Some Issues in the Software Engineering of Verification Tools

Perdita Stevens [*]

Division of Informatics, University of Edinburgh
JCMB, King's Buildings
Mayfield Road
Edinburgh EH9 3JZ

Abstract. The Edinburgh Concurrency Workbench has been the author's responsibility for the past four years, having been under development for eight years before that. Over its lifetime, we have learnt many lessons and encountered many questions about verification tool development, both from bitter experience and from discussion with other tool developers and users. This note raises some of them for wider discussion.

1 Introduction

It is common to hear it said that an important factor in the practical uptake of theoretical work in computer science is the availability of tools that incorporate the theory; and the spread of finite-automata-based verification tools through the US hardware verification industry is indeed one of the more widely visible signs of recent progress in theoretical computer science. Although there are now some cases of verification tools being taken over, or developed in house, by commercial organisations, it is more usual that they are developed in universities, at least partly by people whose jobs also involve research and teaching.

The nature and range of software engineering problems encountered by developers naturally vary with the kind of product being developed and with the nature of the developing organisation. This note draw on a longer paper "A Verification Tool Developer's Vade Mecum" (available from www.dcs.ed.ac.uk/home/pxs) which attempted to bring out the special features of the development of verification tool development in universities. In this note we raise some of the questions, commenting on a few as space permits: the aim is to promote constructive sharing of experience and views.

2 "Business case" level issues

To begin with one of the earliest and thorniest of questions:

Who does the development? And more difficult, **who does the maintenance and support?**

[*] Perdita.Stevens@dcs.ed.ac.uk, supported by EPSRC GR/K68547. Tel: +44 131 650 5195, Fax: +44 131 667 7209

– Professional researchers who are expert in the underlying theory?
– Students, with limited experience in both theory and software engineering? They may receive direction, but this normally comes from someone more expert in theory than engineering.
– Professional software developers, employed for the purpose, under direction from researchers, who supply the detailed theoretical understanding? Will they have to be so minutely directed, in order to ensure that the theory is correctly implemented, that the skilled work of system design is, in effect, again done by the researchers, with the programmers doing routine programming work only?
– Professional software developers who are in the process of becoming professional researchers? The possibility is probably unusual, but I was brought into the Edinburgh Concurrency Workbench project after some years as an industrial software engineer, but with only undergraduate level knowledge of the underlying theory, with the intention that I should acquire whatever knowledge was necessary. My ignorance of the underlying theory posed formidable problems in the beginning – but perhaps it is easier in a university environment to acquire theoretical understanding than engineering understanding?

How do you find the time to spend on tool work? In varying degrees, all of the options for tool development require a serious commitment of time and energy. For someone pursuing an academic career, there is a tension between producing papers and producing tools. Although increasingly universities seem to recognise the importance of producing tools (to gain the advantages cited below), it is very difficult – especially, but not only, for someone who is not engaged in tool development and maintenance – to appreciate the amount of time that is required.

To some extent, it may be possible to combine the goals of producing papers and producing tools: there are fora, including TACAS, for presenting papers about new tools or major new features in tools. However, much of the effort required to maintain a tool, especially one which has many users, is the routine (though skilled) work of updating interfaces to changing external systems, writing documentation, answering email, developing tests, etc. This work is not research.

Do you want to develop a tool at all? We have begun to mention the disadvantages: it's time-consuming, and difficult in ways which often have little to do with research, and it may be difficult to find the resource to do it well. Other options to consider may include:

– Developing a component of another tool set, rather than a whole new tool. The practicality of this will depend on the intended functionality of the tool and the qualities of the tool set.
– Getting someone else to develop the tool. If your intended user group is industrial verifiers, can you build the tool as a collaborative venture with an industrial partner?

Do you want your tool to have users? Or is it better to build a purely experimental system, with lower support needs?

Who are the intended users of your tool? Students and their teachers? Researchers? Industrial verifiers? With what needs and experience?

What do you want to achieve? For example:

– Technology transfer: you may want to improve visibility of some well-established piece of theory (among industrial practitioners, students or both).
– Theory experimentation: you may want to deepen your understanding of some theory by experimenting with different implementations.
– Image manipulation: you may want yourself or your organisation to be seen as doing "practical" work.

3 Architectural issues

What high level structure is appropriate? (This is influenced, for example, by whether you want other people to be able to extend the tool, and if so, in what ways.) Specifically,

Which decisions must be encapsulated so that they can easily be changed?

What programming language is best for the purpose? Considerations will include, for example, support for encapsulation, the type system, the availability of compilers on the relevant platforms.

What user interface is best? For example, do you want a graphical user interface or not, and if so, on what toolkit should it be based, considering maintenance and portability? Much depends on who the users are.

4 Issues concerning the development process and QA

A quality assurance process suitable for academic development of verification tools needs to be extremely streamlined. A meta-question is

What documentation of the process is useful? This depends on, for example, the group of people involved, their distribution and turnover.

Let us consider a couple of important areas.

Version control This is important for all systems, but particularly important for verification tools, where correctness is paramount. I find it helpful to version-control *everything* – code, build files, documentation, tests, "correct" answers to tests, etc. In order to make this feasible given resource constraints, the version control system has to be very easy to use, so that one can check something in and keep working on it without interrupting a chain of thought.

Testing It goes without saying that a verification tool needs to be thoroughly tested; but the effort required to do this is often underestimated. In mainstream software engineering, the usual estimate of how much of a software development project's budget is spent on testing is 30 - 50%; verification tools can be expected to be towards the upper end of this spectrum. It is extremely tempting to cut corners here, and so it is crucial that all time that is spent on testing is used as effectively as possible. Some automated support for regression testing is probably essential – the CWB's system testing software is written in Perl, a language which is well adapted to this kind of task. This simple program enables the CWB developer to run tests and spot newly introduced problems with minimal effort. It is simple-minded; for example semantically insignificant changes to CWB output – printing nominally unordered output items in a different order, for example – are reported as errors: but it has not yet seemed efficient to implement anything more sophisticated, bringing us on to the next question:

What is it efficient to automate? It seems worth remarking that there is a danger of losing time by automating things, too. For example, after making two minor errors in releasing versions of the CWB, I developed a script to automate the release process, when in fact I would have been better off with a checklist of things to do when releasing the CWB: this would have solved the original problem more robustly with less effort.

Other issues in the development process that may need to be considered include

What coding practices are required?

What kind of documentation is needed? Writing *and maintaining* documentation is one of the most time-consuming aspects of tool development, so this needs particularly careful consideration.

5 Dissemination issues

are relevant if you are developing a tool which is to have external users.

What kind of distribution policy is appropriate?

What kind of support will you offer?

In conclusion,

What does your own experience suggest as answers to any of these questions? And what other questions are crucial? What is the single most important piece of advice to give to tool developers?

10.1007/b107031130030