Reachability Analysis Using Polygonal Projections

Mark R. Greenstreet[1] and Ian Mitchell[2]

[1] Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada
mrg@cs.ubc.ca
[2] Scientific Computing and Computational Mathematics
Stanford University
Stanford, CA 94305-9025, USA
mitchell@sccm.stanford.edu

Abstract. This paper presents Coho, a reachability analysis tool for systems modeled by non-linear, ordinary differential equations. Coho represents high-dimensional objects using projections onto planes corresponding to pairs of variables. This representation is compact and allows efficient algorithms from computational geometry to be exploited while also capturing dependencies in the behaviour of related variables. Reachability is performed by integration where methods from linear programming and linear systems theory are used to bound trajectories emanating from each face of the object. This paper has two contributions: first, we describe the implementation of Coho and, second, we present analysis results obtained by using Coho on several simple models.

1 Overview

Reachability analysis is the basis for many verification tasks. This paper addresses reachability for systems modeled by ordinary differential equations (ODEs). In this context, the state of the system is a point in $\mathcal{R}^d$, where d is the dimension (i.e. number of variables) of the system. Given two regions, $A \quad B \quad \mathcal{R}^d$, the reachability problem is to show that all trajectories that start in A remain in B, either during some time interval, $[0, t_{\text{end}}]$, or for all time.

To verify a safety property, one must show that all trajectories are contained in a region satisfying the property. Examples of safety properties include: aircraft are adequately separated [TPS97], an arbiter circuit never asserts grants to both of its clients simultaneously [MG96], and the level of water in a tank is in a specified interval[ACH+95]. To verify these properties, one can determine a region that contains all possible trajectories of the system. If this region is contained in the region that satisfies the desired property, then the safety property holds for the system. This paper presents a method for constructing a region that contains all possible trajectories of a system.

F.W. Vaandrager and J.H. van Schuppen (Eds.): HSCC'99, LNCS 1569, pp. 103–116, 1999.

In this paper we describe a technique for reachability analysis of systems modeled by ODEs. Such systems present two challenges. First, closed form solutions exist only for special cases (e.g. linear models and a few others). Mathematicians have proven enough negative results for closed form solutions that it is clear that little progress can be made by strictly analytical means. Thus, we must use approximation techniques (such as numerical integration) to analyze real systems. With care, these techniques can be designed in a way that ensures the approximations always lead to an over estimation of the reachable space. Thus, our verification is sound—incorrect designs will never be falsely verified, but we may fail to verify a correct system because of our approximations.

The second challenge is that we are interested in systems with moderately high dimensionality. The circuit models that motivate our work typically have five to twenty variables. Algorithms to represent and manipulate general d-dimensional polyhedra typically have time and space complexities with exponents of d or $d/2$ [PS85]. Thus, we will consider a restricted class of high-dimensional objects that can be efficiently represented and manipulated.

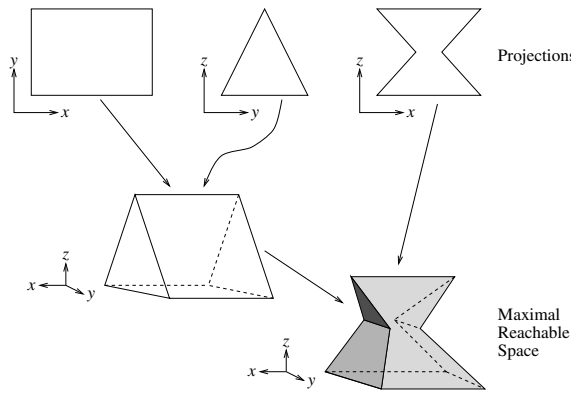


Fig. 1. A three dimensional "projectahedron"

In [GM98] we presented the theory for an approach to reachability analysis where high dimensional objects are represented by their projections onto two dimensional subspaces with each projection corresponding to a pair of variables. For example, figure 1 shows how a three-dimensional object (the "anvil") can be represented by its projection onto the xy, yz, and xz planes. The high dimensional object is the largest set of points that satisfies the constraints of each projection. We call objects that are represented by this technique "projectahedra."

Projectahedra offer several advantages. Ignoring degeneracies, faces of the object represented by a projectahedron correspond to edges of its projection polygons. Our reachability analysis requires flows from each face to be consid-

ered, and the projectahedron representation allows many operations on faces to be carried out as simple operations on polygon edges. The projection polygons are two-dimensional and can be manipulated efficiently using well-known algorithms from computational geometry [PS85].

We do not require the projection polygons to be convex; thus, non-convex, high-dimensional objects can be represented by projectahedra. As is shown in section 4, the reachable regions arising from ODE models are often highly non-convex. However, even non-convex projections cannot represent all possible high-dimensional polyhedra. For example, projectahedra cannot represent objects with indentations on their faces (i.e. a cube where some or all of the faces have hemispherical concavities). Instead, such objects will be mapped to projectahedra where their indentations are filled-in. Since we are only verifying safety properties, the resulting over approximation of the reachable space does not compromise the soundness of our analysis.

Given an object represented by a projectahedron, we must determine how this object evolves according to the ODE model for the system. As this problem cannot be solved analytically for general models, we pursue a numerical approach. We require the ODE model to have bounded derivatives. This means that trajectories are continuous, and it is sufficient to consider the set of points reachable from points on the boundary of the projectahedron. Our algorithm approximates the non-linear model with a linear model and an error bound. By constructing a separate approximation for each face at each time step, the error bounds can be fairly tight. The face is then transformed by the linear approximation and moved outward by the worst-case error. This approach allows general models to be used. Currently, linearization is done manually, and model generation requires significant effort.

This paper makes two contributions to the verification of systems with ODE models. Section 3 describes Coho, our implementation of the techniques mentioned above. For brevity, we focus on the top-level structure of the implementation and describe a few of the "surprises" we encountered. Then section 4 presents several examples where we have used Coho. We start with the linear two- and three-dimensional models presented in [DM98]. We then analyze two systems with non-linear models: a Van der Pol oscillator, and a three-dimensional "play-doh$^{\mathrm{TM}}$" example, where the region is compressed in one dimension while being stretched and folded along another.

2 Computing Reachability

Coho is based on the techniques for reachability presented in [GM98]. This section presents a brief summary of this approach. Consider a general, ODE model:

$$\dot{x} = f(x) \tag{1}$$

where $x \in \mathbb{R}^d$. Given $A_0 \quad \mathbb{R}^d$ such that $x_0 \in A_0$, we are interested in answering reachability questions such as: Given $t \in \mathbb{R}^+$, find $A_t \quad \mathbb{R}^d$ such that $x(t) \in A_t$. For general ODE models, closed form solutions are not possible. Therefore, we do

not hope to compute the smallest reachable set. Instead, we want to compute a conservative projectahedron for A_t. Our reachability computation is an iterative, integration algorithm. We describe a single time-step of this algorithm below.

Each edge of a projection polygon corresponds to a $d \otimes 1$ dimensional face of the projectahedron. At each step, we compute a conservative estimate of the convex hull of this face, and then determine a new convex region that contains any point reachable from this hull at the end of the time step. We then project this hull back onto the basis for the projection polygon corresponding to the face. This computation is performed for each edge of the projection polygon to compute a bounding projection polygon at the end of the time step. By updating each projection polygon in this manner, we obtain a projectahedron that contains all points reachable at the end of the time step.

To compute a conservative approximation of the convex hull of a face, we intersect the constraints for the polygon edge with the constraints for the convex hulls of each of the projection polygons. To compute the set of points that are reachable from this hull, we approximate the model from equation 1 with the differential inclusion:

$$x \in H \Rightarrow \dot{x} \in Ax + b + U \tag{2}$$

where H is the conservative approximation of the convex hull for the face, $A \in \mathbb{R}^d \quad \mathbb{R}^d$ is a matrix, $b \in \mathbb{R}^d$ is a vector, and $U \in (\mathbb{R} \quad \mathbb{R})^d$ is a hypercube (i.e., the Cartesian product of d intervals). For the examples presented in this paper, we compute A, b, and U by performing a power-series expansion about a point near the center of H. Because H is convex, it can be represented by a linear program, and we can use linear optimization techniques to obtain fairly accurate approximations for f.

We now consider the inhomogeneous linear system

$$\dot{x} = Ax + b + u(t) \tag{3}$$

where u is any function such that $u(t) \in U$. By chosing a worst-case u, we obtain a conservative approximation of the reachable region. In our implementation, we approximate u with a linear bounds. This produces a linear program for the approximation of the points reachable from the face, and we obtain a projection of the face back to the coordinates of the projection polygon from this linear program.

3 Implementation

At the top-level, Coho is divided into one component that performs numeric computations and another that performs geometric operations. As shown in figure 2, at each time step the numeric component inputs a projectahedron, updates each face, and outputs a new projectahedron. In this process, each edge of each projection polygon is transformed to a polygon that contains the projection of the corresponding face at the end of the time step. The geometric component merges

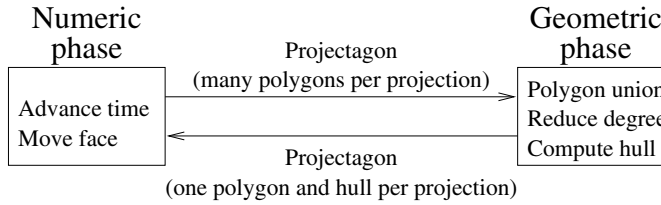


Fig. 2. Top-level of Coho

the polygons associated with a single projection, and computes an approximation with fewer vertices. The geometric component outputs the resulting polygon and its convex hull. This completes a single step of the integration. Details of the algorithm are presented in [GM98].

3.1 Matlab and Java

Currently, Coho implements the numeric component in Matlab [The92] and the geometric component in Java [AG96]. This approach builds on the strengths of both environments. Matlab provides comprehensive, optimized and well-tested implementations of linear programming, matrix exponentiation, and other matrix operations—greatly simplifying the implementation of the integrator. Furthermore, the interactive interface and plotting capabilities facilitated the analysis and visualization of the systems described in this paper.

Although Matlab currently provides a few simple geometric operations, its capabilities are not sufficient for our purposes. Thus, we implemented a computational geometry package in Java. With its type safety, garbage collection, and object oriented abstractions, Java is much better suited for implementing non-numeric algorithms than Matlab's scripting language. These features also gave Java a clear advantage over C or C++: the geometry package was developed without the use of or need for a debugger; our experience suggests that this would not have been possible in C or C++.

The geometric operations were encapsulated as a filter, taking one type of projectahedron and returning a different format of a similar projectahedron (see figure 2). This filter is a Matlab script that writes its parameter projectahedron to a file, invokes the Java program as a shell command, and reads the result from a second file.

We were initially concerned that the time to start the Java Virtual Machine (JVM) would be prohibitive. In practice, starting the JVM each time step takes about as long as the numeric computations for the time step. The time spent performing geometric operations is small by comparison. We believe that the cost of the numeric operations is dominated by the time spent solving the large number of linear optimization problems that occur in our formulation. An implementation optimized for our application would almost certainly result in much improved performance. Likewise, with some programming effort we could change

the interface between Matlab and Java so that the JVM would be started only once. For the examples described in this paper, the total elapsed time was a few seconds per time step running on a 250 MHz UltraSparc 2 workstation with Matlab 5.1 and JDK 1.2-beta4.

3.2 Numerical computation

The numeric phase of a time step begins by loading a polygon and its convex hull for each projection of the system. The convex hulls are then bloated outward slightly for safety. Each projection's bloated convex hull can be translated into a set of linear inequalities in the projection's two coordinates. The combination of all the projections' linear inequalities describes a convex region containing the projectahedron.

At this point, the movement of each edge of each projection's polygon can be computed independently. Each edge represents a face of the projectahedron, and the objective is to compute the furthest outward that points on the face could move during a time step. For each face, the following computations occur.

Restriction: The convex region computed from the convex hulls is further restricted to a box around the edge in the coordinates of the edge by four more linear inequalities. In the full dimensional space this is equivalent to constructing a slab around the face being examined.

Linearize Model: A user supplied function computes a linearization of the system derivatives which is valid over the slab. This model includes linear and constant terms, and must give bounds on the error introduced by the linearization within the slab. The user function has access to the slab's description in terms of the collection of linear inequalities computed in the previous step. Typically, linear programs are run to find bounds on each variable within the slab, from which the linearization and errors computed.

Advance Time: The linear model is used to move the slab forward in time—currently by matrix exponential, although future versions may use better integration routines.

Map Extent: The slab's end of step shape will still be described by a collection of linear inequalities after time is advanced. Building a polygon from these inequalities requires mapping out the region they contain. The mapping is accomplished by running a series of linear programs on the time advanced set of inequalities. Note that the slab may rotate during the time step, so its projection may not be a simple rectangle.

Add Errors: So far, the slab's movement is entirely controlled by the linearized model. To treat the error, we add a constant derivative offset within the error bounds throughout the time step, in such a way as to bloat the slab's projection outward as much as possible.

Each edge of each projection's polygon therefore produces an "edge polygon" at the end of the time step; this polygon contains the projection of all points that could be reached from the corresponding face within the time step. The

union of all such polygons, and the region contained within that union, is the projection of an over approximation of the projectahedron at the end of the time step. Since the next time step must start with a single polygon for each projection, the geometric filter is called at this point to simplify the projectahedron 's description.

3.3 Geometric computation

The input to the geometric phase is a list of edge polygons for each projection: the union of these polygons contains the boundary of the new projection. The resulting polygon may have many more edges than the polygon at the beginning of the time step. To avoid unbounded growth in the number of polygon edges, we conservatively reduce the vertex count. Finally, as projectahedra evolve, degeneracies may occur in the projection polygons: edges may become very short, or vertex angles may become highly acute or highly obtuse. In fact, these degenaracies occur frequently when the projectahedron becomes very narrow along one or more axes, which is typical when approaching an attractor or similar phase space feature.

The five steps of the geometric phase are described below. The constant is used to test for potential numerical degeneracies—the current implementation uses $= 10^{\otimes 12}$.

Short edge removal: If the length of an edge is less than times the distance from the origin of the endpoint furthest from the origin, then one of the vertices is deleted. If only one vertex remains at the end of this step, it is replaced by a square with edges of length 2 . If exactly two vertices remain, the segment is replaced with a bounding rectangle whose major axis is parallel to the segment and that encloses the segment by .

Special case for highly acute vertices: Each edge polygons is convex. If a vertex of one of these polygons is highly acute (angle less than radians), then the edge polygon must be very thin. Such polygons are replaced by a bounding rectangle whose major axis is parallel to the bisector of the angle and that encloses the original edge polygon by .

Polygon merge: For each projection, the edge polygons are merged to produce the boundary of the new projection polygon.

Topological simplification: Having computed the boundary in the previous step, we now find the left most vertex, which must be on the outer boundary. An edge tour starting at this vertex gives the outer boundary of the projection's new polygon. This operation "fills-in" the interior of the projection polygon.

Vertex count reduction: Typically, the polygon produced by the preceding steps of the geometric phase will have many more vertices than the projection polygon had at the beginning of the time step. To prevent an explosion in the number of vertices, we must compute a conservative approximation of the polygon that has a reasonable vertex count.

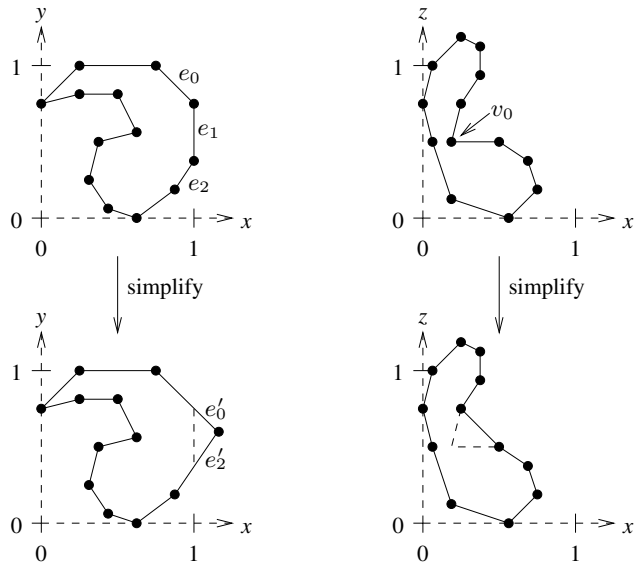


Fig. 3. Vertex count reduction

We say that a vertex is convex (resp. concave) if the polygon is locally convex (resp. concave) at that vertex. Consider a pair of adjacent convex vertices (such as the two rightmost vertices of the xy projection in the top half of figure 3). Let e_1 be the edge joining these two vertices and e_0 and e_2 be the other two edges incident on these vertices. If e_0 and e_2 intersect on the outside of e_1, then the two vertices can be replaced by this intersection. The resulting polygon contains all points in the original polygon; thus, this simplification is safe. Likewise, any concave vertex can be safely eliminated (as in deletion of v_0 from the xz polygon in figure 3).

The selection of vertices to remove is done in a greedy manner. All operations increase the size of the polygon, and each has a cost which is a weighted sum of the increase in the area of the projection polygon and the increase in area of its hull. Currently, the two weights are equal. Vertices are deleted until the total cost reaches a preset fraction of the original polygon area. In the examples below, we used a threshold of 2%—the resulting polygons typically had less than fifteen vertices.

The removal of concave vertices can create short edges or highly obtuse vertices (angles within of). Such degeneracies are eliminated when they occur by deleting appropriate vertices.

3.4 Surprises

Of course, not all went as expected when we first used Coho. Originally, we only used the area of the polygon in computing the cost of deleting a vertex. However,

projection polygons are approximated by their convex hulls in many places in our algorithm; thus, an approximation that enlarges the convex hull is in some sense more costly than one that does not. We found that by including the area of the convex hull in our cost function, we obtained tighter bounds with our reachability analysis.

A second surprise was the difficulty caused by infeasible vertices. Recall that at the beginning of each time step, each edge of each projection polygon corresponds to a face of the projectahedron. The numerical phase of the algorithm computes a convex bound for this face, moves it forward in time, and project it back to the basis plane to produce an edge polygon. The edge polygons for adjacent edges should overlap, and this is guaranteed if the vertex where the edges met was feasible at the beginning of the time step. We discovered that the over approximations used in our algorithm can produce infeasible vertices. However, the extent of the over estimate is not necessarily the same for all projections. This can produce sets of edge polygons that fail to form a closed boundary.

For example, consider the projectahedron as depicted in figure 3. Assume that both polygons have an extent of $[0, 1]$ in x before the vertex reduction step. The vertex elimination operation in the geometric component replaces the two rightmost vertices of the xy projection with a single vertex. This gives the xy projection polygon an x extent of $[0, 1.1]$. Vertex elimination for the xz polygon eliminates a single vertex along the concave section of the boundary, leaving the extent of the polygon unchanged. After vertex elimination, the rightmost vertex of the xy polygon is infeasible (because no point with that x value lies in the xz polygon).

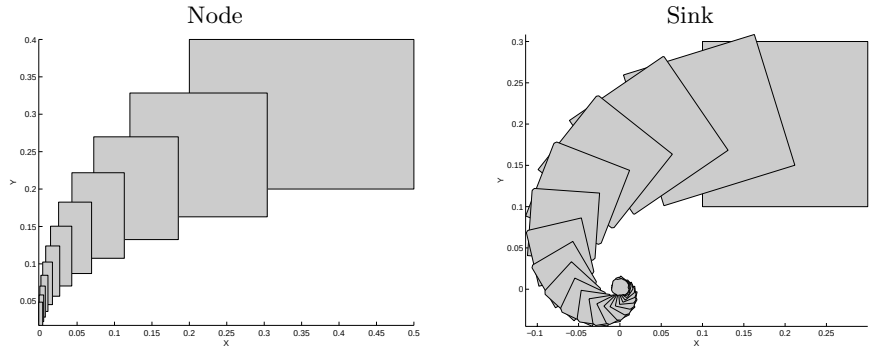
Infeasible vertices led to incomplete boundaries in some of our earlier trials. Our solution is to detect infeasible vertices (using linear programming) and to treat a sequence of adjacent edges as a single piece of the boundary, extending the sequence until both endpoints are feasible. This procedure guarantees that the numeric component will produce a complete boundary that contains the true boundary at each time step. In the example above, edges e'_0 and e'_2 would be treated as a single triangle instead of two separate edges in the next numeric phase of the analysis.

4 Examples

This section presents our initial experience applying Coho to examples from the literature as well as some of our own design.

4.1 Dang and Maler's linear examples

In [DM98], Dang and Maler analyzed five systems with linear models and two with non-linear models. Here, we use Coho to analyze their five linear systems. Four of these are two-dimensional models from [HS74]. Figure 4 show the models and Coho's analysis for two of those examples, the node and sink. We also ran Coho on their center and saddle examples with similar results. When our



$$\dot{x} = -5x \qquad \dot{x} = -2x - 3y$$
$$\dot{y} = -2y \qquad \dot{y} = 3x - 2y$$
$$x(0) \in [0.2, 0.5] \qquad x(0) \in [0.1, 0.3]$$
$$y(0) \in [0.2, 0.4] \qquad y(0) \in [0.1, 0.3]$$

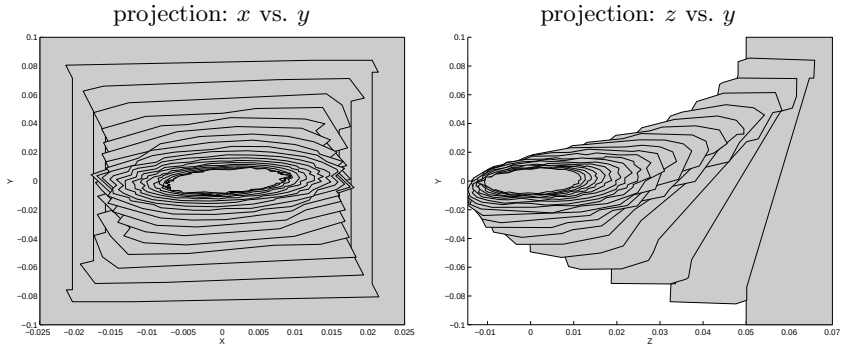
Fig. 4. Two-dimensional, linear models

analysis is compared with the "face-lifting" technique of Dang and Maler, Coho appears to be much more accurate. This can be seen in the node example where the boundaries of our polygons approach the origin without touching the axes, as should be the case for that model. With face-lifting, a much larger area is computed for the node, and it has extensive contact with the axes. Our sink analysis is likewise more accurate, clearly showing distinct cycles of the spiral where face-lifting merges them together.

The greater accuracy of Coho arises from several factors. First, Coho's "approximate" linearizations of the models are exact (to within the accuracy of double precision floating point arithmetic) for these linear examples; the error bound for these models is zero. Second, face-lifting quantizes the reachable region on a relatively coarse fixed grid, while Coho can place polygon vertices at any location representable in double precision. On the other hand, the fixed quantization of face-lifting may make it more amenable for use with symbolic techniques such as timed automata. This does not appear practical for Coho's polygonal projections.

Figure 5 shows Dang and Maler's 3-dimensional example. We have not yet implemented reconstruction of 3-dimensional objects from their projections, so the figure just shows the two projections. The figure in [DM98] does not provide enough detail to support a comparison of accuracy. However, our figure clearly shows how Coho automatically increases the number of vertices in the projection polygons to maintain the requested accuracy. The evolution from rectangle to elliptical "blobs" shows how Coho adjusts the orientation of edges according to its linearization of the ODE model.

Two non-linear models were presented in [DM98]. At the present time, we must manually derive code to linearize a non-linear model and compute the error

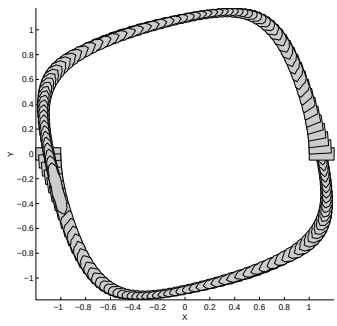


$$\dot{x} = -2x \qquad x(0) \in [-0.025, 0.025]$$
$$\dot{y} = x - 2y \qquad y(0) \in [-0.1, 0.1]$$
$$\dot{z} = y - 2z \qquad z(0) \in [0.05, 0.07]$$

Fig. 5. Dang and Maler's 3-dimensional Model

bounds. This derivation is the most tedious and error-prone aspect of using Coho, and we are looking into ways to automate it. However, this, and the very recent completion of Coho, are the reasons that we have not yet analyzed Dang and Maler's non-linear examples. Instead, we present two examples of our own design that exhibit behaviors that are qualitatively different than those of linear models.

4.2 Van der Pol's oscillator



$$\dot{x} = -y + x - x^3$$
$$\dot{y} = x + y - y^3$$
$$x(0) \in [1.0, 1.2] \cup [-1.2, -1.0]$$
$$y(0) \in [-0.05, 0.05]$$

Fig. 6. Van der Pol's oscillator

Our first non-linear example is a Van der Pol oscillator adopted from [HS74]. Our model is symmetric in x and y; the equations and a cycle of the oscillator are shown in figure 6. This system also shows how invariants can be verified

using reachability analysis: to establish an invariant set, it is sufficient to choose a region Q_0 and integrate for one period of the oscillator to produce Q_1. If Q_1 is contained in Q_0, then the region traced out during the integration is an invariant set.

During our first attempts at this example, the reachable region quickly became very long and skinny, stretching along the trajectory of the oscillation. This stretching occurs because the non-linear terms in the ODE stabilize the amplitude but not the phase of trajectories. Recall that Coho uses error bounds from a model's linear approximation to bloat edges outward—a conservative strategy to maintain the soundness of our analysis. While over estimation of the reachable region's amplitude is damped by the non-linear terms of the oscillator, over estimation of the region's phase tends to accumulate, and the region gets longer and longer.

On the straight segments of the oscillator's trajectory, this stretching causes little harm. However, as the oscillator makes a sharp turn at the corners, the over estimated phase can spill into large over estimates of amplitude. If the amplitude isn't damped before the next corner, the region grows without bound.

To prevent this explosion in region size, we divide a complete oscillation into two portions. The starting region lies on the positive x-axis, and we track this region through half a cycle until it has completely crossed the negative x-axis. We manually identify the segment of the negative x-axis where any portion of the region crossed, and restart from this segment. This second region is tracked around to its crossing of the positive x-axis. Figure 6 clearly shows that all trajectories starting in the original region cross through the second starting region, and all trajectories from the second starting region cross through the original starting region. Thus, we have identified an invariant set.

This technique can be extended to allow verification of a hybrid system whose dynamics depend on a current discrete "mode". As the region being tracked crosses a boundary between two modes, we can record the portion of the boundary touched by the region. Once we have finished tracking the region in the old mode—most likely because it has moved completely into the new mode—the analysis is restarted using the new mode's continuous dynamics. The initial conditions for this restart are those portions of the boundary crossed by the region in the old mode. In the computation of the Van der Pol oscillator's invariant set, for example, different ODEs could have been used for the top and bottom halves of the state space, simulating a system with one mode for positive y and another for negative y. It should be noted that this strategy would have problems dealing with a region which straddled or jittered along the boundary between two modes.

4.3 Squishing "Play-Doh"

Our example of a three dimensional non-linear system corresponds to squishing a lump of modeling clay. Consider a region shaped roughly like an octagonal hockey puck (or tuna can, for those unfamiliar with the Canadian obsession). Orient the puck so that its projection in the x-y plane is the octagon, and in the

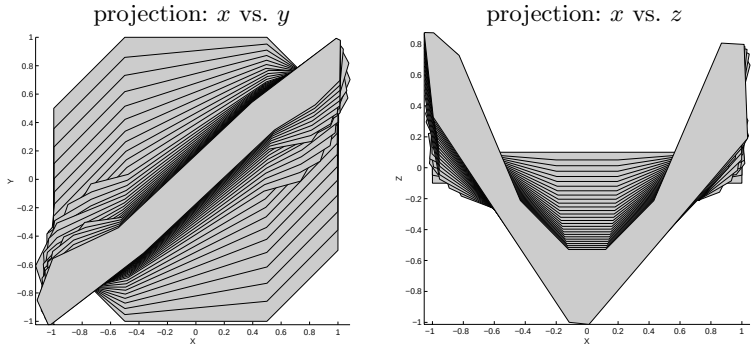


Fig. 7. Play-Doh

x-z plane a rectangle. The playdoh system takes this puck and squishes (in the y direction) toward the $y = x$ line, while bending the puck in its narrow dimension (the z direction) into a "V" shape. A small shrinking factor is applied in the x direction to offset Coho's over approximations. The resulting region is a thick, bent V lying at a 45 degree angle. While the squishing process is linear, bending is accomplished by a quadratic non-linear derivative function.

Although encountering many infeasible vertices and requiring many vertices introduced during numeric phases to be removed, Coho manages to track the two projections without an explosion in vertex count. Note, however, that the regions do not stay symmetric; a result caused by degeneracy handling code in the geometric phase.

5 Conclusions

This paper has presented Coho, a reachability analysis tool for systems modeled by ordinary differential equations. Coho uses projectahedra, an efficient method for representing high-dimensional objects as their projections onto two-dimensional subspaces. Non-linear models are handled by creating local linearizations for each face of the projectahedron. Each linear approximation includes an error bound which ensures the soundness of the analysis. We described the implementation of Coho, and have presented several example analyses, including linear and non-linear systems in two and three dimensions.

Implementation of Coho was completed recently; clearly, more examples will be needed to thoroughly validate our approach. The initial results presented in this paper are encouraging. The reachable state space estimates computed by Coho are more accurate than published results by other methods. The increased accuracy can be attributed to Coho's use of exact methods for analyzing linear

systems combined with a flexible representation of reachable space that does not require vertices to lie on fixed grid-points.

Our set of examples has at least one obvious limitation—in all cases, system derivatives are completely determined by current state. In many real systems, input and modeling uncertainties lead to models where only constraints on the derivatives, but not exact values, can be determined. In [Gre96], we described several such models based on Brockett's annulus construction [Bro89]. We intend to try Coho on similar models. The examples in this paper were based on two and three dimensional systems. Coho was designed with models of up to twenty variables in mind. We are eager to try Coho on higher dimensional models.

References

ACH+95. R. Alur, C. Courcoubetis, N. Halbwachs, et al. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

AG96. Kenneth Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

Bro89. R. W. Brockett. Smooth dynamical systems which realize arithmetical and logical operations. In Hendrik Nijmeijer and Johannes M. Schumacher, editors, *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, volume 135 of *Lecture Notes in Control and Information Sciences*, pages 19–30. Springer, 1989.

DM98. Thao Dang and Oded Maler. Reachability analysis via face lifting. In Thomas A. Henzinger and Shankar Sastry, editors, *Proceding of the First International Workshop on Hybrid Systems: Computation and Control*, pages 96–109, Berkeley, California, April 1998.

GM98. Mark R. Greenstreet and Ian Mitchell. Integrating projections. In Thomas A. Henzinger and Shankar Sastry, editors, *Proceding of the First International Workshop on Hybrid Systems: Computation and Control*, pages 159–174, Berkeley, California, April 1998.

Gre96. Mark R. Greenstreet. Verifying safety properties of differential equations. In *Proceedings of the 1996 Conference on Computer Aided Verification*, pages 277–287, New Brunswick, NJ, July 1996.

HS74. Morris W. Hirsch and Stephen Smale. *Differential Equations, Dynamical Systems, and Linear Algebra*. Academic Press, San Diego, CA, 1974.

MG96. Ian Mitchell and Mark Greenstreet. Proving Newtonian arbiters correct, almost surely. In *Proceedings of the Third Workshop on Designing Correct Circuits*, Båstad, Sweden, September 1996.

PS85. Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.

The92. The Mathworks Inc., Natick, Mass. *Matlab: High-Performance Numeric Computation and Visualization Software*, 1992. `http://www.matlab.com`.

TPS97. Claire Tomlin, George Pappas, and Shankar Sastry. Conflict resolution for air traffic management: A case study in multi-agent hybrid systems. Technical Report UCB/ERL M97/33, Electronics Research Laboratory, University of California, Berkeley, 1997. to appear in IEEE Transactions on Automatic Control.