SE33010 Assignment Two - Alexander D Brown (adb9)

Moving from the Spiral Model to Formal Methods

Introduction

This report will detail the differences between the spiral model of software development currently used to formal methods of software development including reasons to change to a approach using Formal Methods as well as some of the deficiencies of using such a strategy.

Finally, this report will also discuss how to move to a formal methodology and some of the potential tools which can be used to aid this transition.

Comparing Formal Methods of Software Development to the Spiral Model

The spiral model of software development is a lifecycle which is intended for large, expensive and complicated projects, explicitly including risk management as part of the development process.

Formal Methods of software development have many different flavours but focus upon proving that a piece of software satisfies a (formal) specification, typically this specification has a mathematical form. Formal Methods are very useful for safety- or mission- critical systems; where traditional methods would have to rely on large amounts of testing.

It should be pointed out that formal methods do not prove that a program is correct; the methods only focus on satisfying the specification. If the specification is sufficiently detailed to ensure the software is correct then it should follow that the formal methods will ensure the software is correct as well.

Both methods start similarly; gathering requirements for the system. However the spiral model focuses on both requirements and risk analysis whereas Formal Methods these requirements are used to build a formal specification.

Whilst the spiral model builds software incrementally through various prototypes at defined points in the spiral, a formal methods purely builds on the given formal specification. This requires this specification to be very well defined and means it can be very difficult to adapt to changes in requirements.

Rather than the traditional approach of testing software, as is used in the spiral model, formal methods - in general - do not require a heavy suite of tests. The formal specification produced replaces many of the tests required of software and instead focus on proving a piece of code fulfils the specification, rather than passing a test.

The next sections details some of the benefits of using a formal methodology in more depth.

Benefits of a Formal Methodology

There are several advantages to using a formal methodology over a traditional or, as are more prevalent these days, agile methodology to developing software.

One main problem with software development is the inconsistencies, ambiguity and incompleteness of the requirements of the software. By building a formal specification of the requirements it becomes clearer as to where these problems exist early in the project.

Formal methods also allow the evaluation of different design alternatives without the need for expensive prototyping. This can help to improve the overall architecture of the system and, potentially, make it more robust.

Correctness by Construction[1] is one method used in certain formal methodologies to reduce the number of defects by exploiting the benefits of abstraction to achieve clarity and completeness of a specification. This is done by using formality as early in the software lifecycle as is possible and integrates the whole entire team in this formality. Nothing is done unless it add value to the project and two principals are adhered to: *i*) Avoid introducing errors *ii*) Remove errors that are introduced as soon as possible [2]. This is achieved by employing the most rigorous notation at each stage.

Issues with Formal Development Methods

Formal development methods typically require markedly different skill set from traditional software development, a heavy knowledge of mathematics; particularly set theory and logic; is required to be able to write the formal specifications on which code is built to fulfil.

This also impacts the customer as, to retain the most rigorous notation at each stage requires the use of full mathematical notation which not all stakeholders can be expected to be well versed in. Retaining this rigorous notation also prevents the integration of diagrams and English text which are often also needed as part of the project, leading to a large number of different documents.

Another major problem is the programming language used to implement the formal specification: this language should need to be formally proved itself otherwise the software is being built on an unsound base.

One major issue with formal methods is the significant time investment needed to build the formal specification, when done by hand it is a very time consuming process and is very prone to human error. If checked by a tool or 'verifier' then there is the inherent problem of proving that the verifier is performing its job correctly.

For large pieces of software there are usually large lists of requirements. When made into a specification this leads to an unacceptable amount of clutter. Very few formal methods are able to provide the modularity needed to solve this issue. This is especially an issue when items are added to the requirements of a project.

Moving to a Formal Methodology

There are two man methods of moving to a formal methodology:

1. Introduce a small team of experts carrying out the formal work. This is relatively easy to introduce into projects and concentrates the scarcity of resource. However, the benefits are very limited as a small team will only be able to focus on small areas and may lead to divergence between the formal process and the mainstream project.

2. To yield larger benefits the more ambitious strategy of employing formal methods on the entire project can be used. This involves a lot more training but means that formal and informal methods can be integrated and the project can benefit from a good level of formality in its entirety.

Tool Support

Tool support for formal methods tend to revolve around the automation of verification of a formal specification and the conversion of a formal specification into executable code. Both have the benefits of reducing the possibility of introducing human error.

Verification systems have two main categories; automated theorem proving and model checking. Automated theorem proving involves a system attempting to generate a proof from scratch, given a description of the system, the logical axioms and a set of inference rules. Whilst Model checking involves verifying all possible states a system can enter during execution are valid.

Code generators, as the name suggests, take a formal specification and turn it into executable code, this allows the majority of focus to be on the specification and also allows for some flexibility when it comes to the changing of requirements.

Another useful area of tool support is the generation of test data, based on the formal specification. This allows all boundary cases to be covered without an intimate knowledge of the system and frees up the time of those creating the tests to build more complex tests where needed.

Conclusion

This report has begun to detail some of the more important strengths and weaknesses of formal methods as well as discussing the process of moving to a formal method and some of the types of tools which can be used to aid in the development of software with a formal method.

These points should be reviewed carefully before any decisions are made, more detailed research is required if the decision is made to move to formal methods. This report has not discussed the various methods available to implement a formal methodology with and which would suit the business needs most fully.

References

[1] Roderick Chapman. Correctness by construction: a manifesto for high integrity software. In *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[2] Anthony Hall. Realising the benefits of formal methods. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin Heidelberg, 2005.