Experiences of Formal Methods in ‘Conventional’ Software and Systems Design

Ian Oliver
Nokia Research Center
Itämerenkatu 11-13
Helsinki
Finland
ian.oliver@nokia.com

Abstract

We describe the results of the use of formal methods integrated into a conventional, industrial software development environment in a situation where a meaningful comparison can be made against a parallel running project based on the same requirements. Despite the generally chaotic, ad hoc processes and methods that are employed in such environments, the addition into the mix of *some* formality greatly improves the overall quality of the software being constructed.

1 INTRODUCTION

The use of formal methods in software engineering is not widespread; at least not to the degree that methods based upon UML are for example. The primary reasons [9] are that formal methods are too mathematical, not mainstream, too difficult, not applicable etc. Ironically the use of informal and semi-formal methods and the rise of the de facto, ubiquitous UML has revealed a necessity for more formal approaches and the need to specify systems in a clear, precise manner.

Formal methods have their success stories primarily in safety-critical, secure and fault-tolerant systems (NATS, Paris Metro Line 5 are prime examples) but rarely are cases outside of this in more mainstream development heard. One reason may be that the use of formal methods has been hidden by one of the practitioners or that the particular use of formal methods does not stand up to certain rigorous ideals eschewed by formal methods and the results of these are not (sufficiently) reported.

In this paper we describe our experiences¹ of the use of formal methods in more mainstream, day-to-day style software engineering.

2 CONVENTIONAL SOFTWARE DEVELOPMENT

We use the term “conventional software” to relate to those software projects which utilise ad-hoc methods and development techniques in agile or semi-structured processes with design-by-committee style development. Often the development team splits into a number of (semi-)independent groups working on differing abstractions of the system in parallel. Typically such software is constructed through a hash of requirements documents (invariably written in Microsoft Word), models (using the Microsoft PowerPoint ‘modelling tool’) and code (Java, C++). The focus is on good process (CMM level > 1) rather than good method.

Empirical results such as those relating to the amount of work made: size of design/code, bug reports, failure rates etc are rarely made beyond the almost meaningless lines of code measurement

---

¹within the confines of industrial secrecy

[15]. In these environments change requests are often in the same form as bug reports which makes differentiation between features and bugs and tracking these changes difficult if not impossible.

While the above *might* be an over-exaggeration, development of most software does not follow strict conventions or norms and that the need to generate code far outstrips the need of constructing the correct system correctly.

If any detailed modelling is made then semi-formal languages such as the UML with its variable semantics are applied. This further compounds the problem that abstraction levels are mixed (specification vs implementation) and the separation of concerns between architecture, domain and design are not made. Terms such as ‘architecture’ are guaranteed to be over- and misapplied.

Given these environments which can only be described as ‘typical’ it is not difficult to see how addition of formality into the work is needed and what benefits can come because of this. The mandating of languages such as UML into the development flow are actually steps to accepting the necessity of more formal procedures and techniques to provide more detailed analysis of said models.

Despite of all this, software is written and *generally* works according to the customer’s needs. The whole development process emerges out of the chaos and the concepts of the development languages, models, requirements and code are broadly agreed.

It is the author’s belief that wholesale introduction of any technique is either not possible or counterproductive and that development in these types of environment is actually an organic composition of many small techniques any of which can be made more formal without upsetting the overall process.

The rest of this paper describes our experiences in a ‘typical’ project producing ‘conventional’ software where light-weight formal methods were utilised.

3 THE EXPERIENCES

Previous experiences of utilising formal methods have been undertaken utilising the B language in a variety of projects: GSM data structure specification and codec design [10], hardware development [12] and also protocol development with reasonable success [14] - meaning that the specifications form part of the internal documentation and a basis for the implementations relating to those systems. Informally we can state that in those areas where formal methods have been used we can see approximately a 30% improvement in overall productivity. Unfortunately due to the nature of the work environment (projects are passed from research centre to business units) following up on these results is sometimes either very difficult or impossible. The situation where a project is made multiple times with different technologies is rarely, if ever, encountered in industry.

We concentrate here on a unique scenario in industry where two teams working from the same requirements were to work independently: Team One using ‘agile’ methods (5 persons) and Team Two utilising ‘formal’ methods (3 persons). Use of the pronoun ‘we’ invariably refers to Team Two. The workflow followed the route as depicted in figure 1.

3.1 Requirements and Initial Modelling

The requirements of the system - a distributed blackboard system for sharing semantic web information running on mobile devices - were specified by both teams with the customer being present. This process lasted for approximately two months until a ‘final’ requirements document was produced. This document contained not a formal list of features required but a mixture of model fragments, informal description, implementation ideas etc. From this point both teams split with the aim of producing a working prototype within approximately six months.

In order to formalise and understand the concepts, their relationships and the behaviour of the system we undertook an OO analysis [16] of the requirements producing a model expressed using the

FIGURE 1: Process Flow

UML class notation. Animations of use cases and scenarios were made by hand using pictograms to denote individual objects (conforming to the class model) to explain certain behaviors and configurations with the customer and through this confidence with the customer about our ideas, aims and interpretation increased.

3.2 Specification with Alloy

Given the difficulty of writing behavioural specifications with the UML (its state machine semantics are poorly defined and various tools implement the structure in different ways) and the lack of support for OCL we chose Alloy [8] to express the specification for its ease of specification, relative closeness to OO/UML [7, 3] and its animation capabilities [13]. The specification was written by hand rather than tool generated due to the difficulty of reliably exporting models from UML tools and for the exercise of learning Alloy.

The act of constructing a model in a formal language such as Alloy enforces a *discipline of thinking*. Every construct has to be accurately described which then can be checked for its correctness through - in this case - animation [13]. An example of such an animation is shown in figure 2 where we show an example trace across three points in time (2 events) of facts being placed in buffers and then being moved to an information store (space).

The models generated by Alloy were similar enough to the hand-crafted object diagrams from the initial UML models that some degree of regression/refinement checking against the earlier UML animations could be made. We also start to get a trace of how the requirements were developing through the act of modelling and precisely which parts of the initial requirements document were actually relevant to the overall task in hand. Because of the preciseness of the specification we have a *meaningful feedback loop* to the requirements and relevant and focused discussion about requested features and changes to the system can be made.

FIGURE 2: Animation Trace of Sending a Fact

Discussing the results of these simulations with the customer forces both parties to concentrate on specific issues such as the precise behaviour of specific operations or how much locking is required due to the concurrency in the system as a whole.

An interesting point is that the way the Alloy model was written - to support the notion of traces over time - meant that the events in our models became first-class elements in the Alloy. This allowed us to explore the static hierarchy between the operations and then precisely control the frame-conditions[4] for certain classes of events and more accurately decide upon the scope and responsibility of each of the events.

Focusing on the properties of the system proved either too difficult or too distracting as many details of the system were undefined at this point. During the course of the specification work properties emerge through the increasing understanding of what is being constructed. Simple properties such as that between the Nodes, Sessions and Spaces in the system or the disjointedness of the contents of the buffers between Sessions and the Store became ‘known’ and understood.

As confidence in the specification grew, larger and more complex simulations could be attempted which themselves then added to the set of properties that were required. Particular in this case were that the properties became more focused on the behaviour of the system over a period of time and of the concurrency aspects.

One area where we did have problems with Alloy is when trying to specify detailed control structures (such as the operation of a scheduler in part of the system). In these scenarios languages such as B/EventB appear more natural than Alloy to write the specification and a more verification/refinement based approach more suitable. This may be because the scope of the problem here was much better defined and the properties better understood. More work is required on understanding the use and integration of heterogeneous specification languages in a single specification however.

3.3 Towards Implementation

Once the initial specification had been produced and the process of validation (with the customer) was well underway, implementation proceeded somewhat in parallel with the specification. The python programming language was chosen because of its portability², ease for rapid development and that it is uses on object oriented structures. This latter fact makes the move from UML classes and Alloy signatures fairly straightforward.

The reasons for implementing in parallel with the specification were primarily so that we could construct the prototype as fast as possible and use this to understand the specification more. The implementation forces certain choices, particularly regarding the internal structure of the concepts presented in the specification and the organisation and responsibilities of the events.

²Symbian, Linux, Windows etc

These additional details were then reflected back into the specification and the validation tests rerun.

Changes to the implementation because of changes to the requirements and specification were minimal because of the additional work made in understanding the requirements.

Overall the speed at which the implementation can be constructed is significantly faster than working from an informal specification. The code size tends to be smaller and the scope of the operations more focused. Our code size is little over 850 lines of python for all the basic required functionality. Debugging is made easier because of the inclusion of assertions for preconditions and postconditions enabling tracing of error conditions.

4 ‘PROJECT ONE’

It would be ungrateful to state that there was no discussion between the two project groups but certain differences were apparent.

The need for a prototype - a prototype of *something* - based on informal requirements resulted in a prototype that did not accurately reflect the needs and wishes of the customer. This became very apparent when certain features of the system were requested of the prototype but could not either be demonstrated or would have required much refactoring of the prototype to make that feature unimplementable given the current framework.

Discussions between the project teams was severely hampered by the lack of precision in the terminologies used. Because of the effort placed in modelling and specification *we* could unambiguously describe concepts and make very precise discussion about additional features, behaviour etc. In contrast, without this dictionary and definitions, discussion within the team one was often marred by confusion about terms and more critically levels of abstraction. It was very apparent that some terms were being used not only in different ways but to simultaneously relate to concepts from the requirements to implementation. Confusion also arises when terms are borrowed from different areas of computing.

Given the above situation team one rapidly degenerated into a number of subgroups working on different aspects of the system in parallel without meaningful communication. It also was clear that particular aspects of the system were becoming critical in the sense that large amount of development effort was spent upon these parts to the detriment of the system as a whole; one of these was the issue of distribution of the information storage and queries over multiple devices. This effort detracted from the purpose of the system as a whole and was without solution in the end; it never became clear as to actually why this kind of distribution would have been required³.

After production of the prototype a large amount of redesign work was obviously necessary, the prototype abandoned and a further round of specification work with a smaller group of two persons was initiated. The prototype did however serve a purpose in that it was understood by all that the requirements were misunderstood and that there was a need for a more rigorous (if not formal) approach. At this time more collaboration work was being made with our team regarding the interpretation of the requirements and how it should be modelled.

5 CONSOLIDATION

It was decided that due to the unequal progress of both teams that the teams share results and a single, unified specification be produced. This decision was made in order to ensure a single, agreed architecture and set of features be decided and published. From there a ‘final’ implementation could be produced.

The status of the two teams can be summarized as:

---

³The reasons for particular kinds of distribution are still not clear and were never explored initially. No prototyping of the distribution was actually made.

- Team One
  - Prototype (non conformant)
  - PowerPoint ‘models’ of new requirements (from second round of specification)
- Team Two
  - Specifications in Alloy and UML
  - Test suite
  - Implementation
  - Sample Applications

Initial work involved coordinating the different terminologies and architectures and making sure that all understood what was being discussed. It was obvious that the notions of block diagrams, class diagrams, object diagrams were being confused and that there was no attempt to understand the specification in Alloy by the wider group.

Once animations were generated from Alloy with detailed (and interactive) explanation the specification proved useful so that effects of changes in the behaviour and terminology could be visualised and subsequently agreed upon. It however was still necessary to formulate some of the descriptions in terms of block diagrams and telecoms-style protocols; this latter effort sometimes had the effect of confusing the specification with the implementation. It was apparent that changes in modelling language and relating different languages together is actually difficult for some persons. Much of the difficulties arise from not knowing the purpose nor meaning of a specification [11] versus an implementation.

Ironically most of the work here was either political in that terminology was shared between the teams (with more of the focus on the team two’s terminology) or was focused on the syntactical issues of the models and explaining the meaning of notation.

There was a large amount of work made in simplifying the team one’s specification to a point where superfluous features were removed; this was mainly done either by moving that functionality to the application level or by demonstrating (and this was the more important revelation for team one) that much of the functionality and features were conveniences for the four basic operations of a typical blackboard system: `assert(fact)`, `retract(fact)`, `query(expression):Set fact`, `subscribe(expression)`.

While the formal descriptions (and implementation) produced by team two ultimately became the single, unified specification it was still necessary to rework a number of the models and describe the whole path from concept to implementation using much finer grained modelling steps (see figure 3). The primary difficulty being to ensure that the consistency and validity of the specification was not compromised. In the cases of the UML Model and Alloy Specification steps there are up to 5 models within these steps. The simplified domain model (in UML) was produced to explain the domain model without complicating features such as inheritance hierarchies, showing the internal composition of some artifacts and attributes etc.

We ended with a series of models which form a very neat refinement chain (cf: B-Method [1]) but in order to find this particular series of models took many iterations and reworking of the models often with different abstraction levels being worked upon in parallel; such as the implementation driving the specification(s) in some cases. At this point the specifications were team two’s with some syntactical changes and very few structural modifications.

6 CONCLUSIONS

Our use of formal methods can only be described as lightweight [2] and in many cases does not hold to the ideals of formal methods practice. We counter this with saying that we are not formal methods practitioners with processes that are easily amenable to large scale, mandated usage of formal methods. We are very conservative when it comes to the use of new and different methods -

FIGURE 3: Model Structure

although agile methods have taken a foothold due to individuals’ enjoyment of hacking something together (see team one’s prototype for example).

Assessment of the effect of using formal methods is always going to be problematical [5]: rarely are detailed records kept (at least in the research community here) - defects are anyway confidential and even more rarely is the chance to run project with two parallel teams to provide comparison. It is hoped that it is obvious from our informally presented results here that effort in at least formal thinking had a larger payoff than even we expected. This payoff is not only found in the shortened development time but in the testing and debugging process applied to the implementation - the bugs we are finding are primarily to do with localised programming errors rather than larger faults involving the overall architecture.

Of the other issues exposed between the two teams are summarised:

- The lack of knowledge about abstraction
- The overriding need for implementation (of something) versus the need for background research and thought before implementation
- The concentration on small implementation details even at requirements level
- The concentration of syntactical issues and confusion of terms between different domains. (eg: smartspace vs SmartSpace(TM), session vs connection vs ISO 7-layer definitions)
- The usefulness of animation/simulation in understanding the design
- Understanding the completeness of the models.
- Tendency to discover the atomic behaviours
- The lack of usefulness of the UML beyond static structure
- Discovery of the properties of the system during specification and not before

Of these, the first is either attributed to education or to the belief that a single strictly applied method solves everything and this has an effect on the second and third issues which is that some methods don’t enforce a strict separation of abstraction level nor of concerns (architecture is not the same as domain is not the same as a design etc). The effort spent on ensuring consistency,

correctness and acceptance without prejudice to other usages of the terms used is considerable and in many cases is the biggest problem encountered in the development of any system - use of formal methods ensures much focus on this aspect and provides a way of clearly defining terms and their place in the development of the system.

Being able to animate (or simulate) [6] a specification is especially useful⁴ and even more so when the results of the simulation are presentable to the customer. Obtaining customer support and assisting the customer with their understanding of the system and confidence (and progress) in the development work is critical to the overall success of any project. Being able to concentrate on a few key issues early in the development also of course has its benefits later.

It is important to note that the amount of specification necessary in order to make a useful validation or verification is not high - certainly it is a lot less than one sees in a typical ‘UML specification’. It is not necessary either to make complete specifications: our primary model is extremely small (8 Alloy signatures) with further models to express certain less understood parts as necessary, for example the scheduling and creation mechanisms. We advocate somewhat not specifying those parts which are generally agreed up on but focusing on those where ambiguity or complexity exists. As an aside to this, there is a focus on making specifications/models/systems complex for the sake of complexity - simplicity and elegance often been mistaken as too little work made. Specifications must be kept simple and concise.

Writing and being able to animate a specification does provide much needed focus on the behaviour and discovering the basic, atomic behaviours of the system versus the larger convenience functionality. Effectively, formal specification *simplifies* the design because the focus is on what is required and composing larger features out of these simpler behaviours. Using formal methods you become very critical and careful about what you describe.

The failure of UML to sufficiently express behavioural properties is known and the over reliance on state diagrams without semantics (or specific tool based semantics) by some methods is a major problem. Beyond expressing the static (class) structure and very general behavioural aspects with responsibility-based design, the UML provides little to offer. UML’s state diagram and static structure diagrams (mistaken for SDL-like block diagrams) are not sufficient. There are very few tools that are capable of analysing models written using the UML - even static analysis is not generally supported except in some specific cases - again this is an area where formal methods (due to their very nature) have an advantage.

One of the problems with formal methods is the focus on property based design and refinement. It is the case that we do not know what properties we wish our system to have until sometimes significant design work has been made. Properties emerge from the design and are often implicitly expressed in the behavioural specification. Extraction of these properties is difficult and does tend to divert focus away from development - admittedly we are lacking good methodological practice in this area and this is something that needs to be worked upon.

The myths of formal methods [9] are unfortunately deeply ingrained, despite languages such as UML exposing the need for more formality. Particularly important for the uptake of formal methods is the need to understand that integrating formal methods within existing processes does not require major change to those processes nor to the working practices; they do require more thinking and that might be hard. We only exploited formal specification in one development step but applied the principles and thinking throughout.

Overall the use of formal methods has been taken more seriously after this work described here - mainly because of the realisation of the amount of wasted effort due to miscommunication and the imprecise nature of what has been required of the system. The difficulty is how much formality to use and how to balance that across the whole development cycle and provide enough education and support so that ‘regular’ (rather than mandated) use is taken into practice.

---

⁴prior experience with ProB for example

7 ACKNOWLEDGMENTS

Daniel Jackson and Felix Chang for assistance with the Alloy language and tool. Jukka Honkola and Jürgen Ziegler at Nokia Research with the SmartSpace project itself.

REFERENCES

- [1] J-R Abrial. *The B-Book - Assigning programs to Meanings*. Cambridge University Press, 1995. 0-521-49619-5.
- [2] Sten Agerholm and Peter Gorm Larsen. *A Lightweight Approach to Formal Methods*. In *International Workshop on Current Trends in Applied Formal Methods*. Boppard, Germany. October 1998.
- [3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. *UML2Alloy: A Challenging Model Transformation*. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 2007.
- [4] Alex Borgida, John Mylopoulos, and Raymond Reiter. *On the Frame Problem in Procedure Specifications*. IEEE Transactions on Software Engineering, 21(10):785–797, October 1995.
- [5] Jonathan P. Bowen and Michael G. Hinchey. *The Use of Industrial-Strength Formal Methods*. In *Proceedings of COMPSAC*. 1997.
- [6] Norbert E. Fuchs. *Specifications Are (Preferably) Executable*. Software Engineering Journal, September 1992.
- [7] G. Georg, J. Bieman, and R. France. *Using Alloy and UML/OCL to specify run-time configuration management: a case study*. In *Proc. Workshop on Practical UML-Based Rigorous Development Methods — Countering or Integrating the eXtremists*. October 2001.
- [8] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, April 2006. 0-262-10114-9.
- [9] M.G. Hinchey J.P. Bowen. *Seven more myths of formal methods*. Software, 12(4):34–41, July 1995.
- [10] Jean Mermet, editor. *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, 2004.
- [11] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [12] Ian Oliver. *A Demonstration of Specifying and Synthesising Hardware using B and Bluespec*. In *Forum on Design Languages FDL’06*. 2006.
- [13] Ian Oliver. *Design and Validation of a Distributed Computation Environment for Mobile Devices*. In *Proceedings of European Simulation Multiconference ESM 2007, 22-24 October, Malta*. 2007.
- [14] Ian Oliver. *Experiences in Using B and UML in Industrial Development*. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 248–251. Springer, 2007. ISBN 3-540-68760-2.
- [15] J. Rosenberg. *Some Misconceptions About Lines of Code*. metrics, 00:137, 1997.
- [16] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.