`Valigator`: A Verification Tool with Bound and Invariant Generation ⋆

Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács

EPFL, Switzerland

Abstract. We describe `Valigator`, a software tool for imperative program verification that efficiently combines symbolic computation and automated reasoning in a uniform framework. The system offers support for automatically generating and proving verification conditions and, most importantly, for automatically inferring loop invariants and bound assertions by means of symbolic summation, Gröbner basis computation, and quantifier elimination. We present general principles of the implementation and illustrate them on examples.

1 Introduction

In [16], a framework for generating polynomial equations as loop invariants was presented for a rich class of so-called *P-solvable* loops with ignored loop conditions. Implemented in the software package `Aligator`, the approach was successfully tested on many examples. However, `Aligator` was not able to infer properties depending on the loop conditions.

In the current paper we address this problem and present `Valigator`, an automatic tool that extends and uses the functionalities of `Aligator`. More precisely, `Valigator` enables `Aligator` to infer stronger invariants involving polynomial equalities and inequalities by treating loop conditions. In addition, `Valigator` supports generating and proving verification conditions using the inferred loop invariants, and proving the partial correctness of programs annotated with pre- and postconditions. We consider programs containing loops with sequencing, nested conditionals, and assignments, and impose structural constraints on the type of assignments. We require that loop conditions are linear inequalities, variables from loop conditions are changed using affine mappings, and most importantly, branches in loops commute. By commuting we mean that variables in the loop condition are changed by the same affine mappings in all conditional branches.

The purpose of this paper is to discuss the underlying principles of `Valigator`, whose main features are as follows. (1) It contains a prototype verification condition generator based on the strongest-postcondition strategy [13]. (2) It integrates `Aligator` as its invariant inference engine and thus has access to a wealth of powerful algorithms from the computer algebra system `Mathematica` [24]. Moreover, we extended `Aligator` by implementing an approach for automatically inferring polynomial equalities and *inequalities* as invariants from the polynomial closed form of the loop by imposing *bound constraints* on the number of loop iterations. (3) Finally, `Valigator`

⋆ This research was supported by the Swiss NSF.

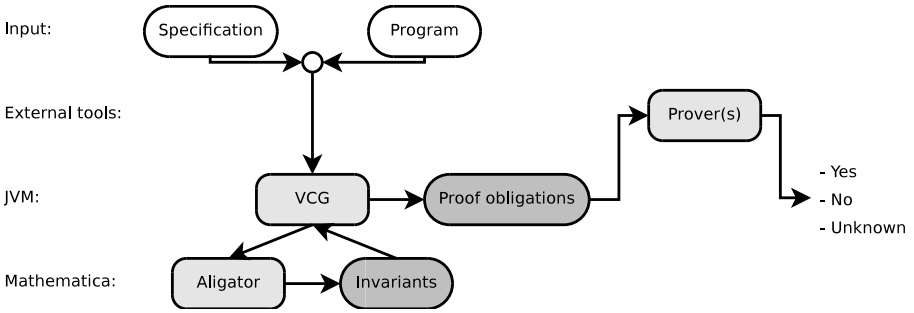

Fig. 1. The `Valigator` tool

uses the automated theorem proving tools `Z3`, `CVC3`, and `STP` [11, 5, 15] for proving the correctness of the verification conditions.

In the sequel, we will discuss in more detail the main ingredients of `Valigator`. We will present general principles of the implementation and illustrate them on examples. In order to improve readability, we present the input and output lines of `Valigator` commands in a simplified form.

Implementation and Installation. `Valigator` is implemented in the `Scala` programming language [20], that compiles to the `Java` bytecode. Beside exploiting the `Scala` capabilities, `Valigator` integrates `Scala` with the computer algebra system `Mathematica` in a transparent way by relying on the `JLink` toolkit [24]. Using and controlling the `Mathematica` kernel directly from a `Scala` program is thus supported in `Valigator`. The overall workflow of `Valigator` is illustrated in Figure 1.

`Valigator` is available at:

`http://mtc.epfl.ch/software-tools/Aligator/Valigator/`.

The current version of the source distribution is 0.1 and runs under most recent `Linux` versions.

Experiments. We have successfully tried our implementation on many examples; – see the mentioned `URL`. For each of the examples, the results were obtained in less than 5 seconds on a machine with a 2.0GHz CPU and 2GB of RAM. The most time-consuming part of `Valigator` lies in proving verification conditions, whereas the generation of verification conditions together with invariant and bound inference requires less than 2 seconds for all examples we tried.

Related Work. We only mention some of the numerous tools that are related to `Valigator`. Such systems include the static program verifiers `Esc/Java` [14], `LOOP` [23], and `JIVE` [18] for sequential `Java`. Inputs to these tools are `Java` source programs with user-supplied annotations expressed in `JML` [17]. Generating verification conditions, these systems are respectively connected with the theorem provers `Simplify` [12], `PVS` [21], and `Isabelle` [19] to verify proof obligations. It is well-known that producing first-order verification conditions requires loop invariants. These tools are thus powerful for programs whose invariants are specified by the user and, due to the nature of the employed provers, they support checking assertions over integers only

if they are linear. A similar verification environment has been developed in the KeY tool [1] for Java programs, and in Spark [2] for Ada code.

A closely related approach to Valigator is the Spec♯ [4] verifier for C♯ programs, which uses the Boogie [3] tool for verification. Boogie combines an invariant inference engine based on the abstract interpretation framework, a verification condition generator, and the theorem prover Z3 [11]. Inferred invariants over integers are again required to be linear, because Z3 handles only linear arithmetic.

The main difference between the mentioned systems and Valigator is that Valigator supports the *automatic generation of polynomial* invariants by means of symbolic computation and offers a richer choice of proof tools over integers for proving polynomial (and not just linear) verification conditions. However, many of the discussed tools handle data types such as arrays and pointers, which is not yet the case for Valigator.

Inferring polynomial invariants is also supported by the Polyinvar tool [22] by bounding a priori the degree of polynomials. Valigator imposes no such bounds, yet Polyinvar can handle more complex loops. We are also not aware of an integration of Polyinvar into a verification environment.

2 Valigator: System Description

Inputs to Valigator are programs formulated in a custom language similar to a subset of the programming language C, with sequencing, assignments, conditionals, and loops, involving addition and multiplication over integers, augmented with pre- and postconditions. Data types are not yet handled. The language uses assume and assert constructs for stating, respectively, pre- and postconditions. In the sequel we call an *annotated program* a program with a precondition and a postcondition.

We require that loops be P-solvable. Namely, (i) they contain only assignments to variables and nested conditionals; (ii) variables in assignments range over numeric types, such as integers; (iii) values of variables can be expressed as polynomials of the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are algebraic dependencies among the new variables; (iv) assignments to variables X' that appear in the loop condition are affine mappings satisfying the matrix equation $X' = AX' + B$ with a square matrix A and column vector B of numeric constants, where the main diagonal of A contains only 1s; (v) conditional branches in loops commute, i.e variables X' from the loop conditions are changed in the same manner on each branch; (vi) and finally, the loop condition is expressed by linear inequalities over loop variables X'. Note that condition (iv) ensures the existence of polynomial closed forms of X' as univariate polynomials in the loop counter, whereas conditions (iv)-(vi) yield polynomial inequalities in the loop counter.

The syntax of considered P-solvable loops is thus as below.

$$\begin{aligned}&\texttt{while}\ (b_0)\ \{s_0;\ \texttt{if}(b_1)\ \texttt{then}\ \{s_1\}\\ &\qquad\qquad \texttt{else}\{\ldots\texttt{else}\{\texttt{if}(b_{k-1})\ \texttt{then}\ \{s_{k-1}\}\ \texttt{else}\ \{s_k\}\}\ldots\};s_{k+1}\}\end{aligned} \tag{1}$$

where $k \in \mathbb{N}$, $s_0, \ldots, s_{k+1}$ are sequences of assignments and $b_0, \ldots, b_{k-1}$ are boolean expressions.

Verification conditions are automatically derived using the strongest postcondition method and `Aligator`. Their validity is checked by state-of-the-art theorem provers.

Command 2.1 : `Valigator[C]`

Input: Annotated program C
Output: `Yes/No/Unknown`, where

- `Yes` means that all verification conditions were successfully proved, and hence the partial correctness of the input is established;
- `No` is returned when at least one verification condition was disproved, and hence either a bug in the program was found or the invariants generated by `Aligator` were not strong enough.
- `Unknown` is answered when at least one verification condition was disproved because, due to some unsafe arithmetic simplifications during invariant inference, the derived invariant is not actually a loop invariant (see Subsection 2.2).

In the last two cases, when the invariants generation fails, it is possible to manually give invariants to `Valigator` using annotations in the program source code.

EXAMPLE 2.1 Consider the annotated program computing the sum of two integers a and b, such that b is non-negative.

```
Input: Valigator[assume (b >= 0);
                 res = a; cnt = b;
                 while(cnt > 0){cnt = cnt - 1; res = res + 1};
                 assert (res = a + b)]
Output: Yes
```

Verification of imperative programs in `Valigator` consists of invariant inference, generation and proving of verification conditions. In what follows, we discuss these steps in more detail, and summarize the main components of `Valigator` in Table 1.

`Valigator`: verification of programs with invariant and bound inference
Input: Annotated program Output: `Yes/No/Unknown`
`VCG`: verification condition generator
Input: Annotated program Output: List of verification conditions
`Aligator`: invariant generation for P-solvable loops with bound inference
Input: P-solvable loop and, optionally, initial values of loop variables Output: Loop invariant
`AnalyseBound`: bound analysis for P-solvable loops
Input: List of assignments over loop variables, loop condition and initial values of loop variables Output: Bound assertions over the values of loop variables

Table 1. `Valigator` commands and their descriptions

2.1 Generation of verification conditions

Generation of verification conditions is performed by the VCG command.

Command 2.2 : VCG[C]

Input: Annotated program C
Output: List of verification conditions (proof obligations)

VCG is a predicate transformer based on a list of inference rules. It treats the program structure recursively, statement-by-statement. Namely, VCG takes an annotated program, and repeatedly modifies the precondition such that at the end the program is "eliminated", and a logical formula is inferred as a collection of verification conditions. In fact, it computes the strongest postcondition in the abstract interpretation framework defined in [10].

For each assertion specified by `assert`, VCG generates a proof obligation. However, contrarily to other verification tools (see e.g. [2, 4]), in Valigator we do not use `assert` for annotating loops with invariants. Instead, VCG invokes Aligator as its invariant inference engine whenever a loop is encountered. Invariant generation is thus part of VCG and the invariant inference takes advantage of the constant propagation performed by VCG. Using the invariant returned by Aligator, VCG generates two proof obligations corresponding to the initial and the inductiveness properties of the invariant.

EXAMPLE 2.2 For Example 2.1, the verification conditions derived by VCG are below.

Input: $\mathtt{VCG}[C]$

Output: $b \geq 0 \ \wedge \ res = a \ \wedge \ cnt = b \ \Rightarrow$
$cnt + res = a + b \ \wedge \ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b)$

$b \geq 0 \ \wedge \ cnt > 0 \ \wedge \ cnt + res = a + b \wedge$
$(b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b) \ \Rightarrow$
$(cnt - 1) + (res + 1) = a + b \ \wedge \ (b > 0 \Rightarrow cnt - 1 \geq 0) \wedge$
$(b \leq 0 \ \Rightarrow \ res + 1 = a \ \wedge \ cnt - 1 = b)$

$b \geq 0 \ \wedge \ cnt \leq 0 \ \wedge \ cnt + res = a + b \wedge$
$(b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \ \wedge \ cnt = b) \ \Rightarrow$
$res = a + b$

where the generated loop invariant is $cnt + res = a + b \ \wedge \ (b > 0 \Rightarrow cnt \geq 0) \ \wedge \ (b \leq 0 \ \Rightarrow \ res = a \wedge cnt = b)$, as shown in Example 2.3, and C is the input of Valigator given in Example 2.1. The first two proof obligations ensure soundness of the invariant; the last one corresponds to the `assert` statement.

2.2 Invariant Inference

Let X denote the set of loop variables, X_0 the corresponding initial values (before entering the loop), and n the iteration counter of the loop.

Command 2.3 : `Aligator[PLoop, IniVal`$\rightarrow$`list of assignments]`

Input: P-solvable loop `PLoop` as in (1) and, *optionally*, a list of assignments specifying the initial values X_0 of X
Output: Loop invariant $\bigwedge_i p_i(X) = 0 \wedge \varphi(X)$, where $\varphi(X)$ is a boolean combination of polynomial equalities $q_j(X) = 0$ and inequalities $r_s(X) \geq 0$, with $p_i, q_j, r_s \in \mathbb{K}[X]$, where $\mathbb{K}$ can be either $\mathbb{Z}$ or $\mathbb{Q}$

EXAMPLE 2.3 The invariant returned by `Aligator` for Example 2.1 is given below.

```
Input:  Aligator[while(cnt > 0){cnt = cnt − 1; res = res + 1},
                IniVal → {res = a; cnt = b}]
Output: cnt + res = a + b ∧ (b > 0 ⇒ cnt ≥ 0) ∧ (b ≤ 0 ⇒ res = a ∧ cnt = b)
```

Note that the initial values of cnt and res are extracted by `VCG` from Example 2.1 (i.e. from the loop precondition), and then fed into `Aligator`.

EXAMPLE 2.4 To illustrate the power of invariant and bound inference, let us now consider a P-solvable loop with conditional branches as given below. Its invariant property returned by `Aligator` is as follows.

```
Input:  Aligator[while(x + y < 10)
                    {if (x < y) then {x = x + y; y = y + 1; c = c + 1; d = d + 1}
                                else {x = x + y; y = y + 1}},
        IniVal → {x = 0; y = 2; c = 1; d = 1}]
Output: c = d ∧ y^2 = y + 2x + 2 ∧ (0 + 2 < 10 ⇒ x + y < 10 ∨ (x = 9 ∧ y = 5))
```

EXAMPLE 2.5 A relatively simple well-known example is taken from [7]. The invariant property returned by `Aligator` is given below.

```
Input:  Aligator[while(x ≠ y){x = x + 1; y = y − 1}, IniVal → {x = a; y = b}]
Output: x + y = a + b ∧ (a = b ⇒ x = a ∧ y = b) ∧ (a + b = 2x = 2y ∨ a = b ∨ x ≠ y)
```

Internally, (i) `Aligator` first checks whether a given input is as (1). If it is not, an error is reported, and the invariant inference stops. Otherwise, polynomial equalities $p_i(X) = 0$ as invariants are generated for loop (1) with *omitted tests*, as follows: the closed form system of X is derived using recurrence solving over the loop body with the summation variable n, and variables depending on n are then eliminated by the Gröbner basis computation [8]. In the sequel, we write $CF_X(n)$ to mean the system of closed form expressions of X as functions of n. As a result of this step, valid polynomial relations among loop variables are inferred [16]. (ii) Next, the *loop condition is taken into account*, and it is checked whether conditional branches *commute*. By commuting we mean that variables $X' \subseteq X$ in the loop condition b_0 are changed by affine mappings in the same manner on each conditional branch. Deriving $CF_{X'}(n)$ is thus also feasible by means of recurrence solving in case of loops with nested conditionals, and, as discussed on page 3, $CF_{X'}(n)$ yields a *univariate polynomial system in* n. If

it is established that the branches commute, quantifier elimination is applied to derive bound assertions on n as additional polynomial inequalities $r_s(X) \geq 0$ and equalities $q_j(X) = 0$. This is based on a relatively simple idea: we seek solution to the formula given below, expressing the upper bound of n.

$$\exists n.n \geq 0 \wedge \Big(\big(\underbrace{b_0[\![n]\!] \wedge CF_{X'}(n)}_{L(n)} \wedge \underbrace{\neg b_0[\![n+1]\!]}_{T(n)} \big) \Big), \tag{2}$$

where $b_0[\![n]\!]$ represents the loop condition in which all variables X' have been substituted by their values $CF_{X'}(n)$ at iteration n; $L(n)$ encodes the behavior of X' during the nth loop iteration; and $T(n)$ corresponds to the termination criteria of the loop after n iterations. The formulas L and T are functions of the loop counter n and are derived from substituting variables X' by their closed forms $CF_{X'}(n)$. As mentioned on page 3, b_0 is expressed by linear inequalities over X', hence $b_0[\![n]\!]$ is expressed by polynomial inequalities in n. This way, solving (2) reduces to the problem of quantifier elimination from a quantified system of univariate polynomial inequalities and equalities in n, which can be solved as presented in [9]. Note that although [9] would also handle the case when the loop condition is a non-linear polynomial inequality, due to efficiency reasons, we only treat loops whose loop conditions are linear.

For solving (2) we rely on the quantifier elimination engine of `Mathematica` integrated into `Aligator` as part of the `AnalyseBound` command for inferring bound assertions. If an exact value of n can be computed, the values $V(X')$ of variables X' when exiting the loop are derived using $CF_{X'}(n)$. Note that equation (2) makes the assumption that the loop will be executed at least once. However, it is trivial to compute the symbolic state at the end of the loop when it is executed 0 times. Turning $V_{X'}$ into a loop invariant is based on the following simple fact. If the loop is executed at least once, then either the loop condition holds or values of the variables X' fulfill $V_{X'}$.

Finally, the loop invariant returned by `Aligator` is the conjunction of the polynomial invariants derived in step (i) and the invariants obtained after the bound analysis from step (ii).

Command 2.4 : `AnalyseBound[`$S_{X'}$`,` b_0`, IniVals]`

Input: Assignment statements $S_{X'}$ of loop (1) corresponding to the variables X', loop test b_0 and initial values of X'
Output: Formula $(b_0[\![0]\!] \Rightarrow b_0 \vee V(X')) \wedge (\neg b_0[\![0]\!] \Rightarrow CF_X(0))$

EXAMPLE 2.6 For the loop of Example 2.1 we obtain:

$$\texttt{Input:}\ \texttt{AnalyseBound}[\{cnt = cnt - 1\}, cnt > 0,\ \{res_0 = a, cnt_0 = b\}]$$

$$\texttt{Output:}\ (b > 0 \Rightarrow cnt = 0 \vee cnt > 0) \wedge (b \leq 0 \Rightarrow res = a \wedge cnt = b)$$

where we denote respectively by res_0 and cnt_0 the variables standing for the initial values of res and cnt. The polynomial invariant inferred in step (i) of `Aligator` is $res + cnt = a + b$. Hence, the complete invariant returned by `Aligator` to `VCG` is $cnt + res = a + b \wedge (b > 0 \Rightarrow cnt = 0 \vee cnt > 0) \wedge (b \leq 0 \Rightarrow res = a \wedge cnt = b)$, as also presented in Example 2.3.

EXAMPLE 2.7 For the loop of Example 2.4 the result of `AnalyseBound` is:

$$\texttt{Input: } \texttt{AnalyseBound}[\{x = x + y; y = y + 1\}, \quad x + y < 10,\ \{x_0 = 0, y_0 = 2, c_0 = 1, d_0 = 1\}$$

$$\texttt{Output: } 0 + 2 < 10 \Rightarrow x + y < 10 \vee (x = 9 \wedge y = 5)$$

where, x_0, y_0, c_0 and d_0 denote respectively the initial values of x, y, c and d.

Note that the invariant generation in `Aligator` might perform unsafe optimizations (since data types are not yet handled). For instance, it always reduces $2(\frac{x}{2})$ to x, which is only valid when x is even. For this reason, in order to ensure correctness of invariants, and subsequently to ensure soundness of `Valigator`, generating and proving proof obligations for invariants' validity is crucial in `Valigator`.

2.3 Proving Verification Conditions

`VCG` generates proof obligations (verification conditions) in the SMT-LIB [6] format in one of the following two logics: Quantifier-free Bit Vectors (QF_BV) or Quantifier-free Linear Integer Arithmetic (QF_LIA). QF_LIA is more limited as it only supports linear relation among variables, but it is usually faster at proving or disproving obligations.

`Valigator` can either feed the proof obligations directly into a theorem prover or dump them to disk. `Valigator` has been tested with `CVC3` and `Z3` for linear arithmetic and `STP` for bit vectors. However, any other prover handling one of the mentioned logics and the SMT-LIB format could be used as well. Note that these two logics are decidable; the provers named above are sound and complete.

EXAMPLE 2.8 Using `CVC3`, `Z3`, or `STP`, all proof obligations listed in Example 2.2 are successfully proved in less than 3 seconds on a 2.0GHz machine.

3 Conclusion

By combining automated reasoning and symbolic computation, `Valigator` allows us to verify programs annotated with pre- and postconditions, and offers automatic support for inferring invariant properties of P-solvable loops. The approach was successfully tested on several examples, some of which are presented in this paper.

So far, using only preconditions of loops, bound assertions are derived under the assumption that quantifier elimination yields an exact integer solution on the number of loop iterations. However, in many cases such a solution does not exist. A possible extension of our approach would be to use an interval approximation of the bound on the number of loop iterations.

The class of loops for which `Valigator` can automatically infer invariants is limited by constraints on the program structure; `Valigator` can handle branches inside the loop as long as they commute. We are trying to extend our approach to programs with non-commuting branches, and thus with more complex flow structure.

We are also interested in generalizing the framework to programs on non-numeric data structures such as arrays and lists.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. J. Barnes. *High Integrity Software - The Spark Approach to Safety and Security*. Addison-Wesley, 2003.
3. M. Barnett, B. Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. of FMC*, 2005.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ programming system: An overview. In *Proc. of CASSIS*, volume 3362 of *LNCS*, 2004.
5. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of CAV*, volume 3114 of *LNCS*, pages 515–518, 2004.
6. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
7. J. Brauburger and J. Giesl. Approximating the Domains of Functional and Imperative Programs. *Sci. Comput. Programming*, 35(1):113–136, 1999.
8. B. Buchberger. An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *J. of Symbolic Computation*, 41(3-4):475–511, 2006.
9. G. E. Collins. Quantifier Elimination for the Elementary Theory of RealClosed Fields by Cylindrical Algebraic Decomposition. *LNCS*, 33:134–183, 1975.
10. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*, pages 238–252, 1977.
11. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
12. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a Theorem Prover for Program Checking. *J. of the ACM*, 52(3):365–473, 2005.
13. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. of PLDI*, pages 234–245, 2002.
15. V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. of CAV*, 2007.
16. L. Kovács. Reasoning Algebraically About P-Solvable Loops. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 249–264, 2008.
17. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06u, Iowa State University, 2003.
18. P. Müller, J. Meyer, and A. Poetzsch-Heffter. Programming and Interface Specification Language of Jivespecification and Design Rationale. Technical Report 223, University of Hagen, 1997.
19. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
20. M. Odersky. The Scala Language Specification. `http://www.scala-lang.org`, 2008.
21. S. Owre, N. Shankar, and J. Rushby. VS: A Prototype Verification System. In *Proc. of CADE11*, 1992.
22. H. Seidl and M. Petter. Inferring Polynomial Invariants with Polyinvar. In *Proc. of NSAD*, 2005.
23. J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 299–312, 2001.
24. S. Wolfram. *The Mathematica Book. Version 5.0*. Wolfram Media, 2003.