A library of Taylor models for PVS automatic proof checker

Francisco Cháves, Marc Daumas

A library of Taylor models for PVS automatic proof checker

Francisco Cháves[‡] and **Marc Daumas**[§]

Laboratoire de l'Informatique du Parallélisme
UMR 5668 CNRS–ENS de Lyon–INRIA
email: Francisco.Jose.Chaves.Alonso@ENS-Lyon.Fr

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
UMR 5506 CNRS–UM2
email: Marc.Daumas@LIRMM.Fr

Visiting Laboratoire de Physique Appliquée et d'Automatique
EA 3679 UPVD
email: Marc.Daumas@Univ-Perp.Fr

Abstract. We present in this paper a library to compute with Taylor models, a technique extending interval arithmetic to reduce decorrelation and to solve differential equations. Numerical software usually produces only numerical results. Our library can be used to produce both results and proofs. As seen during the development of Fermat's last theorem reported by Aczel (1996), providing a proof is not sufficient. Our library provides a proof that has been thoroughly scrutinized by a trustworthy and tireless assistant. PVS is an automatic proof assistant that has been fairly developed and used and that has no internal connection with interval arithmetic or Taylor models. We built our library so that PVS validates each result as it is produced. As producing and validating a proof, is and will certainly remain a bigger task than just producing a numerical result our library will never be a replacement to imperative implementations of Taylor models such as Cosy Infinity. Our library should mainly be used to validate small to medium size results that are involved in safety or life critical applications.

Keywords: PVS, program verification, interval arithmetic, Taylor models.

1. Introduction

Taylor models, see for example (Makino and Berz, 2003) and references herein, have recently emerged as a nice and convenient way to reduce decorrelation in interval arithmetic (Moore, 1966; Neumaier, 1990; Jaulin *et al.*, 2001). Taylor models are even more attractive when one solves initial value problems ODEs as they provide a validated built-in integration operator.

Yet, it is now beyond doubt that programs and libraries contain bugs, no matter how precisely they have been specified and how thoroughly they have been tested (Rushby and von Henke, 1991; Ross, 2005). As a consequence, the highest Common Criteria Evaluation Assurance Level, EAL 7[1], has only been awarded so far to products that provide validation using a formal tool, specifically an automatic proof checker in first or higher order logic.

We present here our library of Taylor models in PVS (Owre *et al.*, 1992). Working with an automatic proof checker, we had to manage two tasks. The first task was to create a data type

[‡] This material is based on work supported by the *Mathlogaps* (Mathematical Logic and Applications) project, an Early Stage Research Training grant of the European Union.

[§] This work has been partially supported by PICS 2533 from the French National Center for Scientific Research (CNRS).

[1] `http://niap.nist.gov/cc-scheme/`.

and operations on this new type to allow users to define and evaluate expressions using Taylor models. The second task was to provide proofs that each operator is correct and a strategy to recursively analyze compound expressions. Both tasks rely on the recently published library on interval arithmetic for PVS (Daumas *et al.*, 2005). As many mathematical developments are not yet available in PVS, we also had to develop an extended library on polynomials and prove a few theorems of analysis and algebra.

Our library on Taylor models can be used to derive quickly more or less accurate bounds. For example, users of formal tools have to provide proofs that radicals are non negative for all expressions using square roots. Some proofs use intricate analysis but most of them are very simple and interval arithmetic or low degree evaluations with Taylor models can produce appropriate proofs. Our library can also be used to expertly derive computer validated proofs of difficult results through an expert use of Taylor models.

The library will be available freely on the Internet as soon as it is stable. Side developments are integrated as they are produced to NASA Langley PVS libraries[2]. Meanwhile, all files can be retrieved from the author's website.

`http://perso.ens-lyon.fr/francisco.jose.chaves.alonso/pvs-files/`

1.1. Working with an automatic proof checker

Software is used extensively for a wide array of tasks. Some pieces of software should never fail. The ones used by transportation means (planes, buses, cars...), for medical care (controlling pumps, monitors, prescriptions...) or in the army (parts of weapons, alarms...) belong to the fast lengthening list of life or safety critical applications. A mindless modification of one parameter reportedly caused human losses in the *Instituto Oncologico Nacional* on Panama where eight people died and twenty others were hurt (Gage and McCormick, 2004). Many lethal and costly failures (Information Management and Technology Division, 1992; Lions and others, 1996) show beyond reasonable doubts that traditional software verification is not sufficient to guarantee correct behavior.

PVS[3] (Prototype Verification System) by Owre *et al.* (1992; 2001a; 2001b) is one environment for the development and analysis of formal specifications that allows the elaboration of theories and proofs. The system deals with theories where users develop definitions, axioms and theorems. To verify that theorems are correct, PVS uses a typed higher order logic language where new types are defined from a list of basic types including booleans, natural numbers, integers... The type system allows the definition of functions, registers, tuples and abstract data types.

PVS uses *predicate subtypes*, subtypes where all objects satisfy a given predicate. For example $\{x : \mathrm{real} | x \neq 0\}$ is the set of non–zero reals. Subtype predicates are used for operations that aren't defined for all possible inputs. This restriction is therefore visible in the signature of the operation. For example the division is an operation of real numbers such that the type of the denominator is a real number different from zero. As a result, all functions of PVS are total in the sense that the domain and the signature must exclude explicitly any input where a function could not be defined.

As predicates used by the system to define types are arbitrary, type verification is undecidable and it usually generates proofs obligations named *type correctness conditions* (TCCs). Users have to provide proofs of generated TCCs with the help of PVS.

In PVS the λ operator defines anonymous functions. Expression $\lambda x.e$ is a function that has parameter x and returns expression e. For example, the function that returns 0 for any value of its single parameter could be defined as $\lambda x.0$ and identity function that returns the same element that is given as parameter is $\lambda x.x$. Function λ `k : nat. if k = 0 then 1 else 0` is the sequence that for input 0, returns 1, and returns 0 for any other input.

[2] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[3] `http://pvs.csl.sri.com/`.

Nowadays, systems such as PVS are fully able to certify that programs are corrects (Ross, 2005) but programmers scarcely use them. Providing a formal proof of correct behavior is a difficult task, it requires a specific training and user interfaces of proof assistants are of little help for all the work that is not done automatically. Hope is that as more and more work is done automatically, users will need only limited interactions with automatic proof checkers down to the point where no interaction is required at all. This trend was recently coined as *invisible formal methods* (Tiwari *et al.*, 2003).

1.2. A few words about interval arithmetic

In interval arithmetic scalar variables x are replaced by pairs (a, b) with the semantic that x lies in the interval $[a, b]$. Later on, we compute bounds rather than values. We use operators commonly found in programming languages such as addition, subtraction, multiplication and so on (Jaulin *et al.*, 2001).

$$\begin{array}{rcl} [a,b]+[a',b'] &=& [a+a', b+b'] \\ [a,b]-[a',b'] &=& [a-b', b-a'] \\ c\cdot[a,b] &=& [c\cdot a, c\cdot b] \qquad c \geq 0 \\ [a,b]\cdot[a',b'] &=& [\min\{aa', ab', ba', bb'\}, \max\{aa', ab', ba', bb'\}] \end{array}$$

Working with automatic proof checkers, we convert operations into properties (Daumas *et al.*, 2005).

$$\text{For all } x \in [a,b],\ y \in [a',b'] \text{ and } c \in \mathbb{R} \begin{cases} x+y &\in [a,b]+[a',b'] \\ x-y &\in [a,b]-[a',b'] \\ c\cdot x &\in c\cdot[a,b] \\ x\cdot y &\in [a,b]\cdot[a',b'] \end{cases}$$

Decorrelation is a problem intrinsic to interval arithmetic. There is decorrelation on interval evaluation of any expression where one or more variables appear more than once. For example, the most simple scalar expression

$$x - x$$

where $x \in [0, 1]$, is replaced in interval arithmetic by

$$[0, 1] - [0, 1] = [-1, 1].$$

Everyone agrees that $x - x$ lies in the interval $[0, 0]$ but interval arithmetic produces the correct but very poor $[-1, 1]$ interval. Decorrelation and other problems lead interval arithmetic to overestimate the domain of results. Techniques are used intensively to produce constrained results.

One of such techniques is based on Taylor's theorem with Lagrange remainder where f is n times continuously derivable between x_0 and x, f is $n+1$ times derivable strictly between x_0 and x and $0 < \theta < 1$.

$$\begin{array}{rcl} f(x) &=& f(x_0) + (x-x_0)f'(x_0) + \frac{(x-x_0)^2}{2!}f''(x_0) \\ &+& \cdots + \frac{(x-x_0)^n}{n!}f^{(n)}(x_0) \\ &+& \frac{(x-x_0)^{n+1}}{(n+1)!}f^{(n+1)}(x_0 + (x-x_0)\theta) \end{array}$$

Adapting Taylor's theorem to interval arithmetic, we obtain the formula below (Daumas *et al.*, 2005) for x and x_0 in I.

$$\begin{array}{rcl} f(x) &\in& f(x_0) + (I-x_0)f'(x_0) + \frac{(I-x_0)^2}{2!}f''(x_0) \\ &+& \cdots + \frac{(I-x_0)^n}{n!}f^{(n)}(x_0) \\ &+& \frac{(I-x_0)^{n+1}}{(n+1)!}f^{(n+1)}(I) \end{array}$$

Using Taylor's theorem was appropriate in (Daumas *et al.*, 2005) but it has many drawbacks:

- It is difficult to hide the use of Taylor's theorem in order to provide *invisible formal methods*. This is due to the large number of quantities involved in instantiating the theorem in its generic form. Progress has been achieved by Muñoz after the publication of Daumas *et al.*.
- To use Taylor's theorem, one has to express the derivatives of function f.
- For large expressions, f alone might be too large to be expressed in PVS.

Taylor models presented in the rest of this text overcome all the previous drawbacks to the price of a less accurate approximation. We have developed a set operations for PVS that includes addition, negation, scalar multiplication, multiplication, reciprocal and exponential. We present our developments in PVS, first quickly on polynomial functions and then on Taylor models. We finish with concluding remarks and a few toy examples.

2. Implementing polynomials in PVS

For the implementation of polynomials we considered a finite list of monomial functions, a finite sequence of coefficients and an infinite power series with finite support. Finite lists or sequences usually imply the construction of a new inductive type *à la* Coq[4] (Bertot and Casteran, 2004). We implemented polynomials as power series with finite support. This scheme is appropriate for a proof system like PVS and is compatible with NASA series libraries[5].

Working with sequences of coefficients rather than monomial functions means that we need the `powerseries` function to evaluate polynomial P on input x. It also means that some theorems can be established on finite support series rather than polynomial functions.

2.1. Finite support series

Our implementation of polynomials is outlined in Figure 1. It mostly describes mathematical objects (definition, function, theorems...) with common words except for the notions introduced in Section 1.1

We define predicate `finite_support (a,N)` just after the preamble. Addition of sequences was already defined and is imported from previous work in the preamble. We had to define a product operator and a composition operator. The first operator applies to generic series. The second operator requires that the first sequences a returns zero for indices above input d.

In the second half of Figure 1 we proved that negation, addition, multiplication by a scalar, multiplication and composition return finite support series provided (both) inputs are finite support series. We also proved that Cauchy's product is meaningful for finite support series. The meaning of composition can only be assessed in regard to polynomial functions.

2.2. Polynomial

As we have mentioned earlier, we use `polynomial (a, n)` function to create a power series from finite support sequence a based on `powerseries(a)(x)(N)` function implemented in previous work. Extended results on polynomial functions are presented in Figure 2 based on NASA libraries.

$$\text{polynomial}(a, n)(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

[4] See for example `http://www.lfcia.org/staff/freire/phd-gilberto/gilberto_phd_html/`.

[5] http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html.

```
finite_support: THEORY
 BEGIN

  IMPORTING series@series, reals@sqrt, series@power_series

  a, b, c: VAR sequence[real]
  N, M, L, n, m, l, i, j: VAR nat
  x: VAR real

  finite_support(a: sequence[real], N: nat): boolean =
      ∀ (n: nat): n > N ⇒ a(n) = 0

  cauchy(a, b: sequence[real])(n: nat): real =
      Σ(0, n,
            λ (k: nat):
              IF n ≥ k
                THEN a(k) × b(n − k)
              ELSE 0
              ENDIF)

  comp(a, b: sequence[real], d: nat): RECURSIVE sequence[real] =
    IF d = 0
      THEN (λ n: IF n = 0 THEN a(0) ELSE 0 ENDIF)
    ELSE LET c = (λ n: IF n = d THEN 0 ELSE a(n) ENDIF) IN
           a(d) × pow(b, d) + comp(c, b, d − 1)
    ENDIF
     MEASURE d

  neg_fs: LEMMA
    finite_support(a, N) ⇒ finite_support(−a, N)
  add_fs: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ∧ L ≥ max(N, M) ⇒
     finite_support(a + b, L)
  scal_fs: LEMMA
    finite_support(a, N) ⇒ finite_support(x × a, N)
  finite_support_mult: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ⇒
     finite_support(cauchy(a, b), N + M)
  finite_support_cauchy: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ⇒
     series(a)(N) × series(b)(M) =
      series(cauchy(a, b))(N + M)
  finite_support_comp: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ⇒
     finite_support(comp(a, b, N), N × M)

 END finite_support
```

Figure 1. Abridged and reordered theory on finite support series (see file `finite_support.pvs`)

```
polynomials_ext: THEORY
 BEGIN

  IMPORTING finite_support, trig_fnd@polynomial_deriv

  a, b, d: VAR sequence[real]
  n, N, M, L: VAR nat
  c: VAR real
  x, y: VAR real

  fs_powerseq: LEMMA
    finite_support(a, N) ⇒ finite_support(powerseq(a, x), N)

  fs_condition: LEMMA
    finite_support(a, N) ⇒
     (∀ (i: posnat): a(N + i) = 0)

  scal_polynomial1: LEMMA
    x × polynomial(a, N) = polynomial(x × a, N)

  powerseries_polynomial: LEMMA
    polynomial(a, n)(x) = powerseries(a)(x)(n)

  polynomial_zero: LEMMA
    polynomial((λ (n: nat): 0), N)(x) = 0

  mul_polynomial: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ⇒
     polynomial(a, N)(x) × polynomial(b, M)(x) =
      polynomial(cauchy(a, b), N + M)(x)

  pow_polynomial: LEMMA
    finite_support(a, N) ⇒
     polynomial(a, N)(x) ˆ∧ n =
      polynomial(pow(a, n), n × N)(x)

  comp_polynomial: LEMMA
    finite_support(a, N) ∧ finite_support(b, M) ⇒
     polynomial(a, N)(polynomial(b, M)(x)) =
      polynomial(comp(a, b, N), N × M)(x);

  geom_polynomial: LEMMA
    (1 − x) × Σ(0, N, λ (i : nat) : x ˆ∧ i) =
     1 − x ˆ∧ (N + 1)

 END polynomials_ext
```

Figure 2. Abridged extensions to the theory on polynomial (see file `polynomials_ext.pvs`)

We proved in this file that Cauchy's multiplication applies to finite support series as well as polynomial functions. We also proved that the series obtained from composing two finite support series as defined in Section 2.1 defines the same polynomial function as the one that would be obtained by composing the polynomial functions associated to the two initial series.

Technical results are also presented in this file to provide more insights to our development.

3. Taylor models

Taylor models (Makino and Berz, 2003) are pairs $t = (P, I)$ where P are polynomial functions of fixed degree N and I are intervals. N is a constant that cannot be changed during the evaluation of expressions. In PVS, pairs are defined using components between `(#` and `#)`. Components can be addressed independently using quotes `‘`, that are `t‘P` and `t‘I`.

Taylor model t is a correct representation of function f if it satisfies the `containment` predicate stated Figure 3,

$$\forall x \in J \qquad f(x) - t'P(x) \in t'I$$

where J is usually $[-1, 1]$.

Our first task was to define operations on Taylor models. Addition, negation and multiplication by a scalar are straight forward and can be read directly from Figure 3. Naive multiplication of Taylor models creates polynomials of degree $2N$. The high order terms of the polynomials must be truncated and are accounted for in the interval part.

The `inv` reciprocal operator uses the following equality where $r \in I$, $p(0) \neq 0$ and $p(x)$ has the same sign as $p(0)$.

$$\frac{1}{p(x)+r} = \frac{1}{p(0)} \cdot \frac{p(x)}{p(x)+r} \cdot \frac{1}{1 - \left(1 - \frac{p(x)}{p(0)}\right)} \tag{1}$$

We define $q(x) = 1 - \frac{p(x)}{p(0)}$ and we expand the last fraction of (1) using the geometrical series $\sum_{i=0}^{N} q^i$ truncated to keep only a polynomial of degree N.

Decorrelation forbids to evaluate the penultimate fraction of (1) directly and we defined a new operator based on the lower bound and the upper bound of $I/p(J)$ that returns directly

$$\left[\frac{1}{1 + \frac{1}{lb'(I/p(J))}}, \frac{1}{1 + \frac{1}{ub'(I/p(J))}} \right].$$

This operator cannot be replaced by a direct implementation of

$$\frac{1}{1 + p(J)/I} \quad \text{or} \quad \frac{1}{1 + \frac{1}{I/p(J)}}$$

because I usually contains 0 preventing anyone to use it as a divisor.

We also implemented the exponential of Taylor models using the following equality where $r \in I$ and $\hat{e}^x$ is a rational approximation of e^x.

$$e^{p(x)+r} = \hat{e}^{p(0)} \cdot e^{p(x)-p(0)} \cdot \frac{e^{p(0)}}{\hat{e}^{p(0)}} \cdot e^r$$

The polynomial part of the result is obtained by developing and truncating the exponential series composed with $p(x) - p(0)$. The interval part is set accordingly to account for all discarded quantities.

The five `_sharp` lemmas of the second part of Figure 3, show that the `containment` predicate is preserved by our operators. It means that we can deduce properties from evaluations of expressions using Taylor models.

```
taylor_model[N: nat, (IMPORTING interval@interval) domInterval: Interval]: THEORY
 BEGIN

  tm: TYPE = [#P: fs_type, I: Interval#]

  tm_equal: AXIOM
    t = u ≡
     polynomial(t‘P, N) = polynomial(u‘P, N) ∧ t‘I = u‘I;

  t + u : tm: tm =  (#P := t‘P + u‘P, I := t‘I + u‘I#);
  −t: tm = (#P := −t‘P, I := −t‘I#);
  c × t: tm = (#P := c × t‘P, I := ⟦c⟧ × t‘I#)
  t × u: tm = (#P := trunc(cauchy(t‘P, u‘P), N), I := ... #)
  inv(t: {t: tm | same condition as below tm_inv_sharp }):
        tm = (#P := ... , I := ... #)

  containment(f: [domIntervalType → real], t: tm): bool =
      ∀ xu: (f(xu) − polynomial(t‘P, N)(xu)) ## t‘I

  tm_add_sharp: LEMMA
    containment(f, t) ∧ containment(g, u) ⇒ containment(f + g, t + u)
  tm_scal_sharp: LEMMA
    containment(f, t) ⇒ containment(x × f, x × t)
  tm_neg_sharp: LEMMA
    containment(f, t) ⇒ containment(−f, −t)
  tm_mult_sharp: LEMMA
    containment(f, t) ∧ containment(g, u) ⇒ containment(f × g, t × u)
  tm_inv_sharp: LEMMA
    ∀ (f: [domIntervalType → nzreal],
         t: {t: tm |
                        t‘P(0) ≠ 0 ∧
                         (t‘I/intervalFromRealSeq(t‘P, N))‘lb ≠ 0 ∧
                          (t‘I/intervalFromRealSeq(t‘P, N))‘ub ≠ 0 ∧
                           (t‘I/intervalFromRealSeq(t‘P, N)) > −1}):
      (∀ xu:
           polynomial(t‘P, N)(xu) ≠ 0 ∧
            (f(xu) − polynomial(t‘P, N)(xu))/polynomial(t‘P, N)(xu)
             ≠ 1
             ∧
             polynomial(λ (i: nat):
                             IF i = 0 THEN 0 ELSE −t‘P(i)/t‘P(0) ENDIF,
                           N)
                         (xu)
              ≠ 1)
        ∧ Zeroless?(⟦t‘P(0)⟧) ∧ Zeroless?( ... )
          ∧ Zeroless?(intervalFromRealSeq(t‘P, N)) ∧ containment(f, t)
        ⇒ containment(1/f, inv(t))

 END taylor_model
```

Figure 3. Abridged and reordered theory on Taylor models (see file `taylor_model.pvs`)

```
example: THEORY
 BEGIN

  IMPORTING tm_exp[5, 5, (#lb := −1, ub := 1#)]

  ch(x: tm): tm =
       (1/2) × (exp(x) + exp(−x))

  sh(x: tm): tm =
       (1/2) × (exp(x) + −exp(−x))

  seq_px: fs_type =
       λ (n: nat): IF n = 1 THEN 1/1000 ELSE 0 ENDIF

  tm_x: tm = (#P := seq_px, I := ⟦0⟧#)

  example1: tm = ch(2 × tm_x) × sh(3 × tm_x)

 END example
```

Figure 4. A toy example of Taylor models (see file `example.pvs`)

In addition to prove mathematical theories, PVS provides a *ground evaluator*. It is an experimental feature of PVS that enables the animation of functional specifications. To evaluate them, the ground evaluator extracts Common Lisp code and then evaluates the code generated on PVS underlying Common Lisp machine.

Uninterpreted PVS functions can be written in Common Lisp. PVS only trusts Lisp codes generated automatically from PVS functional specifications, then one can not introduce inconsistencies in PVS. However, codes are not type-checked by PVS and can break inadvertently.

PVSio[6] is a PVS package developed by Muñoz that extends the ground evaluator with a predefined library including imperative programming language features. PVSio loads in emacs interface using `M-x load-prelude-library PVSio` and then executes with `M-x pvsio`.

4. Toy example, concluding remarks and future work

Figure 4 show how easily we can define expressions. PVSio is used to evaluate Taylor model expressions and Figure 5 shows the polynomial and interval parts of the Taylor model of degree 5 of

$$ch\left(2 \cdot \frac{x}{1000}\right) \cdot sh\left(3 \cdot \frac{x}{1000}\right) = 3 \cdot \frac{x}{1000} + \frac{21}{2} \cdot \left(\frac{x}{1000}\right)^3 + \frac{521}{40} \cdot \left(\frac{x}{1000}\right)^5 + r$$

with

$$r \in 5150892483 \cdot 10^{-28} \cdot [-1, 1]$$

Coefficients are obtained from expressions `example1‘P(0)`, `P(1)` down to `P(5)`. The interval part is `example1‘I`.

To conclude, we would like to say that they have three goals in presenting this report:

- **Present an accurate report of the work involved including the training of a PhD student to PVS**. Though this development is significant, PVS validated projects can be achieved in a reasonable time-frame provided appropriate tutoring is available.

[6] http://research.nianet.org/ munoz/PVSio

```
<PVSio> example1‘P(0);
==>
0
<PVSio> example1‘P(1);
==>
3/1000
<PVSio> example1‘P(2);
==>
0
<PVSio> example1‘P(3);
==>
21/2000000000
<PVSio> example1‘P(4);
==>
0
<PVSio> example1‘P(5);
==>
521/4000000000000000
<PVSio> example1‘I;
==>
(# lb := -1996666003792920908077809559596469417049924988435
675424891258279277724682576954162797931053521035846 47/38763
49604747870233132233643700469577302245603256513727240130672
32422339563866364336668581220000000000000000000000000000,
ub := 1996666003792920908077809559596469417049924988435
67542489125827927772468257695416279793105352103584647/38763
49604747870233132233643700469577302245603256513727240130672
32422339563866364336668581220000000000000000000000000000 #)
```

Figure 5. Trace of our toy example of Taylor models

- **Provide a simple tutorial to our library on Taylor models**. Readers should be able to start validating their own results as soon as they have finished reading this paper.
- **Offer a first easy step to the usage of automatic proof checkers**. It is always frustrating to spend time on questions than can easily be solved by more or less elaborate techniques. As we now provide a PVS library for interval arithmetic and for Taylor models, one should be able to answer quickly to most of the easy questions about round-off, truncation and modeling errors. Concentrating only on intricate questions is rewarding from the academia and ensures financial support from the industry.

In the future, we will implement more operations on Taylor models like square root, sine, cosine, and arctangent. We will also create PVS strategies to hide more and more details of Taylor models to users. Our main goal remains to help provide *invisible formal methods*.

Acknowledgements

The authors wish to express all their gratitude to Cesar Muñoz from the National Institute of Aerospace in Hampton, Virginia, for his tutoring and help in the many manipulations around PVS. The authors would also like to thank NASA Langley Research Center for its free PVS Class held on May 24-27, 2005.

References

Amir D. Aczel. *Fermat's last theorem: unlocking the secret of an ancient mathematical problem.* Four Walls Eight Windows, 1996.

Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th Symposium on Computer Arithmetic*, Cape Cod, Massachusetts, 2005.

Debbie Gage and John McCormick. We did nothing wrong. *Baseline*, 1(28):32–58, 2004.

Information Management and Technology Division. Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992.

Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied interval analysis.* Springer, 2001.

Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.

Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003.

Ramon E. Moore. *Interval analysis.* Prentice Hall, 1966.

Arnold Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 748–752, Saratoga, New-York, 1992. Springer-Verlag.

Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Language Reference.* SRI International, 2001. Version 2.4.

Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS System Guide.* SRI International, 2001. Version 2.4.

Philip E. Ross. The exterminators. *IEEE Spectrum*, 42(9):36–41, 2005.

John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15, New Orleans, Louisiana, 1991.

Ashish Tiwari, Natarajan Shankar, and John Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.