SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Integrated Program Verification Tools in Education

G. de Caso[1*], D. Garbervetsky[1,2] and D. Gorín[3]

[1]*Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina*
[2]*CONICET, Buenos Aires, Argentina*
[3]*Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany*

SUMMARY

Automated software verification is an active field of research which has made enormous progress both in theoretical and practical aspects. Even if not ready for large-scale industrial adoption, the technology behind automated program verifiers is now mature enough to gracefully handle the kind of programs that arise in introductory programming courses. This opens exciting new opportunities in teaching the basics of reasoning about program correctness to novice students. However, for these tools to be effective, command-line-style user-interfaces need to be replaced. In this paper we report on our experience using the verifying-compiler for PEST in an introductory programming course as well as in a more advanced course on program-analysis. PEST is an extremely basic programming language but with expressive annotations capabilities and semantics amenable to verification. In particular, we comment on the crucial role played by the integration of this verifying-compiler with the ECLIPSE integrated development environment. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Education; formal methods; automated program verification; ECLIPSE plug-in

1. INTRODUCTION

E. W. Dijkstra is well-remembered, among many other things, by his passionate beliefs on how programming ought to be conceived as a discipline and, ultimately, how it ought to be taught. His textbooks (e.g., [17]) inspired and influenced the curricula of programming courses all around the world.

In a rather crude simplification, we may summarize his view as follows: worthy programmers must be able to *reason*, in a sound way, about their code. In particular, programmers must possess enough tools and skills as to establish the (partial) correctness of their algorithms. Clearly, a mere understanding of the operational meaning of programming language constructs is not enough to achieve this goal. One needs to be familiar with, at least, the most basic formal methods and concepts: *state descriptions*, *Hoare triples*, *loop invariants*, etc.

Old habits die hard and bad habits die harder. To be effective, formal concepts for program reasoning should be introduced early on in introductory programming courses. This has been a common and recommended practice for more than ten years [1, 7]. Yet, it proves to be a very challenging task, both for students and educators. Formal concepts are much harder to assimilate

*Correspondence to: Intendente Güiraldes 2160, Ciudad Autónoma de Buenos Aires (C1428EGA), Argentina. E-mail: gdecaso@dc.uba.ar

Contract/grant sponsor: UBACyT-20020090300064/20020100100813, MinCyT PICT-2010-235, CONICET-PIP-11220110100596,

for students than the more immediate operational meaning of program statements; moreover the value of learning formal methods tends to be initially unappreciated.

Many things can be said about the difficulties of novice students with formal concepts, but there is one aspect we perceive as a salient limiting factor: for many of them formal methods constitute their first actual contact with mathematical practice. With this we mean that they are required to develop, almost from scratch, an intuitive understanding of what constitutes a *proof*, as understood by a mathematician (in its informal meaning, not in proof-theoretical terms) and what fails to be one.

Getting used to mathematical reasoning takes time and this is, we think, one of the biggest challenges for educators in introductory programming courses. Moreover, for many students this process can take well more than one semester and it is even more challenging for educators to avoid leaving them behind.

Now, in recent years there have been major breakthroughs in automated software verification technology (see e.g., [18]). In particular, it is now possible to extend programming languages with annotations for pre and postconditions, loop invariants, etc., and make a verifying-compiler turn these proof-obligations into verification conditions that are then fed to an SMT solver. ESC/JAVA [21], SPEC# [5] and DAFNY [27] are three well-known examples of this approach. Even if not ready for large-scale industrial adoption, it has already been shown that the state-of-the-art is mature enough to handle classical textbook examples gracefully [28].

These novel tools open exciting new opportunities in teaching basic (and also advanced) programming concepts. For starters, they enable a form of heuristic learning that can be invaluable in the development of sound mathematical intuitions. For example, after a brief theoretical introduction, students can investigate given examples trying to find a loop invariant that the verifying-compiler will actually accept, and learn from the rejected ones. Additionally, the hands-on experience can stimulate those students that are easily put off by more theoretical subjects.

Still, most formal method tools are developed by academics, for academics. Their user interface is simply unsuitable for an effective adoption in the classroom. One needs to present the tool in a format that is intuitive, familiar and attractive for students. Moreover, errors and other valuable feedback must be displayed in a non-threatening way. Our experience shows that this can be achieved in an extremely cost-effective way by writing plug-ins for modern Integrated Development Environments (IDEs). In particular we report here on the usage of our experimental language PEST, as a teaching tool for first-year undergraduate students in computer science, and the role played by the BUDAPEST ECLIPSE plug-in, which was especially built for this task.

The article is structured as follows. We begin in Sec. 2 with an introduction to the PEST language, covering its motivation and aim; its main features as presented to undergraduate students; and also a more formal presentation of its theoretical underpinnings. In Sec. 3 we discuss the BUDAPEST plug-in, which played a crucial role in the introduction of PEST as a teaching tool. This experience is briefly described in Sec. 4. We also report on the usage of PEST in an advanced course on program analysis: also in this rather different context a front-end such as BUDAPEST can be extremely helpful. We conclude in Sec. 5 and 6 with related work and some final words. This is an extended and revised version of [11].

2. THE PEST PROGRAMMING LANGUAGE

Formal automated software verification regained in recent years the attention of the community. There are at least two reasons behind this resurgent success: on the one hand, there were crucial developments in automated theorem proving in the last fifteen years, with SAT and SMT solvers finally reaching industrial strength; on the other, the focus has been shifted to verification of *partial specifications* which somehow overcomes many of the objections raised by De Millo et al. in their classical paper [30]. Automatic verification of partial specifications is regarded today as an error-detection procedure and, as such, akin to traditional forms of testing.

There is a particular form of software verification that interests us, of which SPEC# and ESC/JAVA are prime examples. In this case, the source code is annotated with special assertions,

DOI: 10.1002/spe

normally in the form of method pre and postconditions, loop and class invariants, etc. Verifying-compilers generate verification conditions (VCs) from the annotated sources, which are fed into automated provers [23].

Just like SPEC# is based on a dialect of C# and ESC/JAVA consists of JAVA code with JML [26] annotations, it is fair to say that there are ongoing research efforts in this flavour of automated verification for the majority of programming languages in use (e.g., [8, 36]) The rationale is to lower the barriers to adoption by giving practitioners tools for verifying the code they are writing today. Now, while this is an undeniably sensible plan, the resulting "programming-language-with-annotations" regarded as a whole usually ends-up being not entirely satisfying. We find among the main reasons:

Lack of cohesion. Annotations are usually introduced as a "patch" to the language. Most of the time, this is done in a way such that regular compilers and IDE-tools regard them as mere comments. Moreover, most programming languages provide a way to perform *optional run-time assertion checks* (typically by way of an `assert` directive). These are usually used to validate pre and postconditions or invariants and are, thus, the run-time counterparts of the verification annotations. But despite their dual nature, both mechanisms have normally no syntactical relation whatsoever.

Redundancy. In statically typed languages, the type of the input and output variables of a function are clearly part of its contract. But this means that one ends-up with two completely unrelated ways of specifying contracts: one enforced by compilers (types) and the other by static checkers (the additional annotations).

Missed optimization opportunities. Optimizing compilers cannot leverage on program annotations in the same way they currently do on type information. As a simple example, one would want such a compiler to completely eliminate run-time assertion checks whenever it can statically prove that the assertion always hold (a form of type-erasure). Dead-code elimination can be made more effective if program annotations are taken into account.

Inadequate semantics. We can most certainly exclude *"to ease automated verifiability"* from the list of goals that have driven the design of most modern-day programming languages. We cannot know for sure if today's mainstream languages would have been as popular without features such as complex inheritance mechanisms, uncontrolled method reentrancy or unrestricted aliasing. Nevertheless, the fact that the designers of SPEC# already had to diverge in slight ways from C#'s semantics [5] is indicating, in our opinion, a new driver for the programming languages to come.

Motivated by these concerns, PEST [12, 13] originates as an experiment in programming language design. In its current incarnation, it is a very basic while-style, recursion-free, multi-procedural, monomorphic language; with integers, Booleans and arrays (support for user-defined types and reference types are planned but have not been added to the mix yet). Yet, it has an expressive annotation mechanism and supports several forms of annotation inference to minimize unnecessary verbosity. The minimalistic nature of PEST encourages the experimentation with new language constructs.

While PEST is very far from being a language for every day use, we have found its niche as a teaching aid. As we shall see next, because of its light syntax, PEST can be used almost as pseudo-code, which means it can be added to an existing course without causing much disruption.

Figure 1 shows the definition of a simple procedure in PEST. Keywords **:?** and **:!** introduce pre and postconditions respectively. Procedure arguments are read-only unless listed under **:∗**, in which case they behave as input-output variables; the latter can appear as left-hand-sides of the assignment operator and their initial value can be used using the **@pre** keyword both in annotations and in program text. In this example, all variables are inferred to be of integer type, since the greater-than-or-equal operator[†] takes integers as arguments.

Apart from classical Boolean operators, Boolean expressions may include *bounded* first-order quantification, as illustrated in Fig. 2 where a procedure containing a while-loop iterating over an

[†] As usual, >= denotes greater-or-equal-than; the similar-looking => operator stands for Boolean implication.

DOI: 10.1002/spe

max.pest

```
max(a,b,c)
:? true
:! (a >= b => c = a) && (a < b => c = b)
:* c
{
  if a >= b then
    c <- a
  else
    c <- b
}
```

Figure 1. A PEST procedure definition.

arrayMax.pest

```
arrayMax(A, m)
:? |A| > 0
:! forall k from 0 to |A|-1 : m >= A[k] &&
   exists k from 0 to |A|-1 : m = A[k]
:* m
{
  m <- A[0]
  local i <- 1
  while i < |A|
      :?! 1 <= i && i <= |A|
       && forall k from 0 to i-1 : m >= A[k]
       && exists k from 0 to i-1 : m = A[k]
      :# |A| - i
   do
     local t <- 0
     local e <- A[i]
     max(e, m, t)
     m <- t
     i <- i + 1
}
```

Figure 2. A PEST procedure containing a while-loop.

array is shown. Loop invariants are introduced using the **:?!** keyword and exactly one loop variant must be provided, using the **:#** keyword. Loop variants are needed for the purpose of proving loop termination. A loop variant has to yield a positive monotonically decreasing value from one iteration to the next.

Figure 2 also illustrates a procedure call. Only variables are allowed as actual arguments and they are enforced to be syntactically distinct (this imposes a strict control over aliasing which simplifies reasoning).

2.1. *Operational semantics*

Operationally, what distinguishes PEST from mainstream while-style languages is that annotations have meaning[‡]. Roughly speaking, they are interpreted as runtime assertions, where a failed assertion corresponds to a *stuck* computation. We make this more precise by exhibiting some of the semantic clauses of the language (for the full set of rules, refer to [12]).

[‡]Other languages have followed this path before (see [5] for a historical description), but most commercial/industrial languages do not.

DOI: 10.1002/spe

$$\frac{[\![g]\!]_\sigma = \text{false} \quad [\![inv]\!]_\sigma = \text{true}}{\sigma \triangleright \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \triangleright \sigma}\ \text{(O-WHILE-F)}$$

$$\frac{\begin{array}{c}[\![g]\!]_\sigma = \text{true} \quad [\![inv]\!]_\sigma = \text{true} \\ [\![var]\!]_\sigma > 0 \quad \sigma \triangleright s \triangleright \sigma' \\ [\![var]\!]_{\sigma'} < [\![var]\!]_\sigma \\ \sigma' \ominus \text{locals}(s) \triangleright \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \triangleright \sigma''\end{array}}{\sigma \triangleright \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \triangleright \sigma''}\ \text{(O-WHILE-T)}$$

Figure 3. Operational semantics of PEST's **while**-statement.

$$\frac{\begin{array}{c}[\![\text{pre}(proc)]\!]_\rho = \text{true} \\ \rho \triangleright \text{body}(proc) \triangleright \rho' \\ [\![\text{post}(proc)]\!]_{\rho'} = \text{true}\end{array}}{\sigma \triangleright \textbf{call } proc(cp_1, \ldots, cp_k) \triangleright \sigma\{cp_1 \mapsto \rho'(p_1), \ldots\}}\ \text{(O-CALL)}$$

where $\rho(p_i) \stackrel{\text{def}}{=} \sigma(cp_i)$ and $\rho(p_i\textbf{@pre}) \stackrel{\text{def}}{=} \sigma(cp_i)$ for $0 \leq i \leq k$

Figure 4. Operational semantics of PEST's function call.

Rules are expressed in terms of state transformations. A state σ is a function that maps program variables to concrete values of the proper type. The intended meaning of the expression $\sigma \triangleright s \triangleright \sigma'$ is that of a *big-step* transformation: starting in the initial state σ, program s eventually stops, and the resulting state is σ'. Notice that programs in PEST are total: every program either stops or gets *stuck* in the middle of a computation, the latter is typically because of a failed assertion.

Figure 3 gives the semantics of the **while** statement. We use the following auxiliary notation: $\sigma\{v \mapsto n\}$ denotes the state that coincides with σ except, perhaps, in the value for v which, in the former, is n; $\sigma \ominus \text{locals}(s)$ is the restriction of σ to a domain that does not contain any variable that occurs for the first time in the body s of the loop; while $[\![e]\!]_\sigma$ denotes the value of a *pure expression* e under σ. An expression is *pure* when it is side-effect-free. Pure expressions can occur in annotations; built from variables, arithmetical and/or logical operators, etc.; see [12] for more details.

The upper-half of each rule lists a series of premises that must hold in order to conclude the bottom-half. The premises of the rules for the **while**-statement are (semantically) disjoint, as one expects of a deterministic language. They are not comprehensive, though: when no premise can be matched, execution of the statement gets stuck. Observe that, in particular, if $[\![inv]\!]_\sigma \neq \text{true}$ then no rule applies and the computation cannot continue. Rule O-WHILE-F captures the behaviour of the case when the guard is false, in which case the state is not affected (the language syntax guarantees that guards are free of side-effects). The second rule corresponds to an execution of the body; the variant must be above zero in order to continue and the variant in σ', the state after execution of the body, must evaluate to a smaller value or the computation gets stuck.

Figure 4 shows the rule for procedure calls (for the sake of readability, we shall ignore here the **:*** qualifier and assume every procedure parameter to be of read-write type). The state ρ is used to bind formal and actual parameters and the precondition of the called procedure must hold for this state. The state ρ', if defined, corresponds to the result of executing the procedure body from ρ; the postcondition must hold in ρ'. Finally, actual parameters are updated with the final values assigned to the formal parameters. Notice that PEST disallows global variables.

It is straightforward to write a compiler for the PEST language based on these semantics. Annotations simply need to be evaluated at run-time to decide if the computation continues or gets stuck. Of course, this can be, in general, prohibitively expensive (in fact, because bounded quantification is allowed in annotations, they can define primitive recursive predicates). Now, just like a compiler for a language with a strong, static type system need not include dynamic type checks in a well-typed program, a PEST verifying-compiler is allowed to omit a runtime assertion-check if it can statically guarantee that it will always hold. For this, we need an abstract counterpart of the operational semantics, suitable for machine reasoning. This is presented next.

DOI: 10.1002/spe

$$\frac{p \models \mathrm{safe}(e) \quad \exists\, v'\ (p\lfloor v \mapsto v' \rfloor \ \wedge\ v = e\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow e\ \{q\}}\ \text{(S-ASSIGN)}$$

$$\frac{\begin{array}{c} p \models \mathrm{safe}(g) \\ p \wedge g \models p_1 \quad \{p_1\}\ s_1\ \{q_1\} \quad q_1 \models q \\ p \wedge \neg g \models p_2 \quad \{p_2\}\ s_2\ \{q_2\} \quad q_2 \models q \end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \{q\}}\ \text{(S-IF)}$$

$$\frac{\begin{array}{c} \textbf{true} \models \mathrm{safe}(inv) \quad inv \models \mathrm{safe}(var) \\ inv \models \mathrm{safe}(g) \quad p \models inv \quad inv \wedge g \models p' \\ p' \models var > 0 \quad \{p'\}\ var_0 \leftarrow var\ \ s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q \end{array}}{\{p\}\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \{q\}}\ \text{(S-WHILE)}$$

Figure 5. PEST static semantics (fragment)

2.2. *Hoare-style static semantics*

Instead of dealing with states like in the operational case, the static semantics is based on *predicates* (i.e., Boolean expressions) that describe a (possibly infinite) set of states. For any PEST sentence s, we write total-correctness Hoare triples of the form $\{p\}\ s\ \{q\}$, where p and q are Boolean expressions augmented with unbounded existential quantification. In this context, such a triple must be read "if s is executed starting from a state σ such that $\sigma \models p$, s will stop in finitely many steps in a state σ' such that $\sigma' \models q$". Here $\sigma \models p$ denotes that "predicate p holds at σ"; its formal definition is trivial. As is customary, we also use notation $b_1 \models b_2$, where b_1 and b_2 are predicates, to indicate that b_1 is *stronger* than b_2 (i.e., $\sigma \models b_2$ whenever $\sigma \models b_1$).

In general, pure expressions that refer to program variables can be undefined by many reasons: the use of partial functions (e.g., division); array accesses with out-of-range indices; dereference of null pointers; etc. However, it is usually possible, given an expression e to infer a Boolean expression $\mathrm{safe}(e)$ such that i) $\sigma \models \mathrm{safe}(e)$ is well-defined for all states σ, and ii) $\sigma \models \mathrm{safe}(e)$ implies that $[\![e]\!]_\sigma$ is well-defined, as in the following example:

$$\mathrm{safe}(a[i]\ /\ y) \equiv 0 \leq\ i \wedge i < |a| \wedge y \neq\ 0\,.$$

Boolean expressions use short-circuit semantics and the safe expression predicates need to take this into account. For example, $\mathrm{safe}(\alpha \wedge \beta) \equiv \mathrm{safe}(\alpha) \wedge (\alpha \rightarrow \mathrm{safe}(\beta))$.

A formal definition of these conditions is straightforward and can be found, for the case of PEST, in [12].

Figure 5 lists a selection of the static semantics rules of PEST (for the complete list, refer to [12]). Consider first the rule for assignments. The assignment can only get stuck if e is not well-defined, which is covered by the first premise. The second premise has a definitional role: $e_1\lfloor e_2 \mapsto e_3 \rfloor$ denotes the expression that results from replacing every occurrence of e_2 by e_3 in e_1; therefore, it states that q is a consequence of what was known prior to the assignment ($p\lfloor v \mapsto v' \rfloor$) and its effect ($v = e\lfloor v \mapsto v' \rfloor$). The existentially quantified variable v' stands for the value of v before the assignment (this requires unbounded quantification).

The premises of the rule for the **while**-statement can be seen as both a proof of the Fundamental Invariance Theorem for Loops [16] and a proof of termination using the loop variant. The predicate p' represents any state where the invariant and the guard hold; the loop body is augmented with an initial assignment to a *fresh* variable var_0 that is used to prove that the variant decreases.

A program that is correct with respect to the static semantic rules is called safe. In what follows, if π is a program and p a procedure, then π, p is the program obtained by appending p to π.

Definition 1 (Safe programs)
The set SAFE of programs is inductively defined as follows:

$$\frac{}{\emptyset \in \textsc{Safe}}\ \text{(SAFE-EMPTY)}$$

DOI: 10.1002/spe

```
sumArrayPositions(A, i1, i2, ret)
:* ret
{
  ret ← A[i1] + A[i2]
}
```

Figure 6. PEST procedure to compute the sum of two array positions

$$\frac{\pi \in \textsc{Safe} \quad \{\mathrm{pre}(p)\}\ \mathrm{body}(p)\ \{\mathrm{post}(p)\}}{\pi, p \in \textsc{Safe}}\ (\textsc{safe-extend})$$

It is then straightforward to correlate PEST's operational and static semantics.

Theorem 1 (Safe programs execute normally)
Let $\pi \in \textsc{Safe}$ and let p be a procedure in π. For each σ such that $\sigma \models \mathrm{pre}(p)$, there exists a state σ' such that $\sigma \triangleright \mathrm{body}(p) \triangleright \sigma'$ and $\sigma' \models \mathrm{post}(p)$.

The proof follows using a longish yet straightforward induction on the length of a derivation of $\{\mathrm{pre}(p)\}\ \mathrm{body}(p)\ \{\mathrm{post}(p)\}$. See [12] for all the details.

PEST's semantic rules can be regarded as a form of *typing rules* and, thus, safe programs correspond to *well-typed* programs. Arguably, the main difference with standard type systems is that PEST rules are *semantic* in nature. One can derive a syntactic system by replacing $\models$ by some of its proof-theoretical counterparts $\vdash$ together with its reasoning rules. Of course, the resulting system will be neither decidable nor complete (because it contains a first-order theory of the natural numbers). Alternatively, one can keep $\models$ in the rules and use SMT solvers and other automated reasoning tools as (incomplete) oracles for $\models$ when implementing them.

Just like it is possible to extract a *type inference* algorithm from a type system (e.g., [10]), it is not hard to turn PEST's static semantics rules into a pre/postcondition inference algorithm. In fact, it is possible to devise a set of inference rules that will give both pre and postconditions for a function that is simple enough. More precisely, if a precondition is given, a postcondition can be obtained by a top-down reading of the rules in a way that is reminiscent of symbolic execution. Starting from a postcondition, a precondition can be obtained using a variation of the notion of *weakest precondition*.

We show this by way of an example. Consider the procedure in Figure 6 which computes the sum of the two elements given at the indicated indexes and returns it over *ret*. Notice that this procedure indicates its set of modified parameters, but it does not contain annotations for precondition nor postcondition. On a first pass, one can compute a precondition that ensures that the procedure will stop (that is, without getting stuck). More formally, a predicate P needs to be inferred such that it is the weakest one that satisfies: $\{P\}$ *ret* ← *A*[*i1*] + *A*[*i2*] $\{\mathrm{true}\}$.

Such a P can be obtained by standard techniques; one can get, for instance:

$$P \equiv i1 \geq 0 \wedge i1 < |A| \wedge i2 \geq 0 \wedge i2 < |A|$$

In the presence of **while** statements, such P will depend on the user-provided invariant. In this case, the PEST operational semantics forces the provided invariant to hold on every iteration. The computed weakest precondition must reflect this [§].

Once P is obtained, a postcondition analysis can be used to find a Q such that $\{P\}$ *ret* ← *A*[*i1*] + *A*[*i2*] $\{\ Q\}$ holds. In this case:

$$Q \equiv ret = A[i1] + A[i2]$$

In practice, due to the mechanical nature of the inference, the pre and postconditions obtained may be hard to read. Besides, unbounded quantification is used in order to resolve assignments and,

[§]Observe that, in PEST, adding a loop-invariant may change the operational semantics of a program. For example, a total function may become non-total if a strong invariant is added that precludes certain initial values. It is thus correct to say that the weakest precondition derived from a user-provided invariant is in PEST the weakest precondition of the function.

DOI: 10.1002/spe

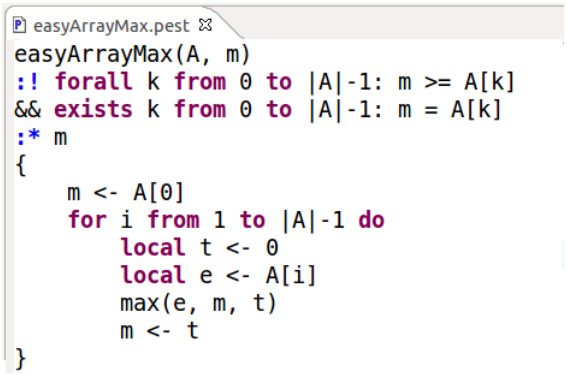

Figure 7. Using **for** to remove annotations.

thus, these conditions may not correspond to contracts that a PEST programmer is allowed to write. Overall, for simple enough functions, this can be a convenient feature and one subject to further research. Further details are given in [12, 13].

It is worth mentioning that inference is greatly simplified by some of PEST's design decisions. In particular, a strictly controlled variable aliasing and lack of recursion allow us to make important assumptions when inferring pre and postconditions. Part of our research agenda involves gradually incorporating to PEST these and other features (perhaps in restricted forms) that are taken for granted in most mainstream programming languages used today.

2.3. *Experimental language features*

One of the main motivations for PEST was exploring the design-space of a programming language focused on verifiability. An interesting example of the kind of constructions one can have in this setting is the *invariant carrying* **for**-statement.

From an operational point of view, it is roughly equivalent to the PASCAL-style *for* loop. The interesting bit is that using the underlying precondition inference engine of the language, one can heuristically guess its loop invariant.

Figure 7 exhibits a function written using this construct. Only the postcondition for the function is explicitly given; the precondition ($|A| > 0$) can be derived from the inferred loop invariant. The heuristic goes, roughly, this way:

1. The iteration variable i can be safely assumed to stay between bounds, provided it is not touched in the loop body, which is syntactically enforced. This is easily expressed in the invariant.
2. Variables that are not touched in the body remain constant during the loop, this is guaranteed by PEST's strong control on aliasing and can also be expressed very easily.
3. For the substantial part of the invariant, a candidate is guessed by syntactically replacing the iteration upper bound by the iteration variable in the loop postcondition (which is inferred from the function's postcondition). Of course, the correctness of this part of the invariant has to be statically verified.

Observe that the procedure in Fig. 7 computes exactly the same function as the one in Fig. 2. However, the former requires no loop annotations and no precondition (although one may want to include it for documentation reasons).

Figure 8 shows how the invariant is correctly guessed for the postcondition of the program depicted in Fig. 7. The differences are marked in bold.

The reader is referred to [13] for more details and for other invariant-carrying looping constructs in the PEST language.

DOI: 10.1002/spe

$$\mathbf{1} \leq \mathbf{i} \leq |\mathbf{A}| \wedge \forall k \ / \ 0 \leq k < \mathbf{i} : \quad m \geq A[k] \wedge \exists k \ / \ 0 \leq k < \mathbf{i} : \quad m = A[k]$$
$$\forall k \ / \ 0 \leq k < |\mathbf{A}| : m \geq A[k] \wedge \exists k \ / \ 0 \leq k < |\mathbf{A}| : m = A[k]$$

Figure 8. Inferred loop invariant for the *easyArrayMax* program (first line), original postcondition (second line)

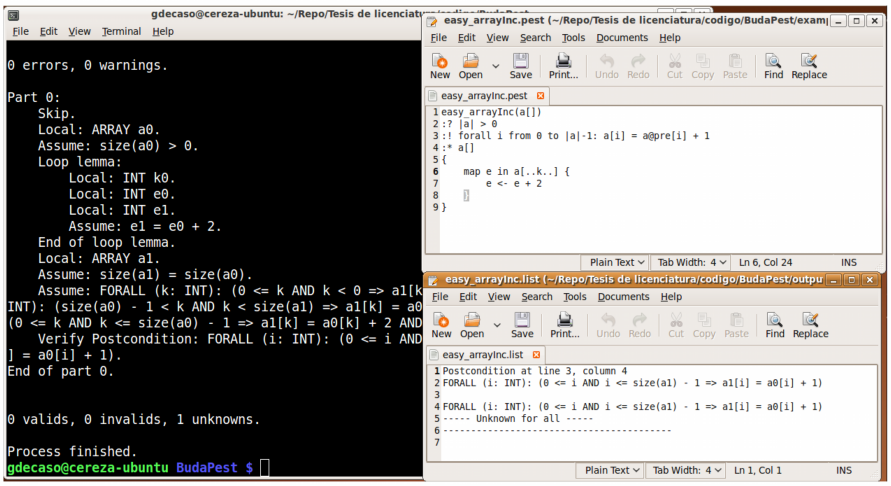

Figure 9. Using PEST before the plug-in was developed

3. THE BUDAPEST ECLIPSE PLUG-IN

The verifying-compiler for the PEST language is a non-trivial piece of software. It consists of around 13 KLOC of Java code and took half a year to develop. It contains a parser, a simple type-inference system, a generator of verification conditions, a symbolic execution engine, a simplifier for logical formulas and is capable of orchestrating several SMT solvers and automated theorem provers. All this internal sophistication was carefully wrapped in an austere, command-line based, user-interface.

Figure 9 exhibits a typical work-flow using the PEST compiler. On the upper-right, a text-editor window containing a PEST program is displayed; no syntax-highlighting is shown. The window on the left shows the output of running the PEST compiler; for complex program, it can span several screens. To figure out why a program fails to compile, one typically redirects the output to a file, and opens it from a text-editor, as shown on the bottom-right window.

Arguably, this is not at all different from most tools in academia. However, once the possibility of using PEST as part of an introductory course on algorithms was considered, it became immediately clear that its presentation had to be radically improved. The time and effort available for this was limited, though. After examination of several alternatives it was decided that a new front-end would be written in the form of a plug-in for ECLIPSE.

ECLIPSE [20] is an industrial strength, widely used, multi-language IDE, extensible via a sophisticated plug-in system. Support for different languages and tools can be added this way. It is used to develop applications in mainstream programming languages such as JAVA, C++ or PYTHON. Due to its popularity, availability and the fact that it is written and extensible in Java, just like the PEST verifying-compiler, it appeared as a natural choice. The plug-in was developed in two weeks' time.

3.1. *Overview of its features*

As in most IDE's, ECLIPSE allows the user to create and manage *projects*; each project is a collection of related files that can be edited, built, deployed, launched, debugged, etc. Each project can be of a different sort; the sort of a project is related to the language and/or frameworks used in it and, ultimately, to the plug-in that implements it. For example, developers can create JAVA-projects,

DOI: 10.1002/spe

C++-projects, etc. Inside an opened project the user can switch between several *perspectives*, which are collections of *editors* and *views*. Each perspective is relevant to one of many aspects of software development (editing, debugging, etc.).

The BUDAPEST plug-in fits well into the ecosystem of ECLIPSE plug-ins. Once installed, it provides a new type of project that allows the development of PEST programs. It defines a new PEST *perspective* which contains a customized text editor that properly highlights PEST syntax. The compiler is automatically invoked on the background every time the user saves a PEST file, which is the standard practice in ECLIPSE projects. Moreoever, the editor is integrated to a customized *Problems view* in which errors and warning are reported back to the user. Each error in this view is associated with a localized error marker in the code displayed in the editor. Finally, inferred pre and postconditions are also shown as blue markers in the PEST editor, allowing the user to hover over them in order to understand what the compiler actually inferred.

Figure 10 displays a snapshot of the PEST perspective and the plug-in integration with some of the most salient ECLIPSE views. The *Problems view* shows a comprehensive list of warnings, errors or pieces of information (such as inferred pre and postconditions) that the PEST program has. Special care was put in making messages as clear, accurate and informative as possible. For instance, observe that while the postcondition of procedure *max* contains two conjuncts, only the one that could not be proved is reported back. This is handled by splitting conjunctions into its conjuncts and attempting to prove each of these independently. This particular feature could not be implemented solely at the front-end level but required support from the underlying PEST compiler. However, we perceive it was worth the effort: novice students can get easily frustrated by long, cryptic or misleading error messages.

The *Console view* is used when executing PEST programs in order to input their parameters and obtain their output. Finally, the *Error log view* shows a list of errors or exceptions that may have been caused due to BUDAPEST malfunction. This is particularly useful during plug-in development or for end-users to report plug-in bugs together with the proper replication information back to the BUDAPEST plug-in developers.

Besides syntax highlighting, the PEST editor will also present the user with the inferred (or guessed) annotations, in case the features presented in Section 2.3 are used. The user can then decide to explicitly incorporate these annotations in the PEST program and then manually refine them.

ECLIPSE provides an extremely convenient plug-in discovery and installing mechanism via so-called *update sites*. These are crawled by ECLIPSE on demand and a list of available plug-ins is offered to the user. Dependencies are automatically tracked and fetched from their update sites as needed. Since it was expected that students would need to exercise at home, a suitable update site was set up for BUDAPEST.

Regrettably, an important part of the installation process could not be properly automated. PEST's verifying-compiler is written in Java and is, thus, portable to any platform capable of running ECLIPSE. It is not however, a stand-alone application, since it relies for the actual verification of the generated conditions on a daisy chain of different SMT solvers, like Z3 [14], CVC3 [6] and YICES [19]. Even though these different SMT solvers are accesed in a standard manner (currently, BUDAPEST uses the SMT-LIB format [35]), each prover has strengths and weaknesses in dealing with the different logical theories that can occur in the verification conditions. If one prover fails to provide a definite answer for a given verification condition, the next one is tried. Now, these solvers are typically distributed in binary form and the platforms supported by each vary. In addition, some of their licenses oblige the user to read and accept an agreement before installing. No simple way was found to handle this problem; instead students were indicated to manually install the SMT solvers. To cope with availability, BUDAPEST defines a preference pane where users can indicate and configure the provers available.

3.2. BUDAPEST *from the inside*

The BUDAPEST plug-in has a modular architecture. The compilation and verification processes are handled by the PEST verifying compiler infrastructure, which was hardly modified during the

DOI: 10.1002/spe

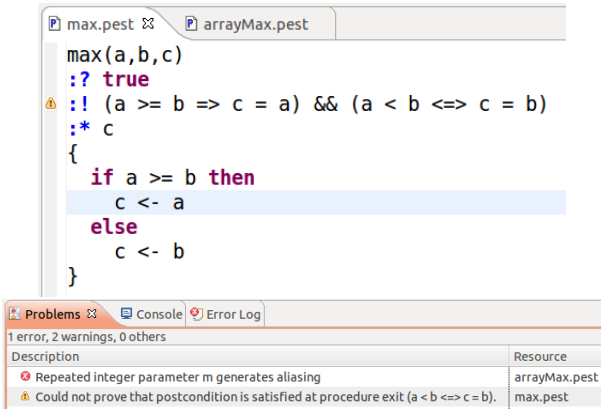

Figure 10. PEST editor with problem markers; problems view depicting VCs that failed.

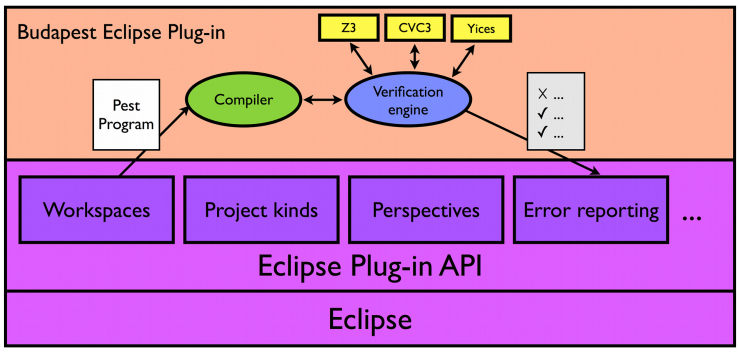

Figure 11. BUDAPEST plug-in architecture

plug-in development. The plug-in is responsible for binding the ECLIPSE editor with the PEST file, sending its contents to the verifying-compiler and fetching the results back from it. In Figure 11 we present a diagram featuring this interaction.

It is worth comparing the effort needed in writing the PEST compiler and its plug-in based front-end. Both projects were designed and developed by the same team, which had a strong background in Java programming at the start of the project, and also some prior experience in ECLIPSE plug-in development (this means we can disregard learning times).

As already mentioned the PEST compiler has 13 KLOC and was developed in approximately 5 months. BUDAPEST, on the other hand, required 2 weeks and 1.5 KLOC; that is, it was simpler by a factor of around 10, both in code size and development time. We can further break-down the effort invested in BUDAPEST: the PEST editor, the largest module in the plug-in has less than 700 LOC and most of them are boilerplate and were derived from templates available on-line. Other components of the plug-in have extremely compact definitions, such as the PEST perspective (22 LOC) or the preferences pane (less than 150 LOC). This can be probably explained by the fact that we were writing the sort of plug-in at which ECLIPSE excels. In any case, in retrospective it has been an extremely cost-effective choice.

4. CLASSROOM EXPERIENCE

We report on two experiences of using the PEST verifying-compiler as a teaching aid in undergraduate courses. The first one is what motivated the development of the BUDAPEST plug-in described in Sec. 3; the second one is an advanced course with a completely different focus

DOI: 10.1002/spe

where the PEST compiler was used as a command-line tool. While no value was initially perceived in introducing BUDAPEST in the latter, we will see that in retrospect it could have been useful too.

Both courses were taught at the Computer Science Dept. of the University of Buenos Aires, a public university and the largest in Argentina. Despite having no tuition, the majority of the Computer Science students come from the middle and upper socioeconomic segments and have comparable secondary education. The proportion of students that arrive with previous programming experience has been continuously decreasing in the last 15 years and are by far a minority by now.

4.1. PEST *in an introductory algorithms course*

The course covers the basics of functional and imperative programming, with a strong accent on formal specification and manual verification. No previous background in formal logic, algorithms or programming languages is assumed. It spans over 15 hours a week (divided in theoretical lectures, problem-solving classes and labs) and lasts weeks.

Typically, between 80 and 130 students enroll in this course every semester. They have previously completed introductory courses on Linear Algebra, Calculus and Algebra. They thus have rudimentary mathematical skills and are familiar, in particular, with basic forms of inductive reasoning.

This is the course where students first learn how to program and to reason about programs. It starts with an introduction to propositional and first-order logics, and how to formally specify problems. They then learn to program simple algorithms using a functional language and prove their correctness by induction. The last part of the course is devoted to (non-recursive) imperative programming, covering the notion of state, Hoare-logic, loop invariants, etc. Common assignments in this last part include writing programs to perform basic operations on arrays, such as linear and binary searching or non-recursive sorting. Emphasis is put in establishing the correctness of the algorithms. In this context, students tend to find the Invariance Theorem for loops particularly challenging.

To prevent excessive disruptiveness, PEST was introduced so far at the very end of the course, with the objective of contributing to a better assimilation of the more difficult concepts. By this time, students have acquired a number of technical skills and are able to write a formal property if asked to. What most of them struggle with is in finding what constitute a suitable loop-invariant.

The methodology used is to hand them algorithms they already learned during the course, but rewritten in PEST and with subtle specification and coding bugs seeded in them. Typical errors include mishandled border cases; incorrect statement ordering; too-weak, too-strong or even missing invariants, etc. Students have to interactively find and correct these errors, with the assistance of the detailed error reporting mechanism the tool provides.

In our observations, students seem to find themselves comfortable with the language and the working environment, even when very little time is devoted to explaining them. PEST resembles the pseudo-code previously used in classes and, since BUDAPEST tries to follow the guidelines of ECLIPSE, the interface is intuitive to them (most had previous experience with ECLIPSE from assignments given during the semester).

The students show great interest once they see the compiler in action. In informal interrogations afterwards some have said that they are happy when they realize that the concepts learned during the course allow them to have an idea of what the compiler is doing behind the scenes. Others seem to like that the tool deals with the formal proof, which they find to be the most tedious and mechanic part of the verification process.

An average of 75% of the students that made it to the last part of the course passed the exams. These involve writing loop-invariants from scratch. Although this constituted a slight increase with respect to previous semesters, there were too many other factors involved so we do not see any statistically valid correlation. However, at a qualitative level, we do think it was a very positive experience, that satisfied both students and educators. It is clear to us that this would not have been possible without the integration to ECLIPSE. During the class, it is crucial to easily and effectively perform verification using the push-button capabilities the tool offers. When combined with the

DOI: 10.1002/spe

syntax highlighting and integrated error reporting, the BUDAPEST plug-in resulted in a captivating classroom experience.

4.2. PEST *in an elective course on program-analysis*

In the second experience, the PEST compiler was used as a simple case-study and not so much for didactic purposes. The context was an elective course on program analysis with a strong emphasis on program verification via theorem proving, data-flow analysis and type theory. Students attending this course are mostly fourth-year undergraduates and there are also some Ph.D. students. They are all already familiar with the concept of formal proofs of correctness and are capable of doing them by hand.

The lectures in this course aim to give a formal theoretical background on the subject, providing a solid foundation for the techniques surveyed. The lectures also present an operational view of the verification process, introducing the concepts behind typical verification tools.

In order to exercise some of these concepts, some key modules were deliberately removed from the PEST's verifying-compiler, most notably, the one responsible for the generation of verification conditions. This incomplete implementation is provided to the students, who are given the assignment of adding the missing bits[¶]. They were given three weeks to finish the assignment, and all of them were able to produce a working solution in that time.

The BUDAPEST plug-in was not introduced in this course: being advanced students, accustomed to using command-line tools, no value was perceived in using an IDE-based front-end. It was observed, though, that one of the things that troubled them the most was debugging their implementations. In particular, they had to continuously go back and forth between the rather long logs of their PEST compiler version and the source code being compiled in order to understand what was going wrong. Since a lot of attention was put in simplifying BUDAPEST's error-handling mechanism, students could have exploited this.

It is interesting to note how this would work. ECLIPSE plug-ins can be developed in ECLIPSE. When doing this, it allows the programmer to run the plug-in under development, for which a second instance of ECLIPSE is launched. Due to BUDAPEST's modular design, which performs the bindings with ECLIPSE but delegates the proper verification and compilation processes to the core compiler, it is possible to test the compiler by running BUDAPEST in developer mode. Should errors or exceptions be raised, the ECLIPSE Error log view can be used in order to understand what went wrong. In retrospective, the actual development of PEST's compiler could have been a little simpler if BUDAPEST had been developed in parallel.

5. RELATED WORK

The static verification of annotated programs, as done in PEST, is the subject of very active research. A complete survey of the area is outside the scope of this note. We will focus, instead, on the use of this sort of tools in education and, to a lesser extent, their integration with IDE's. It must be noted beforehand that the use of automated verification tools in the classroom has very few antecedents, and not many of them have been reported in scientific meetings or publications. Still, we will comment on some previous experiences we are aware of.

The SPEC# programing language [5] is an extension of the C# language including method contracts; object and loop invariants; non-null types and a set of type-based annotations to enforce object encapsulation. The latter relies on an ownership type system [4]. The verifying-compiler for SPEC# generates verification conditions that are passed to the Z3 SMT solver [14] for proving them. The compiler is available as a command-line tool, but it was also integrated to the Visual Studio IDE, which is a typical environment for C# developers although limited in terms of platform availability. Moreover, the verifier has been made available through a web-based front-end from which simple

[¶]Actually, handling loops was an optional assignment.

DOI: 10.1002/spe

programs can be verified without any local requirements other than an Internet connection. This seems as an interesting alternative way of introducing this kind of tools in the classroom [31].

SPEC# was extended in [29] with constructs and operations for handling set comprehensions (e.g., operations such as max, sum, count, etc.). This allowed the authors to naturally encode several textbook examples, and these were successfully verified by the tool. Their stated long term goal is to make SPEC# a suitable language for teaching algorithms both in introductory and advanced courses. One of the authors of [29] recently reported her experience in using SPEC# as a teaching aid, both in introductory courses as well as in some covering advanced topics such as ownership types, inheritance, and aggregation [31]. In both cases the students used the tool as a black box component to verify their programs. Her conclusions are similar to our own observations: students get motivated by the tools and show interest on their inner workings. Error-messages were perceived as hard-to-understand; this reinforces our belief that one of the most important challenges in using these tools in education is developing an adequate error message handling sub-system.

DAFNY [27] is an experimental programming language, mounted on the technology behind SPEC#. Designed for the study of program verification, DAFNY aims to be expressive enough as to achieve full functional correctness of its programs using automatic decision procedures. It is an object based language with dynamic allocation but no subclassing. Unlike PEST, DAFNY allows the user to define inductive data types and recursive functions. At the specification level, it incorporates sets and sequences as native types and also ghost fields. The latter resemble normal object fields, but are used only for the sake of specifications; they are not part of the in-memory representation of objects. One interesting use of ghost fields in DAFNY is in framing method calls, following the dynamic frames methodology [24, 22].

Compared to SPEC#, DAFNY is a relatively simple language and is arguably a better choice for classroom usage. However, it is also a rather unusual language, which heavily relies on ghost-fields (that must be manually updated by the programmer). This may be difficult concept to grasp for novice students, although it is certainly possible that a suitable fragment of DAFNY could be successfully used in introductory courses dealing only with arrays. For advanced courses, DAFNY seems particularly well-suited.[‖] DAFNY includes a VISUAL STUDIO plug-in providing basic features such as syntax highlighting and error reporting. It also includes a sort of debugger [25] which helps users understand the output of the verifier.

In [34] an experience is reported on teaching program verification using JML and ESC/JAVA2 [9]. These tools were presented at the end of a course and the authors report the enthusiasm shown by their students. Interestingly, they list some limitations that were found to be a little discouraging for students: some programs were just too hard for the back-end prover; the ESC/JAVA2 verification engine did not support all Java constructs and the required ECLIPSE plug-in was apparently not easy to install (probably due to unstable dependencies).

Another experience is reported in [2]. In this case, the authors used an approach in which students were encouraged to write loop invariants before writing their code. This was tried in two courses. One was an advanced course for graduate students where they were asked to prove program correctness by manually generating the verification condition and then prove them using the PVS prover [33]. The second experience was in a beginners course, where programming is taught from scratch using their SOCOS environment [3]. The latter is a tool that assists this invariant-based programming methodology by providing a diagrammatic environment for the specification, implementation, verification and execution of procedural programs, relying on the SIMPLIFY SMT solver [15] to perform automatic proving of the verification conditions and PVS for those that cannot be automatically proved.

In this case too, the authors observe that a suitable error reporting mechanism in the tools is a clear need when using this tools in education. In particular, they comment on the difficulties of dealing with PVS. They also warn that the use of tools at the very beginning can be harmful since students

[‖] At the time of writing, it has been indeed used at least in one course on reasoning about program correctness (CS-116) at the California Institute of Technology.

DOI: 10.1002/spe

might be tempted to do a (probably non-converging) test and debug strategy instead of thinking beforehand about the constraints needed for the invariants. We essentially agree with this opinion.

6. CONCLUSIONS

In this work we presented PEST, a simple programming language targeted both for teaching basic programming concepts and for experimenting in language design. It is a basic language designed to facilitate automated program verification, lacking many of the features of modern programming languages that make program development easier but may complicate automated program analysis.

We reported on our experience about using PEST in the classroom, both in an introductory algorithms course, with a focus in reasoning about program correctness, and in an advanced course in program analysis where the goal was teaching automated program verification.

Our experience is encouraging: students were very enthusiastic about using the tool with their own programs. Students less inclined to theoretical aspects seem surprised to discover that the formal tools learnt can be applied in practice and show genuine interest in understanding the tool internals. A key aspect in achieving this was the use of BUDAPEST, an attractive and intuitive front-end to the verifying-compiler.

We believe researchers should take the time to provide an attractive, simple-to-use front-end to their tools by exploiting the plug-in based extension facilities provided by modern development environments. An extensible IDE like ECLIPSE can make the additional effort almost negligible and this make the tools available to a wider public.

Availability and ease of installation is a crucial aspect for the adoption of a plug-in, both inside and outside of the classroom. We have found that, due to its massively ported virtual machine, a JAVA-based plug-in framework like ECLIPSE's can greatly contribute to both aspects. However, many tools in academia often rely on third-party components, usually in binary form; for plug-in developers, packaging, distributing, configuring and licensing these components within the plug-in is still a challenge. We believe there is room for improving this situation.

Regarding the classroom experience it would be interesting to conduct an empirical study in order to evaluate how students perform when presented with a tool like PEST in different stages of the course. In a nutshell, the idea is assess whether students that use these tools from the beginning of a course are, say, more proficient at writing invariants than students that only used PEST at a single lab session in the end of the course. Doing this in a principled way is a challenging task.

With respect to the PEST language itself, we would like to extend it with features that would increase its expressiveness without sacrificing verifiability. We plan to allow the programmer to define and use her own data types and provide means to reason about representation invariants. Adding dynamic memory support is another priority but, in order to keep a verifiable language, we believe we must enforce an alias control mechanism such as [32].

ACKNOWLEDGEMENT

This work was partly funded by the national grants UBACyT 20020090300064, MinCyT PICT-2010 235, PICT-PAE 02278 and Microsoft Research Software Engineering Innovation Foundation award 2010.

REFERENCES

1. J.M. Atlee, R.J. LeBlanc Jr, T.C. Lethbridge, A. Sobel, and J.B. Thompson. Software engineering 2004: ACM/IEEE-CS guidelines for undergraduate programs in software engineering. In *Proc. of the 27th International Conference on Software Engineering*, pages 623–624. ACM, 2005.
2. R.J. Back. Invariant based programming: basic approach and teaching experiences. *Formal Aspects of Computing*, 21:227–244, 2009. 10.1007/s00165-008-0070-y.
3. R.J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. *Tests and Proofs*, pages 61–78, 2007.
4. M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

DOI: 10.1002/spe

5. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*. Springer, 2005.
6. C. Barrett and S. Berezin. CVC lite: A new implementation of the Cooperating Validity Checker. In *Proc. of the 16 th International Conference on Computer Aided Verification (CAV '04)*, pages 515–518, 2004.
7. L.N. Cassel, M. Caspersen, G. Davies, R. McCauley, A. McGettrick, A. Pyster, and R. Sloan. Curriculum update from the ACM education board: CS2008 and a report on masters degrees. In *ACM SIGCSE Bulletin*, volume 40, pages 530–531. ACM, 2008.
8. S. Chatterjee, S.K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. *TACAS'07*, pages 19–33, 2007.
9. D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128, 2005.
10. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.
11. G. de Caso, D. Garbervetsky, and D. Gorín. Pest: from the lab to the classroom. In *Proc. of the 1st Workshop on Developing Tools as Plug-ins*, pages 5–8. ACM, 2011.
12. G. de Caso, D. Garbervetsky, and D. Gorín. PEST formal specification (v1.1). Technical report, Universidad de Buenos Aires, 2012.
13. G. de Caso, D. Gorín, and D. Garbervetsky. Reducing the number of annotations in a verification-oriented imperative language. In *APV'09*, 2009.
14. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
15. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
16. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.
17. E.W. Dijkstra and W.H.J. Feijen. *A method of programming*. Addison-Wesley series in computer science. Addison-Wesley, 1988.
18. V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
19. B. Dutertre and L. de Moura. The Yices SMT solver. *Available at http://yices.csl.sri.com/, August*, 2006.
20. Eclipse Integrated Development Framework. http://www.eclipse.org. [16 September 2011].
21. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02*, pages 234–245, 2002.
22. D. Garbervetsky, D. Gorín, and A. Neisen. Enforcing structural invariants using dynamic frames. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 65–80, 2011.
23. D.I. Good, R.L. London, and W.W Bledsoe. An interactive program verification system. In *ICRE*, pages 482–492, 1975.
24. I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. *FM 2006: Formal Methods*, pages 268–283, 2006.
25. C. Le Goues, K. Leino, and M. Moskal. The boogie verification debugger (tool paper). *Software Engineering and Formal Methods*, pages 407–414, 2011.
26. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
27. K.R.M Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.
28. K.R.M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *FTfJP '07*, 2007.
29. K.R.M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *SAC '09*, pages 615–622, 2009.
30. R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.
31. R. Monahan. Teaching using SPEC# in Europe: An experience report from university teaching and various verification tutorials. Microsoft Research Faculty Summit 2011, July 2011. Oral communication.
32. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP '98*, pages 158–185, 1998.
33. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.
34. E. Poll. Teaching program specification and verification using JML and ESC/Java2. *TFM'09*, pages 92–104, 2009.
35. S. Ranise and C. Tinelli. The smt-lib standard: Version 1.2. *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.
36. D.N. Xu. Extended static checking for Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 48–59, 2006.

DOI: 10.1002/spe