The difference between

Unlike other engineering disciplines, software (methodology) does not have enough precision to allow any formal reasoning about the software itself. Consequently, most requirements and design errors ("details") are discovered typically too late, when the system has been built. One improvement is to introduce mathematical methods in such a way, that they scale, are economic to use, and that the key stakeholders remain involved and are still able to validate the specifications based on such mathematical methods. Analytical Software Design meets these requirements.

• *Leon Bouwmeester* •

Precision and detail: two small words that are often used as if they have the same meaning, but which differ greatly. Precision refers to the ability of a measurement to be reproduced consistently; key factors are predictability and exactness. Detail, on the other hand, refers to something small or trivial enough to escape notice; it has a flavour of "not important" about it.

The Eindhoven area is home to several companies that are involved in building high-precision mechatronic equipment. Precision – next to accuracy (degree of closeness of measurements to the actual value) – often affects their core business (in)directly. Therefore, one can assume that all disciplines related to mechanics, optics, physics, and electronics are well coordinated and aligned within these types of companies. The combination of these disciplines is directly related to the overall precision and accuracy of the resulting equipment they build. Any failure in this area is directly visible – sometimes even literally, as explained by the Hubble example.

Hubble telescope

On 24 April 1990, the Hubble telescope was launched into orbit from aboard the Discovery space shuttle. Almost immediately afterwards it became clear that something was wrong. While the pictures taken with Hubble were clearer than those of ground-based telescopes, they were not the pristine images promised. Analysis of these flawed images showed that the problem was caused by the shape of the primary mirror. Although it was probably the most precise mirror ever made, with variations from the prescribed curve of only 10 nm, it was too flat at the edges by about 2.2 µm, which caused severe spherical aberration: light bouncing off the centre of the mirror focuses in a different place than light bouncing off the edges. Figure 1 shows a schematic overview of the internals of the Hubble telescope.

Fortunately, scientists and engineers were dealing with a well-understood optical problem (although in a unique environment). For which they had a solution: a series of

Author

Leon Bouwmeester joined Verum Software Technologies, based in Waalre near Eindhoven, the Netherlands, in 2006, and is currently working as a managing consultant. He advises business and product management of the benefits of Verum's Analytical Software Design (ASD) technology, and is responsible for the support of the customer's architects and senior designers in the application of ASD.

www.verum.com

precision and detail

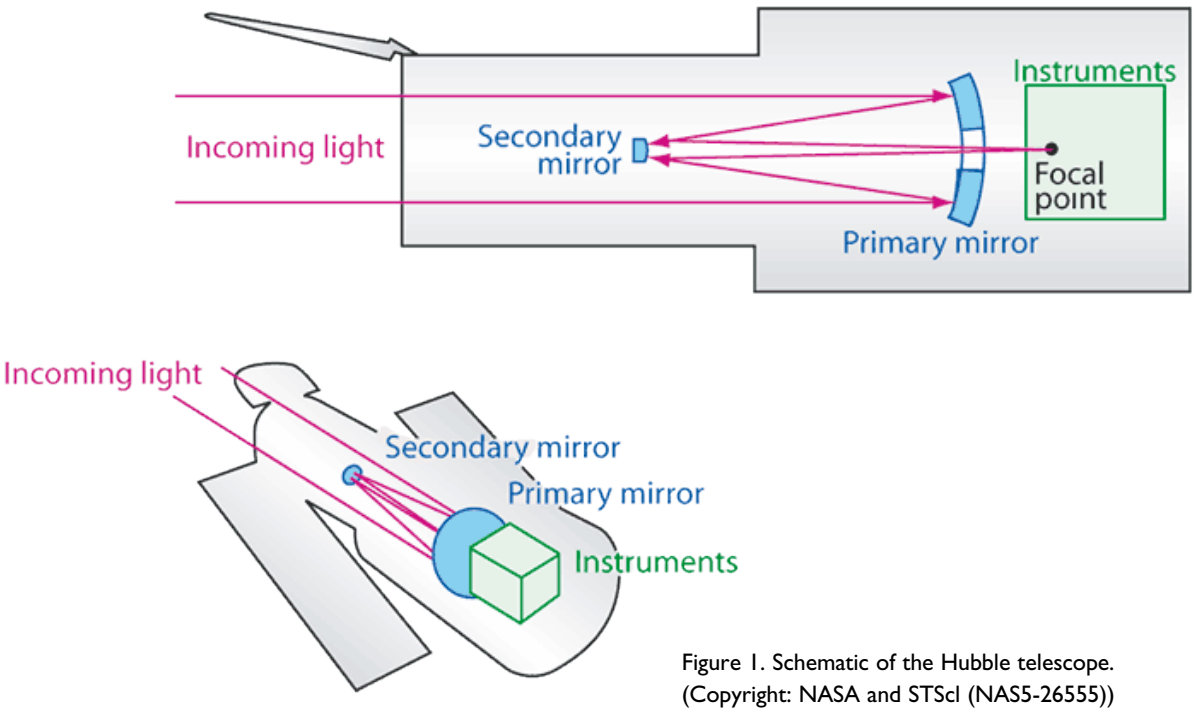


Figure 1. Schematic of the Hubble telescope. (Copyright: NASA and STScI (NAS5-26555))

small mirrors were used to intercept the light reflecting off the mirror, correct for the flaw, and bounce the light to the telescope's science instruments. Several of the telescope's cameras were replaced by newer versions containing small mirrors to correct the aberration [1]. As the Hubble was already in orbit, the costs to resolve the spherical aberration were about $150 million [2] [3]; a figure that does not even include the cost of the shuttle repair mission itself – which may easily have been about three times higher [4].

The root cause

A commission was established to determine how the error could have arisen [5]. They found that the main null corrector, a device used to measure the exact shape of the mirror, had been incorrectly assembled: one lens was wrongly spaced by 1.3 mm. During the polishing, the surface of the mirror was analyzed with two other null correctors, both of which correctly indicated that it was suffering from spherical aberration. However, these test results were ignored as it was believed that these two correctors were less accurate than the primary one that showed that the mirror was perfectly figured.

Observations

A couple of observations can be made about the Hubble telescope and its flaw in the primary mirror. First, it was a well-understood optics problem for which the mathematics were known and described *precisely*. Hence, it was "easy" to figure out a solution to resolve the spherical aberration of the primary mirror. Second, it was this same mathematics that allowed the engineers to verify their designs during design time. They did not need to construct the entire Hubble telescope and check whether the images produced met the required quality. Third, as a consequence all testing that was performed on the telescope was intended to determine the quality of the telescope and not to establish it. Fourth, an expensive mistake was made during the verification of the primary mirror: the engineers missed an important aspect, which was considered a *detail*. They should have investigated more carefully the difference between the main null corrector and the other two null correctors. Fifth, engineering requires one to work *precisely* while not losing sight of important *details*; any mistake may lead to catastrophic failures and most of the time costs a lot of money to fix. In the end, the suppliers of

the telescope agreed to pay $25 million to settle claims over the defects after which they were freed of further liability claims [6].

Software

What about software? Is it as precise as the other engineering disciplines? Unfortunately, the answer is no. Most software (methodology) lacks a sound mathematical foundation, which makes it impossible to reason about it with sufficient precision; let alone perform design-time verification like other engineering disciplines do. Typically, only after implementation it can be determined by testing whether the software is correct (verification: does the software contain errors) and whether it is the correct software (validation: does the software fulfil its intended purpose). This means that only very late during the development process – perhaps even too late – feedback is obtained about the completeness and the correctness of the requirements (including software), architecture and design; paradoxically, the feedback only refers to what can be expected – the unexpected is never tested and therefore makes testing insufficient. Testing is also more than testing code; it is also testing the requirements, architecture, and design. Further, testing is also more than *determining* the quality: it is also often the phase where quality is *established* by resolving all errors that were found and, as a result, testing becomes unpredictable in terms of quality, progress, and cost. Consequently, the decision to release software is often made in a subjective manner [7].

But it gets even worse: during the requirements, architecture, and design phases, reviews are organized to get feedback and improve the quality of the specifications as much as possible. Everybody knows that it is the most cost effective to find and to resolve errors during these phases. However, since most errors are *injected* during the requirements, architecture, and design phases [8], but only a small number are *detected*, a false impression is given about the quality and the progress a software development team is making. In practice, remarks on requirements and architecture are often hand-waived as being *details* that can be resolved later on, whereas in reality such remarks refer to specification points that are not *precise* enough. The consequences only become apparent when it is typically too late: during testing.

A paradigm shift is needed

Is there a solution? Projecting the development of the Hubble telescope onto software development implies the introduction of mathematics or, more precise, formal methods. However, formal software methods in industry were never really successful: they did not scale very well, they were expensive to use as highly-skilled people were needed, and often the solution was more complex than the original problem statement. Lastly, key stakeholders were excluded from the development process as they were not able to read the difficult mathematical notations and judge whether what was described was what they intended. However, when taking a closer look at other disciplines, it can be seen that indeed mathematics are involved, but that most, if not all, of the mathematics are hidden from the engineers. For example, a construction engineer creates a CAD/CAM model of a bridge from which automatically the mathematics is generated needed to check whether the bridge withstands earthquakes, the traffic, strong winds, etc. A similar approach is desired for software development: create a model, automatically generate the mathematics and verify it for design errors and resolve them until the design is error-free, but in such a way that it can be applied in industrial-scale development, that it is general purpose, easy to use and understand, and it is indeed more economic to do so.

Analytical Software Design

The above requirements are met by Analytical Software Design (ASD), a patented technology developed by Verum Software Technologies. ASD is a component-based technology that enables software specifications and designs

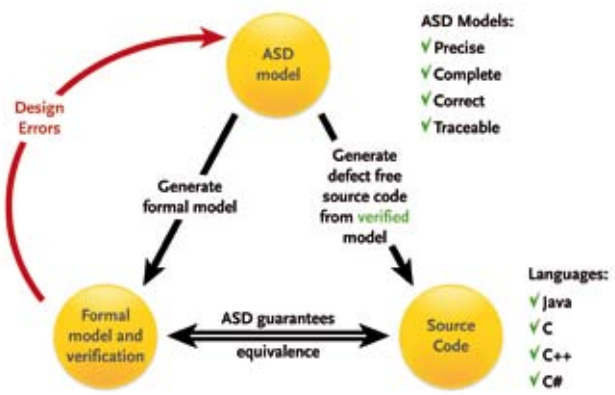
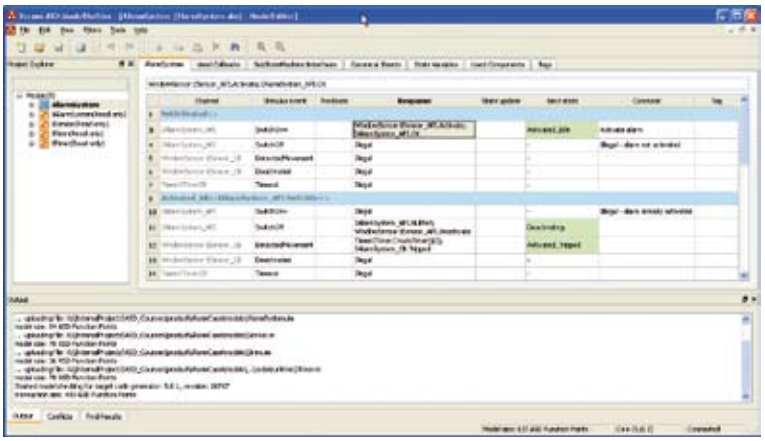
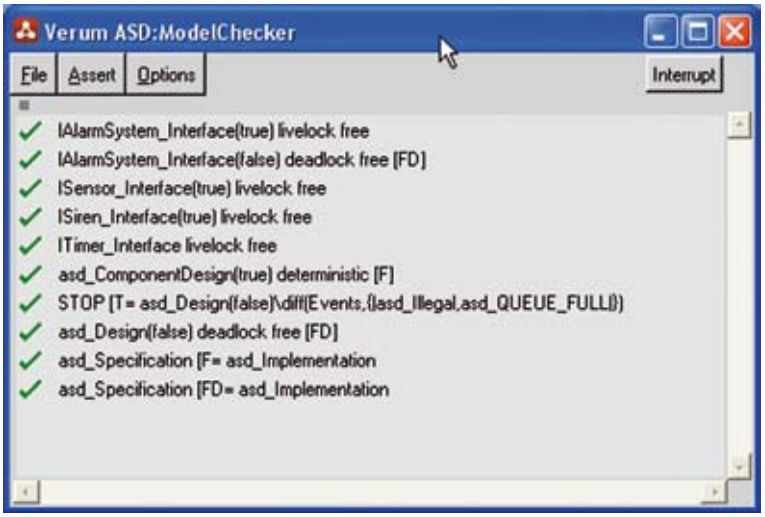


Figure 2. ASD technology provides design verification and code generation.



a



b

Parts of an ASD model.
(a) ModelBuilder.
(b) ModelChecker.

to be mathematically verified at design time. After the design has been verified, code is generated from the verified design; see Figure 2. This technology is incorporated in a tool chain called the ASD:Suite.

ASD uses two kinds of component models: first, interface models that serve as the functional specification of the component which is externally visible. Second, design models that specify the internal design of this component. Although ASD models have no visible mathematical notation and are thus accessible to all project stakeholders, they are sufficiently *precise* so that mathematical models can be automatically generated from them. Where the construction engineer uses tools to generate a finite-element analysis model of a design, the software designer uses the ASD:Suite to generate a process algebra model. The process algebra model is then mathematically verified against the functional specifications by the ASD:Suite by means of model checking. Design errors uncovered by this verification, such as race conditions, deadlocks, and livelocks, are then easily removed by the designer by updating the ASD models. These models are then verified again, the process being repeated until all errors have been removed. At the end of the process, the ASD design model is *correct* and *complete*. The ASD:Suite is then used to generate the corresponding implementation in MISRA C, C++, C#, or Java, in such a way that the execution semantics as described in the models are equivalent to the mathematical model as well as the generated code. Since ASD is component-based, it can be applied to all software components that have discrete control behaviour; rather than specifying and verifying an entire system, the steps above are applied on the individual components. The compositionality rules of the mathematics guarantee that the composition of all these components together also works, and therefore these rules provide the scalability needed for industrial-size systems.

Figure 3 shows two fragments of an ASD model. Using ASD, all behaviour of a component is explicitly described, including all error scenarios. An ASD model is based on a Sequence-Based Specification (SBS) methodology. This methodology ensures that for all possible events in each state that a system finds itself in, proper responses are defined for all the events as well as the next state to go to. During this rigorous specification process, new states can be discovered where again proper responses and next states have to be defined for all stimulus events. This process ensures the completeness, whereas the model checking provides the correctness.

A case: Philips's prototype digital pathology scanner

At the beginning of 2009, CCM (Centre for Concepts in Mechatronics) in a consortium of companies embarked upon an ambitious project to build a prototype digital pathology scanner for Philips's Digital Pathology business venture. Besides the technological challenges, as it had to be a fast scanner with the highest resolution and image quality, it had to be realized in a time span of 12 months where both software and hardware were developed concurrently by several companies leaving only a short period for test and integration.

CCM chose to use ASD for modelling, verifying, and generating the code for the control software of the digital pathology scanner for various reasons. First, at the time the contract was awarded, the customer requirements were good, but like in most other projects, certainly not complete. ASD enabled CCM and the other consortium

Figure 4. Philips's prototype digital pathology scanner, for which the control software was designed using the ASD:Suite.

members to *precisely* specify all possible behaviour of the prototype scanner. Second, during the requirements and architecture phase ASD was used to specify all external interfaces completely and correctly. These *precise* specifications enabled concurrent engineering of the hardware and software with the net result that integration of the graphical user interface was performed within hours and worked first time right, and that integration with the hardware was also successfully performed within a couple of days. Further, the first prototype of the scanner was to be shown at an exhibition for pathologists. The scanner had to be operational during the entire exhibition – even when visitors would push the scanner's buttons in all possible combinations. Another effect of using ASD: all exceptional behaviour has to be specified.

Initially, CCM had estimated to deliver about 70K lines of code and to realize this with a team of 8-12 people; in the end, the software was developed with 7 people while at the same time the code size grew to over 200K lines of code as the actual functionality increased. CCM would not have met the demanding deadlines and quality without the use of ASD, as the benefits went beyond producing defect-free software; it also increased CCM's productivity and facilitated the concurrent engineering.

Conclusion

ASD has been successfully applied to various industrial-scale projects where the digital pathology scanner is the most recent one. It provides the necessary balance between precision and detail since ASD identifies the minimum level of detail required to satisfy its completeness property, which in turn provides the level of precision required for the purposes of formal verification. The only question that remains is which company will be the first one that develops software based on formal methods and accepts liability for the software like other engineering disciplines? Only then will software be a true engineering discipline.

References

[1] hubblesite.org

[2] articles.baltimoresun.com/keyword/repair-mission/recent/2

[3] articles.baltimoresun.com/1992-10-27/news/1992301003_1_elmer-telescope-mirror-hubble-space-telescope

[4] www.nasa.gov/centers/kennedy/about/information/shuttle_faq.html

[5] The Hubble Space Telescope Optical Systems Failure Report, 1990. NASA Technical Report NASA-TM-103443.

[6] articles.orlandosentinel.com/keyword/liability-claims

[7] H. Sassenburg, 2006. Design of a methodology to support software release decisions, Ph.D. thesis, University of Groningen, the Netherlands.

[8] C. Ebert and C. Jones, 2009. Embedded Software: Facts, Figures, and Future, *IEEE Computer*, 42 (4), pp. 42-52.