Languages, Ambiguity, and Verification

The SPARK Team

Praxis High Integrity Systems
20 Manvers Street,
Bath BA1 1PX
UK.
sparkinfo@praxis-his.com

Abstract. This position paper is based on presentations given at the Grand Challenge workshops held at Gresham College in March 2004 and in Newcastle in July 2005. It reports some of our experience from building the SPARK language and its verification tools. We argue that the provision of an unambiguous semantics for a programming language is crucial if the verification framework is to be sound, deep and efficient. Secondly, we offer some reflections on the (mostly non-technical) barriers that we encounter in trying to deploy SPARK within organizations. Finally, we try to set some goals for future work.

1 Design goals for a program verification system

A programming language and verification system that aim to meet this Grand Challenge might have the following design goals:

- Soundness – the system must not give a false-negative result.
- Completeness – the system should issue as few false-positive results (aka "false alarms") as possible.
- Depth – the verification system should be able to verify useful and non-trivial properties of our programs.
- Efficiency – the system must be fast enough to enable constructive and interactive use. If it takes all night to verify anything useful, then no-one will use it! Ideally, the system should be so fast as to wean programmers away from the lure of compilation and test.
- Composition – "separate verification" (somewhat akin to "separate compilation") must be possible. Addition of new program units must not invalidate the verification of existing units.
- Expressive Power – the language must be large enough for use on industrial-scale projects. (It's easy to meet the first five goals for a toy language that no-one else uses…)

2 Languages and Ambiguity

In program verification, we are asking questions of the form "What does this program mean?" or "Does this program have property X?" It seems a reasonable expectation that questions of the first form should only have *one* answer, while the second form should be ideally answered "Yes" or "No".

A central goal then is the provision of an *unambiguous* semantics for our programming language, since any ambiguity can lead to incomplete "Don't know" answers or (worse) unsoundness. Consider the case of a simple evaluation-order dependence—a language feature that is *implementation-dependent* in C, C++ or Ada. If a verification tool encounters a compound expression, should it assume left-to-right, right-to-left, or both evaluation orders? The first two options possibly lead to unsoundness if a compiler disagrees with the choice made by the verification tool. The latter choice leads to an unacceptable explosion in analysis complexity.

Another approach is to define a language (or a subset of some suitable parent language) so that either evaluation order is allowed, but the choice can never make any difference to the program's meaning. This is a rather more subtle trick, but has significant benefit: the system is sound, efficient, and the language can be a true subset of its parent so that standard industrial compilers can be used.

This is the approach taken by SPARK[1], which aims for an unambiguous semantics through:

- Careful sub-setting of Ada95 to remove troublesome language features such as unrestricted use of tasking and generics, and
- Provision of design-by-contract "annotations" in the language to strengthen the specification of units with just the right information required for analysis, and
- Static analyses, such as information flow analysis and aliasing analysis, which are mandatory.

The latter are important: the SPARK Verification-Condition Generator (VCG) is built on the assumptions that programs are free from aliasing of names, function side-effects, and the use of uninitialized variables. These analyses ensure the required properties hold before the VCG can be enabled, and the language is designed so that these analyses are sound, complete and efficient.

3 SPARK in the real world

We now have a small, but non-trivial set of customers using the SPARK system to prove real properties of real programs. These programs range from about 10000 lines of code to several million lines. Most customers attempt proof of the absence of "run-time errors" (e.g. buffer overflow, division by zero etc.) first, since the side-conditions for these are "built in" to the VCG (no additional annotations are required) so it provides an approachable first rung on the ladder. Our theorem prover is typically about 97% complete for the resulting VCs. Machines resources are now available that proof of such properties can be attempted on whole programs with an hour or so, and on a particular subsystem or package within minutes.

We have observed one other effect through the use of the SPARK proof system: it mandates a clarity and precision of expression that offer benefits far beyond the simple verification of program fragments. In trying to complete a proof for a routine, engineers find (with alarming frequency) that their requirements or specifications are incomplete, ambiguous, or contradictory. Finding these mistakes sooner rather than later has an obvious beneficial impact on a project's risk, time-scale and overall defect rate.

3.1 Barriers to SPARK adoption

In SPARK's natural application domain (hard real-time, embedded and critical systems), the technical argument in favour of strong static verification is easily won. The commercial advantage of a low-defect rate is also well established[4]. In spite of these, we often struggle to convince organizations to adopt the technology. The reasons are rarely technical, but seem to fall into the following general categories:

Process-ism
The rise of software process capability models such as CMM has led to a rather unfortunate relegation of technical considerations. We often hear arguments along the lines of "Our process is CMM level X, therefore it doesn't matter what programming language we use..." We also see much "process improvement" effort focused on detection of defects (i.e. "speed up code/test/debug") rather than their prevention earlier in the life-cycle.

Some software process standards, such as DO-178B, give little or no credit for formal approaches, so projects prefer to follow the letter of the standard regardless of the utility of alternative approaches. Ironically, the UK's Def-Stan 00-55 does call for formally-based approaches, yet this seems to have made little difference.

Resistance to change
The SPARK approach implies changing many aspects of software development practice and process—design style, code review, subsequent testing and so are all modified as a result of using static verification. These changes are often perceived as a risk by project managers, who see changing nothing as less risk-prone than adopting something seen as disruptive or new. Large organizations (e.g. defence prime contractors) exhibit a near-glacial inertia that prevents change—this mindset does not prevent the purchase of additional tools as long as they don't require any true change of process or life-style; unfortunately, it is usually the process that is most in need of change.

Snake-oil, wizards, gurus…
The software development industry is rife with approaches, -isms, tools, magics, and wizards—most of which fail to deliver all that they promise. Differentiating SPARK from this crowd remains a constant battle.

Secondly, some engineers are indeed very capable at working with imperfect technologies and processes, and produce excellent results. These "gurus" become the "heroes" of a project, and management come to depend on their skills and advice for this

and future efforts. SPARK undermines these witch-doctors, and so we (ironically) find that the most talented engineers in an organization often find SPARK or similar approaches to be a threat.

Procurement practice

In some industries, there seems to be very little pressure for developers to improve or change their ways. Procurers let contracts that allow for the delivery of a defective product, which is, unsurprisingly enough, what they often get. Procurers rarely ask for any sort of meaningful warranty.

SPARK is mostly associated with highly critical safety-related systems, so we often hear the mantra "our system's not safety-critical, so we don't need SPARK", conveniently ignoring the issue of whether the system actually has to work! This is reinforced by customers' low expectations of success. Since failure is both expected and tolerated, expending extra effort in the hope of success may seem rather pointless, especially when similar attempts using the latest fashion have failed to deliver.

The "A word" and recruitment

The "A word" is, of course, "Ada" on which SPARK is based. Ada still incites a knee-jerk reaction with some potential customers. Customers shy away from the idea of having to adopt a new or different language—they perceive the cost and effort of changing, but ignore the potential pay-off. Recruitment in the industry also seems very focused on languages and tools rather than skills and domain knowledge. This is a real challenge to the Grand Challenge (!)—the world is wedded to fashionable languages, yet these very languages are incompatible with the aims of the Grand Challenge.

3.2 Lessons

Technology transfer remains difficult, but several lessons have been learnt that might be useful for the future of the Grand Challenge:

- Technical strength is required as a basis, but is not enough to get beyond the early adopters. Packaging, training, support, the "user experience" and so on are all crucial for the technology to reach a wider audience. We need to "make the maths disappear."
- Success is not the same as dominance within a market. SPARK exists as a successful product within a small niche, but can hardly be described as dominant. Should the Grand Challenge (given the time-scales available) be aiming for success within a particular niche (e.g. high reliability systems), or a wider goal of program verification becoming dominant in more general software engineering?

4 Future work

This section offers an admittedly incomplete set of ideas for future work, based on our experience and feedback from our customers. Some of these are within reach given time and money. Others seem more challenging. In no particular order:

- Floating point proof. The system does not yet support the proof of floating point numerical algorithms, largely owing to the ambiguity in the popular IEEE specification for such things, and the lack of a notation for expressing relative error in pre- and post-conditions. We need the VCG algorithms, an "equality within relative error" operator, proof rules, and decision procedures for all of this. Recent work in abstract interpretation suggests this ought to be within reach[2].
- Make the language bigger. Some people consider SPARK to be a hopelessly small subset language for general purpose use. It is certainly a *very* small subset of Ada (a good thing!), but there's lots of room for careful enlargement. Tasking was recently re-introduced in a form that's suitable for hard real-time systems, and a subset of generics seems within reach. Beyond that the list is endless: pointers (plus garbage collection), more OO support (particularly polymorphism), call-backs, interfaces, etc. At the far end of this spectrum, we would simply start again and design a new language from scratch or design a subset of a different suitable language.
- Distributed proof. The theorem prover remains a bottleneck in the system, but can be distributed almost arbitrarily. A truly efficient and interactive verification environment could rely on a network of many hundred CPUs working as a "proof engine."
- User interface. SPARK was originally designed as a largely "batch oriented" system. We sorely need an interactive user-interface, both for the basic static analyses and for the proof system.
- Proof management and replay. The current system is rather fragile in the face of change – a small code change can easily "break" a proof script. Automated support for impact analysis, proof management and proof re-discovery would be useful.
- Counter-example finding. Telling a user not to bother looking for a proof of a non-theorem and offering a counter-example would be a direct benefit and could usefully be applied to the small percentage of VCs that the theorem prover does not discharge. We concur the Steel's position paper on this topic[3].
- Concurrency. Support for proving properties of concurrent programs is very limited in SPARK at present.

References

1. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, April 2003. ISBN 0-321-13616-0.
2. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In *ESOP 2004 — European Symposium on Programming*, D. Schmidt (editor), Mar. 27 — Apr. 4, 2004, Barcelona, Lecture Notes in Computer Science 2986, pp. 3-17.

3. Steel, G.: The importance of Non-theorems and Counterexamples in Program Verification. University of Edinburgh. Position paper for this workshop.
4. Humphrey, W.: Winning with Software: An Executive Strategy. Addison Wesley, December 2001. ISBN 0-201-77639-1