Pieter Collins
Department of Data Science and Knowledge Engineering
Maastricht University
pieter.collins@maastrichtuniversity.nl

Model checking dynamic systems

Pieter Collins started work on model checking hybrid systems as a postdoctoral research fellow at the Centrum Wiskunde & Informatica in the group of Prof. Jan H. van Schuppen, working on an EU-funded project 'Control and Computation'. He contributed to the software project Ariadne for reachability analysis and verification of hybrid systems, developed by the company Parades (now Ales) in Rome, and the group of Tiziano Villa in Udine (now Verona). This project led to a Vidi grant on 'Computational Topology for Systems and Control'. Since 2011 he has been a lecturer in the Department of Data Science and Knowledge Engineering and Maastricht University.

Engineers design and build things: mechanical engineers build machines, genetic engineers design lifeforms, and software engineers make computer programs. And of course, these things should serve some purpose and properly perform the tasks they were designed to do. Trains on a railway track should not crash into each other, radiotherapy machines should not deliver a lethal dose, and your computer's operating systems should never display a 'blue screen of death'. To this end, designs are analysed, prototypes tested, and the final product trialled. This process is known as *verification*, and is one of the most important and challenging parts of engineering.

Model checking [5, 11] is a computational approach to verification, in which a (mathematical) model of the system is analysed to determine whether it satisfies its specification as expressed by some temporal logical formula. The analysis must be performed entirely rigorously to avoid the possibility of dangerous or costly errors. Working with models is often cheaper than working with physical prototypes, and may be more powerful, since exhaustive testing may not be feasible, or scenarios may occur which had not been considered during testing.

I first started work in model checking as a postdoctoral researcher in the group of Jan H. van Schuppen at the Centrum Wiskunde & Informatica in Amsterdam, on an EU-funded project 'Control and Computation'. A part of this project was the development of a model checking tool Ariadne for *hybrid systems*, led by the group of Tiziano Villa in Udine, and a small Rome-based company Parades with links to the automotive industry. Hybrid systems [45] first appeared in [18, 41], and are characterised by continuous evolution interspersed by discrete *events* causing an instantaneous jump in the system state, or a switch in the *mode* of behaviour. They typically arise in the control of a physical system via discrete sensors and actuators, but are also encountered in systems with *impacts*, such as robot locomotion, or as electrical systems with switches and/or diodes.

The problem of model checking hybrid systems is a very rich domain, encompassing many of the important parts of continuous mathematics: real numbers, continuous functions, open and compact sets, algebraic and differential equations, and constraint feasibility problems. There are important theoretic questions of which properties of the evolution are even *computable* [48], and practical problems in the development of tools for efficient and *rigorous* numerical computation [25]. As I had already developed an interest in these issues in the context of chaos theory, I saw collaboration on Ariadne as a perfect opportunity to actually develop software, both as an industrially-viable tool for the model checking problem itself, and a flexible, general-purpose package for rigorous numerical computation.

My work on Ariadne led to a rich vein of both theoretical research and practical software development, and to a successful Vidi grant proposal on 'Computational Topology for Systems and Control'. An initial version of Ariadne focused solely on reachability analysis of hybrid systems using affine sets as a fundamental internal representation, and rigorous linear algebra and linear programming to perform computations. While this provided fast computations with reasonable accuracy, the under-

lying technology was only appropriate for fairly coarse safety properties. Subsequent versions of Ariadne featured a full rigorous numerical calculus for numbers, vectors, functions and sets.

Of course, there were already tools for rigorous numerical computation, including the interval arithmetic package INTLAB [43], differential equation solvers AWA [31], VNODE [39] and COSY Infinity [33], and dynamical systems analysers CAPD-Lib [38] and GAIO [14]. Hybrid system solvers included d/dt [2] and HyTech [20] for linear continuous dynamics, and HyperTech [21] and Checkmate [10] for nonlinear dynamics. With Ariadne we aimed to extend the scope and flexibility of these tools, in particular to provide an extensible framework where sets and functions could be manipulated as first-class objects, and to study highly nonlinear and/or nondeterministic systems. Later hybrid systems tools include HSolver [42], Phaver [16], SpaceEx [17], Flow* [9] and AERN [15], with the latter three featuring a similar rigorous numerical calculus to Ariadne.

Model checking hybrid systems

Model checking was originally developed for verifying the flow-of-control of software systems. System models are given by *discrete automata* with a finite state set X specifying the set of possible *executions* $(x_0, x_1, x_2, \ldots)$. Specifications are given by a *temporal logic* formula describing the allowable executions. A *linear temporal logic* formula is built up from *atomic propositions* a, which are sets of states, basic logical operations, and *temporal operators*: *next* (denoted $\bigcirc$), *always* ($\Box$), *eventually* ($\Diamond$), *until* and *release*. For example, the property $\Diamond a \vee \bigcirc\Box b$ (eventually a or next always b) means that either property a holds at some time in the future, or that after one time step, property b always holds. Of particularly importance are *safety* properties $\Box a$ (always a). These can be checked by computing the *reachable set* of states which it is possible to attain from a starting state.

There are many tools for model checking of linear temporal logic for finite automata, such as [23]. They either prove that the specification is satisfied for all executions of the system, or output an execution not satisfying the specification. This *counterexample* may indicate a flaw in the system design, or that the specification has not been properly thought-out.

The main practical difficulty in model checking is state-space explosion caused by the so-called *curse of dimensionality* — for a moderately-sized system with 100 on–off switches, there are $2^{100} \sim 10^{30}$ states, which would take approximately the age of the universe (13.8 billion years) to analyse even when looking at a trillion states per second! Discrete model checkers therefore use a variety of data representations and algorithmic tricks to speed up the calculation. Since the result must be rigorously correct, randomised algorithms cannot be used.

As soon as the number of states becomes infinite, such as by introducing integer variables, then it turns out that model checking linear temporal logic is *undecidable*. A a yes/no question is *decidable*, or a function computing a string or numerical value *computable*, if it can be implemented on a *Turing machine*, which can be thought of as a program running on a digital computer with unlimited memory. It is easy to see that there must be undecidable problems and uncomputable functions, since there are uncountably many functions $\mathbb{N} \to \{0,1\}$, but only countably many programs. The *halting problem*, of determining whether a computer program will terminate or run forever, is the most famous undecidable problem, and can actually be formulated as a model checking problem. Note that undecidability of a problem *class* does not mean that particular *instances* of a problem are unsolvable; for example, termination of a program can always be proved by running it long enough (so halting is *verifiable*), and the infinite loop `while(true) {...}` clearly never terminates.

Hybrid automata were seen by computer scientists as a natural extension of discrete automata to include continuous dynamics. They consist of finitely many *discrete states* $q_1, \ldots, q_n$, in each of which the *continuous state* $x_i \in X_i$ satisfies a differential equation $\dot{x}_i(t) = f_i(x_i(t))$. When the continuous state satisfies a *guard condition* $g_{i,k}(x_i(t)) \geq 0$, a *discrete event* e_k occurs, with the discrete state changing to q_j, and the continuous state being *reset* to $x_j(t) = r_{i,k}(x_i(t))$. Hybrid automata are often described by a graph with state dynamics in the vertices, and guards/re-

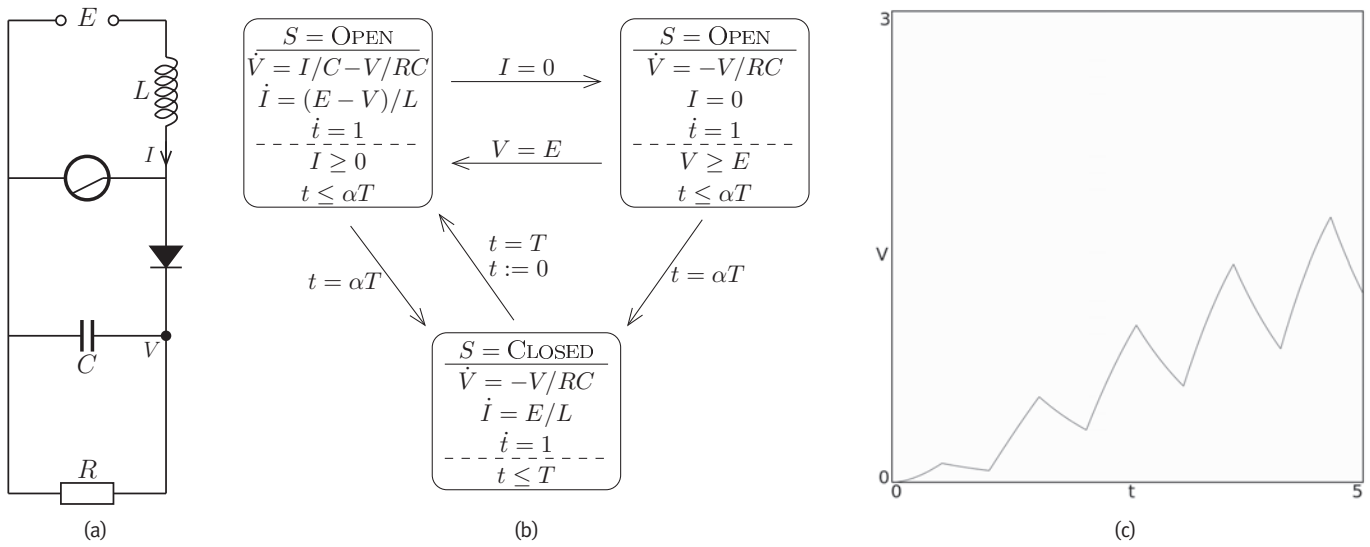


Figure 1 (a) Circuit diagram for a boost power converter. (b) A hybrid automaton representing the boost power converter. (c) Time-evolution of the capacitor voltage.

sets on the edges. An example of a hybrid automaton and its dynamics is shown in Figure 1.

For a simple class of hybrid systems, the *timed automata*, in which the only continuous dynamics are *clocks* satisfying $\dot{t}_i = 1$, model checking was shown to be decidable [1]. This result led to the development of tools such as UPPAAL [29] for the analysis of timed automata. For slightly more complicated classes of hybrid system, including saturated linear systems [7], model checking becomes undecidable [3, 22]. Clearly model checking more general nonlinear hybrid systems is also undecidable; the real question is whether we can verify any such systems at all. Since properties of continuous systems may be highly sensitive to small changes; for example, a small change in a parameter may cause a *bifurcation* or *catastrophe*, we should not expect to be able to say very much unless the property we are considering is *robust* with respect to parameter uncertainty and noise.

Computable analysis

To answer the question of what is possible to verify for nonlinear hybrid systems, we need a theory of (rigorous) computation for continuous mathematics, a subject known as *computable analysis* [26, 48]. The roots of such a theory lie in constructive mathematics, and computable mathematics can be derived from a suitable constructive mathematics using realization theory [6]. However, while in Brouwer's intuitionistic mathematics [12] and other constructive logics, axioms are admitted depending on whether one believes them to be self-evident, in computational mathematics, we have a much clearer criterion: *can we implement it?* Hence the principle of the excluded middle is not accepted (as there are statements that can be neither proved nor disproved), but the axiom of dependent choice (that we can make an unending sequence of choices, each depending on the one before) is accepted. Computable mathematics is uniform in the way that constructive/intuitionistic mathematics is not.

The main difficulty in continuous mathematics is that objects lie in *uncountable* sets, which means we can only describe them exactly using an infinite *stream* of data. The key idea of computable analysis is that it should be possible to extract useful information about a value from a *finite* piece of its describing stream. Computation is still performed using Turing machines, but a program runs forever, reading data from its input streams and writing to its output stream.

It turns out that there is a deep link between topology and computation; a representation of a set of objects by streams induces a natural topology on the set [46], and only continuous functions can be computable. Indeed, the set of all functions with uncountable domain has higher than continuum cardinality, so cannot be described by streams. Two representations of the same set are equivalent if each can be computably converted into the other. A *type* is defined to be a set with an equivalence class of representations.

We start developing computable analysis by constructing the type of real numbers. We are used to describing real numbers either by algebraic formulae, such as $1/7$, π or $\sin(\pi/7)$, or as a decimal expansion, such as $0.\dot{1}42857\dot{}$, or $3.14159\cdots$. Since the real numbers are uncountable, only infinite representations such as decimal expansions can represent them all.

Unfortunately, there is also a problem with decimal expansions: They do not support arithmetic! What is the value of $3\times 0.33333\cdots$? Clearly, if the second number is exactly $\frac{1}{3}$, then the result is $1.00000\cdots$, or equivalently, $0.99999\cdots$. But we can never output $1.0\cdots$, because maybe we have $3\times 0.333332\cdots$, which is definitely less than 1. And we can never output $0.9\cdots$, because maybe we have $3\times 0.333334\cdots$. Hence we can never give the first digit of the answer!

Thankfully, there are ways around this conundrum. The most straightforward way is to use a *signed-digit representation*, which allows negative digits such as $\bar{2}$ (representing '-2') as well as positive digits. The signed-digit representation is equivalent to allowing an error of ± 1 in the last given digit, since $3.1416\cdots$ could represent

$$3.1416\dot{\bar{9}} = 3.1415,$$

$$3.1416\dot{9} = 3.1417,$$

or any number in the interval

$$[3.1415 : 3.1417] = 3.141\,[5:7] = 3.141\,[6\pm 1].$$

Use of signed digits resolves our arithmetical dilemma, since

$$\begin{aligned} &3\times 0.33333\cdots \\ &\quad\in 3\times[0.3333\bar{3}\bar{9}\bar{9}\cdots : 0.3333399\cdots] \\ &\quad= 3\times[0.33332 : 0.33334] \\ &\quad= [0.99996 : 1.00002] \\ &\quad= [1.0000\bar{4} : 1.00002] \\ &\quad\subset 1.0000\cdots. \end{aligned}$$

There are other representations of the real numbers which are equivalent to the signed-digit representation. The *Cauchy* representation describes a real number x by a sequence of rationals q_n with $|x-q_n|\leq 2^{-n}$, and the *Dedekind* representation by a sequence of rational intervals $[a_n, b_n]$ with $a_n \leq a_{n+1} \leq b_{n+1} \leq b_n$ and $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n = x$. Note that the Cauchy representation focusses on the *metric* structure of the real numbers, whereas the Dedekind representation focusses on the *order* structure.

A real number x is computable, as per Turing's definition [47], if there is a computer program which outputs its signed-digit (or an equivalent) representation. All algebraic numbers are computable, as are important constants such as e and π, but only countably many real numbers are computable.

The usual arithmetical operators $+$, $-$, $\times$, $\div$ are computable, as are elementary functions such as $\exp$, $\log$, $\sin$ and $\arctan$. Further, it is possible to take the limit of a sequence x_n with a known a rate of convergence ϵ_n i.e. $|x_{n_1} - x_{n_2}| \leq \epsilon_{\min\{n_1,n_2\}}$.

Comparison of real numbers is undecidable, since we can never tell that a number x actually equals 0 from finite piece of its signed-digit expansion $0.0000\cdots$. For this reason, we need *Kleenean* logic for comparison $x \lesssim y$, with three values $\mathbb{K} = \{\top, \bot, \uparrow\}$ representing *true*, *false* and *indeterminate* ("don't know"), the latter for the case x and y are equal. Note that while a symbolic approach may allow one to decide whether $x = 0$ in some cases e.g. $\sin(3\pi) = 0$, it is an open problem as to whether a number defined in terms of rationals and elementary functions can always be tested for being zero.

To define more complicated types, we can fall back on some very general constructions. For types $\mathbb{X}_1, \ldots, \mathbb{X}_n$, there is a *product type* $\mathbb{X}_1 \times \cdots \times \mathbb{X}_n$ of tuples $(x_1, \ldots, x_n)$ with $x_i \in \mathbb{X}_i$. For types $\mathbb{X}, \mathbb{Y}$, there is an *exponent* type $\mathbb{Y}^{\mathbb{X}}$ which consists of *sequentially continuous* functions $\mathbb{X} \to \mathbb{Y}$. The operations of *evaluation* $(f, x) \mapsto f(x)$

and *composition* $(g,f)\mapsto g\circ f$ are both computable. A foundation of functional programming languages is that the types $\mathbb{X}\times\mathbb{Y}\to\mathbb{Z}$ and $\mathbb{X}\to(\mathbb{Y}\to\mathbb{Z})$ are equivalent. Hence if for every x, the function f_x is computable, then the function f defined by $f(x,y):=f_x(y)$ is also computable.

For real-valued functions $\mathbb{X}\to\mathbb{R}$, it follows from computability of composition that pointwise arithmetic operations $f_1,f_2\mapsto f_1\star f_2$ are computable. For more complicated operators, we may need to return to first principles to prove computability. In particular, solutions of the algebraic equation $f(x)=0$ for $f:\mathbb{R}^n\to\mathbb{R}^n$ are computable if, and only if, they are isolated and have non-zero *index*. The solution of a differential equation $\dot{x}=f(t,x)$ starting at x_0 is computable if it is unique [44].

We define the type of *open* subsets $U\in\mathcal{O}(\mathbb{X})$ in terms of the *membership* function $\chi_U:\mathbb{X}\to\mathbb{S}$, where $\mathbb{S}=\{\top,\uparrow\}$ is the *Sierpinskian* type. Computable open sets therefore have a logical characterisation as sets for which membership is verifiable. The type of *compact* sets is defined as the subtype of $\mathcal{O}(\mathbb{X})\to\mathbb{S}$ satisfying $C\subset(V_1\cap V_2)\Leftrightarrow(C\subset V_1)\wedge(C\subset V_2)$. Computable compact sets as those for which being a subset of an open set is verifiable. It is easy to show computablility of unions and intersections of open and compact sets, of the preimage $f^{-1}(V)$ of an open set V under a continuous function f, and the image $F(C)$ of a compact set C under a compact-valued function F.

Rigorous numerics

Computability theory addresses the problem of what it is possible to compute with a digital computer. To show something is possible, we give an algorithm, but this algorithm is usually given to be theoretically clean and need not be particularly efficient. In order to implement a practical model checker, we also need fast *rigorous numerical* algorithms.

In almost any computational environment, if you type in `x=0.6+0.3+0.1` you will get the answer `x=0.9999999999999999` (and if you get something different, it will probably be `1.00000`, but then `x-1` yields `-1.1102e-16`). The reason for this is that most numerical computational tools use (double-precision) *floating-point* arithmetic, in which numbers are represented by a binary expansion with 53 significant (binary) digits, and results of any arithmetical operation are rounded to the nearest representable value. For many applications, the roundoff-errors are so small as to have negligable effect on the answer. But we don't know this for sure.

In *interval arithmetic* [25, 35, 36], finite-accuracy approximation to a real number is represented by an *interval*, either with lower and upper bounds $x\in[\underline{x},\overline{x}]$or a midpoint and error, $x=\check{x}\pm e$. Ariadne provides intervals with endpoints which can be either builtin double-precision floating-point numbers, or multiple-precision floating-point numbers (as provided by the MPFR library [37]). Operations on intervals are implemented by rounding the results of finite-precision calculations *outward*. For example, working to three decimal places

$$\begin{aligned}\pi\times e&\in[3.141{:}3.142]\,\hat{\times}\,[2.718{:}2.719]\\&=[8.537{:}8.544]\\&\supset[8.537238{:}8.543098].\end{aligned}$$

Interval arithmetic illustrates the core idea of rigorous numerical computation, namely to work with sets of values which are guaranteed to contain the correct result. The sets used are called *basic* sets, and ideally are kept small. This means, for example, that if $\hat{x}$ is a basic set containing x and $\hat{y}\ni y$, then a validated version $\hat{\star}$ of a binary operator $\star$ satisfies $\hat{x}\,\hat{\star}\,\hat{y}\ni x\star y$. In a metric space, basic sets of the form $\{x\mid d(x,\check{x})\leq e\}$ are most useful, where $\check{x}$ lies in a countable dense subset $\check{X}$ of $\mathbb{X}$.

Continuous functions $\mathbb{R}^n\to\mathbb{R}$ can be approximated uniformly accurately by polynomials on bounded domains. In Ariadne, we provide basic sets of continuous functions on domains which are coordinate-aligned *boxes*

$$\prod_{i=1}^{n}[a_i{:}b_i]=[a_1{:}b_1]\times[a_2{:}b_2]\times\cdots\times[a_n{:}b_n].$$

A simple class of representation are *Taylor models* [34], in which basic sets of functions over a box domain $D\subset\mathbb{R}^n$ are given as

$$\max_{x\in D}|\,f(x)-p(s^{-1}(x))\,|\leq e,$$

where $p=\sum_\alpha c_\alpha x_1^{\alpha_1}\cdots x_n^{\alpha_n}$ is a polynomial on $[-1{:}+1]^n$ with floating-point coefficients c_α, and $s{:}[-1{:}+1]^n\to D$ is a coordinate scaling function, as shown in Figure 2. The scaling to the unit box domain improves the accuracy of operations on the polynomial p, and allows for easy simplification of the representation by *sweeping* small terms into the error bound e. Other polynomial model representations are possible by using a different basis. The Taylor basis of monomials is easiest to perform arithmetic on, but the Chebyshev basis or Bernstein bases may yield more accurate evaluation. For problems in partial differen-

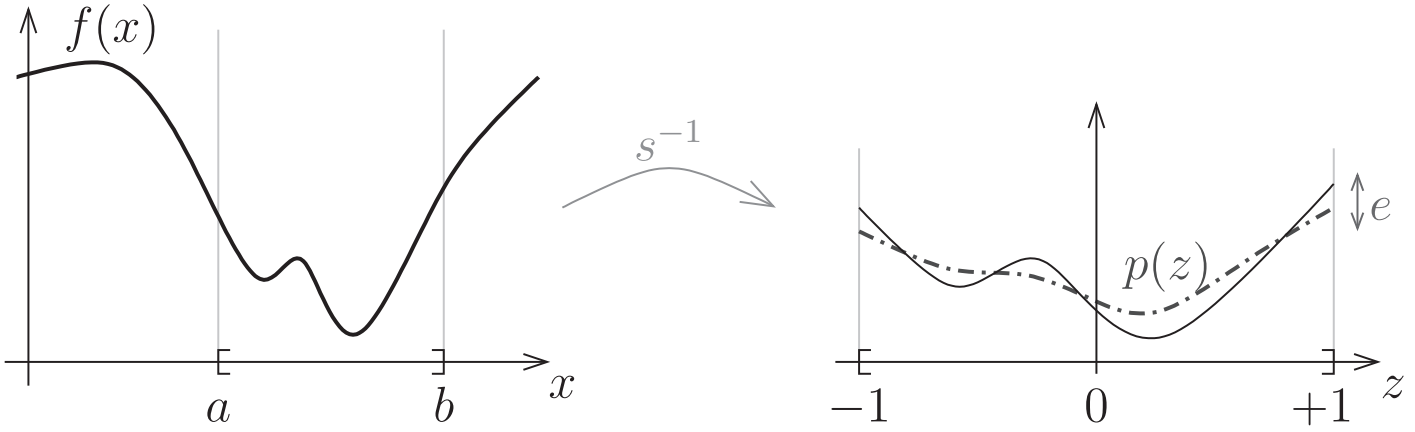
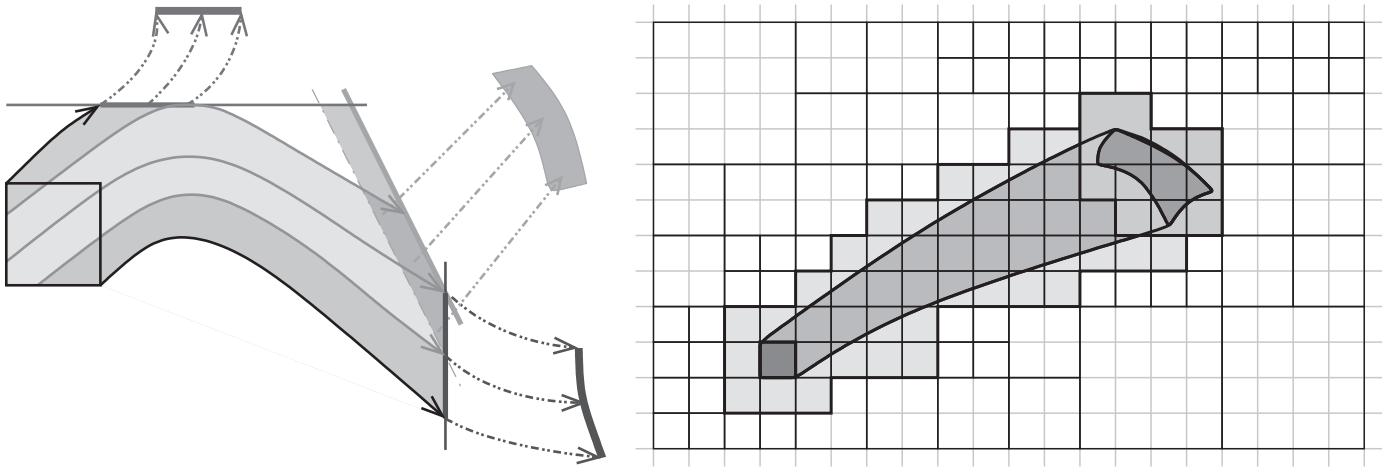


Figure 2 A Taylor model.



(a) (b)

Figure 3 (a) Behaviour of a hybrid system. (b) Discretising a flow tube.

tial equations, a Fourier basis of $\cos(n\pi x)$, $\sin(n\pi x)$ may be used [13].

Parameterised algebraic equations of the form $f(x,h(x))=0$ with $x\in D$ and $h(x)\in E$, are solved in Ariadne using a version of the efficient *interval Newton method* [36]. Differential equations of the form $\dot{\phi}(x,t)=f(\phi(x,t))$ can be solved in Ariadne either using an interval version of the *Picard operator* for the corresponding integral equation or by computing the Taylor series of the flow with error bounds [49].

In Ariadne, we provide two different kinds of sets. Accurate representations are given in terms of functions, with *constraint sets*

$$S=g^{-1}(C)=\{x\in\mathbb{R}^n\mid g(x)\in C\}$$

being regular/open, and *constrained image sets*

$$\begin{aligned}S&=h(D\cap g^{-1}(C))\\&=\{h(w)\mid w\in D\mid g(w)\in C\}\end{aligned}$$

being located/compact. The intersection of a constraint set with a constrained image set, or the image of a constrained image set, can both be realised using function composition. Coarser representations are given by *pavings*, which are unions of boxes with disjoint interiors, and support efficient intersection, union, and subset testing. An over-approximation of a constrained image set by a paving is computed be testing emptiness of sets $h(D\cap g^{-1}(C))\cap B$, which reduces to the constraint satisfaction problem $\exists\, z\in D{:}g(z)\in C\wedge h(z)\in B$.

We now have all the operators we need to verify hybrid systems in Ariadne. We start by considering computation of the evolution $\psi(x,t)$ from at a point x. The continuous dynamics can be computed by solving the differential equation in each mode $\dot{\psi}(x,t)=f_i(\psi(x,t))$, the times $\tau(x)$ of the discrete events can be computed by solving the equation $g_j(\psi(x,\tau(x)))=0$, and the discrete transitions can be computed by composition $\psi'(x,\tau(x))=r_k(\psi(x,\tau(x)))$. The only difficulty which may occur is in computing the crossing time, for if a trajectory $\psi(x,t)$ *grazes* a guard set $g_j(y)=0$, then the further evolution is discontinuous in the initial condition. In this case, we need to consider both the case that a transition occurs, and that a transition does not occur, leading to multivalued dynamics.

By computing the evolution $\psi(x,t)$ as a function of both the initial point and time, we can compute the *reach sets* $\psi(B,[0,t])$ starting from a set of initial states B over a time interval $[0,t]$, and the *evolved sets* $\psi(B,t)$ attained at time t, both as constrained image sets, as shown in Figure 3(a). To verify a safety property, we *partition* the state space X into boxes, and construct an over-approximation of the infinite-time reachable set by discretising the reach- and evolve-sets on this partition, as shown in Figure 3(b), continuing until no more reachable boxes are found. If safety has not been verified, we check for an unsafe evolution, or refine the partition. It can be shown that safety is verifiable if, and only if, the *chain-reachable set*, which is an over-approximation to the set of points which the state of the system may attain, is a subset of the safe set.

Implementation

At the beginning of the project, a decision was made to use C++ as the implementation language, as this is a powerful and efficient language widely used in industry. Later, I added an interface in the scripting language Python to facilitate interactive use of the tool. Considerable attention has been devoted to designing a clean and understandable interface; indeed, most of my time working on Ariadne has been spent on issues of interface design and software engineering rather than algorithm development.

To organise the data types, each class of objects has an underlying mathematical type, such as `Number`, `VectorFunction` or `CompactSet`, and a tag specifying to kind of *information* provided about the quantity. The information may be `Exact`, `Effective`, meaning arbitrarily-accurate approximations can be computed, `Validated`, which means finitely-accurate bounds can be computed, or `Approximate`, which means no information about accuracy is given. These abstractions are implemented by concrete classes, such as `FloatMPBounds` (an interval with multiple-precision floating-point bounds) implementing a `ValidatedNumber`. Although this separation is conceptually clean, how to implement it causes problems, especially regarding automatic conversions which are prevalent in C++. For although any `Effective` description contains more information than a 'Validated' description, so should be convertible to validated, any conversion requires a *precision* or *accuracy* parameter, which must either be given ex-

plicitly, or implicitly from the environment, neither solution being entirely satisfactory.

A major challenge in software engineering is due to the C++ language itself: there are many ways of supporting the data abstractions used in Ariadne, and different approaches require changes to the underlying code. Indeed, a 'good' language would be one in which the underlying numerical, functional and geometric kernels could be implemented in a clean way! Together with Michal Konečný, the developer of [27], another package for rigorous numerics, I have been experimenting with the use of the functional language Haskell in order to find a good semantics.

Since a model checking tool is supposed to prove correct functioning of a dynamic system, we also need to know that the tool itself works correctly, without mistakes in the mathematical algorithms or their implementation. Proven-correct code for real arithmetic exists in Coq [40] and Haskell [30], and for differential equations in Coq [32] and Isbell/HOL [24]. A C implementation of the solution of the wave equation was proved correct in [8], using several different tools including the Frama-C code analyser platform and the Coq theorem-prover. However, this is already a huge effort for a fairly simple C programme, and verified correct implementations of a tool such a Ariadne are a long way off! More promising from the point of view of provably-correct code, is to demonstrate correctness of the algorithms using a theorem-prover such as Coq or Isabelle, and automatically generate code in an efficient and user-friendly language such as the functional language ML.

The future of model checking

Practical use of model checking for large-scale hybrid systems arising in industry will require many advances in algorithm design. A particularly important technical challenge, and currently one of the major bottlenecks in Ariadne, is to improve the core algorithms for working with functions and sets, notably in simplifying the representation of sets while preserving accuracy of an over-approximation.

It will also be important to develop algorithms to handle uncertain systems efficiently. Ultimately, any mathematical analysis of a model of reality is only as good as the model itself, and whenever we make models, there are approximations built-in. Even fundamental physical constants, such as the fine structure constant $\alpha^{-1} = 137.0359991\,[39 \pm 31]$, is only known to finite precision. Systems arising in engineering have larger uncertainties (such as the exact resistence of an electrical component), and our knowledge of biological systems is even less precise. This means that in order to validate properties of real-life systems, we also need to take this uncertainty into account.

Probability is the most widely-used mathematical theory of uncertainty, but in many cases, we cannot assign exact probabilities to the events, and independence requirements needed for stochastic systems models are not met. We therefore need uncertainty models based on what is *possible*, leading to *differential inclusions* [4]. Ellipsoidal methods for their solution are described in [28] and higher-order methods in [19]. Although non-stochastic modelling is sometimes seen as being 'overly pessimistic', in the absence of more information about the system, it is the only way of guaranteeing correctness. Other frameworks, such as *imprecise probability* give more modelling flexibility, but will be even harder to reason about.

Perhaps the main challenge lies in detecting and exploiting system structure. In particular, since model checking has exponential time complexity in the state-space dimension, any methods to reduce the effective dimensionality, or split the problem into sub-problems of lower dimensionality, will be invaluable. Differences in time- and length-scales provide further opportunities for reduction by splitting into fast and slow behaviour, or averaging out spacial fluctuations. Modularity provides the greatest opportunities for dimension reduction, by splitting a system into parts which communicate via external inputs and outputs. Next-generation model checking tools will need to automatically exploit possible reductions in order to make fundamental breakthroughs in engineering practise.

Concluding remarks

The main goal of the Ariadne project was to develop a tool for model checking nonlinear hybrid automata, and the current version is capable of verifying well-conditioned, low-dimensional nonlinear systems. However, I feel that rather than immediately develop software for hybrid systems, it would have been better to first focus on the considerably simpler case of continuous-time systems. This would have allowed testing and showcasing the tool on industrially-relevant problems with considerably less technical demands, leading to more robust software and more opportunity to attract broader scientific interest. The idea that moving straight from discrete automata to hybrid systems may have been a challenge too far, and an incremental approach would have perhaps been better, is also seen in recent workshops explicitly focussing on continuous as well as hybrid systems.

My other main vision with Ariadne was to have general-purpose rigorous numerics package, and the current version does provide a broad interoperative functionality for numbers, functions and sets. Further, the combination of a solid theoretical foundation based on computable analysis, clean generic abstractions and extensible concrete data types is unique amongst rigorous numerical tools. My immediate goals for future development and to stabilise the interface, and sollicit user feedback on naming conventions, data abstractions and documentation.

I still find it odd that in the digital computer, we have a device of astonishing speed, accuracy and reliability, yet in mathematics, a subject justly proud of its application of rigour, most computations are approximative with no guarantees on correctness of the answer! Of course, there are reasons why this situation arose: for many problems, the approximations are usually good enough, speed is a bigger issue than accuracy, and fixed-size floating-point representations can be handled most efficiently with current hardware, and rigorous numerical algorithms are *much* harder to develop. Even so, existing work in rigorous numerics, still a rather niche field of research, shows that we can have the benefits of mathematical rigour with a reasonable efficiency. I believe that in the long-term, rigorous computations will eventually take over from approximate methods, just as formal rigorous mathematics eventually dominated. To facilitate this process, we need to provide tools that are powerful enough for industrial use, and simple enough to train undergraduate students in mathematics, science and engineering. Ariadne is one important step in this direction, and I plan to continue development work in the future to further realise these aims.

References

1 Rajeev Alur and David L. Dill, A theory of timed automata, *Theoretical Computer Science* 126 (1994), 183–235.

2 E. Asarin, T. Dang and O. Maler, The d/dt tool for verification of hybrid systems, in *Proc. 14th International Conference on Computer Aided Verification*, LNCS, Vol. 2404, Springer, 2002, pp. 365–370.

3 Eugene Asarin, Oded Maler and Amir Pnueli, Reachability analysis of dynamical systems having piecewise-constant derivatives, *Theoretical Computer Science* 138(1) (1995), 35–65, Hybrid Systems.

4 Jean-Pierre Aubin and Arrigo Cellina, Differential inclusions: Set-valued maps and viability theory, *Grundlehren der Mathematischen Wissenschaften*, No. 364, Springer, 1984.

5 Christel Baier and Joost-Pieter Katoen, *Principles of Model Checking*, MIT Press, 2008.

6 Andrej Bauer, *The Realizability Approach to Computable Analysis and Topology*, PhD thesis, Carnegie Mellon University, 2000.

7 Vincent D. Blondel, Olivier Bournez, Pascal Koiran, Christos H. Papadimitriou and John N. Tsitsiklis, Deciding stability and mortality of piecewise affine dynamical systems, *Theoretical Computer Science* 255(1) (2001), 687–696.

8 Sylvie Boldo, François Clément, Christophe Jean-Filliâtre, Micaela Mayero, Guillaume Melquiond and Pierre Weis, Wave equation numerical resolution: a comprehensive mechanized proof of a C program, *Journal of Automated Reasoning* 50(4) (2013), 423–456.

9 Xin Chen, Erika Ábrahám and Sriram Sankaranarayanan, Flow*: An analyzer for non-linear hybrid systems, in *Computer Aided Verification*, Springer, 2013, pp. 258–263.

10 E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg and M. Theobald, Abstraction and counterexample-guided refinement in model checking of hybrid systems, *Internat. J. Found. Comput. Sci.* 14(4) (2003), 583–604.

11 Edmund M. Clarke, Orna Grumberg and Doron Peled, *Model Checking*, MIT Press, 1999.

12 Dirk van Dalen, Poincaré and Brouwer on intuition and logic, *Nieuw Archief voor Wiskunde* 5/13(3) (2012), 191–195.

13 S. Day, O. Junge and K. Mischaikow, A rigorous numerical method for the global analysis of infinite-dimensional discrete dynamical systems, *SIAM Journal on Applied Dynamical Systems* 3(2) (2004), 117–160.

14 M. Dellnitz, G. Froyland and O. Junge, The algorithms behind GAIO-set oriented numerical methods for dynamical systems, in *Ergodic Theory, Analysis, and Efficient Simulation of Dynamical Systems*, pages 145–174. Springer, 2001.

15 Jan Duracz, Amin Farjudian, Michal Konečný, and Walid Taha, Function interval arithmetic, in *Mathematical Software–ICMS 2014*, Springer, 2014, pp. 677–684.

16 G. Frehse, PHAVer: algorithmic verification of hybrid systems past HyTech, *Int. J. Softw. Tools Technol. Trans.* 10 (2008), 263–279.

17 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 *Proc. 23rd International Conference on Computer Aided Verification*, LNCS, Vol. 6806, 2011, pp. 379–395.

18 Aleks Göllü and Pravin Varaiya, Hybrid dynamical systems, in *Proceedings of the 28th IEEE Conference on Decision and Control*, IEEE, 1989, pp. 2708–2712.

19 Sanja Gonzalez Živanovič and Pieter Collins, Higher order methods for differential inclusions, Technical Report MAC-1007, Centrum Wiskunde & Informatica, 2010.

20 Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, HyTech: A model checker for hybrid systems, *Software Tools for Technology Transfer* 1 (1997), 110–122.

21 Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar and Howard Wong-Toi, Beyond HyTech: Hybrid systems analysis using interval numerical methods, in N. Lynch and B. Krogh, eds., *Hybrid Systems: Computation and Control*, LNCS, Vol. 1790, Springer, 2000, pp. 130–144.

22 Thomas A Henzinger, Peter W Kopke, Anuj Puri and Pravin Varaiya, What's decidable about hybrid automata? *Journal of Computer and System Sciences* 57 (1998), 94–124.

23 Gerard J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.

24 Fabian Immler, A verified enclosure for the Lorenz attractor (rough diamond), in Christian Urban and Xingyuan Zhang, eds., *Proceedings of the 6th International Conference on Interactive Theorem Proving*, Springer, 2015, pp. 221–226.

25 Luc Jaulin, Michel Kieffer, Olivier Didrit and Éric Walter, *Applied Interval Analysis*, Springer, 2001.

26 Ker-I Ko, Complexity theory of real functions, *Progress in Theoretical Computer Science*, Birkhäuser, 1991.

27 Michal Konečný, Aern-rntorm: Arbitrary-precision arithmetic of multivariate piecewise polynomial enclosures, 2008.

28 Alexander B. Kurzhanski and Istvåan Vályi, Ellipsoidal calculus for estimation and control, *Systems & Control: Foundations & Applications*, Birkhäuser, 1997.

29 Kim G. Larsen, Paul Pettersson and Wang Yi, UPPAAL in a nutshell, *Intern. J. Software Tools for Technology Transfer* 1(1–2) (1997), 134–152.

30 David R. Lester, The world's shortest correct exact real arithmetic program? *Information and Computation* 216, (2012), 39–46. Special Issue: 8th Conference on Real Numbers and Computers.

31 R.J. Lohner, AWA – software for the computation of guaranteed bounds for solutions of ordinary initial value problems, Technical report, Institut für Angewandte Mathematik, 2006.

32 Evgeny Makarov and Bas Spitters, The =Picard algorithm for ordinary differential equations in Coq, in *Interactive Theorem Proving*, Springer, 2013, pp. 463–468.

33 K. Makino and M. Berz, COSY INFINITY Version 9, *Nuclear Instruments and Methods* A558 (2006), 346–350.

34 Kyoko Makino and Martin Berz, Taylor models and other validated functional inclusion methods, *Int. J. Pure Appl. Math* 4(4) (2003), 379–456.

35 R. Moore, *Interval Analysis*, Prentice-Hall, 1966.

36 Ramon E. Moore, R. Baker Kearfott and Michael J. Cloud, *Introduction to Interval Analysis*, SIAM, 2009.

37 MPFR Library, 2000, www.mpfr.org.

38 M. Mrozek et al., CAPD Library, 2007.

39 Ned Nedialkov, VNODE-LP, Technical report, McMaster University, 2006. Technical Report CAS-06-06-NN.

40 Russell O'Connor, Certified exact transcendental real number computation in Coq, in Otmane Ait Mohamed, César Muñoz, and Sofiéne Tahar, eds., *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, Springer, 2008, pp. 246–261.

41 Philippos Peleties and Raymond DeCarlo, A modeling strategy with event structures for hybrid systems, in *Proceedings of the 28th IEEE Conference on Decision and Control*, IEEE, 1989, pp. 1308–1313.

42 S. Ratschan and Z. She, Safety verification of hybrid systems by constraint propagation based abstraction refinement, *ACM Trans. Embedded Comput. Sys.* 6(1), 2007.

43 S.M. Rump, INTLAB – INTerval LABoratory, in Tibor Csendes, ed., *Developments in Reliable Computing*, Kluwer Academic Publishers, 1999. www.ti3.tuhh.de/rump, pp. 77–104.

44 Keijo Ruohonen, An effective Cauchy-Peano existence theorem for unique solutions, *Int. J. Found. Comput. Sci.* 7(2) (1996), 151–160.

45 Arjan van der Schaft and Hans Schumacher, An introduction to hybrid dynamical systems, *LNCIS*, Vol. 251, Springer, 2000.

46 Matthias Schröder, *Admissible Representations for Continuous Computations*, PhD thesis, Fachbereich Informatik, FernUniversität Hagen, 2002.

47 A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc. (2)* 43(1), 1937, 230–265.

48 Klaus Weihrauch, Computable analysis – An introduction, *Texts in Theoretical Computer Science*, Springer, 2000.

49 Daniel Wilczak and Piotr Zgliczyński, Cr-Lohner algorithm, *Schedae Informaticae* 20 (2011), 9–42.