CS-Note[*] 55

Andrew Ireland
July 10, 2003

Visits to NASA Ames and HP's Systems Research Center

1 Introduction

On June 28^{th} and 29^{th} I attended the ASE-2003 Programme Committee meeting that was hosted by John Penix at the NASA Ames Research Center in Moffett Field, Mountain View. On June 27^{th} I made a side visit to the Systems Research Center[1] (SRC) at HP's Palo Alto Labs while on the 30^{th} I attended a workshop at NASA Ames on *Frontiers of Automated Software Engineering*. A summary of the trip, and in particular the 2 side visits, is reported here.

2 HP's Systems Research Center

I gave my standard talk on the use of failure analysis within proof planning and tactic based reasoning, updated with a progress report on the NuSPADE project. The feedback I received was very positive, what follows is a summary of the associated Q/A session:

Jim Saxe: Why inter-leave fertilize with the ripple method?

AI: The presence of second-order meta variables within the goal (a speculative generalized conjecture) means that the ripple method will always be applicable so we rely upon fertilize to terminate the proof.

Cormac Flanagan: Why are run-time check proofs of such interest? In the case of SHOLIS if an exception was raised could one not just fall back on the traditional manual approach?

AI: I believe such guarantees are of real interest to people from the avionics, transportation and defence related industries. In the case of SHOLIS I suspect the expertise for the traditional manual approach may no longer exist, but don't know for certain.

Jim Saxe: In your tool-set slide you have 2 routes from the Examiner to the SPADE (Proof Checker) - one via the Simplifier while the other is direct - could you explain?

AI: Essentially this diagram shows 2 alternative routes. We are interested in the unsimplified version of VCs for which the Simplifier fails on. We find that the Simplifier typically eliminates many of the structural features that our proof methods exploit – so we work on the unsimplified VCs.

Greg Nelson: Just an observation on the idea of generating test cases from failed proofs (mentioned in my future work slide) – we could have developed this for ESC/Java – it was one of the ideas we had but never found the time to consider it.

[*] Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

[1] formerly known as Compaq's Systems Research Center.

My visit also included discussions with Greg Nelson. Of particular interest was the theorem prover (Simplifier) associated with ESC/Java. The Simplifier is refutation-based, but very little has been written about it. A journal paper (some 120 pages) is nearing completion, Greg has promised to send a copy of the submitted version. While the development of ESC/Java is complete[2], Greg Nelson and Rosemary Monahan[3] are currently investigating the possibility of extending ESC/Java in order to support reasoning about dynamic storage. I pointed them at the work by John Reynolds and Peter O'Hearn who have been working jointly on this topic for a number of years. Greg also gave a demo of JUNO-2[4], a drawing tool that is built upon a programming and constraint definition language. It has a neat double-view editing capability, *i.e.* one can either edit the graphics directly or via the corresponding code. I also spoke with Cormac Flanagan, one of the developers of HOUDINI[5], a system for generating proof annotations (invariants) for ESC/Java. The system is based upon predicate abstraction (and invariant "guessing") and refutation based proof via the Simplifier. HOUDINI has been very successful within the context of the light weight safety policy that ESC/Java has been designed around. Like ESC/Java, HOUDINI is no longer under development.

3 NASA Workshop: *Frontiers of Automated Software Engineering*

My talk took the form of a proposal entitled: *The Use of Patterns to Guide Code Certification*. However, I stressed that it was more an idea than a fully worked-out proposal. The idea was motivated by Colin O'Halloran's (QinetiQ) previous ASE presentations on the compliance tool-set. The basic idea is to investigate the the correspondence between patterns of program code and patterns of proof within the context of commercial code generation tools such as Simulink. Ultimately, looking to exploit the correspondence between patterns within Simulink models and the corresponding verification proofs may prove more useful. Not surprisingly, the approach would be to use proof plans as the means of representing proof patterns. The novelty would be to integrate program (model) patterns within the proof planning framework. I used our AutoGap system for generating properties from SPARK source code within the presentation in order to illustrate the rough idea behind program patterns. The talk went down well, the Q/A part is summarized below:

Michael Lowry: Have you found any patterns in the Simulink generated Ada code?

AI: I have not got that far. Basically this idea is very rough and is simply based upon the presentations and related papers by Colin O'Halloran's Software Assurance Group.

Michael Lowry: ... you could do a reverse engineering of Simulink code generation and get the patterns that way?

AI: Yes, as I hinted in my conclusion I think the way to go would be to look for patterns in the models then connect these patterns with patterns in the code and associated proofs.

Julian Richardson: How do you intend to express the program patterns and how do they relate to the patterns in transformation schema work?

AI: I should look at the schema work. Currently we have AutoGap that has been developed as part of a student project. It simply looks for array partitions (and related information) as illustrated in the talk. These patterns are first-order and are represented in Prolog. I guess if we require second-order patterns we will require lambda-Prolog (and lambda-Clam). Unless you have a better suggestions?

Julian Richardson: We should talk off-line.

John Rushby: I noted a couple of missing technologies from your introductory slides. When you passed over ESC/Java you didn't mention the related work by Flanagan on invariant discovery ...

AI: Yes, HOUDINI – I should have mentioned this ...

John Rushby: ... they make a useful trade of intelligence for computation – predicate abstraction and guessing – the outcome looks extremely promising. Then the other technology is the *translation validation* work of Pnuel. They verify each compilation rather than the compiler – that seems very much what you are proposing here?

[2] The Java Model Language (JML) project are looking after it, see: `http://www.cs.iastate.edu/~leavens/JML.html`
[3] Rosemary is a lecturer at National University of Ireland, currently visiting SRC for a few months.
[4] `http://research.compaq.com/SRC/juno-2`
[5] `http://research.compaq.com/SRC/esc/relatedTools.html`

AI: Presumably they have access to the internal details of the compiler (my proposal assumed no knowledge of the internals of the code generator, *i.e.* I assume a commercial code generator)?

John Rushby: I am not sure how much they have because one of their experiments was on the SGI optimizing C-compiler and I don't know if that is open (source) or not.

3.1 Other talks

3.1.1 Challenge and opportunity in mechanized formal methods

John Rushby gave a very stimulating/motivating talk. He cited 3 key reasons why formal methods have failed to deliver. Firstly, requirements are too informal and while code is formal — it is too big and comes too late. Secondly, formal verification is too narrow – doesn't fit the traditional design flows. Thirdly, lack of automation. He then went on to explain why he feels that there is now a real opportunity to succeed. He sees it as the result of the convergence of 3 trends:

- Industrial adoption of model-based approaches, *e.g.* Simulink/Stateflow, SCADE and Esterel, UML.
- New kinds of formal activities, *e.g.* fault tree analysis, test case generation, extended static checking, runtime verification, controller synthesis, ...
- More powerful/automatic deductive techniques, *e.g.* SAT solvers, bounded model checking, k-induction, abstraction, ...

It was nice to see a reference to the SPARK Examiner when he expanded upon extended static checking, *i.e.* exception freedom proofs provide "useful feedback to engineers". With regards to deductive techniques, the emphasis was on combining decision procedures. In order to preserve decidability restrictions are typically required, *e.g.* restrict to quantifier free formulae, and deal with quantification outside the decision procedure. This use of decision procedures and light weight theorem proving is what Shankar calls *little engines of proof.* John summarized his reasons for optimism as follows:

> *Model-based design methods are a (once-in-a-lifetime?) opportunity to get at artifacts early enough in the life-cycle to apply useful analysis within the design loop.*

When I questioned him about the problem of ensuring the correctness of software produced using model-based code generator technology he pointed to the success of SCADE[6] and the potential for generating test cases automatically from the models. Slides from the talk can be found at:

`http://www.csl.sri.com/users/rushby/slides/fase03.pdf`

3.1.2 A category-theoretic approach to composing inconsistent views

Steve Easterbrook talked about his work on reasoning within the context of inconsistent requirements. The basic idea is that one needs a mechanism for classifying requirements as to whether or not they can form part of any formal analysis. This mechanism builds upon a family of multi-valued logics. The overall framework for composing inconsistent requirements is achieved via category theory.

3.1.3 Abstract interpretation - ASE tutorial preview

Arnaud Venet (Kestral Ames Division) provided a preview of his ASE-03 tutorial on Abstract Interpretation. A nice presentation that gave some feel for the strengths and weaknesses of the technology. On the weakness side, the key points I picked up were the approximate nature of the analysis (false positives) and the fact that identifying local bugs requires a complete global analysis of the software under inspection. Arnaud was previously employed by PolySpace.

3.1.4 Explorations, queries and values in model checking

Marsha Chechik talked about work on symbolic multi-valued model-checking. This was developed in conjunction with Steve Easterbrook's work described above.

[6] According to the product advert: The SCADE Code Generator translates block diagrams (system designs) into very efficient, reliable and safe C code (DO178B level A compliance)!

3.1.5 Why testing is better than model checking

John Penix presented/led a kind of debate about testing versus model checking, looking at cost/benefit analysis type issues and where model checking is best utilized. Nothing major to report here. However, I did ask a question about model checking with respect to real-time aspects of systems (in particular scheduling systems – as this relates to Ben Gorry's PhD topic). Basically the NASA response is that they don't do model checking of real-time properties – this is an area covered by testing it appears. John Rushby pointed me at the work of Reinhard Wilhelm (Saarland University) who uses Abstract Interpretation to verify real-time aspects of critical systems:

`http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/index.shtml`

3.1.6 Practical static analysis for NASA applications

Guillaume Brat (Kestral Ames Division) gave an experience report on the use of Abstract Interpretation to analyze existing NASA software. The basic idea underlying the work of Guillaume (and Arnaud) is that in order to make Abstract Interpretation practical (less false positives) it needs to be tailored to the application area. To this end they have developed a kind of open architecture that allows one to modify key parameters. A partitioning component was highlighted as one of the key parameters. More generally, efficiency still seems to be a major problem for the technology. On one of the experiments with NASA software the system gave up after 12 days — the PolySpace license timed-out!