A Verification Tool Developer's Vade Mecum

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 about verification tool development, both from bitter experience and from discussion with other tool developers and users. This paper is the document the author would have liked to read four years ago, and the intention is that it may be useful to other developers in future. We assume familiarity with the basic lore of software engineering, and try to bring out the special issues that arise in the development of verification tools.

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.

Both because of the special nature of the tools, and because of the special circumstances under which the tools are normally developed and maintained, some issues arise which are different from the standard issues of software engineering any product. This paper explores what the issues are, using the author's experience as developer and maintainer of the Edinburgh Concurrency Workbench as illustrations where appropriate. We will assume familiarity with the basic tenets of software engineering, such as might be acquired from a good undergraduate degree, and try to bring out what is special about the situations we address.

For brevity, the paper addresses the potential tool developer as "you" and makes some bald statements with which some readers may disagree. It is difficult not to seem didactic in a paper of this sort, but we welcome debate.

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

1.1 Edinburgh Concurrency Workbench background

The Edinburgh Concurrency Workbench, a tool which supports the manipulation and analysis of concurrent systems, was conceived in 1986 as a focus for the interaction of theoretical, educational and practical concerns.

Its original developers regarded themselves more as concurrency theorists than as software developers, and so, although serious consideration was given to the design of the tool [2], some design decisions were inevitably made which with hindsight are suboptimal. Nevertheless the tool was successfully used both in teaching and in industry, and it became clear that its development should be continued. Several people maintained it over the succeeding years, and many features were added to it. In 1994, when the present author joined the project, the CWB consisted of about 20,000 lines of Standard ML code, implementing about 90 commands pertaining to systems specified in CCS, TCCS or SCCS. By this time it had become clear that modifications to the Edinburgh Concurrency Workbench were becoming harder to make reliably or fast, and the author, who was recruited from a non-academic job as systems and software engineer, undertook to reverse this change.

This task has proved (even) harder than we anticipated, and part of the motivation of this paper is to help the developers of other tools circumvent the process.

1.2 Special features of verification tools

The most obvious respect in which verification tools are different from most software system is in the importance that must be given to correctness. Software systems have bugs: but a verification tool which has semantic bugs – that is, which may give an incorrect answer to a verification question – loses more of its value per bug than most systems. Some people would even say that a verification tool which contains a semantic bug is worse than useless.

The other main class of systems for which this is true is safety-critical systems, and in that area there is a well-developed literature on how to ensure correctness. The main difference between verification tools and safety critical systems is that safety is a property of a whole setup including humans and manual procedures with software as just one part. It can be hard to say categorically whether a given software behaviour is or is not a bug. This is not normally a problem for verification tools, whose functional requirements are rather easy to define.

Verification tools often, but not always, have the property that they are expected to have a long life and undergo many modifications and alterations. The Edinburgh Concurrency Workbench, for example, is intended to be a test-bed for new theory, so it is important that it should be easy to add new components to it.

If there were no limit to the effort which could be spent on a verification tool – or even, if it were possible to spend on the development and maintenance of such systems the same amount of effort as is typically spent on industrial tools of similar size – it would be fairly straightforward to develop a process which would lead to correct (enough) tools. The major challenge for the verification tool developer is that this is normally not possible.

1.3 Special features of verification tools developed by academics

Naturally institutions and the arrangements they are able to make for the development and maintenance of verification tools differ widely. However, there are two major respects in which special difficulties seem commonly to arise.

The first is the slightly delicate question of who does the development – in particular, the design of the software – and how much software engineering experience the developers of the tool have. Ideally, the designer needs to have an in-depth understanding of both software engineering and the theory that underlies the tool.

We know of several models: in what may be descending order of frequency

- The tool may be developed by researchers who are expert in the underlying theory. They may or may not appreciate the importance of software design: but even if they do, as humans they are not likely to be equally expert in both fields.
- The tool may be developed by 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.
- The tool may be developed by programmers under direction from researchers, who supply the detailed theoretical understanding. This sounds like the ideal: but in cases we have heard about, it appears that they had to be so minutely directed, in order to ensure that the theory was correctly implemented, that the skilled work of system design was, in effect, again done by the researchers, with the programmers doing routine programming work only.
- 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. It would be interesting to know of cases where this model has been applied to the initial development of a system: the Edinburgh Concurrency Workbench case was complicated by the fact that the system was a legacy system, in need of reengineering. 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, which would suggest that this is a model worth considering if it is available.

The second major problem is lack of time. If the developer is 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 in section 2.1 below), it is very difficult for someone who is not engaged in tool development and maintenance to appreciate the amount of time that is required. (Indeed, it is difficult for people who *are* engaged in it to do so: research persistently finds that software developers underestimate the time it takes to complete a task.)

To some extent, it may be possible to combine the goals of producing papers and producing tools: there are fora for presenting papers about new tool 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.[1]

It is well understood by the legacy system community that systems suffer "calcification": even systems with good initial architecture tend to lose their good architectural properties with repeated modification over the years. (An example is described in section 3.5 below.) It becomes gradually harder to add new features to the tool. The deterioration can be slowed (though practically not stopped) by spending effort on the initial design of the tool and avoiding hasty modifications: but if the tool developers are always short of time, it is hard to put this into practice.

2 "Business case" level decisions

In this section we consider the questions which must be addressed before the organisation commits to developing a tool at all, bearing in mind the special features of the systems we consider.

2.1 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?

The correct answer will depend largely on what you want to achieve by developing the tool. Some possible goals are:

[1] It is not the case, as is occasionally asserted by theoreticians, that if one spends effort doing software engineering one must be able to write software engineering research papers about it.

- 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. Many people (including the author) find doing detailed design or writing code to be an excellent way to clarify thinking about algorithms or design of formalism.
- Image manipulation: you may want yourself or your organisation to be seen as doing "practical" work.

2.2 Do you want your tool to have users?

Your immediate reaction is likely to be "of course!" but there are many circumstances in which you might want to develop a tool as an internal prototype, to demonstrate the possibility of such a tool existing, or to try out theory in detail, or to investigate average case performance of an algorithm. Users are a nuisance: they need stable, working products with good documentation; they need continued support and development of the tool; and they represent a long-term commitment which it may not be appropriate for you to make. Of course, users are also an extremely valuable source of feedback and ideas.

If your tool does have users, then their interests represent another factor to be taken into consideration when making decisions about the tool. Users' interests will inevitably sometimes conflict with other considerations. For example, faced with limited time and demands both for a paper about a new feature of the tool and for theoretically uninteresting enhancements to the existing functionality, what are your priorities, and do your users understand them? (The same considerations apply even more strongly if you plan for your tool to be extended by other developers: see below.)

2.3 Who are the intended users of your tool?

Different communities of tool users may have quite different requirements. Some existing communities are:

- Other researchers.
- Students and their teachers.
- Seasoned industrial verifiers. For them, performance is likely to be crucial. Interfaces must be efficient to use once learned, but people who use a tool regularly are surprisingly willing to put up with unintuitive interfaces, if the power of the tool provides enough benefit.
- Novice industrial verifiers. In some ways the most rewarding group: but they will need and expect support, which may include help with basic concepts of the underlying theory, as well as help with the tool itself, so you will need to decide what you can provide.

2.4 Do you want other people to be able to add components to your tool?

In the case of the Edinburgh Concurrency Workbench the answer was an emphatic yes, but we've only recently put in place an architecture that makes this reasonably easy. If you decide that people should be able to add components to your tool, then as far as possible, they should be able to add a component by writing their own code using facilities provided by you, ideally without altering any other source code of the tool. (An example is provided in section 3.1 below.)

3 Architectural decisions

We use a broad definition: architectural decisions are those that affect the whole of a system, and hence have to be made early in a project. They are the decisions which define how components work together. This includes, for example, decisions about what programming language should be used and how errors which are not the sole concern of a single component should be handled, as well as decisions about the high-level structure of the system.

3.1 High level structure

The factors that a tool developer has to take into consideration when designing the high-level structure of the tool are largely common with those that any software architect has to consider. We give an example of an improvement made to the architecture of the CWB, to illustrate briefly how standard techniques can be used to make it easier to maintain a verification tool and, in particular, to make it possible for others to add components.

Originally, we had a top level loop that read a user command and called an appropriate function, based essentially on a static case statement. In addition, the on-line help was defined in its own file. So someone who wanted to add a new command had to edit the top level loop and add their help information into the help file. In total four files had to be edited, in addition to the files written by the component writer and excluding the build files. The changes were all small – a few lines at most – so if the CWB maintainer did the additions this situation was simply a nuisance. It was a serious barrier to allowing other people to add components, though, especially since some of these components are likely to be transient features of someone's experimental version, rather than new permanent features of the CWB.

We replaced the original architecture by a layered architecture in which new components are developed at the highest level, making use of facilities provided at lower levels. There are no dependencies from lower levels on higher levels, so in particular no changes are required to the rest of the CWB source code when a component is added at the highest level. Instead, we use standard callback mechanisms. For example, to register a new user-level command the component writer calls a lower level function `registerCommand`, providing the name of the command, the help text with the command, and the function (signature `unit -> unit`) which should be called when the user issues the command. The function must process the arguments (using the lower level functions provided) and carry out the appropriate functionality.

Standard pattern catalogues such as [4, 1] are useful sources of simple but valuable techniques like these.

3.2 User interfaces

One of the more interesting questions, because there's a serious conflict of interests, is what style of user interface a tool should provide. It is widely assumed that a user interface is good – that is, easy to learn to use and to use correctly – if and only if it has a "snazzy" graphical user interface.[2] Both directions of the implication are simply false. If all you cared about were the usability of the tool, you might or might not develop a GUI: you would certainly consider a great deal of agreed knowledge about usability and user interface design ([10, 3, 5], for example). If you care about maximum impact in short demonstrations, e.g. at conferences, however, then you need a GUI.

Designing a highly usable user interface takes time, effort and expertise: the Edinburgh Concurrency Workbench is not yet an example of good practice. For graphical user interfaces in particular, you also need to consider:

- the stability of the toolkit (an especially thorny issue for free toolkits)
- the programming effort required to use it, which may be considerable
- portability issues: what platforms can the resulting interface run on?

3.3 Process paradigm (or: To BDD or not to BDD, that is the question)

Some very basic choices will underlie the choice of what kind of tool to build. Is it to be automata-based for example? How are the states to be represented? There are some trade-offs here. The popular industrial tools are automata-based, by which we mean that a process algebra is used, at most, to enable the user to define the process they want to work with. Internally, the states are represented as the finitely many states of a finite state automaton, not as process algebra derivatives. This has the important advantage that it can be done in a space efficient way – for example, by using BDDs as an efficient representation data structure. The Edinburgh Concurrency Workbench has the grave disadvantage that, because it represents states by the CCS syntax that defines them (and is also written in a notoriously space-inefficient language/compiler), it requires many bits to represent each state, and, worse, the larger the system the more bits per state are required on average. Because of this, the CWB has difficulty in answering questions which require building the full state-space of systems

[2] Sadly, it is difficult not to see the ETAPS requirement, that tool demonstration submissions include a screen-shot, as an instance of the assumption.

which are much larger than 30,000 states, on machines with typical amounts of memory. These are pathetically small systems by the standards of the best automata-based tools. On the other hand, the CWB can happily answer questions about systems which happen to be infinite state, provided that the question does not require the exploration of the entire system. A restriction to finite state systems may not be a problem if your users conceive their systems as finite state automata, but otherwise it conflicts with another good principle of tool development, that you should make easy things easy. [3] The CWB's approach also has the advantage that it is often possible to take advantage of knowledge about the system structure purveyed in the CCS term structure, for example for providing user feedback or for compositional verification.

3.4 Choice of programming language

You need as much support for correctness and for ease of maintenance from the language as possible. In my opinion, that means you want a reasonably strong static type system, and a module system which strongly supports encapsulation – by which is meant not only that it is possible to design systems with good encapsulation, but also that this is the natural thing to do in the language concerned. (The module system might be an integral part of the language, as in C++, or separate, as in ML.) Bear in mind that you are trying to guard against mistakes, not maliciousness.

This said, the obvious candidates are typed object oriented languages such as C++ or Java, or functional languages such as ML, variants such as Caml or O'Caml, or Haskell. If performance of the tool is crucial, you will probably prefer C++. Some users of O'Caml report give encouraging reports, but O'Caml's single-provider status and restrictive licensing rule it out of consideration as a language for the CWB at present. (The stability, support and long-term future of a language are naturally among the things that tool developers need to consider: verification tools are not special in this respect.)

Similar considerations apply to choosing which version of a language to use, which often includes deciding on a compiler. A recent mistake – which, with hindsight, I should have avoided – was to bow to user (and ML developer) pressure to port the Edinburgh Concurrency Workbench code from the original version of Standard ML, known as ML90, to the new version known as ML97, and to make a corresponding change in the versions of the New Jersey libraries it used. There are serious advantages in making the change, most importantly that the new version of the SML-NJ compiler is available on many more platforms than the old, so the port increases the number of users to whom it is available. In particular, the new compiler is available on Microsoft operating systems, and the lack of ability to run the Edinburgh Concurrency Workbench on these platforms has been something I've wished to change for some time. However, it turned out that the compiler was not sufficiently mature to make the porting effort efficient.

[3] As has been pointed out in [9], for example, easiness is a property of questions not systems: some questions about infinite state systems are easy.

Not only was there a serious compiler bug which required non-trivial temporary changes to the Edinburgh Concurrency Workbench code to work around, but also there were many problems of incomplete or incorrect documentation; the porting process took more time and effort than it would have done if I'd waited for greater maturity.

3.5 Encapsulation

The virtues of encapsulation – of software that consists of modules with well-defined interfaces defining their interactions, the interfaces being no broader than is required and serving to enforce invariants on the data – are well known. Two aspects are particularly important for verification tools:

First, encapsulation barriers act as fire barriers for bugs: small, well-defined interfaces limit the possibility of hard-to-understand interactions between different parts of the system. The type, class or module system of the programming language is an important tool, but, for example, using the ML module system is no guarantee of good encapsulation. Here is an example of a poorly chosen encapsulation barrier allowing hard-to-find bugs. It also, perhaps more importantly, serves as an example of how software degrades with repeated modification.

When I took it over, I found that the Edinburgh Concurrency Workbench made great use of an ML module called SortedList, whose interface consisted of various functions whose types involved the built in ML polymorphic list type 'a list. For example, it provided a function add, whose type was

```
('a * 'a -> bool) -> 'a * 'a list -> 'a list
```

The code for the function assumed that the list in the second argument was sorted according to the order given in the first argument. Using this assumption, it added the element in the second argument into the list and returned the result. But there was no compiler support to verify the assumption! The CWB has suffered from very many bugs of the form:

- one piece of code takes a sorted list, does something non-order-preserving to it, and passes it on to another piece of code
- which does something to it, erroneously assuming that it's sorted.

The compiler cannot tell the difference between a list which is intended to be sorted, and one which is not (and even if it could, it would not know by what order function it was supposed to be sorted). Neither, without very scrupulous documentation, can the human maintainer! The result is bizarre bugs, which may, for example, take the form of the tool giving right or wrong answers depending on the names the user chooses for their identifiers.

With 20/20 hindsight, it is clear that what is needed is a separate type for sorted lists, so that the mistaken code above would fail to type-check. There is a variety of ways to implement this, which it is not particularly interesting to go into. The point of the tale, though, is that *there probably never was a wicked programmer who did something idiotic.* As near as I can reconstruct what happened:

1. initially, everything was done using lists, making no use of order;
2. in order to improve performance, some individual pieces of code were (re)written to enforce and take advantage of an ordering.
3. someone (else?) realised that code for sorting and for adding items to list was duplicated in several places, and factored it out into the SortedList module
4. subsequent maintenance used these facilities freely.

At no stage, I hypothesise, did anyone really make the (wrong) decision that this was the right way to do it. This is typical of the way in which systems lose their good architectural properties.

At this point object orientation fanatics are probably feeling smug: this is a perfect illustration of why it is more important to use a language in which it is difficult *not* to encapsulate, than to use one in which encapsulation is possible. This class of bugs would not have occurred in a system written in Smalltalk.[4]

Second, encapsulation makes reuse and replacement of components easier. In a long-lived experimental tool, it is important to be able to replace parts of the tool without having to understand the whole of it. (Reuse, as such, is normally not such an issue: but the factors which make replacement easy are very close to those that make reuse easy, so much of the reuse literature applies.)

ML ML's module system allows well-structured systems to be designed, but in my experience it has serious shortcomings, to the extent that I would not choose ML if developing the CWB from scratch. There is some explanation in my talk "Why I hate ML", slides for which are available [8].

4 Development process and quality assurance

A quality assurance process suitable for academic development of verification tools needs to be extremely streamlined and will probably be largely undocumented. It may be helpful, though, to keep notes on what the process is, in order to help future developers understand what's been learned.

Development will almost certainly need to use an iterative process rather than a waterfall process. The usual reason given for projects choosing iterative development is that not all customer requirements are clear at the beginning of the project. This is not usually a major problem for verification tools, but where correctness is crucial it is convenient to be able to begin system testing early. Moreover in cases where the algorithm to be used is not completed, experimenting with simple-minded versions which are then refined can be helpful. (Of course, while an algorithm that is implemented and appears to give the right answers may be a powerful incentive to proceed with a proof of correctness, it is not in itself evidence of correctness of the algorithm!)

[4] *Pace* Kent Beck, though, this is not a recommendation that verification tools should be written in Smalltalk.

4.1 Refactoring

Refactoring is an important technique for maintaining the architectural integrity of systems that undergo many modifications, whether in an initial iterative development or by undergoing later modifications. The basic idea is that a *refactoring step* is a modification that does not alter the functionality of a program, but clarifies or improves the implementation. It is advocated that one should be clear at all times about whether one is refactoring or implementing new functionality, avoiding mixing the two. This simple idea seems surprisingly powerful: it helps one to avoid allowing the remnants of defunct design decisions from previous iterations to pollute the design of the current system, whilst also avoiding the temptation to get sidetracked into redeveloping the entire tool.

Refactoring was first described by William Opdyke in his thesis [6]; his short paper [7] is a relevant introduction.

4.2 Coding standards

You probably do not know who will maintain your tool later, and – unlike most industrial developers – you probably cannot assume that it will be someone with much experience of development. So it is particularly important to make the code you write as easy to read as possible. Some basics, such as using long-enough meaningful identifier names can help enormously. In functional languages especially, there seems to be a temptation to write very complicated "write only" code, which needs to be resisted.

4.3 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. – and to keep enough information to cross-check the version numbers. For example, we mentioned that CWB test results are kept with a record of exactly which version of every source file was used. In the case that a bug is reported which has been present in the tool for some time, it's immensely helpful to be able to work out what changes to what files introduced it.

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. I find the emacs vc-mode adequate: checking in a new version takes five keystrokes, plus whatever is required to type the comment on the version.

4.4 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. Another commonplace in mainstream software engineering is that developers should write as much test code (code that forms part of test harnesses) as production code. 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. As usual in an iterative development process, some aspects of functionality will have to be retested very frequently, which makes automated support for regression testing essential.

The Edinburgh Concurrency Workbench's system testing software is written in Perl, a language which is well adapted to this kind of task. Systems tests are stored as CWB input files. The test controller program takes a list of test (or test suite) filenames and, for each test, it runs the CWB on the test, directing the output to a known file. It compares the output with a hand-checked "known correct" output file, and reports any discrepancies. It records the RCS version numbers of all the files used to build the version of the CWB it is testing (which the CWB gives in response to the command "cwb version;"). It also records the time taken by each test, as a crude measure of changes in performance. This simple program enables the CWB developer to run tests and spot newly introduced errors with minimal effort. The disadvantage of its simple-mindedness is that semantically insignificant changes to CWB output – printing nominally unordered output items in a different order, for example – are reported as errors. It would be possible – and perhaps an interesting student project! – to design a more sophisticated tool which would enable the test writer to specify more about what constitutes correct output. In the meantime, mindful of the importance of avoiding getting carried away in implementing bells and whistles (see section 5 below), we have a simple separate Perl script which deals with the most commonly occurring instances of this phenomenon, leaving less common occurrences to be checked by hand.

4.5 Defensive programming

Sanity checking in the code is much more important for verification tools than for standard business programs, because there is typically more that can go wrong and it is harder for unit testing to achieve appropriate coverage. Concretely, you will probably need some kind of statement in the code which will arrange for some diagnostic information to be produced only if some flag is turned on. Care is needed to avoid bad interactions with the efficiency considerations of the tool. For example, soon after taking over the Edinburgh Concurrency Workbench I added such statements using a new function called "debug" which took a string as argument and showed it only if debugging was turned on. The problem with this, I soon found, was that it had a serious impact on performance *even when debugging was turned off*. Of course this was because – since ML is a call by value language – the string argument was being built whenever the line containing the function call was executed, whether or not the debug function was going to decide to print the string or not. The simple work around was to thunk it: that is, to use a new function debugFn instead, which takes, not a string, but an anonymous function taking no argument and returning a string. The function is passed to the debug code, which only calls it if the string is required. The effect on the client code is that instead of writing:

```
UI.debug "The number of states is "^foo
```

one writes:

```
UI.debugFn (fn  _ => "The number of states is "^foo)
```

In languages which do not have function passing as a central concept, one might pass an object to similar effect, or use the capabilities of an appropriate macro language.

Testing is a second line of defence after type checking and the encapsulation facilities provided by your module system have failed. As many bugs as possible should be caught by those, first lines of defence.

5 Automating things

We have several times touched on the issue of how automation can help to minimise the effort required to assure quality of verification tools, for example, by making regression testing easy. 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. (The script built the CWB for the platforms I had available, transferred files to the export FTP directory, tar'd and gzipped them, etc.) In fact, the effort of building and debugging the script was not worthwhile. 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.

6 Supporting users and component developers

6.1 Documentation

Documentation is naturally crucial. Tool developers normally realise this, but sometimes forget that out of date documentation can be little better than none. There are several kinds of documentation which you might consider:

- On-line help
- User manual, for printing out
- Web-based documentation

Because these are used by people under different circumstances, ideally you would provide all three written in slightly different styles. For example, the on-line help about a command would provide, tersely, the most important information about a command, whereas the user manual intended to be printed out would be more discursive. Writing separate documents makes good sense in a commercial environment where plenty of resource is available. However, verification tool developers do not normally have this luxury. In the case of the Edinburgh Concurrency Workbench we decided that we could not spend the effort required to maintain three separate documents, so we have a single source of information for each command which is compiled in three different ways (by a very simple home-made Perl script: we avoid using advanced features of any of the text presentation languages) to give the three access methods listed.

6.2 Distribution isses

The Web is the obvious means of distribution, at least for a free tool. You need to consider licensing issues: under what conditions do you want people to be able to use your tool? Do you want it to be free to everyone, or just free for academic and/or non-commercial use? Do you want to make the source code available? You retain the copyright on your code unless you explicitly give it up – in most jurisdictions this does not rely on you inserting copyright statements anywhere – but of course you may not be able to enforce your rights. "Public domain" is often incorrectly used as a synonym for "free": in fact if something is public domain anyone can do anything they like with it. For example, someone could take a public domain tool, package it up removing all reference to the authors, and sell it.

You may want to start a mailing list to carry announcements and possibly discussion of your tool. This is easy to do using majordomo, for example. You will need to consider whether you want to moderate the list.

Platforms A related issue is what platforms you want your tool to be available for; this will be particularly difficult if you don't want to distribute source code. You might be able to find people to compile the code for you on various platforms – but do you really want to distribute code that you didn't compile and can't test? And will the owner of your web site agree? If your chosen language is Java you might get round this by distributing Java byte code: but you might have to rule out Java for other reasons, e.g. performance.

6.3 Support

People will send you email about your tool of a variety of kinds. Some people will send suggestions or requests for new features; some of these will seem like cool ideas, others like crazy ones. A surprising number of people will send you bug reports or queries which demonstrate that they haven't understood the basics of the theory on which your tool is based. Some people will send you bug reports which really are bugs.

It is hard to answer all email helpfully and promptly without letting it take over your life, and I have not mastered this yet. A common piece of advice is to set aside a particular time on one day of each week for answering tool-related email, but I have found this approach to be ineffective. The trouble is that in a high proportion of cases an answer will be needed urgently, either for your peace of mind or for the sender's. For example, if someone reports a serious bug and you delay replying for nearly a week, it gives the bad impression that you do not care or are not surprised.

You cannot completely satisfy everyone who sends you email: tool developers are human, and there is no need to apologise for this! There are some ways you can respond positively without committing time you don't have:

- In the case of requests for new features, encourage the correspondent to do the work themselves. This is only practical if the tool is in a reasonable state for someone else to work on, which is a good reason for ensuring this. In practice most people will find that they don't really want the feature if they have to work on it themselves!
- Is it something that could at some stage be a student project?
- If the correspondent has a serious need for something your tool does not provide, can you recommend another tool that might suit their needs better?

References

1. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley, 1996.
2. Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
3. Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall, 1993. (hardback); 0-13-437211-5 (paperback) only outside USA.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.
5. Thomas K. Landauer. *The Trouble with Computers: Usefulness, Usability and Productivity.* MIT Press, 1996.
6. William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
7. William F. Opdyke. Object-oriented refactoring, legacy constraints and reuse. presented at 8th Workshop on Institutionalizing Software Reuse, available from http://www.umcs.maine.edu/ ftp/wisr/wisr8/papers/opdyke/opdyke.html, 1996.
8. Perdita Stevens. Why I Hate ML. http://www.dcs.ed.ac.uk/home/pxs/talksEtc.html. Slides from talk given to the Edinburgh ML Club and Glasgow Functional Programming group.
9. Perdita Stevens. Abstract games for infinite state processes. In *CONCUR'98*, number 1466 in LNCS, pages 147–162. Springer-Verlag, 1998.
10. Harold Thimbleby. *User Interface Design.* Addison-Wesley Publishing Co. (ACM Press), Reading, MA, 1990. ACM Order number 704907; QA76.9.U83T48 1990.