# Effective Computation for Nonlinear Systems

Pieter Collins*

Centrum voor Wiskunde en Informatica,
Postbus 94079,
1090 GB Amsterdam,
The Netherlands
Pieter.Collins@cwi.nl

**Abstract.** Nonlinear dynamical and control systems are an important source of applications for theories of computation over the the real numbers, since these systems are usually to complicated to study analytically, but may be extremely sensitive to numerical error. Further, computer-assisted proofs and verification problems require a rigorous treatment of numerical errors. In this paper we will describe how to provide a semantics for effective computations on sets and maps and show how these operations have been implemented in the tool ARIADNE for the analysis, design and verification of nonlinear and hybrid systems.

**Keywords:** computable analysis, nonlinear systems, Ariadne.

## 1 Introduction

A simple but important problem in nonlinear systems theory is that of *safety verification*. Given an autonomous system, $x_{n+1} = f(x_n)$ or $\dot{x} = f(x)$, we wish to determine whether the evolution of the system starting in an initial set $X_0$ remains in some safe set $S$. To solve such a problem we need a method which can rigorously compute the image of a set or integrate a differential equation over a set of points. Further, we want our algorithm to be efficient enough to be practically useful, and optimal, in the sense that if it is theoretically possible to decide safety, then the algorithm returns the correct answer. For hybrid systems (see [1]), which combine continuous-time and discrete-time evolution, the problem is even more challenging.

There has been much work recently on *computable analysis* [2,3], which provides a theory of computation on objects from geometry and analysis such as sets and maps, and is essential equivalent to approaches based on Scott domains [4]. This theory can be applied to discuss the computability of the *reachable set* [5], which plays a crucial role in safety verification.

There are many algorithms for rigorously integrating differential equations, dating back to work of Moore [6] from the early days of interval analysis, and.

Many tools have been developed which are capable of computing or approximating reachable sets. The package GAIO [7] may be used to compute the

* This work was partially supported by the Nederlandse Wetenschappelijk Organisatie (NWO) through VIDI project number 639.032.408.

S.B. Cooper, B. Löwe, and A. Sorbi (Eds.): CiE 2007, LNCS 4497, pp. 169–178, 2007.

image of a set, but does not provide a rigorous method for integrating differential equations. The Lohner method for rigorous integration is available in the CAPD Library [8], and higher-order Taylor methods are implemented in VNODE [9]. Other tools are available for reachability analysis of hybrid systems, but only Checkmate [10] can handle nonlinear dynamics. However, there is still a need for a general-purpose open-source tool which can solve a wide variety of problems in nonlinear dynamic systems.

The goal of the computational framework ARIADNE [11] is to provide a syntax, semantics and implementation of fundamental operations from geometry and analysis, guided by formal computability theory. In this paper, we describe how the numerical kernel of ARIADNE, written in C++, implements computation with sets and maps, in a way which can be efficiently implemented and is powerful enough to solve the safety verification problem.

## 2 Reachability and Safety Computation

Consider a discrete-time system with state-space $X$ described by the continuous map $f : X \to X$, a set of initial states $X_0$ and a set of safe states $S$. The *safety verification* problem is decide whether every orbit of $f$ starting in $X_0$ remains in $S$. We can express the safety verification problem in terms of the *reachable set* as:

$$\mathrm{reach}(f, X_0) \subset S, \text{ where } \mathrm{reach}(f, X_0) := \textstyle\bigcup_{n=0}^{\infty} f^n(X_0).$$

To even consider how to solve this problem, we first need to have a description of the sets $S$ and $X_0$ and the map $f$. Since the set of continuous functions on $\mathbb{R}^n$ has continuum cardinality, we cannot represent all functions exactly. Instead, following Weihrauch [2], we describe elements of an infinite set $Y$ by a *representation* $\delta :\subset \Sigma^\omega \to Y$, which encodes $y \in Y$ by a sequences over some alphabet $\Sigma$. To be useful, we must be able to obtain approximations to $y$ from a initial part of some $p \in \delta^{-1}(y)$. Given representations $\delta_0, \ldots, \delta_k$ of sets $Y_0, \ldots, Y_k$, we say a function $f : Y_1 \times \cdots \times Y_k \to Y_0$ is *computable* if there is a Turing-computable function $\mathcal{M} :\subset \Sigma^\omega \times \cdots \times \Sigma^\omega \to \Sigma^\omega$ such that $\delta_0(\mathcal{M}(p_1, \ldots, p_k)) = f(\delta_1(p_1), \ldots, \delta_k(p_k))$.

If $(Y, \tau)$ is a topological space with countable sub-base $\sigma$ labelled by a partial surjective function $\nu :\subset \Sigma^* \to \sigma$, then the *standard representation* $\delta$ of $(Y, \tau, \sigma, \nu)$ is defined by

$$\delta(\langle w_1, w_2, \ldots \rangle) = y \iff \{\nu(w_i) \mid i \in \mathbb{N}\} = \{J \in \sigma \mid y \in J\}.$$

Informally, we say that $\delta$ encodes a list of all $J \in \sigma$ such that $y \in J$.

In this paper, we fix a state space $X$ (such as $\mathbb{R}^n$) with countable base $\beta$. As well as the set of points of $X$, the open sets $\mathcal{O}$, closed sets $\mathcal{A}$, compact sets $\mathcal{K}$, and continuous self-maps $\mathcal{C}$ of $X$ all have natural topologies giving rise to the following standard representations:

- The representation $\rho$ of $X$ encodes $\{J \in \beta \mid x \in J\}$.
- The lower representation $\theta_<$ of $\mathcal{O}$ encodes $\{I \in \beta \mid \overline{I} \subset U\}$.

- The lower representation $\psi_<$ of $\mathcal{A}$ encodes $\{J \in \beta \mid J \cap A \neq \emptyset\}$.
- The upper representation $\psi_>$ of $\mathcal{A}$ encodes $\{I \in \beta \mid \overline{I} \cap A = \emptyset\}$.
- The upper representation $\kappa_>$ of $\mathcal{K}$ encodes $\{(J_1, \ldots, J_k) \in \beta^* \mid C \subset \bigcup_{i=1}^k J_i\}$.
- The compact-open representation $\delta$ of $\mathcal{C}$ encodes $\{(I, J) \in \beta \times \beta \mid \overline{I} \subset f^{-1}(J)\}$.

In [5], we show that the optimal $\kappa_>$-semicomputable over-approximation to $\mathrm{reach}(f, X_0)$ is the *chain-reachable set*

$$\mathrm{chain\,reach}(f, X_0) := \bigcap \{C \in \mathcal{K} \mid X_0 \cup f(C) \subset C^\circ\},$$

which may be much larger than the reachable set. This means that it is impossible to prove safety (using only the approximate information about $f$, $X_0$ and $S$ given by the standard representations) if $\mathrm{chain\,reach}(f, X_0) \not\subset S$, even if $\mathrm{reach}(f, X_0) \subset S$. Similarly, it is only possible to disprove safety if $\mathrm{reach}(f, X_0) \not\subset \overline{S}$. Hence the best possible solution to the safety verification problem (in an approximative setting) is an algorithm which computes:

$$\mathrm{verify}(f, X_0, S) := \begin{cases} \top & \text{if } \mathrm{chain\,reach}(f, X_0) \subset S; \\ \bot & \text{if } \mathrm{reach}(f, X_0) \not\subset \overline{S}; \\ \uparrow & \text{otherwise.} \end{cases}$$

## 3 Representations as Interfaces

The elements of a countable (i.e. discrete) set such as the integers or rational numbers may be described exactly by finite data, and correspond to *concrete types*. It is straightforward to implement types `Integer` and `Rational` which implement integer and rational numbers, such that arithmetic and comparison operators are computable.

The elements of an uncountable set such as the real numbers cannot be described by a finite amount of data. However, we shall see that they can be adequately described by *abstract interfaces* with properties reflecting the standard representation.

Throughout the paper, we use typeface `x` to denote data types and $x$ to denote the corresponding mathematical object.

### 3.1 The Standard Representation of Points

Recall that we can describe an element of a Hausdorff space with countable base $\beta$ using the standard representation, which encodes a point $x$ by listing its basic open neighbourhoods. We could describe the standard representation by a method `neighbourhood` taking an integer argument and returning an element of $\beta$ implemented by a `BasicSet` class:

```
virtual BasicSet Point::neighbourhood(Integer i);
```

In practice, it is more useful to be able to determine whether $x$ lies in some element $I$ of $\beta$. Unfortunately, the standard representation $\rho$ merely yields a semi-decision algorithm for the predicate $x \in I$. If $x \notin \overline{I}$, we can indeed deduce that $x \notin \overline{I}$, since there exists $J \in \beta$ such that $x \in J$ and $I \cap J = \emptyset$. But if $x \in \partial I$, then no such $J$ exists, so we are unable to show $x \notin I$. Hence, any implementation of a method `bool Point::in(BasicSet)` using only approximate information about $x$ will fail to terminate on `x.in(I)` if $x \in \partial I$.

In order to obtain a function which always terminates, we give a *precision* argument $p$, and return an *"indeterminate"* value if we cannot decide $x \in I$ or $x \notin I$ to precision $p$. We therefore have a method

```
virtual tribool Point::in(BasicSet I, Integer p);
```

where `tribool` is the enumerated type {`true`,`false`,`indeterminate`}.

We now give conditions under which the `in` method specifies a point in $X$ uniquely. For consistency, we clearly require,

**(C)** If `x.in(I,p)==true` and `x.in(J,r)==false`, then $I \not\subset J$.

Define $\eta = \{J \in \beta \mid \exists p \in \mathbb{N} \text{ s.t. } \texttt{x.in(J,p)==true}\}$. Then $\eta$ must satisfy the following intersection, refinement and approximation properties:

**(I)** $I_1, I_2 \in \eta \implies I_1 \cap I_2 \neq \emptyset$.
**(R)** $I \in \eta$ and $I \subset J_1 \cup \cdots \cup J_k \implies \exists i \in \{1, \ldots, k\}$ such that $J_i \in \eta$.
**(A)** $J \in \eta \implies \exists I \in \eta$ such that $\overline{I} \subset J$.

Further, if $\eta$ is any set satisfying these properties, then there exists $x$ such that $\eta = \{J \in \beta \mid x \in J\}$.

To simplify code using the `in` method, we can assume that the method is implemented such that the following precision and monotonicity properties are satisfied:

**(P)** If `r>p` and `x.in(I,p)!=indeterminate`, then `x.in(I,r)==x.in(I,p)`.
**(M)** If $I \subset J$, then `x.in(I,p)` $\implies$ `x.in(J,p)`, and `!x.in(J,p)` $\implies$ `!x.in(I,p)`.

### 3.2 Representations of Sets

A closed set is uniquely specified by the basic open sets it intersects, or by the basic closed sets it is disjoint from. These specifications give rise to the *lower* and *upper* representations $\psi_<$ and $\psi_>$, respectively. Since it cannot be true that both $A \cap I \neq \emptyset$ and $A \cap \overline{I} = \emptyset$, we can specify an interface for closed sets by the single method

```
virtual tribool ClosedSet::disjoint(BasicSet I, Integer p);
```

where $p$ is the precision argument. Using the method `disjoint`, we can compute

$$\alpha_< = \{I \in \beta \mid \exists p \in \mathbb{N} \text{ s.t. } \texttt{A.disjoint(I,p)==false}\},$$
$$\alpha_> = \{I \in \beta \mid \exists p \in \mathbb{N} \text{ s.t. } \texttt{A.disjoint(I,p)==true}\}.$$

To ensure that `disjoint` gives consistent results, we require the following condition on $\alpha_<$ and $\alpha_>$.

**(DC)** If $I \in \alpha_<$ and $J_i \in \alpha_>$ for $i = 1, \ldots, k$, then $I \not\subset \bigcup_{i=1}^k \overline{J}_i$.

Given $\alpha_<$ and $\alpha_>$, we can recover sets $A_<$ and $A_>$ by:

$A_< = \{x \mid \exists (J_i)_{i\in\mathbb{N}}$ with $J_i \in \alpha_<$ s.t. $J_{i+1} \subset J_i$ and $\bigcap_{i\in\mathbb{N}} J_i = \{x\}\}$.
$A_> = X \setminus U_>$, where $U_> = \{x \mid \exists I \in \alpha_>$ s.t. $x \in \overline{I}\}$.

For $\alpha_<$, we impose the following monotonicity, refinement and approximation properties:

**($\mathbf{DM}_<$)** If $I \subset J$ and $I \in \alpha_<$, then $J \in \alpha_<$.
**($\mathbf{DR}_<$)** If $I \subset \bigcup_{i=1}^k J_i$ and $I \in \alpha_<$, then there exists $i$ such that $J_i \in \alpha_<$.
**($\mathbf{DA}_<$)** If $J \in \alpha_<$, then there exists $\overline{I} \subset J$ such that $I \in \alpha_<$.

The condition $\mathbf{DM}_<$ ensures that the set $A_<$ is closed, and the stronger condition $\mathbf{DR}_<$ ensures that every basic set $I \in \alpha_<$ contains a point in $A_<$.

In order that $A_>$ is a closed set, we require:

**($\mathbf{DA}_>$)** If $I \in \alpha_>$, then there exists $J \supset \overline{I}$ such that $J \in \alpha_>$.

To simplify algorithms using the `disjoint` method, we usually require either:

**($\mathbf{DM}_>$)** If $\overline{I} \subset \overline{J}$ and $J \in \alpha_>$, then $I \in \alpha_>$.
**($\mathbf{DR}_>$)** If $\overline{I} \subset \bigcup_{i=1}^k \overline{J}_i$ and $J_i \in \alpha_>$ for all $i$, then $I \in \alpha_>$.

In order that $\alpha_<$ and $\alpha_>$ yield equal closed sets, we require:

**(DE)** For all $J \in \beta$, there exists $I \subset J$ such that $I \in \alpha_< \cup \alpha_>$.

The following result gives conditions under which the `disjoint` method contains equivalent information to the standard representations of closed sets.

**Theorem 1**

1. *If* ***($\mathbf{DR}_<$,$\mathbf{DA}_<$)****, then* $A_<$ *is closed and* $\alpha_< = \{J \in \beta \mid A_< \cap J \neq \emptyset\}$.
2. *If* ***($\mathbf{DA}_>$)*** *then* $A_>$ *is closed; if also* ***($\mathbf{DR}_>$)*** *then* $\alpha_> = \{I \in \beta \mid A_> \cap \overline{I} = \emptyset\}$.
3. *If* ***(DC,$\mathbf{DR}_<$,$\mathbf{DA}_<$,$\mathbf{DA}_>$,DE)****, then* $A_< = A_>$.

The proof is omitted.

Since an open set is the complement of a closed set, we immediately obtain an interface for open sets:

```
virtual tribool OpenSet::superset(BasicSet,Integer);
```

The `superset` method defines a set $U_< = \bigcup\{\overline{I} \mid \exists p$ s.t. `U.superset(I)`$\}$. Given properties analogous to those for the `disjoint` method, we can prove that the `superset` interface is equivalent to the standard representation of open sets $\mathcal{O}$.

Since a compact set is just a bounded closed set, we can specify a compact set using the `disjoint` method, together with the method

```
virtual BasicSet CompactSet::BoundingBox();
```

which yields a basic set $I \supset C$.

### 3.3 Representations of Continuous Functions

Continuous functions can be described by the interface

```
virtual BasicSet Map::apply(BasicSet);
```

An object `f` of class `Map` represents a continuous function $f$ if the following consistency and refinement conditions hold:

**(FC)** $f(I) \subset$ `f.apply(I)`,
**(FR)** if $J \ni f(x)$, then there exists $I$ such that $x \in I$ and `f.apply(I)` $\subset J$.

It is also useful in practise to impose the monotonicity property:

**(FM)** if $I \subset J$, then `f.apply(I)` $\subset$ `f.apply(J)`, and

A differentiable function can be specified by giving the Jacobian derivative explicitly as a matrix of interval values

```
virtual Matrix<Interval> C1Map::jacobian(BasicSet);
```

## 4 Implementation in Ariadne

### 4.1 Numerical Types

Ariadne supports various types which can be used to represent real numbers, namely `Float64`, `MPFloat`, `Rational` and `ComputableReal`. These types are classified in terms of the way they handle arithmetic and approximation.

The `ComputableReal` type supports arbitrary arithmetical, algebraic and transcendental functions, which return exact results. The `Rational` type only supports arithmetic, but this is also exact. The *floating-point* types `Float64` and `MPFloat` do not support exact arithmetic, since in general, arithmetical operations cannot be computed exactly for these types. Instead, floating-point types support *interval* arithmetic and functions, which return an object of type `Interval<Float>` which must contain all possible results which could be obtained.

The *precision* of a floating point type is the number of bits/bytes used to store the number. The type `Float64` is a *fixed-precision* type with 64 bits, and the type `MPFloat` is a *multiple-precision* type. The precision of an object of type `MPFloat` is set when the object is constructed, and is only changed on explicit `set_precision` function call. The precision may be given explicitly, but it is usually more convenient to allow the precision to be determined implicitly by the precision of the arguments (and of the result, if the object is a temporary intermediate in a computation), and by a default precision. To avoid accidental truncation, it is an error to assign a number to an `MPFloat` object of lower precision, though assignment to `Interval<MPFloat>` objects is allowed.

The memory for an `MPFloat` is allocated on the heap, which may be slow, but once the object is created, no memory management is required, so computation is reasonably fast.

The fixed-precision types are useful for problems where speed of execution is paramount. The multiple-precision type can be used if the fixed-precision types do not give sufficient accuracy, which may be the case if the problem depends sensitively on initial data or is not robust to perturbations. Rational numbers are useful when exact arithmetical results are required or efficiency is not an issue, and computable real numbers are useful for problem specification.

Currently, the type `Float64` is implemented using the IEEE `double` floating-point numbers, the `MPFloat` type using `mpfr_t` from the MPFR library [12], the `Rational` using `mpq_t` from the GMP library [13], and the `ComputableReal` type using the iRRAM package [14].

### 4.2 Linear Algebra

Linear algebra is important when working with derivatives of maps, and with polyhedral sets. In Ariadne, we provide `Vector`, `Matrix` and `Tensor` classes, which can be used with rational numbers, floating-point numbers and intervals, and also robust linear programming solvers for testing geometric predicates.

### 4.3 Geometric Calculus

The geometric calculus used by Ariadne is based around elementary *basic set* types. A *denotable set* is a finite union of basic sets (of a given type), and can be stored using a finite amount of data. Arbitrary sets are either described by the interfaces given in Section 3.2, or by approximations in terms of denotable sets. Operations on sets can be built on the fundamental geometric predicates of *disjointness* and *subset*, and the operations of *union*, *intersection*, *subdivision* and *approximation*.

To simplify the interface, we do not use an explicit precision argument in Ariadne. Instead, the precision of an operation acting on sets represented using floating-point types is determined by the precision used for the arguments and the default precision. Basic sets with interval coefficients are returned by operations which require arithmetic on sets represented using floating-point types. Operations using sets based on rational numbers are computed exactly.

The simplest type of basic set for Euclidean space are the rational or dyadic hypercubes of the form $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$. In Ariadne, hypercubes are represented by the template `Cuboid<R>`, where `R` is the numerical type used to represent the $a_i$ and $b_i$. This may be a `Float` type, an `Interval<Float>` or a `Rational`. The core interface is given below:

```
class Cuboid<R> {
  Integer dimension();
  R lower_bound(Integer);
  R upper_bound(Integer);
  Integer precision();
};
```

Besides cuboids, ARIADNE provides basic set classes

`Zonotope`: $\{x \in \mathbb{R}^n \mid x = c + Ge \text{ where } c \in \mathbb{R}^n,\ G \in \mathbb{R}^{n\times k} \text{ and } e \in [-1,1]^k\}$.
`Polytope`: $\{x \in \mathbb{R}^n \mid x = Vs \text{ where } V \in \mathbb{R}^{n\times k},\ s \in \mathbb{R}^k_+ \text{ and } \sum_{i=1}^k s_i = 1\}$.
`Polyhedron`: $\{x \in \mathbb{R}^n \mid Ax \leq b \text{ where } A \in \mathbb{R}^{k\times n} \text{ and } b \in \mathbb{R}^k\}$.

The `Zonotope` class is useful to support higher-order integration of vector fields and iteration of maps. The `Polytope` and `Polyhedron` classes are general classes which are useful for computing geometric predicates; additionally, `Polyhedron` class may be useful for specifying input sets. Any polyhedral set can be converted to a `Polytope` or `Polyhedron`, although in most cases the conversion cannot be done exactly using floating-point arithmetic. `Parallelotope`, `Simplex`, `Sphere` and `Ellipsoid` classes are also provided. The basic set classes support the binary predicates given below.

```
tribool contains(Cuboid<R1>, Point<R2>);
tribool disjoint(Cuboid<R1>, Cuboid<R2>);
tribool subset(Cuboid<R1>, Cuboid<R2>);
```

These predicates return `indeterminate` if the result is not robust with respect to changes in the parameters. Basic set types also support the `Cuboid bounding_box()`, `subdivide` and `over_approximation` functions. Other binary operations are provided where natural, such as `minkowski_sum`, `convex_hull`, `open_intersection` and `closed_intersection`.

A *denotable set* is a set which is described exactly as a finite union of basic sets, and typically represents an approximation to some other set. Denotable sets support the fundamental geometric operations, iteration through their elements and the `union` function.

ARIADNE currently supports classes `ListSet<BasicSet>`, `GridCellListSet`, `GridMaskSet` and `PartitionTreeSet`. A `ListSet` is an arbitrary finite union of basic sets. The other types are *partition sets*, since they are based on a topological partition of the state space. The `GridMaskSet` class is easy to work with and is an efficient way of storing unstructured sets. The `GridCellListSet` class is useful to represent approximations to basic sets. The `PartitionTreeSet` class is a highly efficient way of storing structured sets with dynamically-varying resolution.

We say a denotable set is an *under-approximation* to a set $S$ if $\bigcup_{i=1}^k \overline{I}_i \subset S$, and an *over-approximation* if $\bigcup_{i=1}^k \overline{I}_i \supset S$. A list set $\bigcup_{i=1}^k J_i$ is a *lower-approximation* to $S$ if $J_i \cap S \neq \emptyset$ for all $i$. Approximations on partition sets can be specified by giving a `Grid` or other `Partition`, or by directly adjoining elements of an existing partition set:

```
void PartitionSet::adjoin_over_approximation(Set);
```

General sets can be specified by how they interact with `Cuboid` basic sets, as given by the methods `disjoint`, `superset`, `subset` and `bounding_box`. Using `superset` we can compute under approximations, using `disjoint` we can compute lower-approximations, and using `disjoint` and `subset` or `bounding_box`, we can compute over-approximations.

### 4.4 Computing the Image of Sets

One of the most important tasks in ARIADNE is to compute the image of a set under a continuous function. Using just the `apply` method of the `Map` interface described in Section 3.3, we can compute images and preimages of general sets:

```
CompactSet image(Map, CompactSet);
ClosedSet lower_image(Map, ClosedSet);
OpenSet lower_preimage(Map, OpenSet);
```

The image of a closed, but not compact set, and the preimage of an open set are only lower-semicomputable, in the sense that we can only effectively compute convergent lower-approximations to result. The image of a compact set can be computed to arbitrary accuracy.

Although the `apply` method is in principle sufficient to compute set images and preimages, convergence to a good approximation tends to be slow due to the "wrapping effect" of interval arithmetic. It is therefore preferable to use higher-order algorithms if derivatives of the function are available. Since the class of zonotopes is closed under affine maps, we can use zonotopes to compute first-order approximations to the image of a set.

```
Zonotope< Interval<R>> apply(C1Map<R>, Zonotope<R>);
```

The return type is an interval set to avoid expensive approximation operations. The image of a rational zonotope under an affine map can be computed exactly. If higher-order derivatives are available, even more accurate Taylor methods can be used.

### 4.5 The Verification Algorithm

We can now sketch an algorithm to solve the safety problem.

To attempt to verify $\mathrm{chainreach}(f, X_0) \subset S$ for some bounded open set $S$, we compute an over-approximation $\widehat{X}_0$ to $X_0$ and an under-approximation $\widetilde{S}$ of $S$ on a grid $G$ using the `over_approximate` and `under_approximate` functions. We then discretise $f$ by computing an over-approximation $\widehat{f}(\overline{I})$ to $f(\overline{I})$ for every grid cell $\overline{I}$ using the `apply(Map,Zonotope)` and `over_approximate(Zonotope,Grid)` functions. Finally, we can compute the reachable set $\widehat{X}_0$ under $\widehat{f}$ combinatorially. It is straightforward to show that if $\mathrm{reach}(\widehat{f}, \widehat{X}_0) \subset \widetilde{S}$ then $\mathrm{chainreach}(f, X_0) \subset S$, and that the converse holds if the grid $G$ is sufficiently fine.

To attempt to verify $\mathrm{reach}(f, X_0) \not\subset S$, we find a basic set $I$ and a natural number $n$ such that $X_0 \cap I \neq \emptyset$ and $f^n(\overline{I}) \cap \overline{S} = \emptyset$. We can use the `apply` function to compute an over-approximation $\widehat{I}_n$ to $f^n(\overline{I})$, and the `disjoint` method to show $X_0 \cap I \neq \emptyset$ and $\overline{S} \cap \widehat{I}_n = \emptyset$.

In practice, over-approximations to $f^n(X_0)$ are adaptively computed on-the-fly, and counterexamples from the discretised safety verification algorithm are used to guide safety falsification.

## 5 Concluding Remarks

In this paper, we have outlined a scheme for rigorous computation on sets and maps based on the theory of computable analysis and standard representation of topological space. This scheme is being used as the interface for the implementation in C++ of the numerical kernel of the tool ARIADNE for reachability analysis and verification of nonlinear and hybrid dynamical and control systems. The operations required by the interface have a natural syntax and have efficient implementations, some which are based on well-studied general algorithms such as the simplex algorithm or integration algorithms.

Ongoing work includes improving the efficiency of the existing algorithms in ARIADNE, providing more advanced capabilities for applying maps and integration vector fields based on higher-order Taylor methods, providing interfaces to external packages, and extending the interface to cover more advanced problems.

## References

1. van der Schaft, A., Schumacher, H.: An introduction to hybrid dynamical systems. Lecture notes in control and information sciences, vol. 251. Springer, London (2000)
2. Weihrauch, K.: Computable analysis - An introduction. In: Texts in Theoretical Computer Science, Springer, Heidelberg (2000)
3. Brattka, V., Presser, G.: Computability on subsets of metric spaces. Theoretical Comp. Sci. 305, 43–76 (2003)
4. Stoltenberg-Hansen, V., Lindström, I., Griffor, E.R.: Mathematical Theory of Domains. Cambridge University Press, Cambridge (1994)
5. Collins, P.: Continuity and computability of reachable sets. Theor. Comput. Sci. 341, 162–195 (2005)
6. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs, N.J (1966)
7. Dellnitz, M., Froyland, G., Junge, O.: The algorithms behind GAIO-set oriented numerical methods for dynamical systems. In: Ergodic theory, analysis, and efficient simulation of dynamical systems, pp. 145–174. Springer, Heidelberg (2001)
8. Mrozek, M., et al.: CAPD Library (2007), `http://capd.wsb-nlu.edu.pl/`
9. Nedialkov, N.S.: VNODE-LP: A validated solver for initial value problems in ordinary differential equations. Technical report, McMaster University (2006) CAS-06-06-NN.
10. Izaias Silva, B., Keith Richeson, B.K., Chutinan, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: Proceedings of the International Conference on Automation of Mixed Processes. pp. 189–194 (2000)
11. Balluchi, A., Casagrande, A., Collins, P., Ferrari, A., Villa, T., Sangiovanni-Vincentelli, A.L.: Ariadne: a framework for reachability analysis of hybrid automata. In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems (2006)
12. Hanrot, G., et al.: The MPFR library (2000), `http://www.mpfr.org/`
13. Granlund, T., et al.: The GMP library (2005), `http://swox.com/gmp/`
14. Müller, N., et al.: iRRAM (2006), `http://www.informatik.uni-trier.de/iRRAM/`