EPiC Series in Computer Science

Volume 34, 2015, Pages 180–187

ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems

Tool Presentation: Isabelle/HOL for Reachability Analysis of Continuous Systems

Fabian Immler*

Institut für Informatik, Technische Universität München
immler@in.tum.de

Abstract

We present a tool for reachability analysis of continuous systems based on affine arithmetic and Runge-Kutta methods. The distinctive feature of our tool is its verification in the interactive theorem prover Isabelle/HOL: the algorithm is guaranteed to compute safe overapproximations, taking into account all round-off and discretization errors.

1 Introduction

There exist several tools that perform reachability analysis of continuous systems, in this paper, we present a prototype of a tool that is special because of its trustworthiness. Why is trustworthiness an issue? Of course, the model might not represent the real world accurately and we cannot capture errors like that. But there are complex algorithms used and it is easy to get subtle details wrong, signs of expressions, indices, round-off errors. So what the analyzer outputs might not be faithful to the mathematical ideas behind the tool.

Here we present a tool that is verified with the interactive theorem prover Isabelle/HOL [11]. HOL stands for *higher order logic*, a logic in which one can formalize mathematics. Proofs for propositions are mechanically checked by Isabelle/HOL – it is therefore a *theorem prover*. However coming up with proofs needs to be guided by the user – therefore *interactive*.

Higher order logic can also be seen as a functional programming language. Our tool performs reachability analysis (inspired by Bouissou *et al.* [2]) using affine arithmetic [3] and Runge-Kutta methods. It is implemented as a functional program in Isabelle/HOL. Because of the fact that mathematics – and in this case ODEs – are formalized in the same logic, we are able to conduct a proof of correctness and have it checked by Isabelle/HOL.

The advantage of this approach is that it yields highly trusted code and one can be quite certain that reachable sets in the algorithm are safe enclosures for the abstract notion of solution to the ODE. This advantage comes at the cost of an implementation that is not as optimized as comparable tools. The tool is also harder to modify or experiment with, because one needs to change not only the code but also the corresponding formal proofs. But of course this prevents the introduction of bugs.

*Supported by the DFG RTG 1480 (PUMA)

2 Isabelle/HOL

We start with giving some background on interactive theorem proving with Isabelle/HOL and the formalization of mathematics therein. Isabelle is an LCF style theorem prover: theorems are an abstract datatype, new instances of which can only be constructed from existing theorems or axioms via a set of primitive inference rules (like modus ponens) implemented in a small kernel.

2.1 A Short Introduction to Interactive Theorem Proving

The most popular logic in Isabelle is higher order logic, Isabelle/HOL. It can be seen as a functional programming language, i.e. one can for example define datatypes like the type `nat` of natural numbers $\mathbb{N}$ with 0 and a successor function `Suc`.

```
datatype nat = 0 | Suc nat
```

Recursive functions like `add` can be defined via pattern matching:

```
fun add where
  "add 0 y = y"
| "add (Suc x) y = Suc (add x y)"
```

The *logic* part allows to state propositions like commutativity of addition

```
lemma "add x y = add y x"
```

and give the system hints to prove that proposition by induction on the first argument x,

```
apply (induct x)
```

where the system (and this is an example of interaction) requires to prove two subgoals, corresponding to the basis and the inductive step:

```
goal (2 subgoals):
 1. add 0 y = add y 0
 2. ⋀x. add x y = add y x ⟹
        add (Suc x) y = add y (Suc x)
```

For both cases it is helpful to first prove recursive equations in the second argument of add, which is a straightforward induction:

```
lemma [simp]:
  "add x 0 = x"
  "add x (Suc y) = Suc (add x y)"
  by (induct x) simp_all
```

Then the system can solve both subgoals automatically using

```
by simp_all
```

Behind the scenes, the first subgoal is solved by rewriting `add 0 y` with the defining equation and `add y 0` with the just now provided equation for `0` in the second argument. Similarly for the second subgoal: The system rewrites `add (Suc x) y` to `Suc (add y x)` by using the defining equation and then `Suc (add y x)` to `Suc (add x y)` with the induction hypothesis. This equals the right hand side of the proof obligation after rewriting (the right hand side) with the equation for `Suc` in the second argument.

2.2 Formalization of Mathematics in Isabelle/HOL

Now you can imagine that integers can be formalized using natural numbers $\mathbb{Z} = \alpha(\mathbb{N} \times \mathbb{N})$ with $\alpha(n, m) = n - m$, rational numbers as quotients of integers $\mathbb{Q} = \alpha(\mathbb{Z} \times \mathbb{Z})$ with $\alpha(i, j) = \frac{i}{j}$ and real numbers with (Cauchy) sequences of rational numbers: $\mathbb{R} = \alpha(\mathbb{N} \to \mathbb{Q})$ with $\alpha(x) = \lim_{i \to \infty} x(i)$.

All of those types are equipped with operations (like `add` for natural numbers) and proofs of properties about them (like commutativity of addition). Note that not all of the operations need to be defined as functional programs, they often can not even be: there are uncountably many real numbers.

Real numbers have been used to develop a theory of multivariate analysis (as ported from Harrison's work in HOL Light [4]), with e.g. differentiation and integration. This theory has been used to formalize ODEs [9].

2.3 Rigorous Numerical Methods in Isabelle/HOL

There are different formalizations for numerical computations, a library for interval arithmetic [5] (including approximations of transcendental functions) and affine arithmetic, including verified algorithms for geometric intersection of zonotopes with hyperplanes [7]. Those numerical methods are guaranteed in the sense that respective operations in e.g. affine arithmetic always enclose the "real" quantities. For example, addition $\oplus$ of affine forms comes with a correctness theorem like $(x, y) \in \alpha(A, B) \implies x + y \in \alpha(A \oplus B)$, where $\alpha(A, B)$ is the joint range $\{(A_0 + \sum_i^k \varepsilon_i A_i, B_0 + \sum_i^k \varepsilon_i B_i) \mid \forall i.\ \varepsilon_i \in [-1; 1]\}$ of the affine forms given by generators $(A_i)_{i \leq k}$ and $(B_i)_{i \leq k}$.

Crucial for performance is to restrict the precision of the generators. We therefore explicitly round all newly computed generators and account for the round-off errors safely with additional generators. To avoid that the set of generators grows too large, generators below a given threshold are summarized in a small box.

3 Features

We implemented a tool for computing enclosures of ODEs (inspired by Bouissou *et al.* [2]), based on zonotopes and Runge-Kutta methods, with step-size adaptation: we currently use a second-order Runge-Kutta method, where the discretization error is safely overapproximated during the analysis. This requires an explicit bound for the error, which can be used to adapt the step size such that the error is within given tolerances.

We extended this approach by including splitting of reachable sets and reduction at hyperplanes perpendicular to an axis such that the flow is transversal to that hyperplane as described in our recent work [8]. These hyperplanes are what Bak [1] calls "pseudo-invariants". Our algorithm comes up with pseudo-invariants dynamically and decides after the intersection whether it is worth performing the reduction or continue with the "original" sets.

This optimization often helps to improve the precision of the analysis, however it is efficient only in low dimensions: in general an n-dimensional reachable set is reduced to an $n-1$ dimensional set.

4 Formal Guarantees

Our tool has been verified with respect to our formalization of ODEs. The kind of formal guarantees we obtain from that will be presented in the following. We consider systems with dynamics given by an autonomous ODE $\dot{x}(t) = f(x(t))$ with $f : \mathbb{R}^n \to \mathbb{R}^n$. Correctness of the algorithm is formalized with the help of two functions, `flow` and `existence_ivl`. `existence_ivl(x)` gives the maximal existence interval of a solution φ with initial condition $\varphi(0) = x$. For this initial condition, `flow(x, t)` is defined as the solution $\varphi(t)$ at time t (provided $t \in$ `existence_ivl`(x).

```
schematic_lemma vanderpol_fderiv:
  "((λ(x::real, y::real). (y, y * (1 + - x*x) + - x))
    has_derivative (case x of (x, y) ⇒ λ(dx, dy). ?D x y dx dy))
   (at x within X)"
  by (auto intro!: derivative_eq_intros)

approximate_affine vanderpol "λ(x::real, y::real). (y, y * (1 + - x*x) + - x)"
```

The core of our reachability analysis is a Runge-Kutta step `post(X, h)`, which returns a tuple `(Y, Z)`, where `Y` encloses the solution on the interval $[0; h]$ and `Z` encloses the solution at time instant h.

$$\texttt{post}(X, h) = (Y, Z) \implies$$

$$\forall x \in X.\ h \in \texttt{existence\_ivl}(x) \wedge \texttt{flow}(x, h) \in Z \wedge \forall t \in [0; h].\ \texttt{flow}(x, h) \in Y$$

Of course, `post` also needs to certify that the chosen step size h is contained in the `existence_ivl`. It does so (just like Bouissou *et al.* [2]) by proving that the Picard iteration is contracting, which in turn is certified by iterating an overapproximation of the Picard operator (for possible solutions with graph in $[0; h] \times X$ for some set X) until exhibiting a post fixed point. If no such post fixed point is found, then the system retries for a smaller time step.

5 Setup

Setting the tool up for concrete computations involves some manual interaction, which could mostly be automatized, but it shows which proof obligations need to be discharged in order to obtain a provably correct analysis.

Here we sketch the setup for the example of the van-der-Pol system $(\dot{x}, \dot{y}) = f(x, y)$ with $f(x, y) = (y, (1 - x^2)y - x)$

In order to estimate the discretization error of the Runge-Kutta method, one needs to provide derivatives. The second line is the right hand side of the van der Pol system. In the third line, there is a *schematic variable* `?D`, it can be instantiated during the proof. Here it will be instantiated with a term form the map $\mathsf{D}f\mid_x$ given by the Jacobian matrix of f at x. This task can be completed automatically using the set of rules for derivatives, `derivative_eq_intros`.

The same proofs need to be done for higher derivatives. Then affine arithmetic overapproximations can be derived as follows:
As a result, one gets a function `vanderpol` operating on zonotopes, together with a correctness theorem.

After instantiating the framework with the right-hand side f, its derivatives and its corresponding functions in affine arithmetic, code can be generated (code generation will be explained in more detail in the following section) for SML. The compiled code outputs tracing information that can be used for plotting the reachable sets (at discretization points like in Figure 1 or at the time-intervals inbetween, Figure 2).

It is also possible to compile and run the program from within Isabelle. Here we trust code generation as an "oracle". For the following example, we start the reachability analysis from the line segment $[1.25; 1.75] \times \{2.25\}$ and stop when returning to the line $y = 2.25$. The result `vanderpol_limit'` is therefore the Poincaré map and returns within an interval $[24759730456532176 \cdot 2^{-54}; 26264698040279640 \cdot 2^{-54}] \times \{2.25\}$ (`Sctn(n, c)` indicates that the result can be intersected with the hyperplane $\{x \mid \langle x, n\rangle = c\}$):

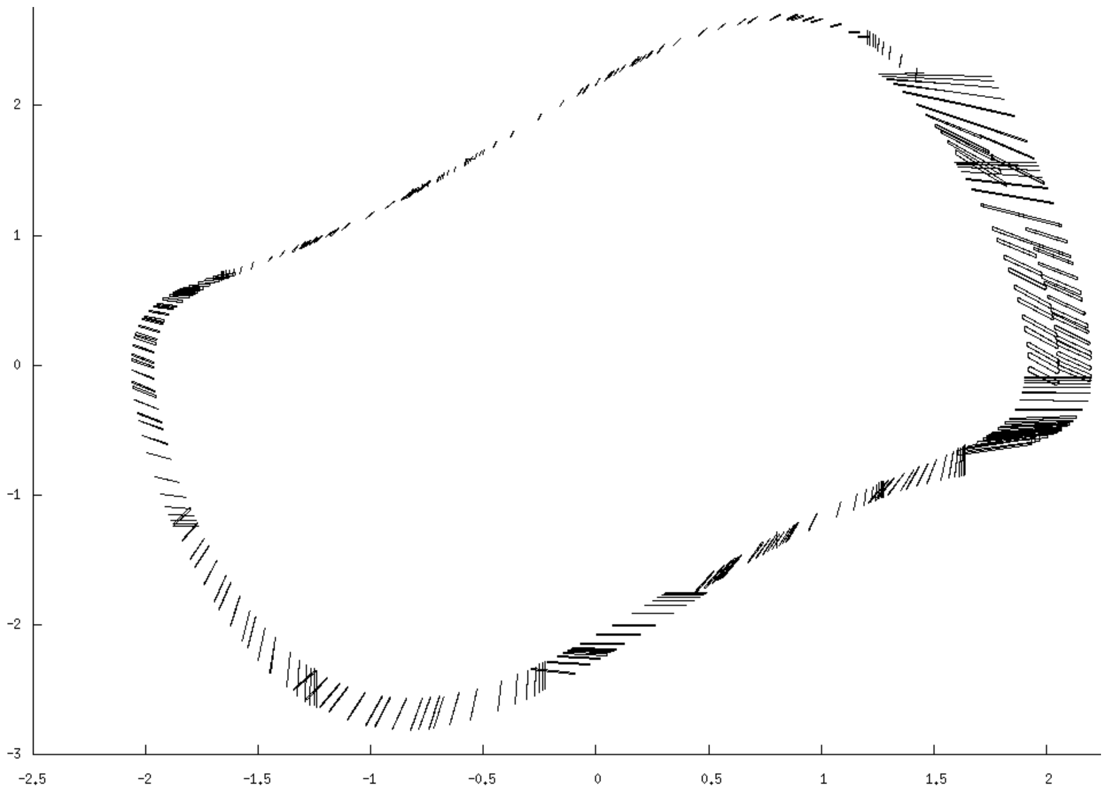

Figure 1: Reachable sets for the van der Pol system at discretization points

```
lemma  vdpl_res: "vanderpol_limit' = dRETURN
  ([(FloatR 24759730456532176 (- 54), FloatR 20266198323167228 (- 53))],
   [(FloatR 26264698040279640 (- 54), FloatR 20266198323167236 (- 53))],
   [Sctn (FloatR 0 0, FloatR (- 1) 0) (FloatR (- 9) (- 2))])"
  by eval
```

The returning interval is contained in the initial set, so after unfolding some definitions, we can prove a theorem that the inital set is forward invariant under the `flow` of the differential equation.

```
lemma vanderpol_flow_invar:
  "∃E. E ⊆ {(1.25, 2.25) .. (1.75, 2.25)} ∧
    (∀x∈{(1.25, 2.25) .. (1.75, 2.25)}.
      ∃t>0. t ∈ vanderpol.existence_ivl x ∧
            vanderpol.flow x t ∈ E)"
```

If one were to formalize more concepts from dynamical systems this result could be used to prove the existence of a stable limit cycle for the van-der-Pol system.

6 Execution

The following sections give a bit more detail into the process of code generation and gives some indications on the performance of the compiled code.

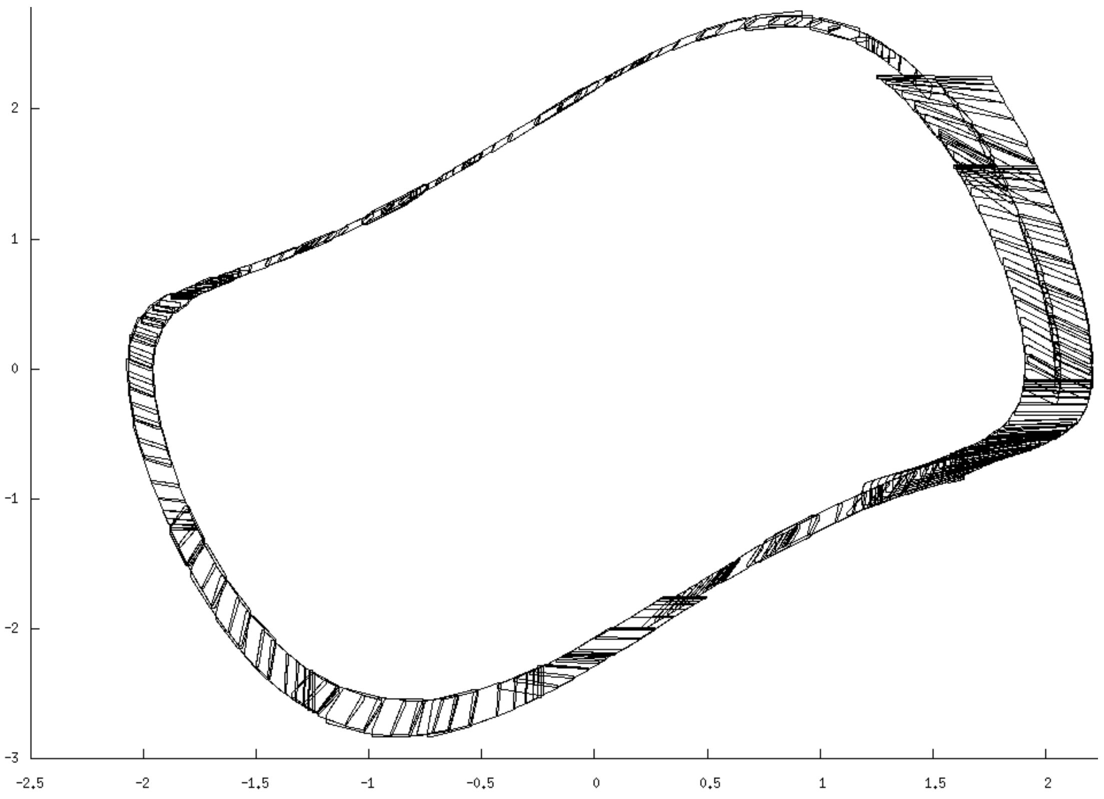

Figure 2: Reachable sets for the van der Pol system during time intervals

Data Refinement. In order to render the specified algorithms executable, data needs to be represented in an executable way. This applies in particular because our algorithms, in particular the generators of the zonotopes, are specified in terms of real numbers. We use data refinement: we replace the abstract type of real numbers with some concrete, executable representation in a sound way. The concrete representation is (arbitrary precision) floating point numbers $m2^e$ for (unbounded) integers $m, e \in \mathbb{Z}$.

To this end we introduce a constructor `Real` defined as $\texttt{Real}(m, e) = m2^e$ in the logic. Then `Real` is used as uninterpreted function in the target language. Operations on real numbers are then reduced to the respective operations on the concrete representation as floating point numbers, e.g. `Real (m1, e1) * Real (m2, e2) = Real (m1 * m2, e1 + e2)`.

The floating point numbers we use work in principle with arbitrary precision, however we introduce explicit round-off operations – the errors are therefore correctly accounted for in the verification, and there is no impact on performance because this yields essentially fixed precision computations.

Code Generation. Isabelle/HOL provides a means to translate the definitions for functions into functional programming languages like SML, Haskell, or Scala. This allows for performant execution while the translation process (which is currently to be trusted) is such that every reduction step taken by the generated code corresponds to a verified equation in the logic.

Performance Comparison. Comparison with Bouissou on his examples, suggests that our tool is one or two orders of magnitude slower, as can be seen in the work [6]. A more detailed comparison with Flow* and VNODE is given in the work [8]: for the van-der-Pol system, our tool is only a factor of two to three slower than Flow*. It is much slower than VNODE, but can handle larger initial sets. Because of the reductions performed by our tool, our tool is even more efficient than Flow* for very large initial sets.

7 Future Work

Including higher order Runge-Kutta methods would be possible by formally proving the corresponding Taylor series expansion correct. But actually one advantage of classical Runge-Kutta methods is that they do not require knowledge of higher derivatives of the ODE. An approach that would (from a verification point of view) better scale to higher orders would be Taylor series based methods with automatic differentiation (like in VNODE [10]).

Setting up concrete instances of our tool still requires a lot of manual interaction, which is where we would like to see more automation. Moreover we aim to extend the machinery developed for handling Pseudo-invariants to deal with zero-crossings of hybrid systems.

References

[1] Stanley 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.

[2] Olivier Bouissou, Alexandre Chapoutot, and Adel Djoudi. Enclosing temporal evolution of dynamical systems using numerical methods. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, volume 7871 of *LNCS*, pages 108–123. Springer, 2013.

[3] LuizHenrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004.

[4] John Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 114–129. Springer Berlin Heidelberg, 2005.

[5] Johannes Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In *Proceedings of the PLMMS'09*. ACM, 2009.

[6] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *LNCS*, pages 113–127. Springer International Publishing, 2014.

[7] Fabian Immler. A verified algorithm for geometric zonotope/hyperplane intersection. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 129–136, New York, NY, USA, 2015. ACM.

[8] Fabian Immler. Verified reachability analysis of continuous systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2015.

[9] Fabian Immler and Johannes Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 377–392. Springer, 2012.

[10] Nedialko S. Nedialkov. Implementing a rigorous ODE solver through literate programming. In Andreas Rauh and Ekaterina Auer, editors, *Modeling, Design, and Simulation of Systems with Uncertainties*, volume 3 of *Mathematical Engineering*, pages 3–19. Springer, 2011.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*. LNCS. Springer, 2002.