Pest: From the Lab to the Classroom

Guido de Caso, Diego Garbervetsky, Daniel Gorín
Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
{gdecaso, diegog, dgorin}@dc.uba.ar

ABSTRACT

Automated software verification is an active field of research which has made enormous progress both in theoretical and practical aspects. In recent years, an important effort has been put into applying these techniques on top of mainstream programming languages. These languages typically provide powerful features such as reflection, aliasing and polymorphism which are handy for practitioners but, in contrast, make verification a real challenge. The PEST programming language, on the other hand, was conceived with verifiability as one of its main design drivers. Although its main purpose is to serve as a test bed for new language features, its bare-bones syntax and strong support for annotations suggested early on in its development that it could also serve as a teaching tool for first-year undergraduate students. Developing an ECLIPSE plug-in for PEST proved to be both cost-effective and a key part to its adoption in the classroom. In this paper, we report on this experience.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*verification*

General Terms

Verification, human Factors

Keywords

Language design, verifiability, teaching, ECLIPSE plug-in

1. INTRODUCTION

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 was shifted to *partial specifications* which somehow overcomes many of the objections raised in [9]. Verification of partial specifications must be regarded as an error-detection procedure and, as such, akin to traditional forms of testing.

We will center on the particular form of verification where source code is annotated with special assertions. These normally take the form of method pre and postconditions, loop and class invariants, etc. Special tools then read these annotated sources, generate verification conditions (VCs) from them and feed these into automated provers [6]. SPEC# [1] and ESC/JAVA [5] are two of the best-known examples. The former is based on a dialect of C# while the latter takes JAVA code with JML [7] annotations.

In fact, there are other ongoing research efforts in automated verification for almost every major programming language in use [2, 12, 10]. The rationale is to lower the adoption barrier 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* (e.g., `assert`). 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 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.

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

TOPI '11, May 28, 2011, Waikiki, Honolulu, HI, USA

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: Simple Pest procedure definition.

ways from C#'s semantics [1] is indicating, in our opinion, a new driver for the programming languages to come.

Motivated by this concerns, Pest [4] originates as an experiment in programming language design. It is a very simple while-style, multi-procedural, monomorphic language; with integers, booleans and arrays (support for user-defined types is planned but has 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 enables experimenting 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 tool for first-year undergraduate students in computer science. Students at this point typically struggle to understand key concepts such as *correctness proof* and *loop invariant*. Because of its light syntax, Pest can be used almost as pseudo-code and students can develop valuable intuitions in a hands-on way using an heuristic approach.

For this to work in practice, though, one needs to present the tool in a format that is intuitive, familiar and attractive for students and where errors and other valuable feedback is presented in an nonthreatening way. Our experience shows that this can be achieved in an extremely cost-effective way using plug-ins for state-of-the-art Integrated Development Environments (IDEs) and we report on this.

The paper is structured as follows: §2 gives an overview of the Pest language; §3 describes BudaPest[1], the Pest plug-in for Eclipse; in §4 we briefly discuss our classroom experience with this tool; §5 presents some related work and; finally, we conclude in §6 with some final remarks and items for future improvement.

2. PEST LANGUAGE OVERVIEW

Pest is a multi-procedural, non-recursive, while-style, imperative programming language whose syntax and semantics natively incorporate various kinds of annotations. Variables are typed and monomorphic (only booleans, integers and arrays of integers are supported for the moment) and their types are inferred from use (for a formal description of the language, refer to [4]).

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

[1]Available from: `http://lafhis.dc.uba.ar/budapest`

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 loop.

variables and can appear as left-hand-sides of the assignment operator $\leftarrow$. In this example, all variables are inferred to be of integer type, since the $\geq$ operator takes integers as arguments.

Apart from classical boolean operators, boolean expressions may include *bounded* first-order quantification, as illustrated in Figure 2 where a procedure containing a while-loop iterating over an array is shown.. Loop invariants are introduced using the **:?!** keyword and exactly one loop variant must be provided, using the **:#** keyword.

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).

Operationally, annotations in Pest correspond to runtime assertions. Roughly speaking, pre and postconditions are evaluated on procedure entry and exit, respectively; invariants are checked prior to the evaluation of the loop guard; etc. Program execution fails when an assertion does not hold. Of course, it is expensive to check annotations at run-time. Alternatively, a Pest compiler can remove an assertion whenever it is statically verified. In a way, this is reminiscent of a *type erasure*.

A Pest compiler can also *infer* pre and postconditions of a procedure. If only the precondition is given a postcondition can be obtained by way of a symbolic computation; on the other hand, starting from a postcondition, a precondition can be obtained using a variation of the notion of *weakest precondition*. In simple cases, like the procedure in Figure 1, one can remove the annotation altogether and rely solely on inference. Details are given in [4].

2.1 Language Extensions

The simplicity of the Pest language makes it a natural testbed for experimenting with new ideas, be it verification strategies, program constructs or language features. As an example, we consider the *invariant carrying* **for** iteration construct.

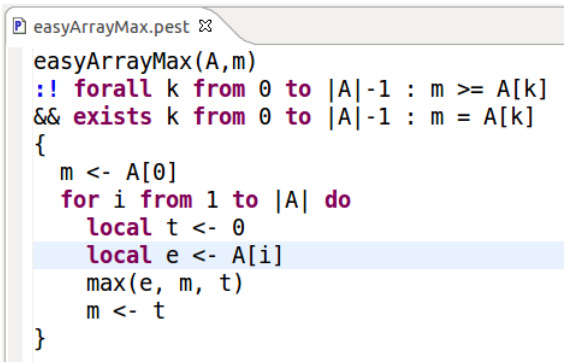

Figure 3: Using for to remove annotations.

This iteration construct is roughly equivalent to the PASCAL-style *for* loop; but, in addition, the invariant is heuristically derived from the postcondition as follows:

1. The iteration variable (i in Figure 3) will stay between bounds, provided it is not touched in the loop body.
2. Variables that are not touched in the body remain constant during the loop (remember PEST programs are alias free).
3. For the remaining part of the invariant, a candidate can be guessed by syntactically replacing the iteration upper bound by the iteration variable in the loop postcondition (which may be inferred). Of course, the correctness of this part of the invariant will have to be statically verified.

Using this construct, the procedure in Figure 2 can be rewritten as shown in Figure 3. Notice that this version requires no loop annotations and no precondition either: (they are all correctly inferred by the compiler using the procedure's postcondition).

The reader is referred to [4] for more extensions of the PEST language.

3. THE BUDAPEST ECLIPSE PLUG-IN

ECLIPSE[2] is an industrial strength, widely adopted, multi-language IDE. It can be used to develop applications in mainstream programming languages such as JAVA, C++ or PYTHON. Support for new languages and tools can be added via a sophisticated plug-in system. It is written almost entirely in Java, just like our PEST verifying-compiler, and available for every major platform. Writing an ECLIPSE plug-in appeared as a natural choice when discussing how to take PEST to the classroom.

BUDAPEST is the result of this effort. Once installed, it provides a new type of ECLIPSE project that allows the development of PEST programs in a way that is natural for ECLIPSE users. For this, it defines a new PEST *perspective* which contains a customized text editor that is integrated to a *Problems view*. PEST syntax is properly highlighted and the compiler is invoked every time the user saves. Errors and warnings are reported back in the problems view, together with localized error markers in the code.

Figure 4 depicts the PEST perspective. Younger students can often be frustrated by cryptic or misleading error messages; therefore, especial care was put in making messages

[2]`http://eclipse.org`

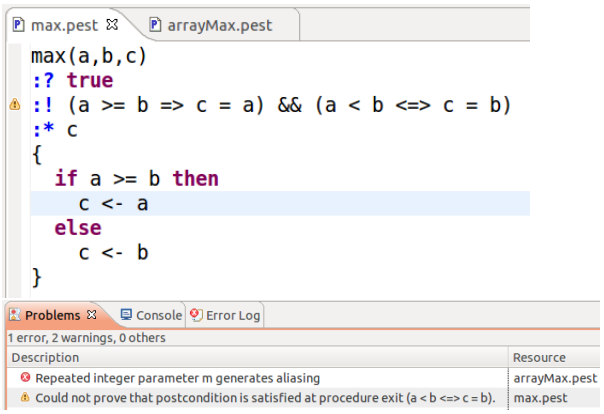

Figure 4: Pest editor with problem markers; problems view depicting VCs that failed.

as clear, accurate and informative as possible. For example, if an annotation that failed to be verified contains several clauses, we add a problem marker to the ones that failed.

The installation process for BUDAPEST needed to be very simple if one expects students to be able to exercise at home. Fortunately, 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. This means that we simply had to setup an update site for BUDAPEST and leave the rest to ECLIPSE's plug-in management tools.

Regrettably, there is a part of the installation process that could not be properly automated. The PEST verifying-compiler discharges the verification conditions (VCs) to a daisy chain of different SMT solvers, which are used as black-box components. If one prover fails to provide a definite answer for a given VC, the next one is run. These solvers are typically distributed in binary form and the supported platforms vary. Moreover, some of their licenses oblige the user to read and accept an agreement before installing. We could not find a simple way to handle this problem and, therefore, require the user to manually install the SMT solvers that will be used. BUDAPEST defines a preference pane where the available provers can be indicated.

It is worth comparing the relative complexity of the PEST compiler and the BUDAPEST plug-in. Both were 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). The PEST compiler was developed in 5 months and consists of 13 KLOC. BUDAPEST, on the other hand, needed only 2 weeks and 1.5 KLOC; that is, it was simpler by a factor of around 10, both in code size and development time.

4. CLASSROOM EXPERIENCE

We tried the PEST language in the context of a first-year undergraduate course on algorithms. This course has a strong accent on formal specification and manual verification. Students have to write programs that perform basic operations on arrays (e.g., searching and non-recursive sort-

ing) and establish their correctness. They typically find the Invariance Theorem for loops particularly challenging.

To prevent excessive disruptiveness, so far we introduced PEST at the end of the course, as a way to contribute to the assimilation of the concepts learned. We plan to introduce it earlier in the course in upcoming semesters.

We observed that, at first glance, students find themselves comfortable with the PEST language and its working environment. PEST resembles the pseudo-code used in classes and, by following ECLIPSE's guidelines, BUDAPEST's interface is very intuitive.

We present them algorithms they already learned during the course, but rewritten in PEST and with subtle specification and coding bugs seeded in them (e.g., mishandled border cases, incorrect statement ordering, weak or missing invariants, etc.). They have to interactively find and correct these errors, with the assistance of the detailed error reporting mechanism the tool provides.

We perceive that, after seeing the tool in action, students get excited. On the one hand, they realize that the concepts learned during the course allow them to understand what the compiler is doing behind the scenes. On the other, the tool deals with the formal proof, which they find to be the most tedious and mechanic part of the verification process.

Finally, we would like to point out that this positive experience in the classroom would not have been possible without the plug-in. During the class, it is crucial to easily and effectively perform verification using the push-button capabilities the tool offers. When combined with the syntax highlighting and integrated error reporting, the BUDAPEST plug-in resulted in a captivating classroom experience.

5. RELATED WORK

For reasons of space, we shall only discuss prior reported experiences on using program-verification tools in the context of teaching.

The SPEC# programing language was extended in [8] with new constructs and techniques aimed at improving the automatic handling of set comprehensions (e.g., max, sum, count, etc.). This allows them to naturally express and verify several textbook examples. Their long term goal is to make SPEC# a suitable language for teaching algorithms both in introductory and advanced courses. SPEC# has a very attractive IDE although limited to only a few platforms.

In [11] the authors report their experience teaching program verification using JML and ESC/JAVA2 [3]. They too present the tools at the end of their course and report the enthusiasm shown by their students. They mention also some limitations that could be a little discouraging for them: 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 seems tricky to install (probably due to unstable dependencies).

In comparison, PEST is a very simple language but this makes it easy to learn. Since it was designed with verifiability in mind, all its constructions are fully supported. Being an introductory course, the algorithms we used were probably not too hard for our cocktail of state-of-the-art SMT solvers. It is not clear if we can have a similar successful experience in a more advanced course. Since we were targeting students, we tried to make our tool very easy to install.

6. FINAL THOUGHTS

We reported on our experience transplanting to the classroom an experimental programming language designed to serve as testbed for new verification-related language features. A key aspect in achieving this was providing an attractive and intuitive front-end to the verifying-compiler.

While developing a complex graphical user interface can often consume a large portion of development time, we have observed that an extensible IDE like ECLIPSE can make the additional effort almost negligible. This should encourage researchers to take the time to write appropriate plug-ins for their tools; it is a very cost-effective way of making their 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.

7. REFERENCES

[1] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*. Springer, 2005.

[2] S. Chatterjee, S.K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. *TACAS'07*, pages 19–33, 2007.

[3] 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.

[4] G. de Caso, D. Gorín, and D. Garbervetsky. Reducing the number of annotations in a verification-oriented imperative language. In *APV'09*, 2009.

[5] 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.

[6] D.I. Good, R.L. London, and W.W Bledsoe. An interactive program verification system. In *ICRE*, pages 482–492, 1975.

[7] 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.

[8] K.R.M. Leino and R. Monahan. Reasoning about comprehensions with first-order smt solvers. In *SAC '09*, pages 615–622, 2009.

[9] 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.

[10] N. Norwitz. PyChecker. *SourceForge project http://pychecker. sourceforge. net.*

[11] E. Poll. Teaching program specification and verification using JML and ESC/Java2. *TFM'09*, pages 92–104, 2009.

[12] D.N. Xu. Extended static checking for Haskell. In *SIGPLAN workshop on Haskell*, pages 48–59, 2006.