The massive application of digital controllers for the control of continuous (e.g. physical) systems raises the need for verification approaches for such *hybrid* systems with mixed discrete-continuous behaviour. Though the reachability problem for hybrid systems is in general undecidable, a variety of incomplete safety analysis approaches have been developed. Besides verification methods based on theorem proving, SMT solving or rigorous simulation, these include techniques based on *flowpipe construction*, e.g. [1–3].

Starting from a set of initial states, flowpipe-construction-based methods iteratively over-approximate *flowpipes*, i.e. the set of states reachable from a given state set via time evolution according to the system's continuous dynamics, and sets of successors via discrete execution steps. Due to non-determinism, these computations generate a tree with state sets as nodes, where the root includes all initial states and the children of a node include all discrete successors from the node's flowpipe. These computations are usually bounded in the time duration

This work was supported by the German research council (DFG) in the context of the HyPro project and the DFG Research Training Group 2236 UnRAVeL.

for flowpipes and the number of discrete steps executed (unless a fixedpoint can be detected).

If none of the flowpipes and discrete successor sets contain unsafe states then the model is safe. Otherwise, due to over-approximation, no conclusive information can be derived. Therefore, it is important to provide possibilities to reduce the over-approximation error by increasing the *precision* of the computations [4–6]. To avoid complete re-starts of the analysis upon parameter refinement for increased precision, some approaches use counterexample-guided refinements [7,8].

For applicability, it is also important to increase the *scalability* of these methods. A piece of work in this direction is [9], where the authors propose a scalable approach to compute the set of all states reachable by fixed-step simulation. Approaches like [10–12] decompose the state space into lower-dimensional subspaces in which reachability computations can be executed faster (but usually with less precision). One of the few parallelization approaches is presented in [13]; besides speeding up sequential computations, the authors propose to parallelize flowpipe computations for the over-approximation of reachability between two discrete state changes.

In this paper we propose a *parallelization* approach for a sequential algorithm [8], which applies flowpipe-construction-based reachability analysis in an iterative counterexample-guided refinement loop for error reduction. In contrast to [13] we do not parallelize the construction of a single flowpipe, but compute several flowpipes independently by parallel threads. An extension could additionally apply parallelization according to [13], but it is left for future work. Without a refinement loop, different flowpipe computations would be independent and thus their parallelization would be natural. However, our experience shows that achieving a work-load balance at low communication costs is challenging. Furthermore, the refinement loop makes additional synchronization necessary, which we keep at a minimum to reduce unnecessary synchronization costs. We implemented our method and provide some experimental results.

The rest of this paper is structured as follows: Sect. 2 contains preliminaries on flowpipe-construction-based reachability analysis and its embedding in a refinement loop as introduced in [8]. Section 3 presents our parallelization approach, followed by experimental results in Sect. 4. We conclude the paper in Sect. 5.

2 Preliminaries

2.1 Hybrid Automata

Hybrid automata are a well-established formalism for modeling hybrid systems.

Definition 1 (Hybrid automata: Syntax [14]). *A* hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$ *with the following components:*

– *Loc is a finite set of* locations *or* control modes*.*

- *$Var = \{x_1, \ldots, x_d\}$ is a finite ordered set of real-valued* variables*; sometimes we use the vector notation $x = (x_1, \ldots, x_d)$. The number d is called the* dimension *of $\mathcal{H}$. By $\dot{Var}$ we denote the set $\{\dot{x}_1, \ldots, \dot{x}_d\}$ of dotted variables (which represent first derivatives during continuous evolution), and by Var' the set $\{x'_1, \ldots, x'_d\}$ of primed variables (which represent values directly after a discrete change). Furthermore, given a variable set X, let $Pred_X$ denote a set of predicates with free variables from X.*
- *$Flow : Loc \to Pred_{Var \cup \dot{Var}}$ specifies for each location its* flow *or* dynamics*.*
- *$Inv : Loc \to Pred_{Var}$ assigns to each location an* invariant*.*
- *$Edge \subseteq Loc \times Pred_{Var} \times Pred_{Var \cup Var'} \times Loc$ is a finite set of* edges *(ℓ_1, g, r, ℓ_2) with* source *location ℓ_1,* target *location ℓ_2,* guard *g, and* reset *function r.*
- *$Init : Loc \to Pred_{Var}$ assigns to each location an* initial *predicate.*

While the presented approach can be generalized, in this work we focus on *linear* hybrid automata, where $Pred_{Var}$ is the set of all conjunctions of linear equalities and inequalities over Var, $Flow$ assigns to each location a linear ordinary differential equation (ODE) system of the form $\dot{x} = Ax$ with some $A \in \mathbb{R}^{d \times d}$, and where reset functions on discrete transitions are defined by affine mappings $x' = Ax + b$ with $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$.

A *state* $, = (\ell, \nu)$ of a hybrid automaton consists of a location $\ell \in Loc$ and a variable valuation $\nu : Var \to \mathbb{R}$. We refer to a set of states with a common location ℓ and valuations from a set $\mathcal{V}$ by $(\ell, \mathcal{V}) = \{(\ell, \nu) \mid \nu \in \mathcal{V}\}$.

The state of a hybrid automaton can be changed either by time or by discrete steps. A *time step* $(\ell, \nu) \xrightarrow{t} (\ell, f(\nu, t))$ (also called *flow*) of length t models the passage of t time units: the control location remains unchanged and the variable values evolve continuously according to a solution f of the ODE system $Flow(\ell)$; the time step is enabled only if the invariant $Inv(\ell)$ is satisfied during the whole time step, i.e., by all $f(\nu, t')$ with $0 \leq t' \leq t$. A *discrete step* $(\ell, \nu) \xrightarrow{e} (\ell', \nu')$ (also called *jump*) models a discrete change of the control mode: it follows an edge $e = (\ell, g, r, \ell') \in Edge$ which is enabled (i.e., ν satisfies g and ν' satisfies $Inv(\ell')$), where ν' results from ν by applying the affine mapping specified by r. Note that hybrid automata are in general non-deterministic, as a time step and several jumps can be enabled at the same time.

An *execution* or *path* $\pi = \sigma_0 \xrightarrow{t_0} \sigma'_0 \xrightarrow{e_0} \sigma_1 \xrightarrow{t_1} \ldots$ is a (finite or infinite) sequence of alternating time and discrete steps, starting in an initial state $\sigma_0 = (\ell_0, \nu_0)$ such that ν_0 satisfies $Init(\ell_0)$. A state is called *reachable* if there is a finite path leading to it. Given a hybrid automaton $\mathcal{H}$ and subset T of its states, the *reachability problem* poses the question whether some state of T is reachable in $\mathcal{H}$.

2.2 Reachability Analysis Based on Flowpipe Construction

In this work we use a *bounded flowpipe-construction-based reachability analysis* method for linear hybrid automata. As the reachability problem for linear hybrid automata is in general undecidable, this approach computes over-approximations of bounded reachability (with upper bounds on the number of jumps and on the

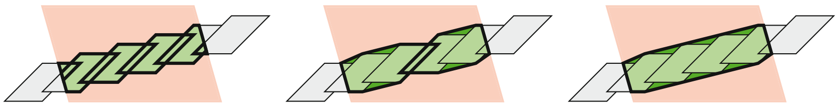

Fig. 1. Jump successors can be processed individually (6 sets on the left), clustered (2 sets in the middle) or aggregated (1 set on the right).

length of time steps). The computation starts from a set $(\ell_0, \mathcal{V}_0)$ of initial states and over-approximates, alternatingly, time successors within a *time horizon* T and jump successors iteratively up to a given *jump depth* J. As datatypes for state sets $\Omega = (\ell, \mathcal{V})$, different geometric or symbolic state set representations (e.g. boxes, convex polyhedra, zonotopes, support functions or Taylor models) can be used to over-approximate the valuation set $\mathcal{V}$ (when interpreted as a subset of $\mathbb{R}^d$).

To compute bounded time successors from a given set of valuations $\mathcal{V}$ in a location ℓ, the time horizon T is divided into N time segments of size $\delta = \frac{T}{N}$. For each $i = 0, \dots, N-1$ the set of states reachable from $\mathcal{V}$ in ℓ within time $[i\delta, (i+1)\delta]$ is *over-approximated*, intersected with the invariant of ℓ and stored as a state set in Ω_i (called the ith *flowpipe segment*). The union $\bigcup_{i=0}^{N-1} \Omega_i$ of the flowpipe segments is referred to as the *flowpipe* and over-approximates the set of states reachable from $\mathcal{V}$ in ℓ within T time. If any of the flowpipe segments has a non-empty intersection with the set of unsafe states then the algorithm terminates (with an inconclusive answer due to over-approximation).

Otherwise, for each flowpipe segment $\Omega_i = (\ell, \mathcal{V}_i)$ and jump $e = (\ell, g, r, \ell')$ rooted in ℓ we determine an over-approximation $\Omega_{e,i}$ of the jump successors from Ω_i along e; this includes the intersection of $\mathcal{V}_i$ with g, the affine transformation of the result according to r, and the intersection with the invariant of ℓ'.

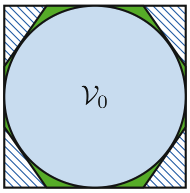

Fig. 2. Valuation set $\mathcal{V}_0$ over-approximated by a box (blue) and a convex polytope (green) [8]. (Color figure online)

One possibility is to apply the algorithm now iteratively to all non-empty jump successors $\Omega_{e,i}$ with $i = 0, \dots, N-1$ and e being a jump leaving ℓ, until the jump depth has been reached. However, this approach is computationally very expensive. Alternatively, we can group the jump successors into a fixed number k of clusters (if there are more than k segments), over-approximate each cluster by one set, and continue the computations for each cluster over-approximation. If $k > 1$ then we call this procedure *clustering*, and for $k = 1$ we call it *aggregation* (see Fig. 1).

The choices of time segmentation, state set representation and clustering/aggregation parameters influence the over-approximation error. Usually, a smaller time step size δ, a more precise state set representation and finer clustering leads to a smaller error on the cost of increased computation time.

For instance, boxes require little computational effort for set operations but in general introduce more over-approximation error as e.g. convex polytopes do (see Fig. 2). We refer to [15,16] for further details.

During the analysis, we store the state sets for which flowpipes and jumps successors need to be computed in a *search tree*, whose depth is limited by the jump depth (see Fig. 3). The root node stores the initial states, whereas each other node n_i stores either the jump successor states of the flowpipe segment given by the parent, or a clustering/aggregation of such sets, depending on the parameter setting. If we label each parent-child connection with the union of the time intervals of the considered flowpipe segments and the jump taken, then the path from the root to a node describes a *symbolic path* $\Pi = I_0, e_0, \ldots, I_k, e_k$, which *represents* all paths $\mathit{Paths}(\Pi) = \{\sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{e_0} \sigma_1 \ldots \sigma_{l+1} \,|\, l \leq k \wedge \forall 0 \leq i \leq l. t_i \in I_i\}$. A path $\pi \in \mathit{Paths}(\Pi)$ that does not exist in the hybrid automaton is called *spurious*.

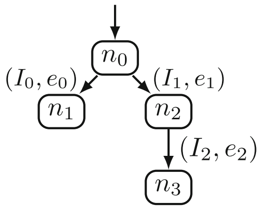
$$\begin{tikzcd}
 & n_0 \arrow[dl, "{(I_0,e_0)}"'] \arrow[dr, "{(I_1,e_1)}"] & \\
n_1 & & n_2 \arrow[d, "{(I_2,e_2)}"] \\
 & & n_3
\end{tikzcd}$$

Fig. 3. An example for a search tree.

The structure of the search tree depends not only on the analyzed hybrid automaton but also on the analysis parameters. Non-determinism naturally causes a branching in the search tree, but over-approximation might cause not only larger sets in the nodes but also additional branching.

If the algorithm has terminated due to the detection of an unsafe state then the symbolic path to one of the nodes represents a *counterexample* path leading to an unsafe state. However, due to over-approximation, we do not know whether this counterexample is spurious or not.

2.3 Counterexample-Guided Parameter Refinement

Most available algorithms terminate at this point; the user needs to restart the search with adapted parameters to achieve a higher precision. To avoid a complete restart, in [8] we presented a counterexample-guided approach to repeat the search with refined parameters along (potentially spurious) counterexample paths only.

A user-defined collection of parameter settings is stored in an ordered list, the *refinement strategy*. We say that we compute at *refinement level* i when we use the $(i+1)$st setting in the refinement strategy. The refinement levels might differ e.g. in the state set representation, the time step size or in the clustering/aggregation settings.

The search starts at refinement level 0, i.e., with the first setting in the refinement strategy. When a potential counterexample is detected at refinement level i then we enforce an iterative re-computation of reachability within the counterexample's symbolic path Π (called the *refinement path*) at refinement level $i+1$ (unless i was the last defined level, in which case the algorithm terminates without any conclusive answer). These re-computations start at the root node, for which the successors are computed at refinement level $i+1$, however, only its successors along Π will be further processed by the refinement (i.e. only successors with symbolic path Π' for which $\mathit{Paths}(\Pi) \cap \mathit{Paths}(\Pi') \neq \emptyset$).

Note that several refinements might be applied to the same symbolic path. A special case is when a counterexample is detected *before* the whole previous counterexample has been refined, i.e., before reaching the end of the previous counterexample. In this case the counterexample must be spurious, because the previous over-approximative computations did not detect any unsafe states at that point; we continue the computations without additional refinement.

The refinements stop if either the counterexample could be shown to be spurious (path is safe) or we have tried all settings in the strategy but the potential counterexample could not be excluded. In the first case, the analysis continues with further successor computations; if the path with the spurious counterexample had less jumps than the jump depth, then also successors for its last state set are further processed, however, for these computations we jump back to refinement level 0.

Due to space restrictions, we cannot explain how we store the refined sets at all levels in a single search tree, and how we switch back from a higher refinement level to level 0 after the elimination of a spurious counterexample. Regarding the aspects of parallelization, it is not necessary to understand these mechanisms in detail. It is however important to notice that a node in the search tree can store several state sets, each computed at a different refinement level. Thus a node and a refinement level uniquely specify a state set stored in the tree.

3 Parallel Reachability Analysis

In the following we propose a parallelization approach for the previously introduced reachability analysis method with counterexample-guided parameter refinement for hybrid automata. We first discuss some aspects of a sequential implementation (Sect. 3.1) before we describe our parallel approach (Sect. 3.2) and implementation details (Sect. 3.3).

3.1 Sequential Analysis

In this section we recall from [8] some implementation-related concepts for the sequential analysis with counterexample-guided parameter refinement, as they will play basic roles for the parallelization.

Task. A task collects all information that is needed to compute flow and jump successors for a state set stored in a node of the search tree. In a classical approach without refinement, storing a reference to the search tree node would be sufficient for this purpose, assuming that all search parameters are globally accessible. With refinement, tasks are either *basic* at refinement level 0 or they are *refinement tasks* storing the refinement work at a positive refinement level for a node on a (potentially spurious) counterexample path. In both cases, the task additionally needs to store the current refinement level (specifying the parameter values for the computations). In the case of refinement, the task also needs to store the symbolic path of the counterexample, to which also the refinement of the successors should be restricted.

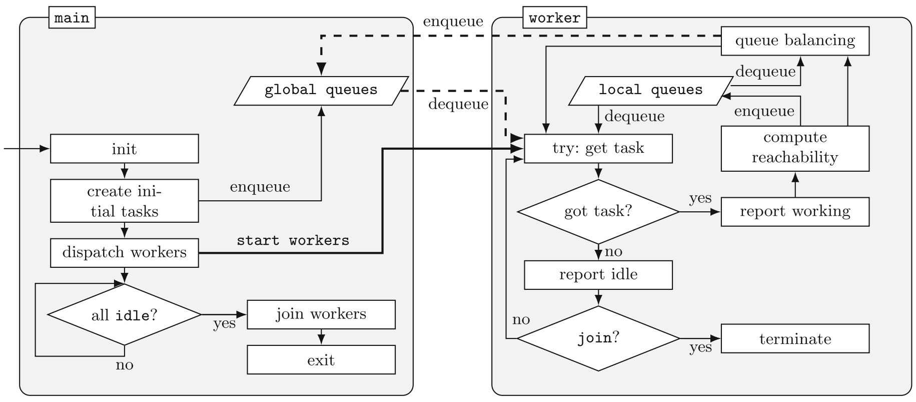

Fig. 4. HyDRA's execution structure. Dashed lines denote synchronized access.

Worker. A worker (or in the sequential setting *the* worker) is responsible for the execution of tasks. In our setup we employ one type of workers, which uses a state-of-the-art method for computing flowpipes and jump successors (see Sect. 2). However, as stated in [17], we could also consider specialized workers (e.g. applying different successor computation approaches dedicated to certain types of dynamics). We could even consider the decomposition of the state space as described in [10] and the invocation of specialized sub-workers on the components, but these ideas are not yet implemented.

Task Queue. Once a worker completed a basic task, it adds the corresponding jump successor state sets as new nodes to the search tree and creates tasks for them to trigger their processing (unless the jump depth has been reached or a potential counterexample has been detected). To keep track of the tasks that still need to be processed, in the sequential setup the worker maintains its own *task queue* – we will extend this concept for parallelization. In the implementation we use priority queues which allow to implement different search heuristics by modifying the order inside the queue.

Refinement Queue. Whenever a worker detects the potential reachability of some unsafe states, it triggers a refinement of the symbolic path to the current node (the *refinement path*) as presented in [8]. As counterexamples might share a prefix, when refining a node, the worker first checks whether the node has already been refined to the required level; if so then there will only be created requests for processing the children along the refinement path (in the form of new tasks).

For their storage, we want to prioritize refinement tasks over basic tasks. Instead of changing the ordering of the task queue, we do so by using a separate *refinement queue*, as it also allows for separate queue balancing methods (see Sect. 3.2).

3.2 Parallel Analysis

In this work we develop parallelization based on *multi-threading*. The tasks are natural units for parallel processing: multiple threads can implement workers (in a one-to-one correspondence between threads and workers) processing different tasks in parallel.

Local and Global Queues. As in the sequential case, each worker has a *local task queue* and a *local refinement queue*. Access to these local queues is restricted to the owning worker, therefore it does not require any synchronisation and is thus fast. Additionally, for work balancing, we need a mechanism to distribute tasks between threads. For this purpose we use a *global task queue* and a *global refinement queue*, which can be accessed by all workers in a synchronized fashion.

Initially there are some initial tasks (for initial state sets) in the global task queue, and the global refinement queue and all local queues are empty.

When idle, each worker tries first to obtain a task to process from its local refinement queue or its local task queue, in this order, to keep the synchronization overhead as small as possible. Only if both of its local queues are empty, the worker tries to obtain a task from the global refinement or the global task queue, using synchronized access. If both global queues are also empty, the worker re-checks the global queues regularly, until they are filled or until also all other local queues are empty, which leads to a synchronized completion of the algorithm.

If a worker processes a task, resulting new tasks will be added to the worker's local queues. I.e., without further balancing, the subtree under the currently processed node in the search tree will be analyzed by this worker only.

To allow work-balancing, workers can *move tasks* from their local queues to the corresponding global queues (from local task queue to global task queue, from local refinement queue to global refinement queue). We consider three heuristics for this balancing step, which apply after each completion of a task: (i) the worker pushes all but one tasks from its local queues to the global queues; (ii) only when the local queue size is larger than a certain threshold, tasks exceeding that threshold are moved from the local to the global queues; (iii) push a certain ratio of tasks from the local queues to the global queues. We expect that approaches (i) and (iii) will result in balanced work distribution at higher synchronization costs while approach (ii) should be better suited to limit these costs but lead to a less balanced execution. Note that the queue balancing happens after the completion of each task by a worker, i.e. when potential successor tasks have been added to the thread-local queues.

We also consider a different setting, where *only global queues* are present. In this setting, work is automatically distributed but getting work from the queues and adding new tasks to the queues require synchronization.

Node Synchronization. All workers share a single search tree. Without path refinement, the workers need to synchronize on the access to the global queues, but not on search tree nodes: each search tree node (below the jump depth level) will be referred to by exactly one task, which will be processed by exactly one worker. However, this is not the case for path refinement, as counterexample paths might share a prefix. To ensure thread-safety during path refinement, each worker first gets a lock on the tree node it intends to refine, processes the node, and gives the lock free before starting to process any other node.

3.3 Implementation

We extended our HyDRA tool by the presented approach, based on a previous implementation of the sequential counterexample-guided parameter refinement method. HyDRA uses the HyPro [15] `C++` library for state set representations, and it has been developed in a modular fashion to be easily extensible.

The general data flow of HyDRA is illustrated in Fig. 4. Similarly to the design principles presented in [17], the reachability analysis core (compute reachability) of HyDRA is built up of separate components dealing with the computation of continuous and discrete successors. Our implementation extends the existing concepts by distributing the analysis process among multiple threads.

The main thread of the tool is responsible for management operations e.g. invoking the parser, dispatching workers or plotting the computed reachability over-approximations, if required. Reachability is computed by a fixed number of separate worker-threads. After pre-processing and initialization by the main thread, i.e. parsing and creation of tasks from the initial states, the worker threads are created. Tasks are shared via globally accessible work queues – as stated before we maintain separate queues for refinement tasks and regular analysis tasks. Following the concept of work stealing, an idle worker with empty local queues obtains its next task to work on from a global work queue and processes it. Each worker extends the shared search tree by jump successor state sets and creates the corresponding new tasks for the work queue.

Signaling. Inter-thread communication is necessary to join workers after completion of the analysis. A worker reports idleness via an event system whenever there is no task in its local queues and no task available in the global queues. During idling, the worker repeatedly tries to get a task from the global queues; if this succeeds, the worker signals the end of its idling period (this signalling happens inside the synchronized access to the global queues). When all workers reported idleness i.e. all queues are empty, the main thread signals the worker threads to terminate. All workers are joined and post-processing of the computed sets e.g. plotting (in the figure: *exit*) can be performed. As signaling requires synchronization, the number of signals should be limited as far as possible.

Queue Access. To reduce the overhead introduced by synchronization, we equip our global queues with synchronized as well as non-synchronized methods for access. Idle workers can utilize non-synchronized methods for the global queues to check for emptiness and only use synchronized access methods whenever the queue is not empty (after a second, synchronized check for emptiness while holding the lock for the queue). This allows to avoid unnecessary synchronization in scenarios where there are many idle workers constantly accessing the global queues which are empty most of the time but at the same time ensures that dequeuing of tasks is still synchronized.

Thread-Safe Linear Optimization. Despite synchronized access to the task queues and the single search tree nodes, adjustments to the implemented state set representations in HyPro have to be considered to make the tool thread safe. In general this does not require specialized approaches, however adapting an embedded linear optimization engine required some effort. HyPro allows to use different linear solving backends with a fallback to GLPK which are wrapped into an optimizer class. It is known that GLPK is not thread-safe, however with minor modifications it is possible to obtain a re-entrant version. This can be achieved by changing the maintained global GLPK-context object to a thread-local context. Now that each thread maintains its own GLPK-context, special care has to be taken to avoid memory-leaks. We extend our optimization wrapper class by mapping the unique thread id to the corresponding GLPK context and its problem instances. State set representations (e.g. support functions) which hold their own optimizer class instance now have to make sure the GLPK context for this instance is properly deleted upon joining threads, as for every thread which accesses this state set the corresponding mapping in the optimizer class is extended. To avoid this we provide clean-up methods, which should be called before a thread is joined. Clean-up deletes all GLPK-problem instances and removes the thread-local GLPK-context instance (which can only be deleted by its creating thread). In general creating a GLPK-problem instance upon request and deleting it afterwards would solve this issue as well – however as the same problem instance usually is used several times we reduce the overhead of creating and deleting these instances by keeping them as long as possible.

4 Experimental Results

We tested our implementation on several well-known benchmarks with a timeout (denoted as *to*) of 10 min on a machine equipped with 48×2.1 GHz AMD Opteron CPUs and a memory limit (*mo*) of 8 GB.

Benchmarks. Three benchmarks have been selected for empirical evaluation. We include two instances of the navigation benchmark [18] – instance 9 (`na09`, time horizon $T = 3\,\mathrm{s}$, jump depth $J = 9$) and instance 11 (`na11`, time horizon $T = 3\,\mathrm{s}$, jump depth $J = 8$). Both instances model a point mass moving on a two-dimensional plane subdivided into cells which each model different acceleration affecting the movement of the point mass. Due to the large set of initial states, these benchmarks usually exhibit strong branching behavior and thus should be well-suited to evaluate the capabilities of our implementation. Furthermore, we include an instance of Fisher's mutual exclusion protocol benchmark (`fish`, $T = 12\,\mathrm{s}$, $J = 13$) which also was used in [19]. This benchmark models several processes competing for a shared resource which can be accessed in a mutually exclusive way. As all processes have the same priority the model is non-deterministic and we expect to obtain a shallow search tree during analysis. In our evaluation we did not include benchmarks with little non-deterministic choices. Our central goals are to investigate on the influence of queue balancing methods and on the potential speed-up which can be achieved by parallelization. Additionally, our results show that the overhead caused by synchronization is small (see below) so we can expect little influence on running times for benchmarks with little branching.

Table 1. Parameter settings: Refinement strategies are lists of configurations, each configuration specified by a triplet (1) state set representation (box, support functions (sf)), (2) time step size, (3) aggregation (*agg*)/clustering in k clusters ($cl.k$). Additionally, the last column specifies the queue balancing rate.

Name	Refinement strategy	Work balancing
s_0	$(box, 0.1, agg), (sf, 0.01, agg), (sf, 0.001, agg)$	100%
s_1	$(sf, 0.1, agg), (sf, 0.01, agg)$	100%
s_2	$(sf, 0.1, agg), (sf, 0.01, cl.5)$	100%
s_3	$(box, 0.1, agg), (box, 0.01, agg)$	100%
s_4	$(box, 0.1, agg), (box, 0.01, cl.3)$	100%
s_5	$(box, 0.1, agg), (box, 0.01, cl.3)$	10%
s_6	$(box, 0.1, agg), (box, 0.01, cl.3)$	50%
s_7	$(box, 0.1, agg), (box, 0.01, cl.3)$	Global queue only

Settings. For our experiments we consider 8 different settings (see Table 1). Even though path refinement is not the main focus of our presented approach, all 8 settings support path refinement as this involves synchronization (see Sect. 3).

Each setting specifies a refinement strategy and a work queue balancing heuristics. A refinement strategy is a sequence of triplets, each triplet specifying (1) the state set representation used, (2) the time step size for flowpipe construction and (3) settings for aggregation/clustering. In regard to queue balancing, we made experiments with pushing all tasks above a threshold from the local queues to the global queues, but this was far less stable in efficiency than pushing a certain percentage of the local queue contents, therefore here we include only experiments with the latter. In Table 1, the work queue balancing heuristics specifies which portion of the local queues is moved to the global queues after the completion of each task (at least one task is always left in non-empty local queues, i.e., 100% means all but one).

Settings s_0–s_4 differ in their refinement heuristics, but they are all eager in pushing all but one task from the local to the global queues after the completion of each task. Contrary, settings s_4–s_7 share the same refinement heuristics but they differ in their work balancing method. Especially, setting s_7 completely avoids thread-local queues: every worker operates on the global queues directly. The difference is that, while in all other settings the work distribution takes place at the end of the flowpipe computation in a batch, s_7 pushes single successor tasks to the global queues during its computations such that idle workers potentially could start computation earlier. As the experimental results will show, this works surprisingly good, even though the increased synchronization effort is recognizable.

Table 2. Running times [sec.] for settings s_0–s_7, timeout (to) = 10 min, memout (mo) = 8 GB, † = safety cannot be shown. Running times averaged over 10 runs.

Benchmark	Setting	#threads						
		1	2	4	8	16	32	48
`na09`	s_0	**21.99**	20.32	20.32	20.40	20.34	20.29	20.35
	s_1	24.87	**15.72**	**11.87**	**11.70**	**11.68**	11.70	11.72
	s_2	to	to	to	to	mo	mo	mo
	s_3	†	†	†	†	†	†	†
	s_4	263.8	134.9	69.34	36.87	21.68	16.70	15.63
	s_5	252.8	127.9	64.79	32.85	17.00	**10.41**	**7.51**
	s_6	263.5	132.8	68.70	36.20	20.90	15.53	13.95
	s_7	78.52	46.60	32.01	29.52	34.21	42.23	45.03
`na11`	s_0	70.49	45.72	45.39	45.42	45.41	45.47	45.44
	s_1	**18.47**	**9.81**	**6.15**	**5.03**	**4.68**	4.49	4.50
	s_2	to	290.7	146.4	75.53	39.92	22.45	16.50
	s_3	†	†	†	†	†	†	†
	s_4	95.73	47.05	24.04	12.21	6.42	**3.60**	3.13
	s_5	93.68	45.85	23.28	12.03	6.57	4.02	3.54
	s_6	92.11	47.16	24.02	12.20	6.62	3.74	**3.02**
	s_7	95.92	49.12	25.12	13.03	8.02	6.16	6.49
`fish`	s_0	40.66	20.46	**10.43**	**5.49**	**2.96**	1.84	**1.61**
	s_1	to	to	to	393.9	201.5	107.2	79.02
	s_2	to	to	to	394.3	201.4	107.4	79.07
	s_3	40.57	20.44	10.47	5.54	2.97	**1.82**	1.78
	s_4	**40.56**	20.45	10.49	5.55	2.97	1.79	1.83
	s_5	40.63	20.47	10.87	6.76	4.56	3.92	3.96
	s_6	40.67	**20.42**	10.47	5.53	**2.96**	1.84	1.70
	s_7	42.73	21.79	11.26	6.06	3.68	3.45	3.93

Results. The running times for our experiments are listed in Table 2. In general, we can observe a speed-up when increasing the number of worker threads – we could achieve a speedup of up to factor 33 (`na09`) which in this case results in ∼70.1% efficiency ($efficiency = \frac{speedup}{\#threads}$) of the parallelization (`na11`: max. factor 30, `fish`: max. factor 25). Even though a general speed-up when using more worker threads can be observed, some instances (e.g. `na09`, s_0) stabilize in their running times. This indicates that either work is not well balanced or there is a heavy synchronization overhead.

For interpreting the results, it is important to mention that processing each single task is in general computationally expensive: the time required to compute a flowpipe is usually long in comparison to the time it takes to acquire a lock for synchronization and move tasks to global queues. Consequently the running times using one thread in our implementation resemble the running times of a purely sequential approach. Furthermore, with aggregation/clustering the number of generated new tasks is often relatively small. For example, for a deterministic system a task might generate just a single successor task, in which case no work balancing would take place at all. This might lead to insufficient work balancing and explain why for some benchmarks and some settings involving more workers does not lead to any additional speedup.

To further investigate upon this we ran the benchmarks with up to 48 threads. For benchmark instances such as the navigation benchmark in combination with settings where aggregation was used (s_0, s_1, s_3) we can observe that the running times already converge for a low number of threads as there are not enough tasks created during analysis such that most threads idle. The running times for these settings do not significantly increase when using more threads which confirms that our implementation successfully minimizes the synchronization effort required. An exception is setting s_7 on benchmark `na09`, where the running times increase when using more than 8 threads; as this setting only uses global queues, the increased need for synchronization is reflected in the running times.

To investigate on the actual work distribution we collected the number of tasks processed by each worker thread. Table 3 shows the coefficient of variation (CV) of these results to allow for statements about variance in the work distribution. The coefficient of variation as a relative measure for variance gives the influence of the variance of data on the mean in percent. Lower percentages hereby indicate a lower variance in data.

We can observe the influence of different queue balancing methods for benchmarks with settings which produce a lot of tasks (s_4–s_7). With increasing number of threads the average number of processed tasks per worker decreases. When using settings which produce too few tasks, many worker threads idle, thus increasing the variance of processed tasks per worker (see e.g. `na09`, s_0). As expected the setting using only global queues shows the lowest CV throughout the experiments as all available tasks are immediately shared.

Settings with local queues where 100% of the created tasks are shared are expected to exhibit a similar CV as when using global queues only, there are only two differences: firstly, when using global queues only, tasks are shared

Table 3. Coefficient of variation (left) and idle time (right) in percent for settings s_0–s_7, "–" marks failures (timeout, memout). Unsuccessful settings are left out.

Benchmark	Setting	#threads						#threads					
		2	4	8	16	32	48	2	4	8	16	32	48
`na09`	s_0	87.5	85.4	102.2	133.5	197.2	220.6	18.72	34.96	36.52	33.5	15.28	12.2
	s_1	32.7	43.6	39.0	44.8	92.5	118.4	10.63	28.78	36.52	28.29	12.76	10.21
	s_4	**0.1**	0.7	1.0	**1.3**	**1.7**	**2.2**	**0.04**	**0.18**	0.44	0.85	1.09	1.22
	s_5	0.4	1.1	1.8	2.7	4.0	4.9	0.16	0.46	1.07	2.30	4.33	6.16
	s_6	0.2	0.4	1.0	1.4	1.8	**2.2**	0.05	**0.18**	0.45	0.86	1.33	1.69
	s_7	0.4	**0.6**	**0.9**	**1.3**	2.2	2.7	0.11	0.23	**0.30**	**0.41**	**0.38**	**0.41**
`na11`	s_0	45.3	44.3	70.4	130.8	175.1	215.8	7.52	6.05	3.54	2.44	1.36	0.74
	s_1	24.0	15.3	29.2	45.1	90.5	121.0	4.13	20.95	40.91	47.22	33.84	25.93
	s_2	**0.4**	**0.9**	**1.9**	3.6	6.0	7.6	0.11	0.51	2.33	5.76	12.10	17.2
	s_4	0.9	1.7	10.3	11.0	15.7	13.5	0.11	0.44	1.21	3.45	5.79	6.17
	s_5	2.2	3.2	5.3	8.8	13.6	16.2	0.17	0.64	1.43	3.82	6.62	7.75
	s_6	1.4	2.1	2.6	17.3	11.9	12.6	0.07	0.46	0.81	3.35	6.35	7.74
	s_7	2.5	3.0	3.6	**3.3**	**3.9**	**5.5**	**0.01**	**0.30**	**0.72**	**1.63**	**2.67**	**2.78**
`fish`	s_0	0.6	3.6	7.6	8.8	11.6	14.3	0.32	1.22	3.32	5.94	10.21	12.33
	s_1	–	–	6.8	8.0	10.0	13.2	–	–	**0.44**	**1.09**	**2.44**	3.7
	s_2	–	–	7.6	8.5	10.0	12.7	–	–	**0.44**	**1.06**	**2.65**	3.8
	s_3	0.8	3.3	7.4	9.3	11.8	13.9	0.29	1.43	3.84	5.88	10.45	11.37
	s_4	0.8	3.4	7.1	8.0	11.3	13.9	0.29	1.19	4.39	6.41	9.90	11.70
	s_5	**0.3**	2.6	14.9	24.8	67.6	99.8	0.24	2.00	1.96	7.73	15.30	14.81
	s_6	0.9	3.4	8.1	8.4	11.6	14.0	0.32	1.29	3.98	6.01	10.42	11.85
	s_7	0.5	**1.2**	**2.6**	**2.9**	**4.1**	**4.9**	**0.23**	**0.67**	1.44	2.56	2.83	**2.50**

immediately after their creation, whereas in the presence of local queues sharing happens after task completion; secondly, 100% sharing with local queues is not exactly 100% as one single task is kept for further processing in a local queue. Strategies where a worker only shares part of its created tasks (s_5, s_6) show a larger variance i.e. work is less equally distributed. With regard to the observed running times we can deduce that sharing work comes at a price – even though setting s_7 has the lowest variance, the running times in comparison to settings s_4–s_6, which share the same analysis parameters are worse.

Note that a low CV can also be achieved when many threads are taking turns in processing a small number of such tasks. Therefore, we also analyzed the average share of idle time for all threads (see Table 3, right). We can conclude that the increased running time for setting s_7 indeed can be amounted to synchronization, as the idle time for the workers is amongst the lowest ones.

5 Conclusion

We have presented a natural approach to parallelize reachability analysis for linear hybrid systems. Experimental results show a general reduction of the analysis times. The observed synchronization overhead is minor compared to what we gain from the parallel execution and thus this approach is usable also for systems for which the search tree has a low level of branching (e.g. for deterministic systems). Naturally, the possibilities of work sharing are restricted to problem instances with a low level of non-determinism. The used modular approach allows for several extensions and improvements as future work: (i) combining this method with the approach presented in [13], and (ii) using specialized workers which allow for subset-computations which can be performed in parallel.

References

1. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Trans. Robot. **30**(4), 903–918 (2014)
2. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
3. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
4. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
5. Frehse, G., Kateja, R., Le Guernic, C.: Flowpipe approximation and clustering in space-time. In: Proceedings of HSCC 2013, pp. 203–212. ACM (2013)
6. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Anal. Hybrid Syst. **4**(2), 250–262 (2010)
7. Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T.T., Ladan, H., Podelski, A., Wehrle, M.: Guided search for hybrid systems based on coarse-grained space abstractions. STTT **18**(4), 449–467 (2016)
8. Schupp, S., Ábrahám, E.: Efficient dynamic error reduction for hybrid systems reachability analysis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 287–302. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_17. Accessible for reviewers under https://ths.rwth-aachen.de/research/publications/
9. Bak, S., Duggirala, P.S.: Simulation-equivalent reachability of large linear systems with inputs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 401–420. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_20
10. Schupp, S., Nellen, J., Ábrahám, E.: Divide and conquer: variable set separation in hybrid systems reachability analysis. In: Proceedings of QAPL 2017. EPTCS, vol. 250, pp. 1–14. Open Publishing Association (2017)
11. Bogomolov, S., Forets, M., Frehse, G., Podelski, A., Schilling, C., Viry, F.: Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices. CoRR abs/1801.09526 (2018)
12. Chen, X., Sankaranarayanan, S.: Decomposed reachability analysis for nonlinear systems. In: Proceedings of RTSS 2016, pp. 13–24. IEEE Computer Society Press (2016)
13. Ray, R., Gurung, A.: Parallel state space exploration of linear systems with inputs using XSpeed. In: Proceedings of HSCC 2015, pp. 285–286. ACM (2015)

14. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of LICS 1996, pp. 278–292. IEEE Computer Society Press (1996)
15. Schupp, S., Ábrahám, E., Makhlouf, I.B., Kowalewski, S.: HyPro: A C++ library of state set representations for hybrid systems reachability analysis. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 288–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_20
16. Schupp, S., Ábrahám, E., Chen, X., Ben Makhlouf, I., Frehse, G., Sankaranarayanan, S., Kowalewski, S.: Current challenges in the verification of hybrid systems. In: Berger, C., Mousavi, M.R. (eds.) CyPhy 2015. LNCS, vol. 9361, pp. 8–24. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25141-7_2
17. Frehse, G., Ray, R.: Design principles for an extendable verification tool for hybrid systems. In: Proceedings of ADHS 2009, pp. 244–249. IFAC-PapersOnLine (2009)
18. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_22
19. Bu, L., Ray, R., Schupp, S.: ARCH-COMP17 category report: bounded model checking of hybrid systems with piecewise constant dynamics. In: Proceedings of ARCH 2017. EPiC Series in Computing, vol. 48, pp. 134–142. EasyChair (2017)