Analytical Software Design Case MagLev Stage Software Project for Philips Applied Technologies

Guy H. Broadfoot, George Kielty

Product innovation, quality and time to market are key elements in the battle to achieve and sustain competitive advantage. For a growing number of businesses, this means software development. Software is now an essential component embedded in an ever increasing array of products. It has become an important means of realising product innovation and is a key determinant of both product quality and time-to-market. For many businesses, software has become business-critical *and software development is a* strategic *business activity. At the same time, software development continues to suffer from poor predictability. Existing development methods appear to have reached a quality ceiling that incremental improvements in process and technology are unlikely to breach. To break through this ceiling, a different approach is needed. In this paper, we describe how Verum applied Analytical Software Design (ASD), a new approach that applies software engineering mathematics to industrial software development, to develop the software controlling an advanced mechatronics subsystem being developed by Philips Applied Technologies in the Netherlands.*

"*ASD is a formal method that is informal enough to be applied in practice.*" G.P.M. Haagh
Senior Software Architect, Philips Applied Technologies

Introduction

Product innovation, quality and time to market are key elements in the battle to achieve and sustain competitive advantage. For a growing number of businesses, this means software development. Software is now an essential component embedded in an ever increasing array of products. It has become an important means of realising product innovation and is a key determinant of both product quality and time-to-market. For many businesses, software has become *business critical* and software development is a *strategic* business activity.

Today, software development continues to suffer from poor predictability. Business managers need reliable answers to the questions "When will it be ready?", "What will it cost?" and "How well will it work?" These are the very questions that software developers are least able to answer.

In recent years, in an attempt to overcome these problems, companies have invested heavily in software development process improvements, technology, infrastructure and training. In spite of this, the rapidly increasing complexity and amount of software still presents a serious challenge. According to studies, 40% - 50% of total development costs are typically lost on avoidable rework [4]; 15% - 25% of software defects are delivered to customers [4]; in 2002, software failures cost the U.S. economy an estimated $59.5 billion [7].

Existing development methods appear to have reached a quality ceiling that incremental improvements in process and technology are unlikely to breach. The amount and complexity of in-product, embedded software is growing exponentially; according to the SEI, productivity in the most successful organisations is improving linearly, at best. The challenges many businesses experience developing embedded software and being able to guarantee its quality and correct functioning is a testament to this growing capability gap and inability of current testing-centric software development practices to bridge it. To bridge this divide, a different, more formal, approach is needed.

The Case: The MagLev Stage

Philips Applied Technologies[1] is part of the Dutch Royal Philips Electronics group of companies. Part of its mission is to develop innovative solutions for advanced manufacturing. It is a leading developer of industrial vision systems and high precision mechatronic systems.

One of its latest products is a highly accurate, high performance "stage" known as the MagLev Stage (see figure 1). This is a subsystem designed to be incorporated in a variety of industrial systems that require high speed, highly accurate material positioning, especially in high vacuum situations, for applications such as e-beam inspection and laser cutting. It uses advanced magnetic levitation servos and achieves a repeatable sub-micron scanning accuracy (130 Nanometers or better).

An essential part of the MagLev Stage is the control software embedded in it. This software coordinates the actions of the multi-axis controllers and provides an Application Program Interface (API) to customer developed domain specific application software.

Early in 2004, a "proof of concept" version of the control software was developed to enable the mechanical and electronic sub-systems to be developed and tested. This development took about 10 weeks in order to achieve a level of "good weather" functionality useful for the product development. Although suitable for its purpose as a "proof of concept" prototype, this software was not of the industrial quality levels considered suitable for the final product. Since its initial development, defects have emerged at regular intervals, including software crashes and race conditions. By January 2005, more than 20 versions were present in the software configuration management system, each representing a major release to fix multiple errors.

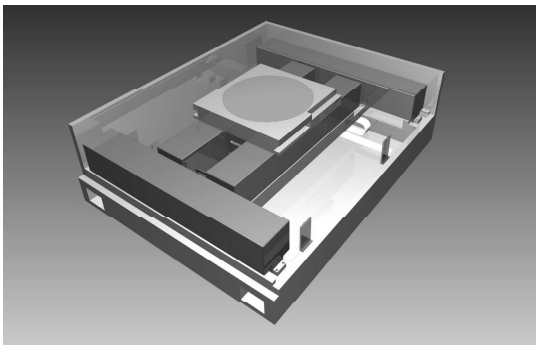


Figure 1: MagLev Stage

In September 2004, it was decided that a new version of the controller software would have to be developed in order to achieve required quality standards. Given the complex nature of the software, Philips Applied Technologies and Verum together applied Analytical Software Design (ASD) techniques in which the complete software design is modelled mathmatically and model-checked for correctness before implementation starts. After verifying the design mathematically, Verum's ASD techniques enabled 90% of the new code to be generated in C++ automatically from the verified design specifications.

Technical Details

The MagLev Stage consists of two *substages* called the *Intermediate Substage* and the *Carrier Substage* respectively. Each substage is controlled by its own dedicated multi-axis controller. The multi-axis controllers are existing subsystems previously developed by Philips Applied Technologies and used in other products.

[1] `www.apptech.philips.com`

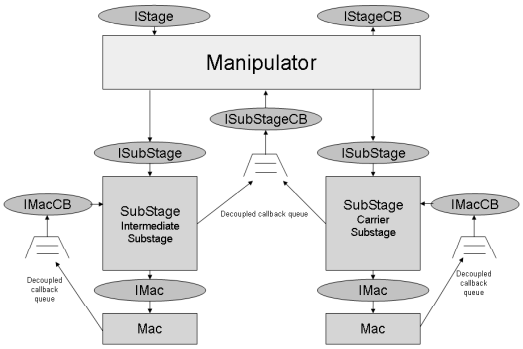


Figure 2: MagLev Software Overview

Figure 2 shows the overall organisation of the software into two major components, namely the *Manipulator* and the *SubStage*. In the diagram, software components are depicted by rectangles; major interfaces are depicted by the labelled ovals.

The SubStage component is responsible for controlling a single substage via its dedicated multi-axis controller known as a *MAC*. The diagram shows two instances of the SubStage component, one controlling the Intermediate Substage and one controlling the Carrier Substage. The Manipulator component coordinates and controls the two substage components. All actions that are specific to a single MAC are implemented in the SubStage component; all actions requiring coordination between the substages, such as most movements and all exception and error handling, are implemented in the Manipulator component. As is usual in such systems, the "good weather" behaviour, although complex, is relatively straight forward; the majority of the program logic is required to handle all the various exception conditions that can occur.

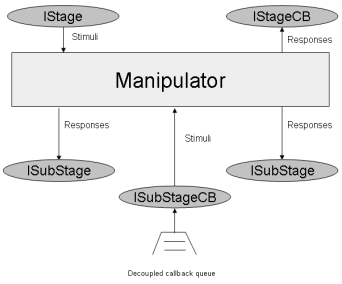


Figure 3: Manipulator Context Diagram

The Manipulator component implements the API to be used by the customer-developed domain specific application software. It must be "thread safe"; that is, able to support multi-threaded client applications while handling asynchronous call-back events from the two substages. For efficiency reasons, the Philips Applied Technologies senior architect required the execution architecture to minimise context switching with execution remaining under the caller's thread context as long as possible. Figure 3 is a context diagram of the Manipulator component. This shows the Manipulator as implementing its client API (IStage), sending asynchronous call-back notifications to the client application (IStageCB), using the SubStage API (ISubStage) and receiving notifications from the two substages via the ISubStageCB call-back interfaces. All the ISubStageCB events are routed to the Manipulator via a queue and are executed under the context of a separate deferred procedure call (DPC) server thread.

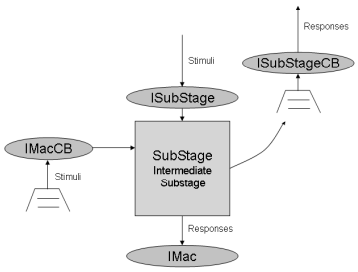


Figure 4: SubStage Context Diagram

Figure 4 shows the context of each SubStage component instance. It implements an API used by the Manipulator (ISubStage and ISubStageCB) and uses the interfaces provided by the MAC (IMac and IMacCB). The SubStage receives API calls from the Manipulator and asynchronous event notifications from the MAC via a queue and a separate DPC server thread. The ISubStage interface realises a high level abstraction of the physical substage, with high level moves implemented in terms of the primitive move operations provided at the MAC interface.

An Overview of ASD

Analytical Software design is based on two design principles:

- Business critical software must be based on designs that are verified before implementation starts;
- Software Architects and Designers must use designs and architectures that can be verified using currently available tools and techniques.

With one exception, all branches of engineering routinely apply their specific branches of mathematics to specification and design. Modelling a design is cheaper than building a prototype and testing it. It is also more certain; testing is by definition an exercise in sampling and can never provide complete coverage or certainty of correctness. An architect charged with designing an earthquake-proof building does not build it and wait for an earthquake to test it! Instead, the design is mathematically modelled and subjected to rigorous mathematical analysis.

The one exception is software engineering. Apart from those few domains (mostly safety critical) where formal design and verification methods are mandated, mathematics are not routinely applied to software specification or design. Instead, reliance is placed on informal inspection-based methods and testing. As a consequence, defects injected early in the life-cycle during specification and design activities are frequently not detected and removed until after implementation is substantially complete and integration testing begins. This is the most expensive time to correct defects and occurs at a point in the life-cycle that results in the maximum impact on time to market. For many kinds of errors, such as race conditions and deadlocks, this is also the least certain way to find them.

Analytical Software Design[2] combines the practical application of software engineering mathematics and modelling with specification methods that avoid difficult mathematical notations and remain understandable to all project stakeholders. In addition, it uses statistical techniques for software component testing and advanced code generation techniques. From a single set of design specifications, the necessary mathematical models, program code and statistical test cases are generated automatically.

ASD uses the Sequence-based Specification method [8, 9] to specify functional performance requirements and designs as black box functions. These specifications are traceable to the original requirements specifications and remain completely accessible to the critical project stakeholders. This allows them to play a key role in verifying the ASD specifications and retain control over them. At the same time, ASD specifications provide the degree of rigour and precision necessary for mathematical analysis.

ASD applies the Box Structured Development Method [5, 6] following the principles of stepwise refinement to transform the black box design specifications into state box specifications from which programming is based.

The ASD Model Generator generates mathematical models from the black box and state box specifications and designs automatically. These models are generated in the process algebra CSP [3, 10] and can be formally analysed and verified using the model checker FDR [1]. For example, we can use the model checker to verify (i) whether a design satisfies its functional requirements; (ii) whether the state box coding specification is behaviourally equivalent to the black box design; and (iii) whether the design uses other components according to their external functional specifications.

The ASD Code Generator can generate significant amounts of code automatically from the ASD specifications. The principle advantage of code generation is correctness; the code is generated automatically from the ASD specifications that have already been formally verified. Code generation may not be applicable to every project but in those cases where it is, significant development efficiency gains can be achieved.

ASD uses Statistical Testing methods based on Usage Models derived directly from the ASD Specifications to test software components against the verified designs. The ASD Test Case Generator and Analyser generates large numbers of self-running test cases and analyses their results.

Figure 5 shows the main elements of ASD. The functional specification is analysed using the Sequence-based Specification method extended to enable nondterminism to be captured. This enables the externally visible behaviour of the system to be specified with precision and guarantees completeness.

[2] Patent applied for under patent application number GB 0410047.5

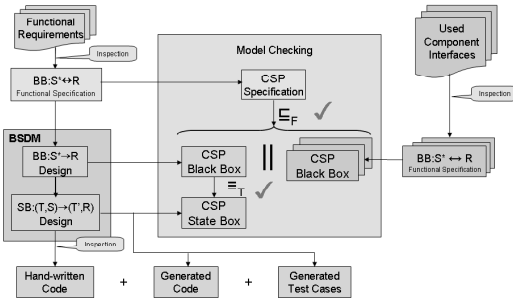


Figure 5: ASD Overview

Because ASD specifications avoid difficult mathematical notations and are fully traceable to the original specifications, they can be validated by inspection with project stakeholders. Next, the design is specified using Sequence-based Specification. This still remains a creative, inventive design activity requiring skill and experience combined with domain knowledge. With ASD, however, the design is typically captured with much more precision than is usual with conventional development methods, raising many issues early in the life cycle and resolving them before implementation has started.

The ASD model Generator is used to generate process algebra models of both the specification and the design so that the design can be verified for compliance with the specification. In most cases, a design cannot be verified in isolation; it depends on its execution environment and the components it uses for its complete behaviour. In ASD, used component interfaces are specified using Sequence-based Specification, the corresponding mathematical models are generated using the ASD Model Generator and these models, combined with those of the design, are verified for compliance with the specification. For CSP models, this verification is done mathematically using the model checker FDR. Errors detected during the verification are corrected in the design specification, new CSP models are generated and the verification is repeated. (This is typically a very rapid cycle.)

When the design has been verified, the ASD Code Generator is used to generate program source code in C++ or C or other similar languages. The percentage of the total code that can be generated this way varies from project to project. Experience suggests this is typically between 70% and 90%, but it can be lower on some projects.

Finally, from the same set of design specifications, large numbers of statisitically selected test case can be generated in the form of self running tests and the results analysed by the ASD Test Analyser.

Applying ASD to the MagLev Stage Development

The design team consisted of a senior software architect and software engineer from Philips Applied Technologies and two employees from Verum. The goals of the project were:

1. To re-develop the MagLev Stage control software to industrial quality standards as quickly as possible;
2. For Philips Applied Technologies to gain practical experience of applying ASD in practice with a view to assessing its applicability to other typical software developments carried out within Philips Applied Technology.

The work proceded as follows: firstly, an ASD specification of the MAC interface (IMac and IMacCB) was made based on existing specifications and the expert knowledge of the senior architect. This black box function was plotted in the form of a state transition diagram and reviewed by the team.

Next, an ASD specification of the client application API (IStage and IStageCB) was made based on the existing implementation and with frequent references to the existing code. The process of making the ASD specification raised a significant number of specification issues, most of which were resolved by the senior architect based on his extensive domain knowledge and experience gained in developing the original “proof of concept” prototype.

The architecture was then developed, partitioning the major functions of the control software between the Manipulator component and the two instances of the SubStage component; an ASD specification of the SubStage interfaces (ISubStage, ISubStageCB) was made, reflecting the first “guess” at the SubStage abstraction.

The first major design task was the design of the Manipulator. This was specified using Sequence-based Specification. As the design evolved, the precision of the ASD interface specifications of

the client interface and the SubStage interface was extremely beneficial. As the design neared completion, the ASD Model Generator was used to generate the CSP models of the client interfaces (IStage, IStageCB), the SubStage interfaces (ISubStage, ISubStageCB) and the Manipulator design. The parallel composition of the Manipulator design model plus two instances of the SubStage Interfaces (one for the Intermediate SubStage and one for the Carrier SubStage) were verified against the client interface model using the model checker. This was done after first verifying with the model checker that the design was free from divergence, internal inconsistencies and deadlocks.

During this process, many design and some specification errors were discovered by the model checker. As they were discovered, the appropriate ASD specification was updated to correct the error, new mathematical models generated and the verification continued. This cycle occurs quite rapidly, finding and fixing several errors per hour. This differs significantly form conventional, testing-based development method. Unlike conventional testing:

- All of this is done without having written any program code or executing any test cases.
- This form of verification is based on mathematical proof and is total. It is equivalent to 100% *execution path* coverage, something unacheivable by testing.
- This form of verification is particularly good at uncovering dynamic behavioural errors such as deadlocks, race conditions and design behaviour that violates interfaces specifications. Such errors are extremely difficult to detect and diagnose using conventional testing because their nondterministic nature makes them to reproduce and repair.
- This verification is done before investing in implementation, at the most economic point in the life-cycle.

When the Manipulator design was completed and verified, work began on the SubStage design. Again, this was specified using the Sequence-based Specification Method. In this case, the implemented interface is the SubStage interface (ISubStage, ISubStageCB) and the used interface is the MAC interface (IMac, IMacCB). As the design neared completion, the CSP models were generated automatically using the ASD Model Generator and FDR was used to check the SubStage design plus the MAC interface against the SubStage interface. The SubStage interface model was the same one used to verify the Manipulator design.

During the SubStage design, it proved impossible to implement the SubStage interface exactly as it had been specified and it was necessary to change it. This involved changing the ASD SubStage interface specification, regenerating its mathematical model and then verifying the Manipulator design against the changed SubStage interface to assess the impact of the changes on the Manipulator design. Where necessary, the Manipulator design was changed and verified against the modified SubStage interface specification. This enabled the impact of ISubStage design alternatives on the Manipulator to be assessed quickly and provided additional input for making often difficult technical choices.

When both the Manipulator and SubStage designs were completed and verified by model checking, the C++ code of both the Manipulator and th SubStage was generated using the ASD Code Generator.

Results

The ASD specification of the MAC interfaces took about 1 week; the specification of the client API interface and the SubStage interface took about the same time. The MAC interface specification has 493 transition rules and 12 canonical sequences, the longest of which is 5 stimuli long. The ASD specification of the client API has 345 transition rules and 13 canonical sequences. The SubStage interface has 370 transition rules and 13 canonical sequences.

The ASD design and verification of the Manipulator took about 4 weeks to complete. The design was extremely complex due to the complex behaviour of the MAC as this was still visible at the SubStage interface plus the event driven and concurrent nature of the behaviour. Due to its complexity, the Manipulator design was hierachically decomposed into a top level design together with 3 significant lower level sub-designs. In total, the design has 1,700 transition rules and 28 canonical sequences. This hierarchical design structure was carried through into the generated mathematical models and the generated C++ code, providing full traceability between these different views.

During the design verification of the manipulator, about 200 errors were detected in the model checking phase. Most of these fell into one of two categories: i) internal inconsistencies where the design violated the interface specifications of the used components or was unable to react correctly to notifications arriving asynchronously from the used interfaces; ii) race conditions that were particularly difficult due to a) the number of unstable states in the SubStage specification arising from the nature of the underlying MAC behaviour and b) the loose coupling between the Manipulator and the SubStage introduced by the event notification queue mechanism. The order of verification was as follows: i) to verify freedom from race conditions, divergence and deadlocks; ii) to verify complicance with the used SubStage interface specifications; iii) to verify compliance with the client API specifications, the interface implemented by the design.

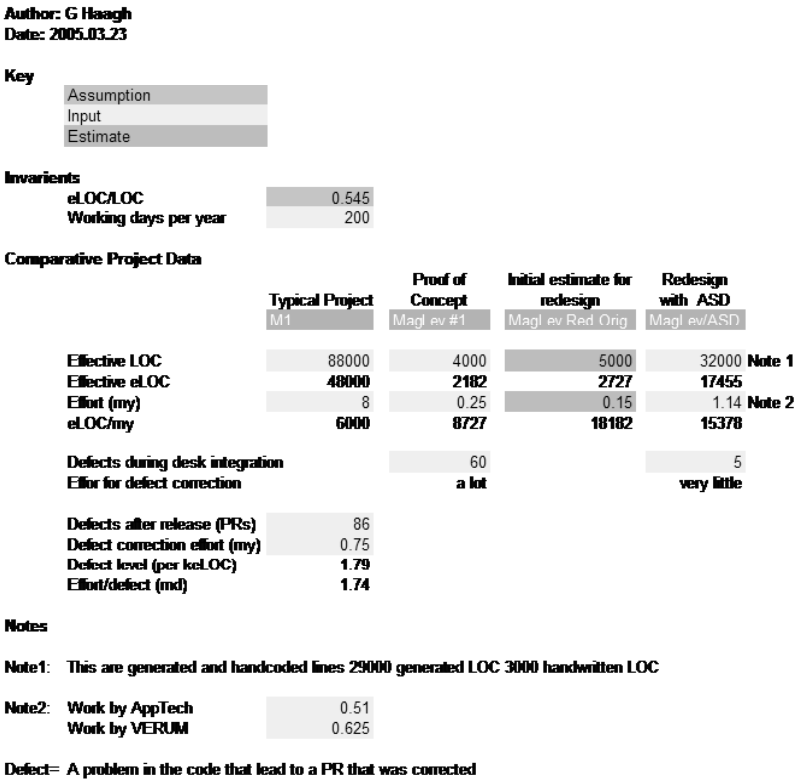
The ASD design and verification of the SubStage took about 4 weeks to complete. Due to its complexity, the design was hierarchically decomposed into a top level design plus 5 lower level subdesigns. In total, the design has 4,700 transition rules and a total of 84 canonical sequences. The order of verification was the same as described above. Again, the number of unstable states in the behaviour of the MAC together with the decoupling caused by the event notification queue resulted in a very complex design with many possibilities for race conditions and other unexpected behaviour. During the verification, in the order of 200 errors were detected by model checking and removed.

After all designs were completed and mathematically verified, the C++ code was generated. The generated code is structured according to the well known State Pattern [2] and was tailored to meet the code architecture required by Philips Applied Technologies. This is a normal part of the ASD code generation process; experience shows that "standard" code generators are frequently too inflexible in the style and strucure of the code they generate. Every project and development environment has specific requirements for the generated code to ensure that it properly integrates with the rest of the code base and the run-time platform. In this project, the run-time platform was VxWorks. In total 17,000 executable lines of code were generated, representing more than 90% of the code. The hand written code was either concerned with domain specific issues such as coordinate transformations or "glue" code interfacing the software to the rest of the run-time environment. Although the final run-time platform was VxWorks, component testing was done by Verum under Windows XP. Testing in the final VxWorks environment is being performed by Philips Applied Technologies.

The comparative results of this project are shown in figure 6. Philips Applied Technologies calculates that its code production rate for a typical software development, including design, specification, coding and testing effort, is about 6,000 executable lines of code per man year. The original MagLev "proof of concept" software was produced at a rate of 8,727 executable lines of code per man year. Desktop integration testing of this software, using simulated hardware, found 60 defects, resulting in a large - but undocumented - amount of rework.

In 2004, when considering the redesign of the MagLev software using traditional methods, the Applied Technologies design team expected to produce approximately 5000 lines of code in 6 man weeks: a productivity equivalent to 18,000 executable lines of code per man year. Based on their experience with the "proof of concept" version, they also obviously expected an increase in the quality of the end result.

Ultimately the redesign of the MagLev software was performed together with employees of Verum using ASD. The result was the production of 17,000 executable lines of code in 45 man weeks of effort, including all specification, design, design verification, coding and desktop integration testing effort. This equates to a production rate of 15,000 executable lines of code per man year. The stated effort captures the contributions from both Applied Technologies design staff and Verum's employees. It also captures the learning curve needed by both parties; Verum's employees to learn about the MagLev application and Applied Technologies engineers to learn how to work with ASD. Much of this learning curve would not be required for future projects. Furthermore, application of ASD to the design of this system resulted in the discovery of about 400 defects during design verification. The average effort to find and fix each defect was approximately 1 man hour per defect. Consequentially the software delivered to Philips Applied Technologies has a very low defect rate. During desktop integration



Author: G Haagh
Date: 2005.03.23

Key

Assumption
Input
Estimate

Invariants

eLOC/LOC	0.545
Working days per year	200

Comparative Project Data

	Typical Project M1	Proof of Concept MagLev #1	Initial estimate for redesign MagLev Red Orig	Redesign with ASD MagLev/ASD	
Effective LOC	88000	4000	5000	32000	**Note 1**
Effective eLOC	**48000**	**2182**	**2727**	**17455**	
Effort (my)	8	0.25	0.15	1.14	**Note 2**
eLOC/my	**6000**	**8727**	**18182**	**15378**	
Defects during desk integration		60		5	
Effor for defect correction		**a lot**		**very little**	
Defects after release (PRs)	86				
Defect correction effort (my)	0.75				
Defect level (per keLOC)	**1.79**				
Effort/defect (md)	**1.74**				

Notes

Note1: **This are generated and handcoded lines 29000 generated LOC 3000 handwritten LOC**

Note2:	**Work by AppTech**	0.51
	Work by VERUM	0.625

Defect= A problem in the code that lead to a PR that was corrected

Figure 6: Comparitive Results

testing with simulated hardware, only 5 errors were found and very little effort was required to correct these errors.

Of course, the number of executable lines of code is a poor indicator of the complexity of a piece of software. Comparison of hand versus automatic code generation techniques leads to a discussion of the relative efficiency of each technique, with no obvious conclusions except that automatically generated code leads to far lower error rates. Unfortunately, there are no other common metrics that give an indication of complexity in this case. However, the design team judged the complexity of the MagLev design to be at least twice that estimated at the beginning of the project, even with the experience of having produced a proof of concept version.

As a result, Philips Applied Technologies drew the following conclusions from the application of ASD to the redesign of the MagLev software:

- Overall the use of ASD in the design/code/unit test phases is cost neutral w.r.t. traditional ways of working; that is, the benefits were gained at no extra cost as compared to traditional working methods.
- The number of defects found during desktop integration is reduced by a factor 12
- The perceived quality of the code is MUCH higher (supported by the figures)

Verum's employees also observed that:

- The complexity of the MagLev (re)design problem was much greater than that anticipated by the Applied Technologies design team
- The use of ASD exposed the complexity of the (re)design problem during the earliest moments of the development
- The MagLev software was delivered on time according to original expectations
- The MagLev software was delivered in line with effort estimates, bearing in mind the unexpected complexity of the design problem.

At the time of writing this report, the MagLev software remained to be tested with real hardware and released to real customers. Therefore Applied Tech-

nologies has only measured the effect of ASD on early lifecycle phases and has yet to experience the benefits ASD brings to system testing, release and maintenance.

Conclusions

This project has demonstrated to Philips Applied Technologies:

1. The application of ASD is cost neutral over conventional design methods during the first half of the project lifecycle.
2. The application of ASD results in a factor 12 reduction in defects found during initial integration testing.
3. The application of ASD results in a predictable completion date for the project.
4. ASD specifications are understandable and usable by project stakeholders without knowledge of software engineering mathematics; there is no complex mathematical notation to be learned.
5. ASD enables experienced employees of Verum to work productively together with domain experts in a joint design team in an existing software development environment and with software engineers and architects not specifically trained in the method.
6. ASD is applicable to a wide variety of projects within Philips Applied Technologies.
7. ASD results in designs and implementation of a much higher quality than can be achieved by conventional methods.

The senior architect on this project stated that he has a much higher level of confidence in the quality than he has using conventional methods. He said: *"This is the first formal method informal enough to be applied in practice."*

Acknowledgements

We are grateful to Philips Applied Technologies[3] for allowing us to present the case study and for their cooperation and support when we applied these techniques together to develop control software for the MagLev Stage. We are particularly indebted to G.P.M. Haagh Senior Software Architect and Rutger van Beusekom Software Engineer, both of Philips Applied Technologies, for their cooperation and positive contribution in applying ASD to this development.

References

[1] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 2003. See `http://www.fsel.com`.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] Thomas McGibbon. A business case for software process improvement revised. Technical report, Data & Analysis Center for Software, 1999.

[5] H. D. Mills. Stepwise refinement and verification in box structured systems. *Computer*, 21(6):23–26, 1988.

[6] H. D. Mills, R. C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design*. Academic Press, 1986.

[7] The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology NIST, US Department of Commerce, 2002.

[8] S. J. Prowell and J. H. Poore. Sequence-based software specification of deterministic systems. *Software - Practice and Experience*, 23(3):329–344, 1998.

[9] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions of Software Engineering*, 29(5):417–429, 2003.

[10] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[3] Philips Applied Technologies B.V., Eindhoven, The Netherlands.

Contact Information

Guy H. Broadfoot
guy.broadfoot@verum.com

George Kielty
george.kielty@verum.com

Verum Consultants B.V
Paradijslaan 28-28a
5611 KN, Eindhoven
The Netherlands
Phone +31-40-2359090
Fax +31-40-2359099