A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq

Pieter Collins, Milad Niqui, Nathalie Revol

A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq

(Extended Abstract)

Pieter Collins[1], Milad Niqui[1], and Nathalie Revol[2]

[1] Centrum Wiskunde & Informatica, The Netherlands
{Pieter.Collins,M.Niqui}@cwi.nl
[2] INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon, France
Nathalie.Revol@ens-lyon.fr

Abstract. We present a framework for the verification of the numerical algorithms used in Ariadne, a tool for analysis of nonlinear hybrid system. In particular, in Ariadne, smooth functions are approximated by Taylor models based on sparse polynomials. We use the Coq theorem prover for developing Taylor models as sparse polynomials with floating-point coefficients. This development is based on the formalisation of an abstract data type of basic floating-point arithmetic . We show how to devise a type of continuous function models and thereby parametrise the framework with respect to the used approximation, which will allow us to plug in alternatives to Taylor models.

1 Introduction

Hybrid systems are used to model phenomena involving both discrete and continuous state space. The usual way of verifying hybrid systems is to apply model checking, e.g. based on methods such as the predicate abstraction, and to devise a hybrid automata model. In practice this approach is helpful in verifying several types of properties of systems, however model checking can be prone to state explosion. A satisfactory solution is to combine model-checking with *theorem proving* in the logically rich environment of a theorem prover. This enables one to *validate* the correctness of model checking algorithms. Further on, one can *enhance* and simplify model checking algorithms by proving properties about classes of systems such as modular decomposition and symmetry reduction. This is the approach we take in this work: we use the Coq theorem prover, to implement and verify the algorithms of Ariadne [2] which is a tool for the analysis of nonlinear hybrid systems. Coq is an integrated theorem prover and a logical framework that is also capable of formalising mathematics. This means that in addition to the model checking of hybrid automata, we can use Coq to verify the algorithms for approximating elementary functions and for numerical differentiation and ultimately for solving ODEs. In the long run this will result in a hybrid systems analyser embedded in Coq. This analyser will enable the

user to state and prove properties by model checking or other techniques and it will use the Ariadne tool as an oracle for computations.

2 Ariadne

Ariadne is a tool written in C++ for the analysis of nonlinear hybrid systems. It has algorithms to analyse evolution, reachability and safety problems. The reachability analysis is performed using a predicate abstraction method [1] which is based on reducing the state space to a finite one. The new states correspond to a subdivision of the (bounded) state space into a grid. Each element of the grid is defined by guards, by reset values and by the solutions of the flow equation governing the dynamics of the system. For solving these equations, a numerical function calculus is implemented, based on Taylor models with floating-point coefficients: it yields approximations with guaranteed bounds on the errors.

Validating Ariadne Algorithms using Coq
Our aim is to provide a framework to validate Ariadne's analysis, in order to increase user's confidence in its results. We aim to do this by:

(1) verifying, in Coq, Ariadne's primitives for function calculus (Ariadne's kernel), which are based on Taylor models;
(2) verifying, in Coq, Ariadne's algorithms for reachability analysis.

In this article we describe how to tackle (1). This will be achieved by formalising the algorithms for basic operations on Taylor models, considered as sparse polynomials with floating-point coefficients. To this end we first describe our use of floating-point numbers and then demonstrate how to deal with the specific algorithms for basic arithmetic on floating-point Taylor models.

An important point is that we would like a framework that is easy to adapt to future changes in the Ariadne tool. For example, at the moment Ariadne uses Taylor models to approximate functions. Thus we would like our framework to be parametrised over the current specific implementation of Taylor models, but also over any other approximation model for smooth functions. To this end we present in Section 5 how to devise an abstract data type for computable functions that can be used for parametrising the main properties of the approximation models.

3 Floating-point Data Type

We describe our implementation of an abstract data type for the floating-point numbers: we introduce a type `Float` together with some of the basic IEEE 754 operations. Let us point out the fact that operations with various rounding mode are directly added to the signature of our abstract data type. For instance there are three addition operations $\oplus_u$, $\oplus_d$ and $\oplus_n$ for summing up two floating-point numbers using respectively upwards, downwards and to-nearest rounding. In the long run, we plan to extend our axiomatisation to a library compatible with

IEEE-754 specification of the basic operations $(+, \times, -, \div, \sqrt{})$ and the recommended elementary functions. At the moment we only handle operations that are necessary in formalising the proofs for Taylor models.

Our axiomatisation includes a type $\mathbb{F}$ together with the binary operations $\oplus_u, \oplus_d, \oplus_n, \otimes_u, \otimes_d, \otimes_n$, the unary operation $-$ and the constants $0_{\mathbb{F}}$ and $1_{\mathbb{F}}$. Note that currently $\mathbb{F}$ is not handled as a bounded set and that there is no reciprocal and no symbols for `NaN` and $\pm$`Inf`. (This will be added in the future.) The type $\mathbb{R}$ from the standard library of Coq is used as the type of idealised real numbers, and we assume the existence of an injection $inj : \mathbb{F} \longrightarrow \mathbb{R}$. The rest of the axioms will govern the arithmetic operations in $\mathbb{F}$ and are stated in terms of inj. For example the axioms for the addition operations are as follows.

```
Axiom F_0_u: ∀ x, x⊕_u 0_F = x.
Axiom F_0_d: ∀ x, x⊕_d 0_F = x.
Axiom F_0_n: ∀ x, x⊕_n 0_F = x.
Axiom F_0_inj: inj(0_F) = 0.
Axiom F_1_u: ∀ x y, | inj(x⊕_n y) - (inj(x) + inj(y)) | ≤
                                          inj(x⊕_u y) - (inj(x) + inj(y)).
Axiom F_1_d: ∀ x y, | inj(x⊕_n y) - (inj(x) + inj(y)) | ≤
                                          (inj(x) + inj(y)) - inj(x⊕_d y).
Axiom F_2_u: ∀ x y,  inj(x) + inj(y) ≤ inj(x⊕_u y).
Axiom F_2_d: ∀ x y,  inj(x⊕_d y) ≤ inj(x) + inj(y).
Axiom F_3: ∀ x y, | inj(x⊕_n y) - (inj(x) + inj(y)) | ≤
                                    1/2 ×  inj( (x⊕_u y) ⊖_u (x⊕_d y) ) .
```

Note that the absolute value and the $\leq$ are evaluated in $\mathbb{R}$ and $\ominus_u$ is defined in terms of the primitives $\oplus_u$ and $-$. Axioms for multiplication are similar.

We can *instantiate* this abstract data type by providing the required operations and proving the axioms, although this is not needed in current project. Nevertheless the concrete type in [4] provides instances for these axioms.

4 Taylor Models based on Floating-Point

Taylor models [6] provide an approximation of continuous functions by polynomials along with bounds or enclosures of the approximation and roundoff errors. In this work we are interested in basic scalar operations, addition, multiplication of Taylor models as these are the main ingredients of the Ariadne's function calculus kernel. The rigorous proof of correctness of these algorithms is given in [7]. The present work can be considered as a formalisation of [7] in Coq. There are several other developments of Taylor models in theorem provers: in Coq [8] using as coefficients the constructive real numbers; or in PVS [3] using rational interval arithmetic. Our work is different in that we use floating-point coefficients. As pointed out in [8] this makes the formalisation more cumbersome but it enables us to be as close to the actual Ariadne kernel as possible. As a final note, here we present the formalisation for univariate polynomials and one-dimensional functions. In the future we will extend this to any dimension using Coq's dependent data type for vectors.

First we need a type for univariate sparse polynomials with coefficients in $\mathbb{F}$, which is simply a record.

```
Record Sparse_polynom := { polynom:> list (nat*𝔽)
                         ; polynom_sparse: is_sorted polynom}.
```

Here `is_sorted`(l) is an inductively defined predicate l specifying that l (which is a list of pairs consisting of degrees and coefficients) is sorted with respect to to the degrees. The precise definition is given in Appendix A. A Taylor model is a pair composed of such a sparse polynomial and a floating-point error bound.

```
Record Taylor_model := { spolynom :> Sparse_polynom
                       ; error: 𝔽 }.
```

The function `eval_Taylor_model` that evaluates (recursively, by descending degrees) a Taylor model in a point in $\mathbb{R}$ is given in Appendix A.

Then the binary predicate `Models`$(\langle t, \epsilon\rangle, f)$ between a Taylor model and a function $f\colon \mathbb{R} \longrightarrow \mathbb{R}$ specifies that f is approximated by t with error ϵ on interval $[-1, 1]$ (cf. the *containment property* in [7]).

```
Definition Models (t: Taylor_model) (f:ℝ → ℝ) :=
    ∀x, -1 ≤ x ≤ 1 → |(eval_Taylor_model t x) - f(x)| ≤ inj(t.(error)).
```

We define the following basic operation on Taylor models
`scalar_mult_Taylor`$\colon \mathbb{F} \times$ `Taylor_model` $\longrightarrow$ `Taylor_model` ,
`monomial_mult_Taylor`: `Taylor_model` $\longrightarrow$ `Taylor_model` ,
`add_Taylor`: `Taylor_model` $\times$ `Taylor_model` $\longrightarrow$ `Taylor_model` ,
`mult_Taylor`: `Taylor_model` $\times$ `Taylor_model` $\longrightarrow$ `Taylor_model` ,
which respectively correspond with scalar multiplication by a float, multiplication by the unit monomial, addition and multiplication of Taylor models. As an example the algorithm, in Coq, for `add_Taylor` is given in Appendix B.

From now on, proving the correctness of the above algorithms with respect to the `Models` predicate is tantamount to formalising the correctness proofs in [7, § 4.4,§ 5.4,§ 6.4]. For example, the correctness theorem for addition will have the following type.

```
Theorem add_Taylor_correct:
    ∀ t1 t2 f1 f2, Models t1 f1 → Models t2 f2 →
                           Models (add_Taylor t1 t2) (λx⇒ f1(x)+ f2(x)).
```

5 Data type for Abstract Function Models

Taylor models provide useful approximations for continuous real functions. However, especially for dealing with transcendental functions, other approximation models may lead to better relative errors [5]. For this purpose we intend to axiomatise a type of function approximations for continuous or even measurable functions. Such a type will be augmented with operations and predicates for specifying the domain, the *evaluation* of these function models on computable

real numbers, floating-point numbers and interval values, and the *composition* of two function models, or of a computable function with a function model.

The goal is to use only operations and properties of the abstract models when performing higher-level operations, such as solving algebraic equations and integrating differential equations. If this can be realised, then the generic algorithms can be implemented in a way which is provably correct for all instantiations.

6 Conclusion

In hybrid systems, different phases of modelling deal with different mathematical objects. Table below shows the objects, from left to right, in the increasing order of approximation (uncountable, countable and finite in the first line).

Real number	Rational Interval	Floating-point number
Real Function	Taylor Model	Taylor Approximation

Since we base our validation work on a rich logical environment where all these objects can coexist (we provide the missing ones), we can reason about all these facets in combination and interaction with each other: the actual C++ functions working with the floating-point numbers are translated to Coq and their behaviour is validated using the idealised notion of real numbers $\mathbb{R}$. We apply this methodology both to the kernel of the Ariadne tool (current work) as well as to the higher level algorithms for analysing hybrid systems (future work).

We focused here on the Ariadne tool, but the theories that we develop in Coq deal with fundamental objects such as floating-point numbers and Taylor models, thus they could be applied to the implementation of other tools in Coq.

References

1. R. Alur, T. Dang, and F. Ivancic. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embedded Comput. Syst.*, 5(1):152–199, 2006.
2. A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. L. Sangiovanni-Vincentelli. Ariadne: a framework for reachability analysis of hybrid automata. In *Proc. MTNS 2006*, pages 1259–1267, Kyoto, Japan, July 2006.
3. F. J. Cháves Alonso. *Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves*. PhD thesis, École Normale Supérieure de Lyon, Sept. 2007.
4. M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In R. J. Boulton and P. B. Jackson ed., *Proc. TPHOLS 2001*, pages 169–184. LNCS 2152, Springer, 2001.
5. C. Lauter, S. Chevillard, M. Joldes, and N. Jourdan. *Users' manual for the Sollya tool*. LIP, Sept. 2008. http://sollya.gforge.inria.fr/.
6. K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *Int. J. Pure Appl. Math.*, 4(4):379–456, 2003.
7. N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *J. Log. Algebr. Program.*, 64(1):135–154, 2005.
8. R. Zumkeller. Formal Global Optimisation with Taylor Models. In *Proc. IJCAR 2006, U. Furbach and N. Shankar ed.*, pages 408–422. LNCS 4130, Springer, 2006.

A Auxiliary Coq Types and Terms for Taylor Model

```
Inductive is_sorted {A:Type} : list (nat*A) → Prop :=
   | is_sorted_nil : is_sorted nil
   | is_sorted_one : ∀ m a, is_sorted (cons (m,a) nil)
   | is_sorted_cons : ∀ m (a:A) xs a0, head xs = Some a0 → (m<fst a0) →
                                is_sorted xs → is_sorted (cons (m,a) xs).

Fixpoint eval_polynom  (p: list nat*𝔽) (x:R) {struct p} : R :=
    match p with
    | nil ⇒ 0
    | fn :: p0 ⇒  inj_R(snd fn)*x^(fst fn) + eval_polynom p0 x
    end.

Definition eval_Sparse_polynom (sp : Sparse_polynom) (x:R) : R  :=
    match sp with
    | Build_Sparse_polynom p H ⇒ eval_polynom p x
    end.

Definition eval_Taylor_model (t: Taylor_model) (x:R) : R  :=
    match t with
    | Build_Taylor_model sp _ ⇒ eval_Sparse_polynom sp x
    end.
```

B Addition of Two Taylor Models in Coq

```
(* coefficients *)
Function pre_merge (κ1:𝔽→𝔽) (κ2:𝔽→𝔽→𝔽) (pp:list(nat*𝔽) * list(nat*𝔽))
                   {measure (λll⇒ length (fst ll) + length (snd ll)) pp}
                                                          : list (nat*𝔽) :=
    match pp with
    | (nil, nil) ⇒ nil
    | (nil, fn2 :: p2') ⇒ fn2 :: p2'
    | (fn1 :: p1', nil) ⇒ fn1 :: p1'
    | (fn1 :: p1', fn2 :: p2') ⇒
       match lt_eq_lt_dec (fst fn1) (fst fn2) with
       | inleft (left _) ⇒
             (fst fn1, κ1(snd fn1)) :: pre_merge κ1 κ2 (p1',(fn2 :: p2'))
       | inleft (right _) ⇒
           (fst fn1, κ2 (snd fn1) (snd fn2)) :: pre_merge κ1 κ2 (p1',p2')
       | inright _ ⇒
             (fst fn2, κ1(snd fn2)) :: pre_merge κ1 κ2 ((fn1 :: p1'),p2')
       end
    end.

Lemma pre_merge_sorted: ∀ κ1 κ2 (p1 p2: list (nat*𝔽)),
     is_sorted p1 → is_sorted p2 → is_sorted (pre_merge κ1 κ2 (p1,p2)).
```

```
Definition pre_merge_add_near (p1 p2: list (nat*𝔽)) : list (nat*𝔽) :=
                                                  pre_merge id ⊕ₙ (p1,p2).

Definition pre_merge_add_near_sorted:= pre_merge_sorted id ⊕ₙ.

Definition merge_add_near (sp1 sp2: Sparse_polynom) : Sparse_polynom :=
    match sp1, sp2 with
    | Build_Sparse_polynom p1 H1, Build_Sparse_polynom p2 H2 ⇒
       Build_Sparse_polynom (pre_merge_add_near (p1,p2))
                            (pre_merge_add_near_sorted p1 p2 H1 H2)
    end.

(* error *)
Function pre_merge_error (pp:list(nat*𝔽) * list(nat*𝔽))
                      {measure (λll⇒ length (fst ll) + length (snd ll))}
                                                          : list (nat*𝔽) :=
    match pp with
    | (nil, nil) ⇒ nil
    | (nil, fn2 :: p2') ⇒ nil
    | (fn1 :: p1', nil) ⇒ nil
    | (fn1 :: p1', fn2 :: p2') ⇒
       match lt_eq_lt_dec (fst fn1) (fst fn2) with
       | inleft (left _) ⇒ pre_merge_error (p1',(fn2 :: p2'))
       | inleft (right _) ⇒
                    (fst fn1, (snd fn1 ⊕ᵤ snd fn2)⊖ᵤ(snd fn1 ⊕_d snd fn2)
                    :: pre_merge_error (p1',p2')
       | inright _ ⇒ pre_merge_error ((fn1 :: p1'),p2')
       end
    end.


Definition sum_add_up := fold_right (λnf⇒ ⊕ᵤ(snd nf)) 0_𝔽.

Definition error_add (t1 t2: Taylor_model) : 𝔽 :=
    t1.(error) ⊕ᵤ
    ( t2.(error) ⊕ᵤ
      sum_add_up (pre_merge_error (t1.(spolynom),t2.(spolynom)))).


(* addition of Taylor models *)
Definition add_Taylor (t1 t2: Taylor_model) : Taylor_model :=
    Build_Taylor_model (merge_add_near t1.(spolynom) t2.(spolynom))
                       (error_add t1 t2).
```