EPiC Series in Computing

Volume 43, 2017, Pages 80–90

ARCH16. 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems

EPiC Computing

High-level Hybrid Systems Analysis with Hypy (Tool Presentation)

Stanley Bak[1], Sergiy Bogomolov[2], and Christian Schilling[3]

[1] Air Force Research Laboratory, Information Directorate, USA
[2] IST Austria, Austria
[3] University of Freiburg, Germany

Abstract

Hybrid systems play an important role in many application domains. A range of powerful analysis methods for this class of systems perform *high-level analysis*, where, iteratively, (1) a reachability computation is performed on a system model, (2) the result of the analysis is examined, and (3) the model is modified and the process repeats. For example, a well-known high-level analysis method is counter-example guided abstraction refinement (CEGAR), where, at each iteration, the model is refined based on the counter-example produced by the reachability computation.

In this paper, we present hypy, a python library which strives to ease the development of high-level analysis approaches. Hypy provides the necessary machinery to run a number of up-to-date hybrid systems analysis tools, parse their outputs, and modify the models. The modifications are performed using HyST, a source-to-source model transformation framework, which supports output formats including SpaceEx, Flow*, dReach, and HyCreate. HyST, however, does not run reachability tools nor interpret their output. The developed hypy library fills this gap, providing an extendable and flexible architecture which simplifies development of complex analysis strategies. We demonstrate its practical potential on three non-CEGAR case studies: abstraction for parameter identification, generation of pseudo-invariants to reduce reachability overapproximation error, and completely automatic tool parameter tuning for the Flow* reachability tool.

1 Introduction

Hybrid systems [1] play an important role in many application domains, and various tools and algorithms for reachability analysis have been developed recently. Many analysis approaches for these systems perform *high-level analysis*, where the computation is decomposed into a number of iterations handling modified versions of the input model or analysis parameters.

For example, a widely-used instance of iterative analysis is counter-example guided abstraction refinement (CEGAR) [2, 13]. The approach works by refining an initial abstraction until it is fine enough to reason about the given property. In other words, the approach analyzes a sequence of increasingly precise abstractions and calls a verification engine for each of them.

A second instance of an iterative workflow is in the falsification setting, where the primary goal is to find an error trajectory. Approaches based on pattern databases [6], for example, use the results of an abstract analysis to guide the error-trajectory search in the concrete state space. Here, the methods first use the parameterization capabilities of modern reachability analysis tools in order to later guide a concrete counter-example search.

The above-mentioned approaches exemplify the importance of high-level analysis in hybrid systems. We have created the tool hypy in order to simplify the process of developing such strategies. Our tool provides the necessary machinery to run a number of up-to-date hybrid systems analysis tools, parse their output results, and modify the input model between reachability tool runs. Hypy can even be used for the development of composite methods relying on multiple tools / reachability algorithms as part of their workflow. Hypy uses the model modification and conversion machinery provided by HyST [5], a framework for source-to-source model transformation which supports a number of up-to-date tools including SpaceEx [16, 7], Flow* [11], dReach [17], and HyCreate [3]. It does not, however, support running the tools or interpreting their output. Hypy fills this gap by providing a python-based interface to automate calls to HyST, run each of the tools, and programmatically retrieve tool results. The script-based architecture allows fast extensions to individual needs. We demonstrate practical potential of our tool on three short case studies: a parameter identification abstraction process, automatic generation of pseudo-invariants to reduce reachability overapproximation, and automated tool parameter tuning for Flow*.

The remainder of the paper is organized as follows. First, Section 2 provides an outline of the organization of hypy and its capabilities. In Section 3, we discuss three short case studies which show practical benefits of hypy. This is followed by a discussion about related work in Section 4 and a conclusion in Section 5.

2 Hypy Organization

Hypy is written in python, and provides both a way to run a set of hybrid analysis tools, parse and interpret their output, and perform model modification. A simple hypy script which performs a single reachability computation using SpaceEx and produces a plot is given below.

```
import hybridpy.hypy as hypy

e = hypy.Engine()
e.set_model("toy.xml")
e.set_tool("spaceex")
e.set_output_image("result.png")

if e.run() != hypy.RUN_CODES.SUCCESS:
    print "engine.run() returned error: " + str(code)
    exit(1)
```

Internally, hypy consists of:

- tool-specific python scripts, which contain the logic for running each tool, producing a `.png` plot, and parsing tool-specific output into python objects;
- a tool-independent core, which can generate models for each tool by running HyST, copy intermediate files to and from temporary directories during processing, and call the tool-specific scripts to perform reachability analysis, produce plots, and parse output.

Each tool-specific script contains a python class which inherits from the core's `HybridTool` class, and implements abstract methods to run the tool (`run_tool`), produce a plot (`make_image`), and parse tool output into a python object (`make_output`). The input format is SpaceEx's SX [14] model format, which is converted using HyST into the formats of the individual tools. The generic `HybridTool` class contains logic to create temporary directories and copy model

`Engine` Method Name	Purpose
`set_tool`	selects which tool to run
`set_model`	sets the input model path
`set_timeout`	sets a timeout while running
`set_save_model_path`	sets whether to save converted model output by HyST
`set_output_image`	sets the output image path
`set_print_terminal_output`	sets whether to print stdout
`set_save_terminal_output`	sets whether to save stdout
`set_create_result`	sets whether the tool-specific output object should be created
`set_tool_params`	sets HyST parameters

Table 1: Methods callable on `Engine` objects in `hypy`.

files to them, run the tool, produce plots and parse output (by calling the appropriate abstract method from the tool-specific script), and copy result files back. Tool-specific scripts are also individually runnable on model files for that particular tool, which enables easy GUI integration (for example, right-click a model file, 'Run with Flow*').

The `hypy` core's main programmatic interface is through an `Engine` class, which contains instances of every implemented `HybridTool` object. Scripts that use `hypy` can instantiate an `Engine` object, and then call various methods to customize the computation, outlined in Table 1. Scripts can then call the `run` method to perform the conversion, computation, and/or produce the image output, and then `get_result` to get the tool-specific parsed output object.

Hybrid systems analysis tools require individual attention in order to be run, produce plots, and interpret output.

In terms of running, SpaceEx and dReach accept the input model file on the command-line. SpaceEx splits the input model into two files, an automaton description and a configuration file which specifies the initial states and various tool parameters. dReach has a command-line parameter indicating the number of discrete transitions to unroll when performing a check. HyCreate also has the model file passed in on the command line, but a flag is needed to run it in batch mode where the GUI does not appear. Flow* reads the model file input from stdin.

Plotting a `.png` image output result is also tool-specific. SpaceEx outputs a data file which can be processed to the `graph` tool (with the appropriate flags to produce a `.png` image). dReach, when run with the appropriate flag, can output a visualization JSON file, which can then be displayed by starting a local web server and setting up an HTML file which runs Javascript to load the JSON data[1] [20]. HyCreate directly outputs a `.png` plot. Flow* can output a `gnuplot` script. This script is called using the `gnuplot` program which creates an encapsulated postscript (`.eps`) image. This image is then loaded using `gimp`, rotated to be upright, and converted to a `.png` using `gimp`'s `script-fu` interface.

Each tool's output is in a slightly different format. Currently, the tool-independent core automatically saves any stdout output produced while a tool is running, along with time stamps of when the output was produced. Tool scripts can perform additional processing on either the output produced by the tools, or the text printed to stdout. The SpaceEx-specific script can read the output file produced when the INTV output mode is selected, which gets parsed into a python object containing the names of every reachable mode, and the interval ranges for every variable both globally and within each mode. Additionally, the stdout output printed is examined to determine if SpaceEx converged or reached the maximum number of iterations. The Flow*-specific script processes the stdout output into a python list that describes each mode where computation was performed, and the amount of reachability time computed in each mode. Additional output processing can be added to `hypy` as the needs arise. The lines of python code

[1]This process would then require further processing to convert it to a `.png` and has not been implemented in `hypy`.

Script Filename	Lines of Code
`tool_spaceex.py`	274
`tool_flowstar.py`	128
`tool_hycreate.py`	59
`tool_dreach.py`	57

Table 2: The lines of code for the hypy tool-specific scripts.

for each of the tool-specific scripts is provided in Table 2. Tools which currently support more advanced output processing (SpaceEx and Flow*) have slightly larger implementations.

3 Case Studies

Hypy eases implementations of high-level analysis for hybrid systems models. For each application, a script must be written that can make use of hypy's ability to run HyST and then interpret the output of the computation. The script contains application-specific logic on how the model should then be modified for subsequent computations. In this section, we present three different applications that demonstrate various use-cases for hypy.

3.1 Model Parameter Identification

In the first case study, we extend a recent approach to do parameter identification for multiaffine systems [9], implemented in the Hydentify tool. Given a parametric system with multiaffine dynamics, a parameter domain, and a safety specification, the authors present a technique for identifying polytope-shaped parameter classes which guarantee that the system meets its specification. The algorithm identifies parameter equivalence classes and performs a hierarchical search in this space. For each equivalence class, the algorithm abstracts the system to a linear hybrid automaton (LHA) which is then analyzed by SpaceEx. When SpaceEx reports a violation, the search algorithm proceeds recursively by partitioning the currently considered parameter equivalence class into two subclasses.

We make two key observations here. First, the LHA are considered in a hierarchical tree-like manner, which induces a parent-child relation between them. Therefore, by construction, it always holds that the trajectory set of a parent-LHA encompasses the trajectories of every child-LHA. Second, while the dynamics become more restrictive and transitions may become unused, all LHA have the same discrete modes.

From these observations we conclude that, if a mode is not reachable in a parent-LHA, then it is also not reachable in all child-LHA and thus can be removed from consideration here. This step improves performance in two computationally time-intensive phases of the parameter identification algorithm: (1) input model parsing and data structure initialization in SpaceEx, which involves slow hard-disk operations; and (2) computation of LHA abstractions for every parameter equivalence class, which relies on expensive polytope operations.

Hydentify by default uses a SpaceEx version which stops as soon as a specification violation is detected. In this way, the run time is improved as exhaustive fixpoint analysis is avoided. We observe that, for the modification described above to be sound, modes can be removed in the child-LHA only after a fixpoint has been detected. Therefore, an exhaustive analysis is of necessity, which raises the question of whether the time saved due to model trimming outweighs the overhead due to exhaustive reachability analysis.

We incorporated hypy into the existing Hydentify computation flow to explore this trade-off. Rather than running SpaceEx directly, a 59-line python script was made which used hypy to run the tool and interpret its output. When SpaceEx terminates, it also outputs all reachable

Max. Level	Modes Saved	Run Time (sec)
0	–	7,172
1	85	4,368
2	85	4,397
∞	92	6,867

Table 3: The largest run time saving occurs when the hypy trimming strategy is used only at the first level of abstraction.

states. Hypy parses this output into a data structure to the custom script, which then writes a file containing the names of all modes that are reachable. In addition, hypy adds the information of whether a fixpoint was found or not, which is normally output on stdout from SpaceEx. Then Hydentify parses the output of the script and, if a fixpoint was found, marks the reached modes in an internal data structure. Afterward, Hydentify proceeds in the normal way while ignoring all unmarked (unreachable) modes.

To evaluate our extension, we consider a cardiac cell model which can be used to analyze excitement conditions of the ventricular myocytes [18]. We compare the performance of the original Hydentify implementation and the adjusted version which embeds hypy as part of the workflow. The standard version of Hydentify produces 54 different LHA, each with 128 modes, which takes 2 hours in total.

We first ran the new modification using the hypy trimming strategy for all LHA in the search tree. The run time was slightly reduced by 5 minutes, which is not too significant. Notice, however, that saving a mode on a higher level of the search tree, near to its root, has a larger impact compared to mode trimming on lower levels as absence of modes is propagated to all child-LHA. Hence, we then investigated a hybrid approach and used our extension in the first k levels of the search hierarchy *only* and switched back to the old behavior of Hydentify for all later runs. Having applied an exhaustive reachability analysis to the root LHA only, i.e., $k = 1$, we already saved 85 modes and 46 minutes of run time. While applying hypy also for the second level, i.e., $k = 2$, we cannot remove more modes, and consequently the run time gets worse by 29 seconds due to two (unnecessary) fixpoint computations in SpaceEx. We observed a continuing increase in run time for $k > 2$, and the number of removed modes only goes up to 92 at most. The results are summarized in Table 3. Using hypy, we were able to quickly investigate a trade-off between model trimming and exhaustive reachability analysis.

3.2 Pseudo-Invariants

The method of pseudo-invariants [4] is a technique used to improve the accuracy and speed of reachability computation. Many tools that do reachability computation are flow-pipe construction methods. These methods compute the continuous-post operation by tracking the currently reachable set of states at specific points in time. The set of states being tracked can quickly become quite complex, and the tools will compute overapproximations of this true set of reachable states, which is sufficient for safety guarantees. For example, Flow* uses a Taylor model [22] representation, and SpaceEx can use a polyhedral, zonotope, or support function representation. This simplification, however, leads to wrapping-effect error [21] in the computation.

Wrapping-effect error can sometimes be reduced in the computation by splitting a single mode of the hybrid automaton into multiple modes. The dynamics and discrete transitions remain unchanged, and the only difference is that artificial invariants and guards are used to create discrete transitions at specific boundaries in space. When the invariant becomes false, a discrete transition is taken to the copied mode, which has identical dynamics and discrete successors. Theoretically, this does not change the reachable set of states. Practically, however,

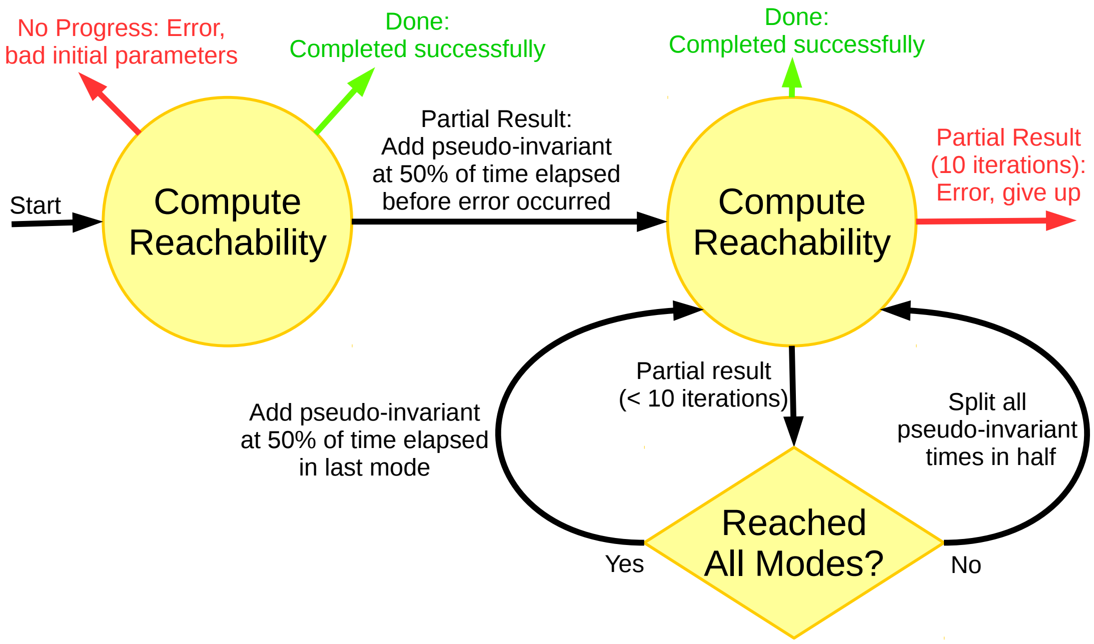

No Progress: Error, bad initial parameters
Done: Completed successfully
Done: Completed successfully
Partial Result: Add pseudo-invariant at 50% of time elapsed before error occurred
Start
Compute Reachability
Compute Reachability
Partial Result (10 iterations): Error, give up
Partial result (< 10 iterations)
Add pseudo-invariant at 50% of time elapsed in last mode
Split all pseudo-invariant times in half
Reached All Modes?
Yes
No

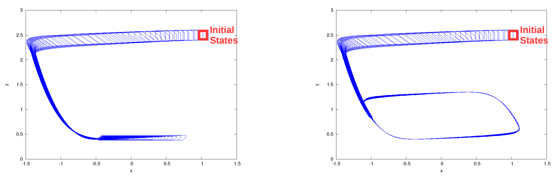
Figure 1: The logic to automatically create pseudo-invariants is encoded into a 123-line python script which uses the hypy library to run Flow* and parse its output.

the discrete transition effectively pauses the flow-pipe construction process along its boundary. This can significantly reduce the size of the set of states that needs to be tracked, reducing computation time and overall error.

A key concern with pseudo-invariants is how to come up with them: at what boundaries in space should discrete transitions be forced? HyST provides two options for applying pseudo-invariants to a model. First, the user can manually provide a sequence of auxiliary hyperplanes to act as the pseudo-invariant boundaries. Second, the user can provide a set of simulation times, and HyST will perform a simulation for those amounts of time from the center of the initial set. Auxiliary hyperplanes are then inserted at the resultant points, in directions orthogonal to the gradient of the dynamics at each point. Both options, however, require the user to provide parameters.

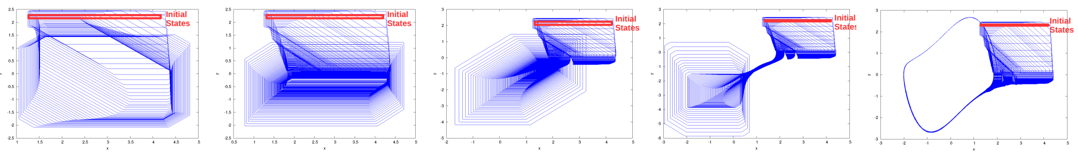
We wrote a script which uses hypy to make the process completely automatic. The script first runs Flow* on the initial model. When Flow* reports a problem due to too much representation error during flow pipe construction ("The remainder estimate is not large enough."), the script uses hypy to look at the tool's output in order to examine the point in time of the computation (reachability time, not wall time) when the error occurred. A time of 50% of the error time is then used as the simulation time in order to derive the auxiliary hyperplane for use in the HyST transformation pass. In this way, a discrete transition is forced before too much error is accumulated by Flow*. The tool is then run again. If another error occurs during computation, and all the modes were reached, then another auxiliary hyperplane is inserted by splitting the time elapsed in the final mode. If not all modes were reached, all pseudo-invariant times are split in half. Using the new simulation times, HyST is run again and the process repeats, up to 10 iterations. The logic for this hypy script is shown in Figure 1.

We evaluated the proposed strategy on two examples.



(a) Initial Reachable Set (b) With Pseudo-Invariant

Figure 2: Reachability using Flow* for the neuron system.



(a) Initial run (b) Iteration 1 (c) Iteration 2 (d) Iteration 3 (e) Iteration 4

Figure 3: Reachability plots using Flow* for the Van der Pol system.

The first example is a 2-D neuron model which was previously used to evaluate nonlinear reachability techniques [15]. The dynamics for this system are $\dot{x} = x - x^3 - y + 0.875$, and $\dot{y} = 0.08 * (x + 0.7 - 0.8 * y)$, and we use the same initial set of states as the earlier work, $(x, y) \in [0.9, 1.1] \times [2.4, 2.6]$. Using the unmodified model, Flow* does not successfully finish computing reachability, but rather produces a remainder-estimate error after 25.45 seconds of reachability time. The created script detects this, uses HyST to insert a pseudo-invariant at simulation time 12.725, and reruns Flow*. Reachability then completes successfully for the modified model. The reachability plots for both runs are shown in Figure 2.

Second, we evaluate the Van der Pol oscillator, with dynamics $\dot{x} = y$, $\dot{y} = (1 - x^2) * y - x$. We use a parameterized initial set, where initially $y \in [2.28, 2.32]$ and $x \in [1.25, 1.25 + 0.01 * w]$. In earlier work [19], a novel Runge-Kutta based method which uses rigorous numerics based on affine arithmetic was formalized in Isabelle/HOL, and could successfully compute reachability on this model up to $w = 175$ ($x \in [1.25, 3]$) in about 70 seconds. Furthermore, the approach was compared to Flow* with manually derived auxiliary hyperplanes up to $w = 75$, after which the authors reported that they were "unable to come up with hyperplanes that would allow Flow*-PI integration for larger values of w." Manually coming up with pseudo-invariants can be difficult.

Using the developed script, reachability for this model was computed using a *significantly* larger initial set, with $w = 300$ ($x \in [1.25, 4.25]$). After multiple runs of HyST and Flow*, the automated approach eventually succeeded after inserting pseudo-invariants at simulation times 0.22, 0.77, 1.84, and 3.185. The script took 215 seconds for all the iterations to complete, 73 of which were on the final (successful) iteration. The reachability result for each iteration of the tool is shown in Figure 3.

With the created hypy script, the method of pseudo-invariants then can be used in a completely automated fashion. This also demonstrates the integration of hypy with both HyST and a reachability tool inside a larger iterative loop.

3.3 Tool Parameter Tuning

Tools for computing reachability often contain many parameters which may need to be carefully tuned in order to successfully analyze reachability for a particular model. An expert for a particular tool has an intuitive sense as to how changing these parameters is likely to affect accuracy and computation time. A novice in the tool is thus likely to spend significant time finding out and tuning each of the parameters, or may falsely conclude a given tool cannot analyze a model because the parameters she used were incorrect. Hypy can be used to create parameter-tuning strategies, where a model is analyzed multiple times using different parameters in order to try to find a valid parameter setting.

Flow* contains several options which affect its accuracy and performance. In particular, the time step, order of the Taylor models, and the remainder estimate will affect the quality of the result. An incorrect setting in any of these and the tool may exit before completing the reachability computation, or take excessively long to complete.

We created a proof-of-concept 112-line hypy script which selects these parameters with the goal of producing a configuration that can complete the computation for a given model. The input to the tuning process is the model file together with the duration of desired reachability time. The encoded strategy is simple, and consists of three nested loops. In the outer loop, the time step is selected, trying, in order, $\frac{1}{100}$, $\frac{1}{400}$, $\frac{1}{800}$, $\frac{1}{3200}$, and $\frac{1}{12800}$ of the total desired reachability time. In the middle loop, the Taylor model order is picked, trying, in order, 2, 4, and 6. Finally, the inner-loop iterates on the remainder estimate, which is used as a check to ensure the Picard Iteration used in the Taylor Model reachability approach is converging. The inner loop checks the values 1e-9, 1e-5, and 1e-2. The intuition behind these parameter choices is that it is generally better to try more restrictive parameters first, as these are more likely to result in an error quickly, and generally take less time if they are successful.

In addition to the nested loops, HyST is used to generate the Flow* models, which automatically chooses which of the integration modes to use in each location of the automaton based on the types of dynamics and the number of dimensions ('poly ode 1' for less than 3 dimensions, 'poly ode 2' for 4 to 6 dimensions, 'poly ode 3' for more than 6 dimensions, and 'nonpoly ode' if nonpolynomial dynamics are used). Based on the advice in the Flow* manual, QR preconditioning is used if there are less than four variables; otherwise identity preconditioning is selected. As soon as a set of parameters is found where reachability can be computed to the desired time, the process is considered successful. The process can fail if Flow* takes more than an hour computing with a given set of parameters, or if all the parameters are tried and none succeed.

We ran the automatic parameter selection script on a number of models from the Flow* case studies [12], where model files with hand-tuned parameters were provided for comparison. The results are shown in Table 4. The 'Tune (sec)' column shows the amount of time spent tuning the model, before the final (successful) run of the tool. The 'Run (sec)' column is the run time for the final run to be completed.

Of the 7 models tried, the strategy succeeded in finding working parameters for 4 of them. For the lorenz long model, the tool reached the one hour timeout while computing, and may have completed if given more time. For the biology7 model, the tool also timed out after running for an hour, but without indicating any progress on stdout, which could indicate a bug in Flow* and would require more investigation. For the tank model, all parameters were tried in less than a minute without completing successfully. Examining the hand-tuned parameters in the Flow* case study model, the provided file uses per-variable remainder estimates, rather than having the same value for all variables. Since our script does not do this, it could not find a valid set of parameters for this model.

Model	Tune (sec)	Run (sec)	Step	Order	Rem
brusselator	3.90	18.47	0.15	6	1e-2
lorenz_short	2.48	12.26	0.01	4	1e-2
bio9	15.19	10.14	0.0005	2	1e-2
3dstable	7.82	110.64	0.025	4	1e-2
tank	50.90	-	-	-	-
lorenz_long	7787.46	-	-	-	-
bio7	3604.07	-	-	-	-

Table 4: Automatic tool parameter tuning in Flow*.

The purpose of this case study was to demonstrate using hypy to perform automated parameter tuning, not necessarily to develop the best parameter tuning strategy. Despite its simplicity, the developed script was successful on more than half of the models tried. More complicated strategies, for example, strategies that perform per-variable remainder estimate selection, could be created to increase the number of models that can be analyzed automatically. Completely automatic analysis is not just useful for new users of tools, but could also enable tool competitions, which have proven beneficial to the SAT solving tool community.

4 Related work

In the last decade, significant effort has been devoted to address semantical differences between tools for hybrid systems.

Sangiovanni-Vincentelli *et al.* suggested the hybrid systems interchange format (HSIF) [10, 23] for this purpose.

Bogomolov *et al.* [8] suggested to use the Function Mock-up Interface (FMI) in order to co-simulate hybrid systems. In that setting, the authors give up on finding a "common denominator" of the tools semantics, but augment the simulation engine with a common API which enables tool synchronization. Unfortunately, at the current stage the FMI framework can be applied only in the *simulation* setting and does not provide any means to ensure interoperability between tools in the *verification* setting.

As already outlined, Bak *et al.* [5] presented the HyST framework for source-to-source transformation of hybrid models. In this work, hypy builds on the HyST machinery and particularly internal hybrid systems representation to transfer hybrid system models between different tools.

5 Conclusion

Hypy is a python-based library that enables quick development of complex, high-level analysis strategies that require multiple runs of hybrid systems reachability tools.

We demonstrated three applications which use hypy: Iterative parameter identification, completely automatic synthesis of pseudo-invariants, and tool parameter tuning for Flow*. Each of these applications contained a short script file, which called the appropriate hypy methods in order to run the analysis tools or modify the system model using the HyST model transformation tool.

Further analysis methods are envisioned using hypy. Portfolio approaches are possible, where multiple tools are run in parallel and the result is used from the one which terminates first. Additionally, ensuring that computed reach sets intersect between outputs of different tools can be used as a way to detect implementation bugs.

Acknowledgments

DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA #88ABW-2015-5083)

This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 1995.

[2] R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 250–271, 2003.

[3] S. Bak. Hycreate: A tool for overapproximating reachability of hybrid automata. Demo and Poster Session, ACM/IEEE 16th International Conference on Hybrid Systems: Computation and Control (HSCC 2013), 2012.

[4] S. Bak. Reducing the wrapping effect in flowpipe construction using pseudo-invariants. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, CyPhy '14, pages 40–43, New York, NY, USA, 2014. ACM.

[5] S. Bak, S. Bogomolov, and T. T. Johnson. HyST: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC, Seattle, WA, USA, April 14-16, 2015*, pages 128–133. ACM, 2015.

[6] S. Bogomolov, A. Donzé, G. Frehse, R. Grosu, T. T. Johnson, H. Ladan, A. Podelski, and M. Wehrle. Guided search for hybrid systems based on coarse-grained space abstractions. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[7] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. S. Pasareanu, A. Podelski, and T. Strump. Assume-guarantee abstraction refinement meets hybrid systems. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014*, LNCS, pages 116–131. Springer, 2014.

[8] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikucionis, T. Strump, and S. Tripakis. Co-simulation of hybrid systems with SpaceEx and Uppaal. In *Proceedings of the 11th International Modelica Conference*, 2015.

[9] S. Bogomolov, C. Schilling, E. Bartocci, G. Batt, H. Kong, and R. Grosu. Abstraction-based parameter synthesis for multiaffine systems. In *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference (HVC)*, LNCS. Springer, 2015. to appear.

[10] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1, 2006.

[11] X. Chen, E. Abraham, and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. *2013 IEEE 34th Real-Time Systems Symposium*, 0:183–192, 2012.

[12] X. Chen, E. Abraham, and S. Sankaranarayanan. Constructing flowpipes for continuous and hybrid systems: Case-studies. Online, 2013.

[13] E. Clarke, A. Fehnker, Z. Han, B. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[14] S. Cotton, G. Frehse, and O. Lebeltel. The SpaceEx modeling language. http://spaceex.imag.fr/documentation/user-documentation/spaceex-modeling-language-33, 2010.

[15] T. Dang and R. Testylier. Reachability analysis for polynomial dynamical systems using the Bernstein expansion. *Reliable Computing*, 17(2):128–152, 2012.

[16] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[17] S. Gao, S. Kong, and E. Clarke. Satisfiability modulo ODEs. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[18] R. Grosu, G. Batt, F. Fenton, J. Glimm, C. L. Guernic, S. Smolka, and E. Bartocci. From cardiac cells to genetic regulatory networks. In *In Proc. of CAV'11, the 23rd International Conference on Computer Aided Verification*, LNCS, Cliff Lodge, Snowbird, July 2011. Springer Verlag.

[19] F. Immler. Verified reachability analysis of continuous systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2015.

[20] S. Kong. How to use ODE visualization. Github, 2014.

[21] R. Moore. *Interval analysis*. Prentice-Hall series in automatic computation. Prentice-Hall, 1966.

[22] M. Neher, K. R. Jackson, and N. S. Nedialkov. On Taylor model based integration of odes. *SIAM J. Numer. Anal*, 45, 2007.

[23] A. Pinto, L. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli. Interchange format for hybrid systems: Abstract semantics. In J. P. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 491–506. Springer Berlin Heidelberg, 2006.