Formal Techniques in Large-Scale Software Engineering

Mathai Joseph
Tata Research Development and Design Centre
Tata Consultancy Services
54B Hadapsar Industrial Estate
Pune 411 013 India

Draft of Paper

1. Introduction

Formal techniques have been used effectively for developing software: from modeling requirements to specifying and verifying programs. In most cases, the programs have been relatively small and complex, many of them for safety critical applications. Use of formal techniques has also become relatively standard for the design of complex VLSI circuits, whether for processors or special purpose devices.

Both of these areas are characterized by having a small state space, closely matching the capability of the formal techniques that are used. This capability has of course increased greatly over the last two decades, just as processing speeds have improved, and the problems that are solved today would have been out of reach, or even beyond expectation, some years ago. However, from the point of view of software engineering practice, formal techniques are still seen as exotic and suitable only for particular problems. Software engineers will agree with promoters of formal techniques, but for very different reasons, that these techniques are not applicable to today's software engineering methods. They will usually disagree on why this is so, one pointing to the lack of contribution that formal techniques make to the practice of these methods and the other questioning the methods themselves.

Successful applications of formal techniques have tended to be in the few areas where their use has become part of the practice, for reasons of complexity, cost or regulatory pressure. But the mechanisms supporting formal techniques have now reached a level of maturity where they can be used for many more purposes than those attempted today. Software engineering methods themselves are now better understood and systematized, especially when they are measured against the requirements of the so-called maturity models such as CMM.

There is urgent need for formal techniques to be used to support software engineering practice on a wide scale. There are improvements that are becoming increasingly essential in different areas, such as requirements modeling, software architecture and testing. And there are major contributions that formal techniques can make to improving practice.

This paper proposes that the Grand Challenge of the Verified Compiler should be seen in a wide context. The use of the term 'compiler' in the challenge may suggest that the activity will start with a program, perhaps with some formal annotations. I will argue that the specification, program and verification steps that are often the context for using formal

techniques must be extended to start earlier, with requirements, and end later, with testing. In fact, there is good reason to assume that for a great deal of application development, verification of a program will be a far less important activity than is perceived today.

2. Formal 'Methods' and 'Value'

The use of formal techniques in large-scale software engineering faces two major obstacles:

a. The techniques have no defined role among the steps of existing software engineering methods, and

b. There is no 'method' to the use of a formal technique: while the goal may be precise, there is no specified series of steps that can be followed to reach the goal.

Nevertheless, there are ways in which formal techniques can be used in the software engineering process:

(i) To give greater assurance in the application of steps of the development methods, and

(ii) To augment the steps themselves with formal operations.

However, for a software engineer, any change in current practice needs to be justified by a perceivable (and, ideally, measurable) improvement in terms of reducing the effect of a commonly accepted problem. Moreover, the use of a formal technique should be seen to require no more effort than is currently required (described by Bob Kurshan (1994) by saying 'for the use of a formal technique to be acceptable, it must be seen by the project team as a zero cost option!').

A formal technique should therefore:

a. Provide useful data otherwise that is not available;

b. Reduce the time or the effort needed for completing a task;

c. Add measurable quality to the resulting software;

d. Scale up to meet the project needs; and

e. Be capable of being used reliably and repeatedly.

Few formal techniques available today meet all these requirements. The low acceptance of formal techniques in the industry points to the difficulty in promoting their use for benefits that do not include solving known problems.

3. Problems

Rather than starting with a formal technique and looking for its possible uses, it is rewarding to first look at where there are known difficulties in the software engineering process.

a. In any large software development project, it will be seen that requirements change throughout the development and maintenance lifecycle. Some of these changes may due to inadequate planning or lack of common understanding between different groups. Others may be forced due to changes in the operating environment, or (most commonly) due to new requirements arising. In practice, it is not unusual that requirements change; indeed, it would be unusual if they did not.

b. Each change in requirements will usually result in changes in the program design and the program code. Further, changes will be needed to ensure that the testing procedures cover the changed requirement.

c. Few programs work in complete isolation so in general the requirements, design, coding and testing must take place in the context of the chosen operating environment, which includes software at platform and higher levels. Changes are possible here too, as mentioned in (a).

Tracking changes through the development cycle is a major challenge because it also requires ensuring that the effects in different steps in the process are consistent.

As an example, consider the following data. An analysis of efforts used in a wide cross-section of projects using Java, Cobol and Oracle/Developer 2000 in a large company showed the following averages:

Requirements analysis	4%
Design	16%
Construction	64%
Testing	12%

The remaining time was spent on project management and review. Note that a lot of the requirements analysis was done with the customer, not in the company office, and is not logged as part of the efforts listed here.

These figures suggest immediately that since a great proportion of the total time was spent on construction, that is where improvements would have the most effect. An indication of how to make improvements could come from looking at figures for the amount of *re-work* that was being done, since this will give an indication of things being done incorrectly.

Analysis of the percentage of effort spent on re-work gives an interesting picture: of all the defects found before testing, as much as 59% were corrected in the construction phase. In other words, there a great deal of effort during construction on correcting errors. These errors could have been introduced during coding or in an earlier phase. One could conclude that either (a) the programmers were particularly poor at their task, or (b) there is some other cause. In discussions with project teams in the company, a different picture emerged.

Invariably, project teams were able to show that the extent of rework during the construction phase was almost entirely due to *requirements* being corrected, augmented or changed right through the development cycle, far less often due to coding errors.

Studies of program development projects and so-called software 'maintenance' projects show clearly that requirements *will* change throughout the lifecycle. When the resulting program changes have to be made manually, the cost rises and the possibility of introducing new errors increases. In general, requirements written in any notation are shorter and more compact than the corresponding program code. If code is generated automatically from models created from the requirements, changes in requirements could be automatically converted into program changes.

In the next section, we will look at some results obtained from automatic code generation from UML models in large software engineering projects.

4. Automatic Code Generation

Manual coding has limitations in the amount of code that can be produced by a person. A programmer produces around 15 lines of code per day when averaged over a project; there will be outstanding programmers who far exceed this average, and there will be others whose average is significantly lower. Taking the average as a representative figure, it is not difficult to calculate the time and effort it will take to produce programs of the size typically expected in a reasonable-sized project. For example, a final program of one million lines will require just under 300 person years of effort, or a team of 100 programmers working for 3 years.

These figures are of course merely indicative. However, even if divided by a factor of 2 or 3, the numbers are still large. Moreover, in practice, project overheads multiply with team size and total productivity drops; on the other hand, reducing team size will increase the duration of the project.

Given the speed at which business opportunities change, no customer will be prepared to wait for years before a required software solution is delivered. If the team size must be kept high, then just to assemble, train and organize the team will itself take considerable time.

Many application software projects are expected to produce a final program of anything between 1M and 5M lines of code and to do so within as short a time as possible. This leads to the need for *generating* as much code as possible, rather than producing it by hand.

Example

A large and highly complex financial services software system was produced using a software development environment that included automated code generation from UML models and compact operational definitions of operations. The figures for different phases of the project are shown below:

Requirements analysis	36%
Design, construction and testing	64%

A team of 20-30 engineers were responsible for defining the requirements over 5 months and a team of 60 programmers developed the UML models and operational definitions to generate over 6M lines of code in 6 months. Few errors were discovered during testing and very little re-work needed to be done. The productivity per programmer was as high as 300 lines of generated code per day.

Given the inefficiencies in size of generated code, dividing the final program size by 2 or 3 would give a figure closer to the size that may be expected from a team of good programmers working using the usual methods. However, with manual programming both the team size and the project duration would increase very greatly.

End of Example

This is not an isolated example. 60-70 systems of large size have been produced using the same software development environment and many more are under development.

There are many important advantages to using automated code generation:

a. Great reduction in programming effort and development time;
b. Almost complete absence of testing associated with programming errors;
c. Uniformity in code structure, making inspection and maintenance easier;
d. Reduction in time and effort needed to accommodate changes in requirements;
e. Ability to generate programs for different platforms from the same requirements, with no increase in effort.

This comes with a price: increased program size. However, this is something that matters less in these days of falling memory sizes.

5. Software maintenance

There is more software under so-called maintenance than is being developed at any point of time. Figures vary widely but to say that over 70% of the total software work is for maintenance would not be far wrong.

It is often assumed that software maintenance consists of correcting the bugs left by bad programming and inadequate testing. However, industry figures (http://fox.wikis.com/wc.dll?Wiki~SoftwareMaintenance~SoftwareEng) tell another story.

Remedial effort (correcting errors)	21%
Adapting to operating changes	25%
Enhancements (adding new features)	50%
Improvements (making it more robust)	4%

These figures are not dissimilar to those collected at a large software company:

Remedial effort (correcting errors)	27%	
Adapting to operating changes	< 0.5%	(handled separately)
Enhancements (adding new features)	58%	
Improvements (making it more robust)	4%	

It is clear from both cases that the major part of the effort goes into making changes in the software in response to changes in requirements. While the figure of 20-25% shows that bug fixing takes about half the time spent on adding new features during maintenance, it is still large.

6. Automated Test Generation

To be added.

7. Discussion

There are a few conclusions that can be drawn from the figures shown in the previous sections.

c. Improving the way requirements are modeled could have a substantial effect on software is developed and maintained.

d. Automated code generation can cut down greatly on program development time.

e. Improving and automating methods of testing can reduce further reduce the time needed for development and maintenance.

Can a Verifying Compiler remove the need for all but cursory testing? Most large applications run in an environment where there are interfaces to many other software systems: existing software systems, middleware and communication layers, database systems and many others, apart from the operating system. It is hard to see how these interfaces can be modeled and checked during even an extended ‘compilation’ time. Moreover, the interfaces are subject to change so verification checks would need to be repeated for each change. It is already sufficiently onerous to repeat the full cycle of regression tests each time that a change is made in the software. If to this is added the cost of re-verifying the program for each change, it would add considerably to the overhead of using the Verifying Compiler.

7. Conclusions

The goal of producing the Verifying Compiler is seen as a task requiring international effort between teams contributing in different ways. It is a remarkable challenge and one that should help to bring together many different approaches, techniques and skills towards a common purpose.

This purpose must also be seen by practitioners as common to the evolution of software engineering techniques. If the already wide gap between the use of formal techniques and

software engineering practice is allowed to increase, getting the Verifying Compiler used in practice will certainly prove to be a much larger challenge.

There can and should be many smaller steps towards reaching this goal, motivated by the need to produce a Verifying Compiler. There are things that can be done today and an excellent example is the use of verification techniques for checking properties of device drivers for Windows systems, described in papers by Rajamani and Ball and others. There are many other examples of the use of formal techniques in software engineering practice in large companies, each of which is producing measurable improvements and a level of acceptance by software engineers that is very promising. In this paper, we outline a few such initiatives (one given above, more to be added) and discuss the directions where new work is focused.

8. References

To be added.