Automatic and Verifiable Synthesis of Implementations from Mathematical Models

Jacques Carette Alexandre Korobkine Mark Lawford

Department of Computing and Software, McMaster University, Hamilton, Canada

carette@mcmaster.ca, korobkao@mcmaster.cs, lawford@mcmaster.ca

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming

General Terms Model Driven Development, Code Generation, Verification, Computer Algebra

Keywords Computer Algebra, Theorem Proving

Abstract

Many applications in image processing, control systems and other areas, have very well understood mathematical models. Optimization problems, in particular, are a class of (implicit) models which is particularly useful. When faced with the task to develop such an application, a software engineer aware of best practices might use a computer algebra system to compute some problem-specific quantities which are then put into a simulation environment (such as Matlab), and ultimately translated to code. Such a process requires the development of multiple versions of the mathematical model, often by hand, in tools with widely varying levels of formalism, which are generally susceptible to soundness problems. Moreover, it may be very difficult to decide whether the various models are in fact equivalent. Through an extended example of a multidimensional Newton's Method, we demonstrate how computer algebra and theorem proving systems can be used to largely automate this process. We compute problem-specific quantities, including higher-order ones, verify their correctness, and automatically generate code which is verifiably correct. The process combines untrusted components with trusted ones so that failure in an untrusted component is always detected.

1. Introduction

The Model Driven Development (MDD) methodology can afford developers significant advantages such as traceability of requirements, executability of the model, platform independence, etc. Selic states that for a model to be useful it must have the five key attributes of abstraction, understandability, accuracy, predictiveness, and inexpensive [Selic 2003]. An additional attribute required to realize the full benefits of MDD is the ability to manipulate and calculate new results from the models. A significant problem is that most models currently are neither easy to manipulate (if they can be manipulated at all), nor can they be used to derive new results.

While current technology is still unable to manipulate and calculate new results with models of general software applications, there are significant classes of problems in areas like image processing, control systems design, mechanical design, etc., which have well-understood mathematical models. These can be represented exactly (although not always in *closed-form*) via the use of a computer algebra system (CAS). For certain classes of problems, the mathematical models may even be coaxed into closed-form solutions. Even from implicit representations, numerical approximation code is frequently (and easily) derivable. These mathematical models are usually quite abstract (from a computational point-of-view) as they usually correspond to the physical model of the underlying problem. But that is also a strength, as they also tend to be accurate descriptions of the problem, and can be used to predict the behavior of the system being modeled. In these cases, one can profitably use the symbolic computation and code generation capabilities of a CAS, such as Maple, to perform model driven development. For example, the mathematical model of a target image for a vision processing system can be modeled in Maple and then manipulated to derive an efficient solver based upon Newton's method [Korobkine 2002]. Similarly, symbolic methods can be profitably applied to control systems problems [Munro 1999].

Mathematical models described in a CAS clearly possess the first four of Selic's key attributes while his fifth - "inexpensive" should soon be achieved – engineers are usually already using mathematical models, but they frequently have to be taught to work at the model-level rather than relying solely on simulation. As we will see through our example, some models are easily manipulated in a CAS, so that a proper MDD methodology can be used with them.

However, as pointed out in the literature [Harrison and Théry 1998, Kramer 2007, Wester 2000b], CAS typically have soundness bugs. For example, consider finding the minimum of the function $f(x) = 2x^2 + x^4 + sin(x + 1)$. When Maple 13[1] attempts to solve this symbolically (using the command `minimize(f, x, location)`), it produces the clearly incorrect result $+\infty$; asking for a numerical solution, we however get the "correct" answer -0.163103667753977111. Such soundness concerns in CAS, as well as limitations on the kinds of computations that automated theorem provers (ATP) can perform, have led to efforts to integrate CAS with theorem provers such as HOL, Coq and PVS (e.g., [Harrison and Théry 1998, Delahaye and Mayero 2005, Adams et al. 2001]). Such works typically focus on using an ATP to check the soundness CAS results and/or using the CAS to help the ATP with proofs. In our case, we are interested in performing provably correct model driven development. In other words, we want to

[1] Similar problems can be found for all CASs.

verify not just the operations performed by the CAS, but also the code generated from the model. Checking the soundness of the algebraic manipulations and generating correctness proofs are the "necessary evils" to verify and validate our implementation.

The methodology we use is to first create a general MDD environment for solving a classes of problems which can be well-modeled using classical mathematics. We then specialize this to a specific class, and then further specialize to a particular method – our main example uses an N dimensional Newton's Method (section 2.5). To facilitate the development of a high reliability solution to specific problem instances, we use the Maple model to not only compute a solution and generate an implementation, but also to generate problem specific verification tasks in PVS. This helps us certify the correctness of both, the Maple computations and the generated code. We use a unified representation of all aspects of the mathematical model in Maple, which is then used for all subsequent tasks, and in particular, to generate the code and to generate the PVS proof obligations which "certify" the correctness of the generated code.

We note that currently we are only verifying that the generated code agrees with the (syntactic representation of the) model from which it came, where by "agree" we mean that they define the same computation over the abstract theory of the reals. In other words, we prove various theorems relating syntactic equivalence of computations, and we are not consider low-level semantic issues, nor are we considering numerical issues due to floating-point computations – although many others are actively working on such problems. There is much that needs to be done to deliver on the full promise of provably correct MDD from mathematical models. We hope that this work helps motivate further research and development to this end.

This paper is organized as follows. Section 2 introduces readers to the mathematical tools employed. Section 3 provides details on our proposed methodology, and the next section provides a full length example. Section 5 discusses related work, while Section 6 provides the conclusion and outlines future work.

2. Mathematical Tools

This section provides a brief introduction to the mathematical tools used in our research. We first illustrate the basic ideas of symbolic computation in Maple, as well as its code generation capabilities. We then briefly introduce the proof capabilities of PVS (based on the introduction in [Formal Methods Program 2003]), as well as its built-in computational capabilities (PVSio).

2.1 Symbolic Computation in Maple

Maple is a software system for symbolic and numeric computations, scientific visualization, as well as providing a full-fledged programming language. Such systems are usually known as computer algebra systems (CAS), but this is misleading. While a CAS usually does indeed excel in performing algebraic computations (such as computations with integers, rationals, polynomials, elements of finite fields, matrices over all the above domains, formal series, and so on), such systems are really *useful* through the *symbolic computation* features they provide.

In general, symbolic computation is any computation done on a term algebra which contains un-interpreted symbols. In "computer algebra", there are simple interpretation functions between the syntactic objects in the term algebra and the semantic objects of the underlying model. Furthermore, there is a simple correspondence between computations on the term models and functions on objects in the models. While in many instances of "symbolic computation" there are also interpretation functions and correspondences between computations and functions, the adequacy and soundness of these can be much more problematic [Stoutemeyer 1991, Wester 2000b].

Nevertheless, it is the power of *symbolic computation* which we wish to harness here. In particular, we want to manipulate *mathematical models* represented in exact form by closed-form mathematical formulae. This is a very powerful paradigm, with a long history of successful applications [Barton and Fitch 1972, Pavelle 1985, Wester 2000a].

We should note that Maple supports multiple modes of interaction; one can write "interactive documents", use a top-level Read-Eval-Print-Loop (REPL), or run batch programs. Each mode has different uses, and we will not detail them here. For simplicity we show transcripts of the use of the REPL. Such interactive sessions are useful for exploration and prototyping, but they should be turned into proper programs once a proper methodology for solving (a class of) problems is found.

We can illustrate the power of this paradigm with a few examples of what such a system can do. As a first example, we might want to determine the value of $a^4+b^4+c^4$ under the constraints

$$\begin{aligned} a+b+c &= 6 \\ a^2+b^2+c^2 &= 10 \\ a^3+b^3+c^3 &= 25. \end{aligned}$$

One way to encode this problem is as one of *simplification with side-relations*; in Maple, this translates to the following input:

```
> simplify (a^4+b^4+c^4, {a+b+c=6,
>     a^2+b^2+c^2=10, a^3+b^3+c^3=25});
```

$$106$$

Of more direct interest to us are computations of derivatives and symbolic expressions for a matrix to be positive definite.

```
> assume(x::'real',y::'real'):
> H1 := VectorCalculus:-Hessian
>     (2*exp(x)-y^3-8*x*y,[x,y]);
```

$$H1 := \begin{pmatrix} 2e^x & -8 \\ -8 & -12y^2 \end{pmatrix}$$

```
> LinearAlgebra:-IsDefinite(H1);
```

$$0 < 2e^x \textbf{ and } 0 < -12ye^x - 64$$

The above already shows an example of why we need to combine computation and deduction. While the symbolic expression for `H1` to be positive definite is correct, it is easy to prove that in fact $2e^x > 0$ is true, but harder to do a simple "computation" to simplify this to `true`.

2.2 Code Generation

For the Maple programs which correspond to traditional (numerical) computations, there are facilities for doing program-to-program translation, to a variety of languages. As of now[2], C, Java, Matlab, Fortran and VisualBasic are supported target languages.

We will illustrate some of these capabilities via a simple example. Using a simple function f (defined as an expression),

```
> f := 1-x/2+3*x^2-x^3+x^4:
```

We can create a straight-line-program, or in other words, an optimized list of computation steps to compute f

```
> g := codegen:-optimize (f);
```

$$\texttt{g} := \quad t_2 = x^2, \quad t_5 = t_2^2, \quad t_6 = 1 - \tfrac{x}{2} + 3t_2 - t_2 x + t_5$$

[2] i.e. in Maple 13.

This can be translated to a C code fragment

```
> CodeGeneration:-C([g], 'declare'=[x::float]);
```

$$\begin{aligned} t_2 &= x * x; \\ t_5 &= t_2 * t_2; \\ t_6 &= 0.1e1 - x/0.2e1 + 0.3e1 * t_2 - t_2 * x + t_5; \end{aligned}$$

If we are willing to take more time at the stage of code-generation, we can ask for more efficient code to be generated

```
> g2 := codegen:-optimize(f, 'tryhard'):
> CodeGeneration:-C([g2],'declare'=[x::float]);
```

$$\begin{aligned} t_1 &= x * x; \\ t_2 &= 0.1e1 - x/0.2e1 + (0.3e1 - x + t_1) * t_1; \end{aligned}$$

The efficiency difference in this toy example is minimal, but nevertheless the generated code to compute f has fewer floating point instructions and uses fewer locals (which may result in lower memory usage). In larger examples, the `tryhard` option can result in very significant gains [Carette et al. 2008].

It should certainly be noted that such symbolic computations are not new [Kahrimanian 1953, Gibbons 1960, Wengert 1964], and the literature on using CASes to help in the production of code is rather large, with [Dall'Osso 2003, Carette et al. 2008] as two typical examples.

2.3 PVS

PVS, or Proof Verification System, provides automated support for specification and verification. It consists of a specification language integrated with support tools and a theorem prover. The language is used to define theories and conjectures. Assuming they are correct, these conjectures can then be discharged using the interactive theorem prover.

The specification language of PVS is based on higher-order logic, which is extended with predicate subtypes and dependent types, as well as a theory system. Type constructors include functions, tuples, records, recursive datatypes (e.g., lists and trees), and enumerations. Furthermore, sets are represented by their characteristic predicates. On top of that, PVS provides a large prelude of theories of useful definitions, axioms and proved theorems.

Generally, the PVS theorem prover is used as an interactive system, through a Read-Eval-Print-Loop (REPL), much as Maple is. The PVS theorem prover is interactive and is based on a sequent calculus presentation. PVS offers a graphical representation of proofs in the form of proof trees. These proofs can be saved as scripts and rerun either automatically, or in a single-step mode. While basic proof commands are built-in, most are programmed as strategies. The built-in commands provide powerful automaton that include decision procedures for unquantified integer and linear arithmetic, automatic rewriting, and BDD-based propositional simplification and symbolic model-checking. Predicate subtypes offered by the specification language allow for a great deal of specification to be embedded in its types. The predicate used for defining a predicate subtype can be arbitrary. Thus, typechecking can become undecidable, and may lead to proof obligations called Type Correctness Conditions (TCCs). Generally, the proof strategies built into the theorem prover can automatically discharge some of these obligations. The not so obvious ones, however, are left for the user to prove. Although PVS has a model checker integrated with its theorem prover, it lacks the counterexample generation capability.

2.4 PVSio

PVSio is a PVS prelude library extension that extends its ground evaluator with "a predefined library of imperative programming language features such as side effects, unbounded loops, input/output operations, floating point arithmetic, exception handling, pretty printing, and parsing" [Munoz 2005].

The PVSio library is implemented using semantic attachments. It provides a simple Emacs interface, as well as a stand alone interface to the ground evaluator. Furthermore, PVSio includes proof rules that safely integrates the ground evaluator into the PVS theorem prover. These proof rules use the Common Lisp code generated by the ground evaluator to simplify ground expressions in sequent formulas [Munoz 2005].

1. *Express the Model* - the model is declaratively expressed in mathematical terms (symbolic DSL),
2. *Transform the Model* - transform the initial model into a form more suitable for computational solutions,
3. *Extract Structure* - structure and properties are directly extracted from the model,
4. *Optimize the Computation* - the structure is used to optimize the computational "solution" of the model,
5. *Generate Correctness Conditions* - all of the above information is used to generate correctness conditions, and frequently their proofs as well,
6. *Generate the Code* - low-level code is generated for the solution.

Figure 1. Methodology

2.5 Newton's Method

To demonstrate an application of our methodology, we have chosen our problem class to be function optimization using multivariate Newton's method. It is well-known that if the objective function being optimized is convex, then Newton's method will converge in theory, although in practice it may be very slow and numerical errors may prevent it from converging. On the other hand, if the objective function is not convex, it will not converge in general. However, under mild assumptions, every local minima is contained in a neighborhood on which the function is convex, which is contained in the basin of convergence of the local minima under the Newton iteration.

Let $J_{\mathcal{U}}$ be the Jacobian of F and $H_{\mathcal{U}}$ the Hessian of F with respect to the variables $\mathcal{U}$. The Newton iteration is defined by the recursion:

$$u_{n+1} = u_n - H_{\mathcal{U}}(u_n)^{-1} J_{\mathcal{U}}(u_n).$$

At each iteration, Newton's method is equivalent to forming a quadratic model of the objective function F around the current iterate u_n. When the Hessian matrix, $H_{\mathcal{U}}(u_n)$ is globally positive definite, the model has a unique minimizer. If the Hessian matrix is locally positive definite (which is more frequent than being globally positive definite), then we are at least insured that Newton's method will take a 'step' in the direction of a (local) minimum.

3. General Methodology

Our methodology (Figure 1) starts with a model of the problem class. Contrary to many other model-driven development methodologies, our model is not a software model, or even a software-inspired model (such as UML), but rather a mathematical model of the physical process we are interested in. Our model is a mathematical model of the *environment* and the abstract problem we are trying to solve, and not a model of our software. In other words, our models would be recognizable to any traditional engineer as a "model". Furthermore, the kinds of processes we are interested in

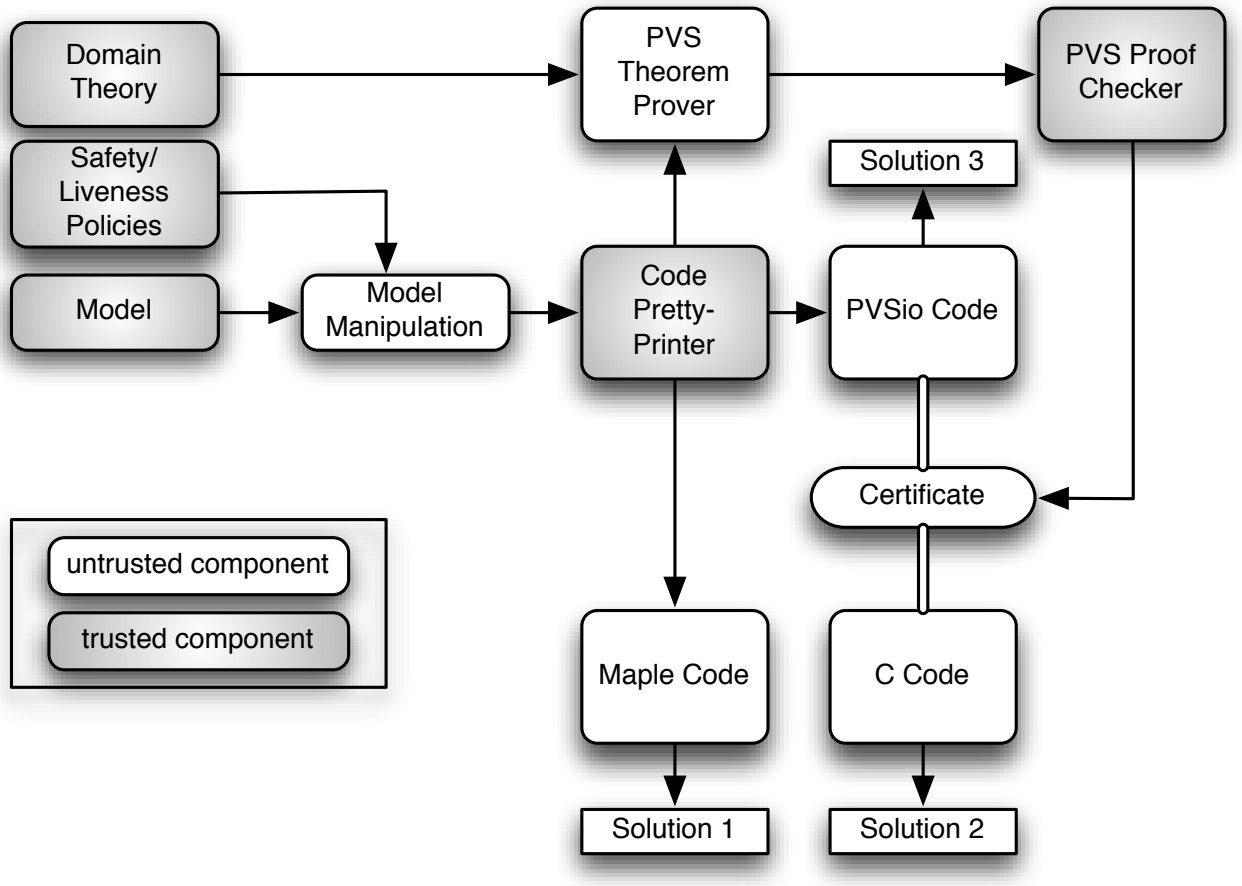

Figure 2. System Architecture.

already have well-understood models, generally involving smooth functions, differential equations, linear algebra and optimization.

We then carry out our task via model manipulations (and optimization), code generation and verification condition generation, directly on (a representation of) the mathematical model. Note that this is an extension of the methodology of [Carette et al. 2008] to include verification condition generation.

This method can be implemented in software. Figure 2 shows the overall system architecture of our implementation. This is closely related to the "Certifiable Program Generation" of [Denney et al. 2005] (see section 5.2 for discussion of their work). As they do, the architecture distinguishes between trusted (grey) and untrusted (white) components. Trusted components must be correct. The presence of errors in any of these components can compromise the assurance provided by the overall system. Furthermore, the assurance provided by our program generation approach does not depend on the correctness of its two main components: the Maple computations, and the PVS theorem prover – instead, we need only trust the problem specification, the safety policies, domain theory, the pretty-printer and the PVS proof checker. Even in the case of a failure of the PVS proof checker, in order to miss detecting an error, Maple must have made an error and/or the pretty printer.

As should be apparent from Figure 2 though, our method crucially relies on the correctness of the various code pretty-printers. However, we believe that it is considerably easier to write correct pretty-printers than it is to write a correct Verification Condition Generator (VCG). Further it might be possible eliminate this potential single point of failure by using another tool to parse the generated C code and generate another proof obligation to compare (the model of) the code to the original model in maple which was used to generate it.

The next section will describe the steps in more detail through an example, but we first give a high-level outline of the ideas. More specifically, *Express the Model* involves writing the *Domain Theory*, and *Model* parts of the architecture, as well as setting the *Safety/Liveness Policies*. This frequently involves creating a simple DSL (domain specific language), which is straightforward in any symbolic computation system. As the model is expressed directly in mathematical terms, which is what a CAS is designed for, *Transform the Model* and *Extract Structure* are in some sense the easiest step to perform! Various computations like finding derivatives, carrying out linear algebra computations (including computing inverses, LU decompositions, finding eigenvalues), and so on are quite easy. The difficulty lies in deciding which transformations are useful and what structure is actually present in any given problem. This is the aspect of the work where human ingenuity still has a significant rôle to play. Given a transformed model and structural information about that model, there are usually well-known

solutions methods that can be applied. This is what *Optimize the Computation* embodies. And it is at this step where advanced techniques, like Active Libraries [Veldhuizen and Gannon 1998], Telescoping Languages [Kennedy et al. 2001], and SANS [Dongarra and Eijkhout 2003] can also come in.

In any case, at this point we still possess a *model* of a computation. From this, we need to *Generate Correctness Conditions*. These fall into various classes:

(a) That the new model is equivalent to the original model,

(b) That the extracted structure is correct,

(c) That the model (of a computation) will produce the "answer" we want when run.

We then have to encode, partly in the *Domain Theory* and partly in the *Safety/Liveness Policies*, the various specifications that correspond to model equivalence (for a), definition of "extracted structure" (for b), and the overall process (for c). Each of these is quite specific to the class of problems being considered, and are best given by example. In the end, these are also modeled in our CAS, so that they can be pretty-printed as theories (in PVS, currently).

Finally, we *Generate the Code*. More specifically, we simply "pretty print" the model of the computation as code in various languages.

Another aspect of our work is that we use both proofs and tests. In particular, our process generates 3 separate pieces of "code" from the same model: one each in C, Maple, and PVSio. All 3 can be executed (albeit at vastly different speeds), and their results can be checked against each other. These are the `Solution` $1, 2, 3$ in Figure 3. In other words, our *Models*, in both Maple and PVS, are sufficiently precise that they can be executed.

4. Example

Figure 3 gives an overview of the various levels of our modeling process. Figure 4 breaks down our code generation and verification process.

To illustrate our process, consider the following (made up) unconstrained optimization problem: Consider the following optimization problem:

$$\min_{y_0,y_1,y_2\in\mathbb{R}} \frac{1}{2}y_0^2\left(\frac{1}{6}y_0^2+1\right)+\frac{1}{2}e^{y_1-1}+(y_1-10)^4+(y_2-4)^2. \tag{1}$$

The function to minimize, as well as the dimensionality 3 (i.e. that we are in $\mathbb{R}^3$) will be our starting point. This is our model, and gives us `func` and `N` in Figure 4, and the starting point for the *Symbolic Problem Model* in Figure 3. A priori, we also choose *Newton's Method* on the gradient of our function as our method to solve 1. This method requires an initial guess, which we also provide.

Turning to the mathematics of solving an unconstrained minimization problem via Newton's method, there are several derived quantities we need: the gradient of our function (as the minimum will be amongst its zeroes) as well as the Hessian matrix, both because Newton's method requires it (or, more precisely, its inverse), and because we need to check that we get a minimum (Hessian should be positive definite) and that we are iterating in a feasible region (again, Hessian should be positive definite). Mathematically

```
> InverseThy := proc(A::LetC,B::LetC,argVec,N)
>   local idM, i, j;
>
>   idM := LetC("identityMatrixNxN",
>               LinearAlgebra:-IdentityMatrix(N), 'None');
>   i := TypedVar("i", "below(N)");
>   j := TypedVar("j", "below(N)");
> [ Import("MatrixNxN[N]"),
>   CMatrix(idM),
>   CMatrix(A),
>   CMatrix(B),
>   Lemma("checkInverse",
>     ForAll([argVec, TypedVar(["i","j"],"below(N)")],
>     Equal( Call(Let("multiplyVectors",NONE),
>              Call(A,argVec), Call(B,argVec), i, j),
>              Call(Call(idM, i), j)))),
>   Proof("checkInverse",
>         "(then (grind) (grind-reals))") ]
> end proc:
```

Figure 5. Maple generator for inverse theory

speaking, we can describe the process as

$$\begin{aligned} g &= \nabla f && (2)\\ h &= H(f) && (3)\\ hh &= h^{-1} && (4)\\ p? &= \{y_0, y_1, y_2 \mid h \text{ is positive definite}\} && (5)\\ \mathsf{step}_1 &= y - hh \cdot g && (6)\\ \mathsf{step}_2 &= \mathcal{OSL}(\mathsf{step}_1) && (7) \end{aligned}$$

where f is our function, g is treated as a row-vector, and $\mathcal{OSL}$ is the operation of creating an 'Optimized Straight Line' program equivalent to its input. Note that, in Maple, equations 3–7 are each a single line of code!

Thus, in stage 1, we use Maple to carry out all those necessary symbolic computations, generating derived model information. Simultaneously, we generate representations for the verification conditions which will need to be checked to certify that these computations are correct. In Figure 4, the first 3 computations correspond to the `checkInverse`, `checkJACOBIAN`, and `checkHESSIAN` obligations, and the last to `Equiv`.

Of course, we do not wish to do explicit linear algebra, especially for such small dimensions, when actually solving our code. We thus perform these computations too, getting 3 closed-form expressions for the Newton step update. In this stage, we also perform some code optimization. Since the natural expression for the update step in a multi-dimensional Newton's method contains many duplicate computations, a common-subexpression elimination is very fruitful. Instead of using a 3-vector of expressions, we turn the code into a straight-line program. This generates another obligation, namely that this straight-line program does in fact compute the "same function", and this is the obligation `Equiv`. This is also the "middle section" of the part labelled "Auto Code Generation" in Figure 3.

In Figure 5, we see a bit of Maple code which generates a PVS theory that checks if the matrix B is related to matrix A, via $A \cdot B = I$, where I is the N-dimensional identity matrix.

In more detail, a `LetC` structure is a triple of a name, a (Maple) value and a (possibly empty) sequence of free variables, and represents a named expression (of any type). `argVec` is a representation of an N-dimensional abstract vector, i.e. y_0, y_1, y_2, encoded as $("y", 3, "real")$. A `TypedVar` is a typed variable (given as a name and a type), `Import` brings in part of a domain theory, `CMatrix` is for defining a constant matrix, and `Lemma` and `Proof` are self-explanatory. `multiplyVectors` is defined in the *Domain Theory* to be the usual vector multiplication, and the lemma ensures that $A \cdot B = I_{i,j}$ pointwise. This procedure is parametric in the 2 matrices, their free variables and the dimension, and generates a special-

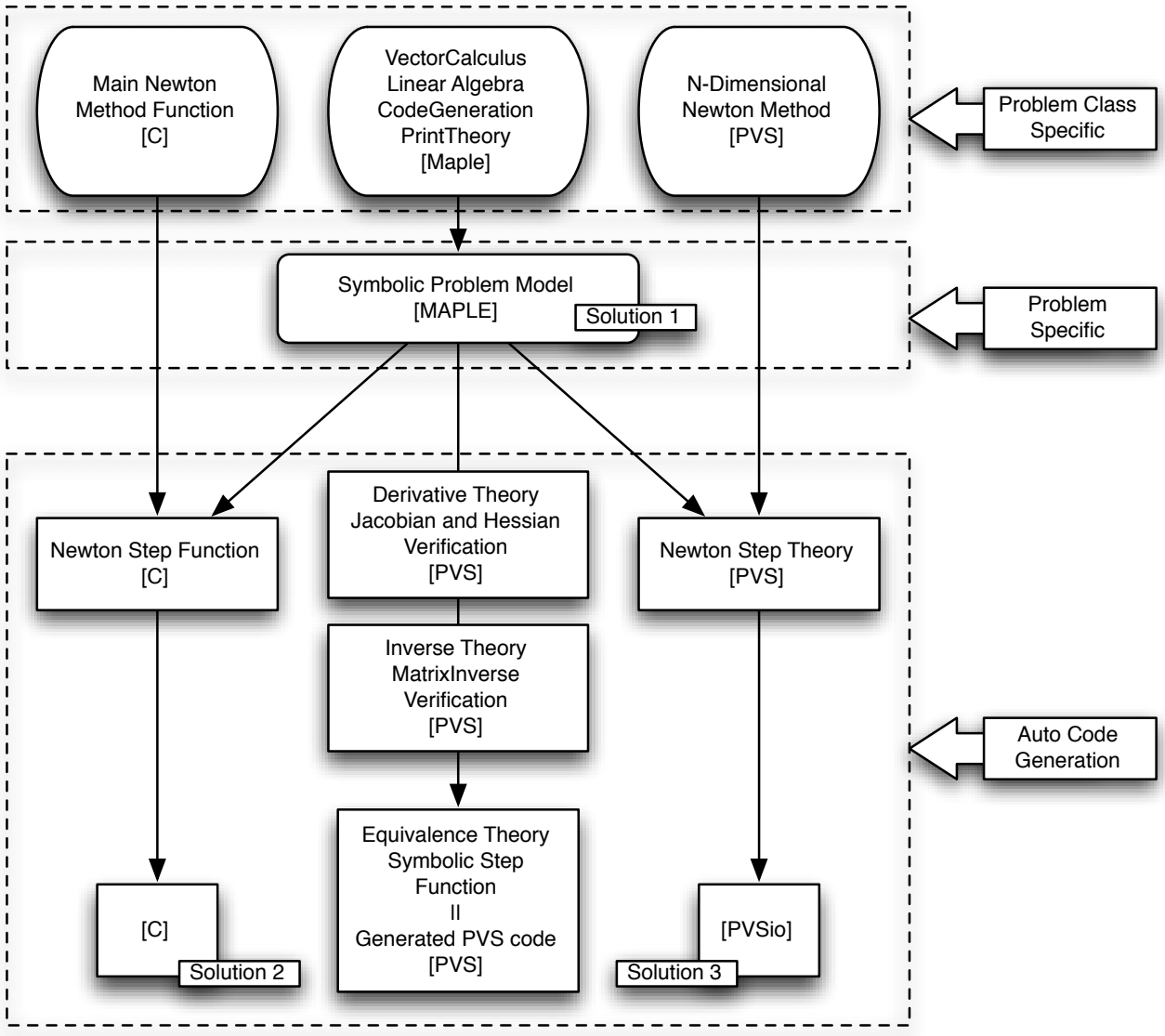

Figure 3. Abstraction levels of modeling.

ized PVS theory for any given input. More precisely, `InverseThy` generates an abstract data type (a "model") of a specialized PVS theory, which is then passed to a pretty-printer which "prints" a PVS file.

This pretty-printer is completely generic, and only knows how to print a specific DSL for mathematical models.

The process for the other theories in stage 1 follow exactly the same pattern.

In stage 2, step_1 is inserted into a more complex theory (not shown here). This theory contains a part (`checkNewton`) is meant to be run by PVSio. It contains an instantiation of the Newton Method, as well as the results of running the same computation (of the Newton Method) numerically in Maple (labelled Solution 1 and Solution 3 in Figure 3. A number of TCCs need to be verified at this point to ensure termination and convergence, and this is where the predicate $p?$ is used.

While setting up the theory is tedious, it is not complex. Eventually we get to

```
PVSioPredicate("checkNewton",
    ForAll(i, Close(Call(pvsres,i), Call(mapres,i),
        prec)))
```

which generates a PVSio predicate

```
checkNewton : bool = FORALL (i: below(N)):
    abs(Result(i) - MapleResult(i)) < precision,
```

which, for this example, is evaluated in PVSio to `TRUE` when `precision` equals to 10^{-6}.

Finally, now that we are quite confident that all of our pieces work smoothly together, we can pass to stage 3, where we take step_2 and pretty-print it as C code. We actually have 2 ways of doing this, one where we generate a C file for just the Newton Step

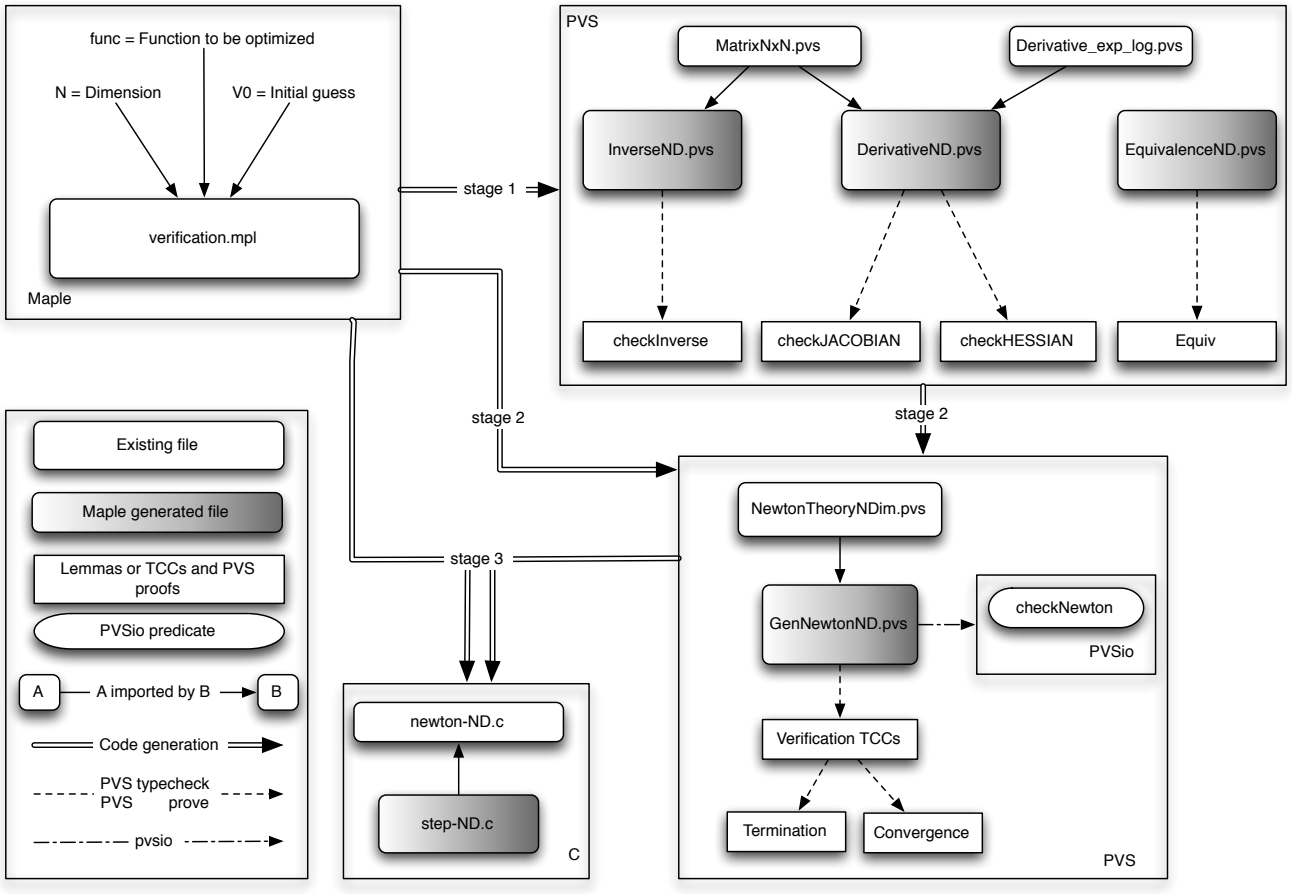

Figure 4. Detailed overview of code generation and verification process.

to be plugged in to a driver routine (shown on Figure 4), as well as a late-breaking method (not shown, but in the accompanying code) to carry out whole-program generation.

For example, with an initial guess of $(y_0, y_1, y_2) = (-1000.0, -1000.0, -1000.0)$, we come to a minima at $(y_0, y_1, y_2) = (0.0, 6.67965113451880832, 4.0)$ in 20 iterations, and all 3 solutions agree.

Lastly, in accordance with the skeptic's approach to combining CAS and ATP [Harrison and Théry 1998], all we need to record in our theories are the input-output relations associated to 3–7, which are quite simple. We do not need to worry about the (immense!) complexity of some of the underlying implementations.

The full code for this paper is available at [Full Code].

5. Related Work

This section discuss the paper's relationship to the existing literature. It is broken down into subsection on related CAS-ATP integration efforts, certifiable program generation, and testing of auto-code generators.

5.1 Integration of CAS and ATP

There are ambitious projects based upon the OMDoc [Kohlhase 2000] and OpenMath [Buswell et al. 2004] standards to provide correct semantic translations in between CAS and ATP (e.g. [Armando and Zini 2000]). These projects are typically focused upon facilitating the use or verification of results of one type of system in the other. These standards are still evolving, and there is not yet consistent widespread support for them in either types of tools. Furthermore, there is no native concept of "theory" in Maple. We have therefore chosen to embed a DSL of theories (into Maple), along with a pretty-printer to generate the required PVS theories for certification of our results.

A direct connection of a CAS and ATP is described in [Delahaye and Mayero 2005]. It connects Maple to Coq to help Coq deal with algebraic expressions over a field. This is typical of many of the efforts to improve the proving capability of an ATP using a CAS. In contrast we using the combined features of a CAS and an ATP to do certifiably correct code generation for a class of models.

In doing this interconnection, we follow the "skeptics' approach to combining CAS and ATP" of [Harrison and Théry 1998] where the ATP (PVS in our case) is used to certify results produced by Maple. We take a similar approach to checking the correctness of Maple's results where we generate appropriate theorems as we perform the derivations. For example, when computing $H_{\mathcal{U}}(u_n)^{-1}$, the inverse of the Hessian to be used in Newton's method, we generate a theorem that $H_{\mathcal{U}}(u_n)H_{\mathcal{U}}(u_n)^{-1} = I$ to be discharged in PVS. However our goal is to facilitate provably correct model driven development from a mathematical model in the CAS, so rather than have the ATP as the master and the CAS as the slave (as in [Harrison and Théry 1998]), we have the CAS as the master. This is similar to the Maple-PVS system [Adams et al. 2001]. Where

we differ significantly from Maple-PVS is in the way in which we do the translation to PVS and the fact that we also generate and verify code. The Maple-PVS interface is string-based; more precisely, the user directly writes PVS code in Maple strings, but can directly refer to any Maple value by its name. As a proof-of-concept, the authors achieved their goals, but this is clearly not a viable methodology. Furthermore, we did not want to be tied to PVS, and so we preferred to create a theory DSL which we could pretty-print to PVS (now) and other theorem provers (later)[3]

In [Hardy 2006], Hardy develops a decision procedure for problems relating to polynomial and transcendental functions. A decision procedure is implemented using a modified version of Maple-PVS, where PVS is connected to the QEPCAD implementation of quantifier elimination by partial cylindrical algebraic decomposition. The implementation of the decision procedure is then applied to the analysis of Nicols Plots that are frequently used to determine the stability of feedback control systems. The approach of [Hardy 2006] is similar to our work in that it combines tools to solve a specific problem, but it is mainly concerned with implementing the decision procedure, and the analysis of control systems is secondary.

In this area, the significant difference with previous efforts to combined CAS and ATP is that we focus on certifiably correct program generation from models in such a way that there should be no single point of failure that could result in an undetected error. This then brings us to the discussion of previous efforts in certifiable program generation.

5.2 Certifiable Program Generation

Auto-coding techniques are becoming increasingly important. Automatically generating complete programs from models is essential to the success of MDD [Selic 2003]. Program generation has significant potential to improve the software development process and promises many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. However, "the key to realizing these benefits is clearly generator correctness – nothing is gained from replacing manual coding errors with automatic coding errors" [Denney et al. 2005].

For large systems, verification of automatic code generations is extremely difficult due to size and complexity. However, much effort has been directed towards a product-oriented certification approach [Schumann et al. 2003, Whalen et al. 2002]. There are five major principles in this approach:

1. Trustworthiness of the code generator is reduced to the safety and liveness of each individual generated program,
2. Program safety and liveness are defined as adherence to explicitly formulated policies,
3. The safety and liveness policies are formalized by a collection of logical program properties,
4. Hoare-style program verification is used to show that each generated program satisfies the required properties, and
5. The code generator itself is extended to automatically produce the code annotations (e.g., loop invariants) required for verification.

It should be noted that the above principles refer to both, the safety and liveness policies.

The local annotations for a statement capture the changes in variables imposed by that statement, without the need to describe the global information that may later be necessary for proofs. Then, the annotations are propagated throughout the program. "The approach is feasible because the code generator has full knowledge about the program under construction and about the properties to be verified."[Schumann et al. 2003] It can thus often generate all auxiliary code annotations a theorem prover would need to confirm all arising verification obligations fully automatically.

This approach is similar to the methodology of proof carrying code (PCC) [Necula and Lee 1998]. It also relies on a small kernel of verified components. "These components, the safety policy, verification condition generator, and the proof checker, are very simple and can be verified using standard software development techniques."[Schumann et al. 2003] In case of PCC, the developer produces the auxiliary annotations (e.g., loop invariants) which are required to make the proofs possible. In product-oriented certification approach [Schumann et al. 2003, Whalen et al. 2002], the code generator itself generates the required auxiliary annotations. Even in the latter case, errors in the various parts of the software (i.e., the code generator and theorem prover) are intended to cause the proof process to fail at some step.

5.3 Testing of Automatic Code Generators

The main approaches for the verification of auto-code generators are formal verification and testing. As can be seen from the previous discussion above, formal verification methods can give very strong guarantees about the correctness of auto-code generators. Formal reasoning requires mathematically proving

$$(\forall m : \mathsf{model}, i : \mathsf{input}) : \mathtt{ModelExecSem}(m, i)$$

$$\equiv$$

$$\mathtt{CodeExecSem}(\mathtt{CodeGen}(m), i),$$

where `ModelExecSem` and `CodeExecSem` are the model and code execution semantics respectively [Sampath et al. 2008]. `CodeGen`(m) denotes the code generated by the auto-code generator to be verified, and $\equiv$ compares the output generated by a model with the output generated by the corresponding code. Due to the quantification over the models and the inputs, the verification process becomes a very challenging task.

To simplify the approach to formal verification of the auto-code generators, comparing of the model and the generated code could be carried out separately for each execution of the code generator. Now, given a *m: model*, the proof in hand simplifies to the following:

$$(\forall i : \mathsf{input}) : \mathtt{ModelExecSem}(m, i)$$

$$\equiv$$

$$\mathtt{CodeExecSem}(\mathtt{CodeGen}(m), i).$$

Furthermore, one can firstly and wisely choose a small subset of inputs in order to meet some coverage criteria (statement, branch, condition and state/transition coverage). Then one can verify the above assertion for this chosen subset of inputs, which will provide a degree of confidence that the program behaves correctly for all possible inputs.

Ultimately, testing of automatic code generators challenges the correctness of the following: for a particular subset of models and inputs,

$$\mathtt{ModelExecSem}(m, i) \equiv \mathtt{CodeExecSem}(\mathtt{CodeGen}(m), i).$$

Now, a test-case to an automatic code generator consists of the following: the model, the chosen set of inputs to this model, and the corresponding expected outputs from this model [Sampath et al. 2008].

We believe it is important to understand the various ways of dealing with verification of automatic code generation and the level

[3] We realize that our current DSL is probably still too PVS-specific right now, and certainly the proofs cannot be reused, but believe that the modifications necessary will be manageable.

of complexity of this verification of each method. We believe that our method offers some compelling advantages in certain situations.

6. Conclusion and Future Work

We believe formal methods can provide the highest level of assurance of the safety of any given piece of code. This is especially challenging when code generation is used. Nevertheless, the benefits of code generation are too many to ignore, and thus it is important to develop various methods to combine high-level assurance with generative techniques. Like others before us, we wanted to demonstrate that assurance in this case does not necessarily mean code-generator trustworthiness, but follows from a constructed set of assertions derived from the model and its gradual transformation into code. In other words, a model-driven approach, at least when the model consists of well-understood mathematical models, can simplify the job of getting high-assurance code by generative means.

During the evolution of our DSL for models and theories, it became more and more apparent that we were encoding *biform theories* [Farmer 2007]. A biform theory combines axiomatics definitions with computational transformers, coupled via *meaning functions*. Our theory generators correspond exactly to those meaning functions, with the Maple computations corresponding to the transformer and the Domain theory providing the axiomatic definitions. We have some prototype work in this direction which is promising, and should allow us to further simplify our models.

Naturally, we are also working on extending our methodology to larger and more complex examples. In particular, many optimal control problems should fit quite well in our development and verification environment. Furthermore, we plan to apply our methodology to specific applications, such as the class of image processing problems described in [Korobkine 2002]. Lastly, issues of numerics should be considered in verifying the implementation, as has been done for digital filters in [Akbarpour and Tahar 2007].

References

A. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. BoultonRichard J. Boulton, editor, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42. University of Edinburgh and University of Glasgow, Edinburgh, United Kingdom, September 2001. ISBN 3-540-42525-X. http://www.dcs.qmul.ac.uk/~hago/Papers/tphols2001.pdf.

Behzad Akbarpour and Sofiène Tahar. Error analysis of digital filters using hol theorem proving. *J. Applied Logic*, 5(4):651–666, 2007.

Alessandro Armando and Daniele Zini. Interfacing computer algebra and deduction systems via the logic broker architecture. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic computation and automated reasoning: The CALCULEMUS-2000 Symposium*, pages 49–64. A K Peters, Ltd, St Andrews, Scotland, August 2000.

D. Barton and J. P. Fitch. A review of algebraic manipulative programs and their application. *j-COMP-J*, 15(4):362–381, November 1972. ISSN 0010-4620. http://www3.oup.co.uk/computer_journal/hdb/Volume_15/Issue_04/150362.sgm.abs.html.

S. Buswell, O. Caprotti, D. P. Carlisle, M. Gatano M. C. Dewar, and M. Kohlhase, editors. *The OpenMath Standard (Version 2.0)*. The OpenMath Society, 2004.

Jacques Carette, Spencer Smith, John McCutchan, Christopher Anand, and Alexandre Korobkine. Case studies in model manipulation for scientific computing. In *Proceedings of First Conference of Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 24–37. Springer Verlag, 2008.

Aldo Dall'Osso. Using computer algebra systems in the development of scientific computer codes. *Future Gener. Comput. Syst.*, 19(2): 143–160, 2003. ISSN 0167-739X. http://dx.doi.org/10.1016/S0167-739X(02)00126-7.

David Delahaye and Micaela Mayero. Dealing with algebraic expressions over a field in coq using maple. *Journal of Symbolic Computation*, 39(5): 569 – 592, 2005. ISSN 0747-7171. Automated Reasoning and Computer Algebra Systems (AR-CA).

Ewen Denney, Bernd Fischer, Dieter Hutter, and Mark Jones. Software Certificate Management (SoftCeMent'05). In *Automated Software Engineering, Software Certificate Management*, 2005. http://doi.acm.org/10.1145/1101908.1102003.

Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science, June 2–4 2003, St. Petersburg (Russia) and Melbourne (Australia)*, 2003.

W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.

Formal Methods Program. Formal Methods Roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003.

Full Code, 2009. http://www.cas.mcmaster.ca/~carette/newtongen.

A. Gibbons. A program for the automatic integration of differential equations using the method of Taylor series. *Comp. J.*, 3:108–111, 1960.

Ruth Hardy. Interactions between pvs and maple in symbolic analysis of control systems. *Electronic Notes in Theoretical Computer Science*, 151 (1):111 – 125, 2006. ISSN 1571-0661. Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005).

J. Harrison and L. Théry. A skeptic's approach to combining hol and maple. *J. Autom. Reason.*, 21(3):279–294, 1998. ISSN 0168-7433. http://dx.doi.org/10.1023/A:1006023127567.

H. G. Kahrimanian. Analytical differentiation by a digital computer. Master's thesis, Temple University, May 1953.

Ken Kennedy, Bradley Broom, Keith D. Cooper, Jack Dongarra, Robert J. Fowler, Dennis Gannon, S. Lennart Johnsson, John M. Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *J. Parallel Distrib. Comput.*, 61(12):1803–1826, 2001.

Michael Kohlhase. Omdoc: an infrastructure for openmath content dictionary information. *SIGSAM Bull.*, 34(2):43–48, 2000. ISSN 0163-5824. doi: http://doi.acm.org/10.1145/362001.362021.

Alexandre O. Korobkine. Model-Based Visual Tracking via Maple Code Generation. Master's thesis, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, 2002.

Walter Kramer. Bugs, Errors, and Unexpected Results in Computer Algebra Packages. *PAMM*, 7(1):2140009–2140010, 2007.

C. Munoz. PVSio Reference Manual – Version 2.b. (DRAFT). http://research.nianet.org/~munoz/PVSio/PVSio-2.d.pdf, 2005.

Neil Munro, editor. *Symbolic Methods in Control System Analysis and Design*. Institution of Electrical Engineers, Stevenage, UK, UK, 1999. ISBN 0852969430.

George C. Necula and Peter Lee. Efficient Representation and Validation of Proofs. In *Logic in Computer Science*, pages 93–104, 1998. http://citeseer.ist.psu.edu/article/necula98efficient.html.

Richard Pavelle, editor. *Applications of computer algebra (Philadelphia, PA, 1984)*, Norwell, MA, USA, and Dordrecht, The Netherlands, 1985. Kluwer Academic Publishers Group. ISBN 0-89838-173-8.

Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. Behaviour Directed Testing of Auto-code Generators. *International Conference on Software Engineering and Formal Methods*, 0:191–200, 2008.

Johann Schumann, Bernd Fischer, Mike Whalen, and Jon Whittle. Certification Support for Automatically Generated Programs. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.1, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1874-5.

B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, Sept.-Oct. 2003. ISSN 0740-7459. 10.1109/MS.2003.1231146.

D.R. Stoutemeyer. Crimes and misdemeanors in the computer algbebra trade. *Notices of the AMS*, pages 701–785, 1991.

Todd L. Veldhuizen and Dennis Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.

Michael J. Wester, editor. *Computer algebra systems: a practical guide*, New York, NY, USA, 2000a. John Wiley & Sons, Inc. ISBN 0-4719-8353-5.

Michael J. Wester. A critique of the mathematical abilities of CA Systems. In *Computer algebra systems: a practical guide* Wester [2000a]. ISBN 0-4719-8353-5.

Michael W. Whalen, Johann Schumann, and Bernd Fischer. Synthesizing Certified Code. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 431–450, Copenhagen, Denmark, July 2002. Springer-Verlag. ISBN 3-540-43928-5.