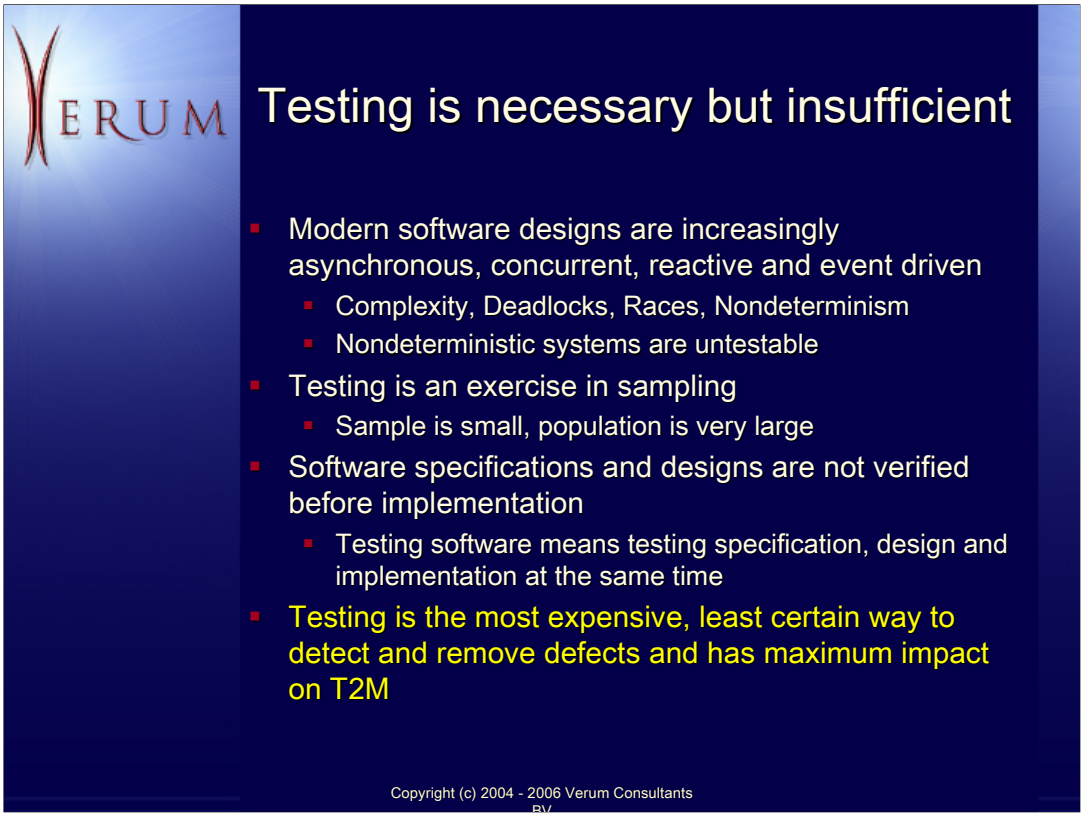
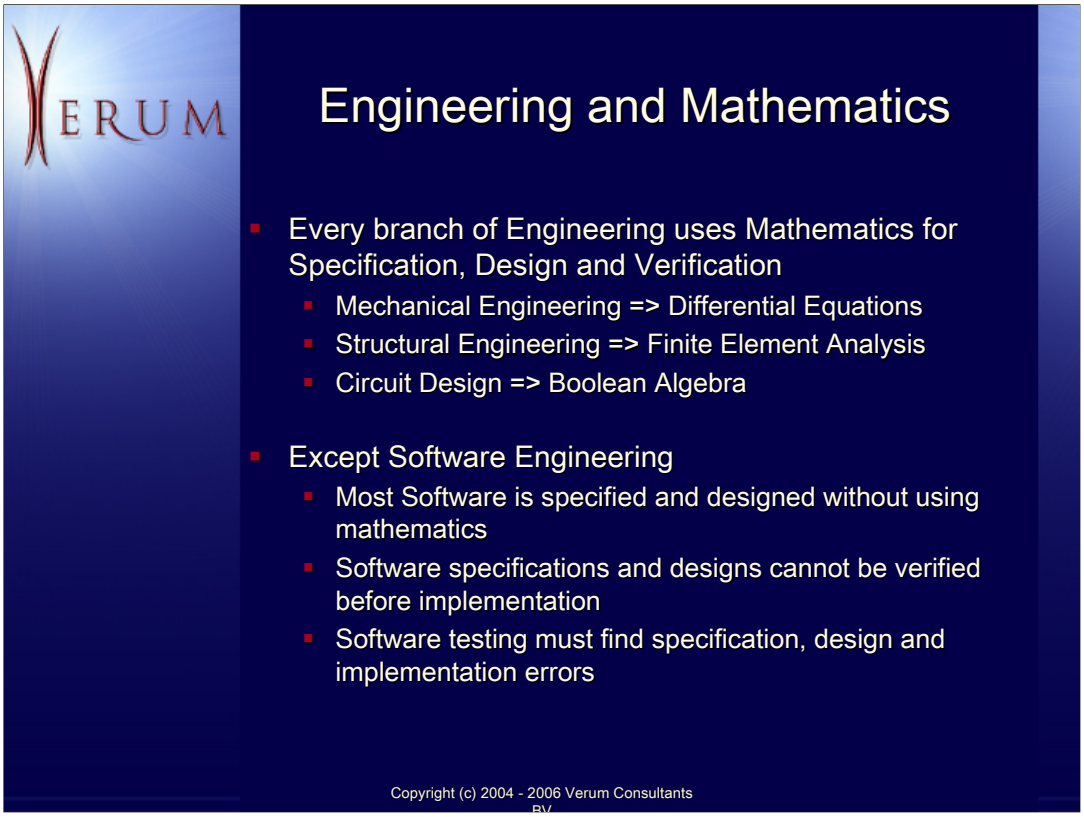
The previous speaker talked about testing software *after* it has been implemented; I want to talk about how we can test software *before* it is implemented and before we have spent any money on programming or test specification. In most product sectors, the amount and complexity of in-product software is increasing *exponentially* – roughly in following Moore's law. This is challenging existing software engineering methods and processes, especially the testing phases. In addition to this increase, the nature of the systems we are called upon to develop is changing. Almost every system we make today is event driven, reactive and involves some degree of distributed processing and concurrency. This has an important consequence:

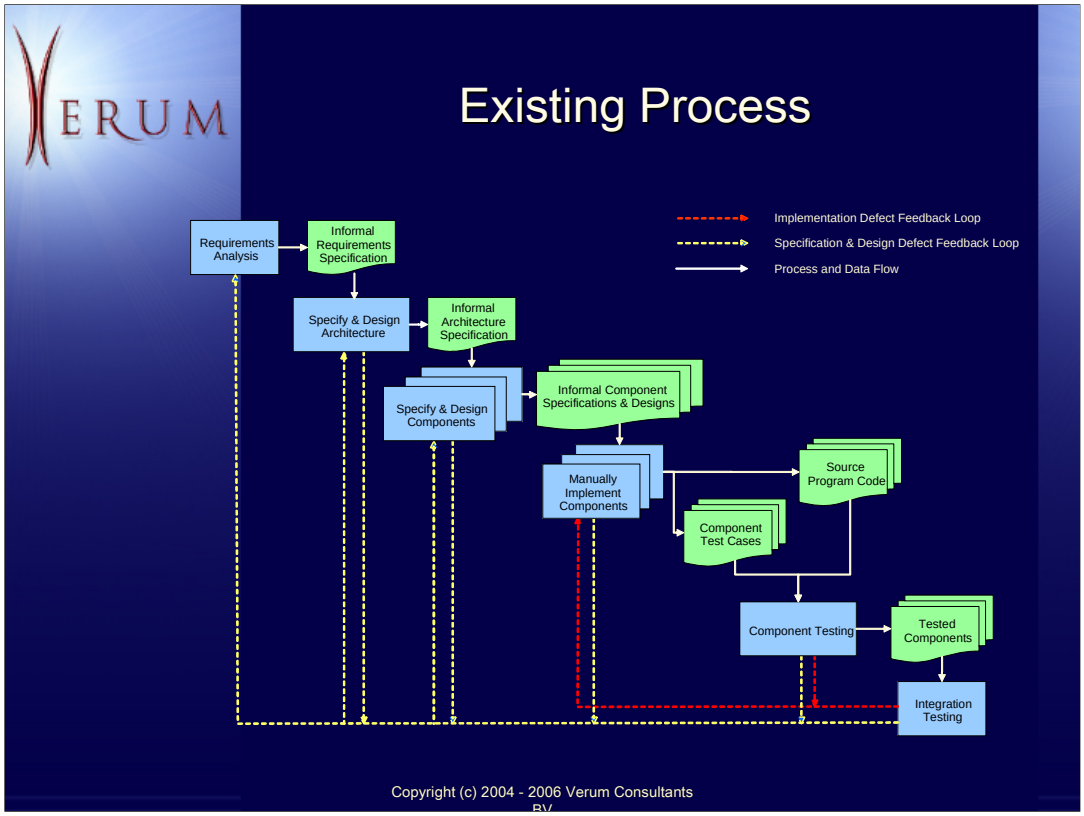


Testing is necessary but not sufficient. Note: this is not an argument to do less testing; it is an argument for doing something else, in addition to testing, so that software enters testing with a higher quality than is usually the case. Why is testing alone not able to meet the modern software development challenge? Modern software designs are increasingly *asynchronous* and *concurrent.* Such systems are, by definition, *nondeterministic*, increasingly complex and introduce the potential for design errors such as deadlocks, divergence and race conditions. These are among the most difficult errors to detect and remove by testing. It is axiomatic that *nondeterministic systems are untestable.* There is no economically feasible amount of testing that can give us any meaningful measures of correctness and freedom from errors. Such designs are not restricted to a few niche domains; many embedded / in-product systems are designed this way. For example, the GSM protocol stack on your mobile phone; the software in your digital TV; the software controlling a wafer stepper, component mounter, electron microscope and a body scanner are designed this way. As are the many systems in cars, the most complex distributed platform in mass production! All of these software systems have something in common; they are an essential part of some core product and are *Business Critical* to the companies that make them.

All testing is an exercise in sampling, but in testing software systems, the sample size is very small compared to the population size. Consider a simple software module with an alphabet of 20 stimuli and a maximum sequence length of 10 (that is, the longest sequence of input stimuli that results in unique behaviour). There are in the order of 1.08E13 potential execution scenarios. Now imagine two different components of this complexity executing concurrently and communicating on a shared an alphabet of 10 events. How many potential execution scenarios are there? Now imagine compositions of 20 such processes, or a 100 or more. How can conventional, informal design methods address such complexity? What does testing coverage mean in this context?



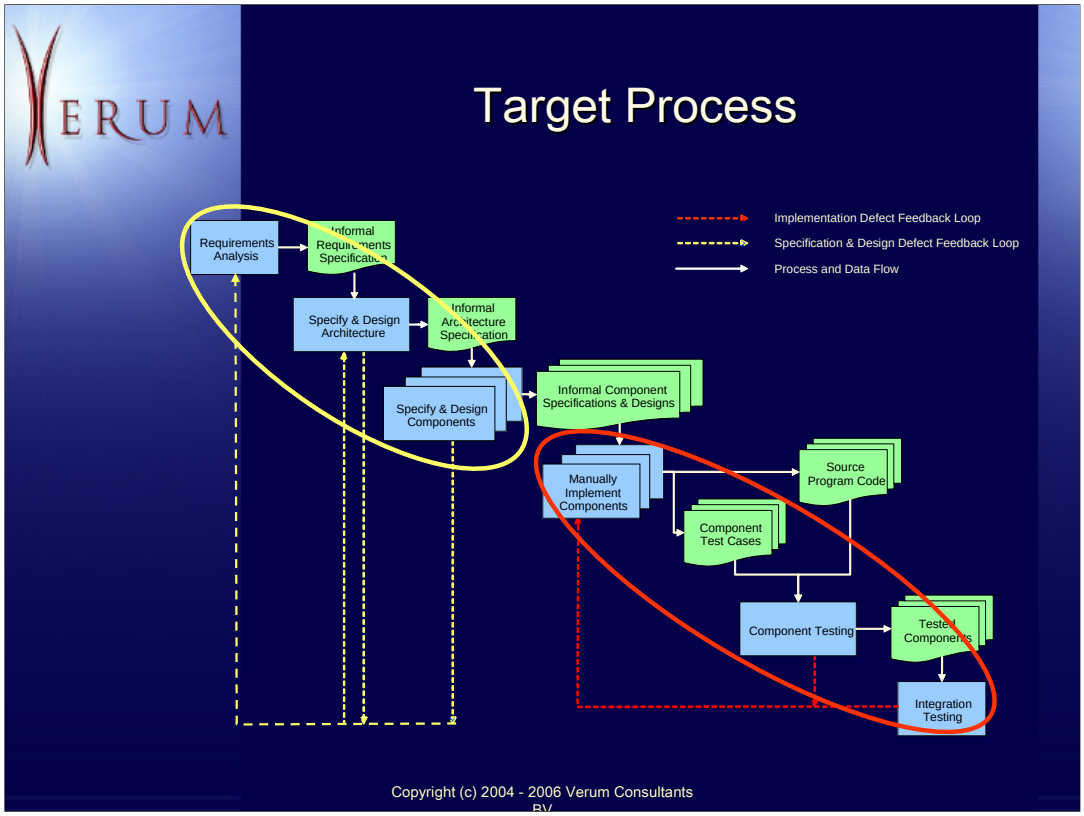
This is what distinguishes software engineering from other branches of engineering – the routine application of mathematics during specification and design to eliminate errors before implementation.



I have explained the background to ASD – the fact that software complexity is increasing, product liability issues arising from increased software use in consumer products, T2M pressures and the need to improve development predictability. I mentioned that software engineering, unlike all other branches of engineering, does not routinely apply the mathematics that enable specifications and designs to be verified before implementation. This means that when software is tested, not only are we testing the implementation and looking for implementation defects, it is the first moment in which we are able to concretely test the architecture, design and specifications. This occurs late in the life cycle.

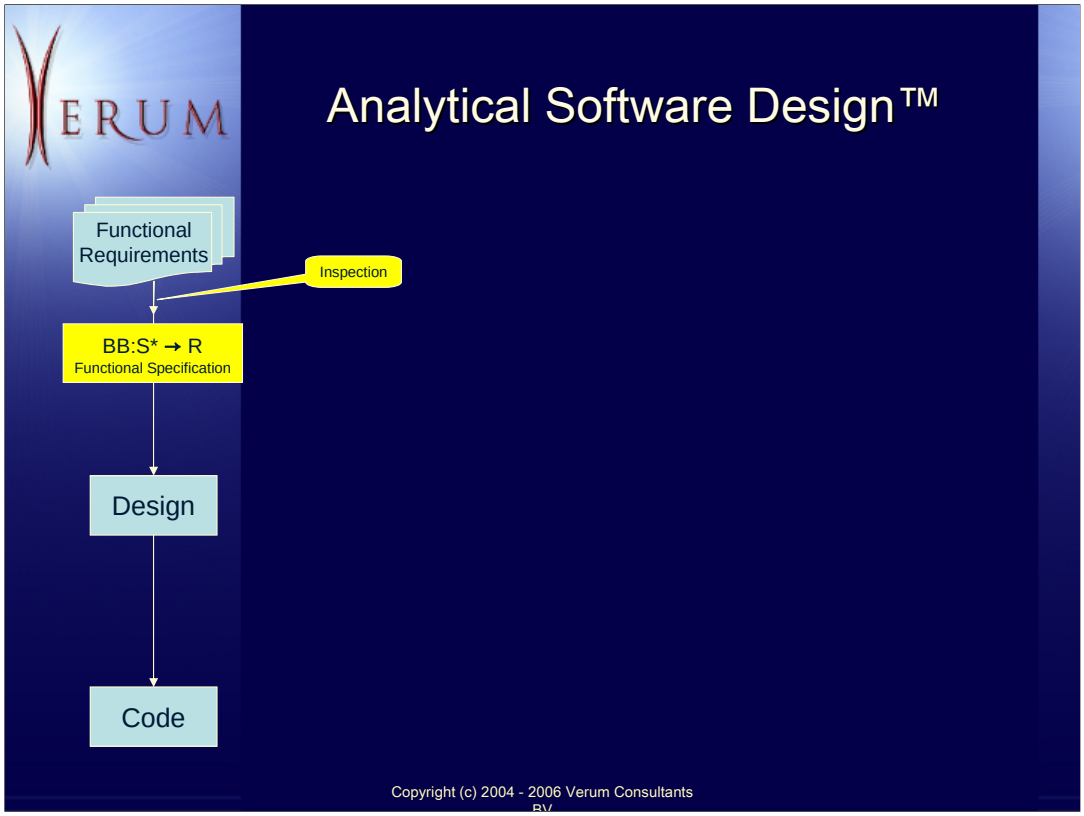
This slide illustrates the typical sequence of phases that must be accomplished to develop software. It is not intended to represent any specific development process; it merely serves to illustrate that the work occurs in a certain natural order. The dotted lines represent the feedback that occurs as each phase finds defects in previous phases. We expect this to an extent; each phase is a refinement of the work done in previous phases. The problem arises because current development methods generally do not provide any formal means of verifying architectures and designs against function specifications before we spend the money implementing the software. We have informal methods, such as design reviews and inspections and these make a substantial contribution. But at best, they are only partial; they are reviews of static descriptions of the system presented in informal specifications, which lack precision and any formal means of establishing completeness or consistency. As a result, when testing starts, errors are found in specifications, architectures and designs, resulting in expensive rework and delays in completion.

What ASD aims to do is this:



It leads to breaking the feedback loop into two parts: the yellow part removes specification and design errors before spending money on implementation and does so using mathematically based proof techniques. This means that the implementation is based on designs known to be correct and the red feedback loop is only about implementation errors – programming mistakes if you will. We do not use testing to find design errors, only programming errors.

How do we do this? By applying software engineering mathematics.

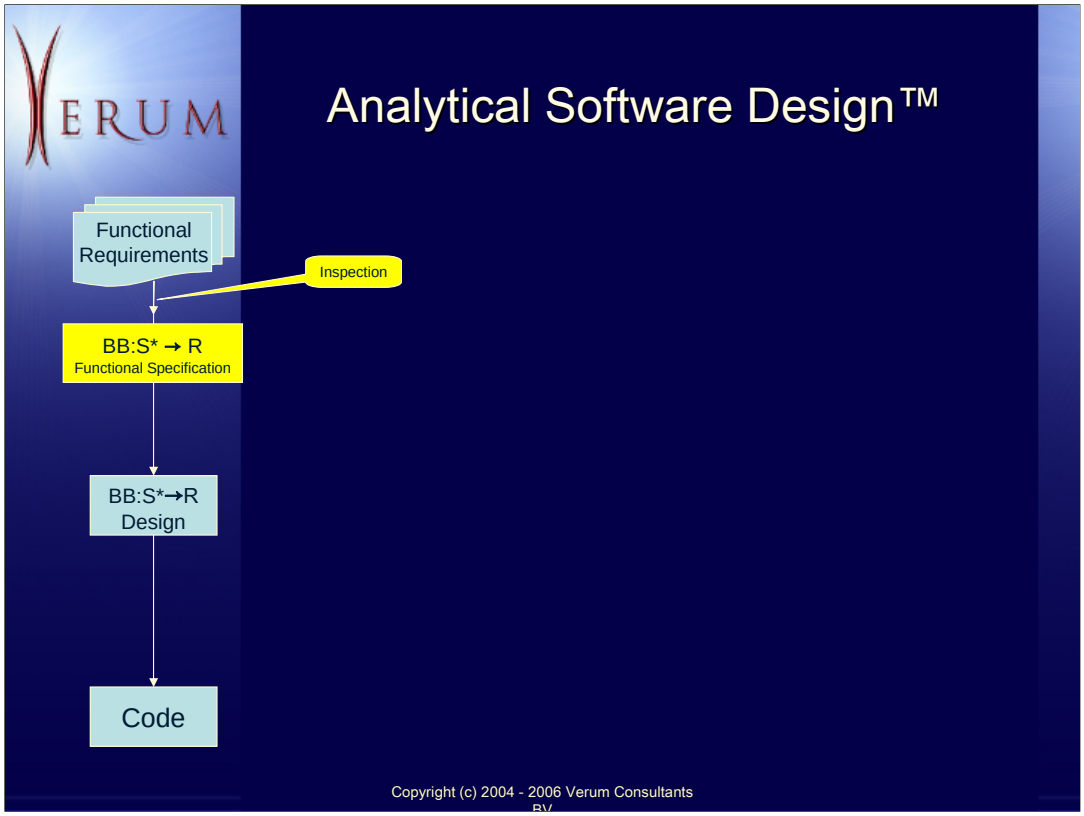


For new software, either for new systems or for new parts of existing systems, we start with a conventional “informal” specification in the form of the work products already produced by our Customer’s existing development process. Step 1 is to make an ASD specification using Sequence-based Specification techniques (SBS) to produce a so-called Back Box Function (BB) specifying the required functional behaviour. This is a total mathematical function mapping all possible sequences of input stimuli (events, messages method calls etc.) onto the specified system response. We do this together with Customer domain / technical experts. The goal here is *precision*, not detail as such.

For reengineering existing software components either because of required changes or because conventional testing based approaches have been unable to solve stability or reliability problems, we may also reverse engineer the specifications from the existing code base, again with the involvement as needed from those familiar with the code.

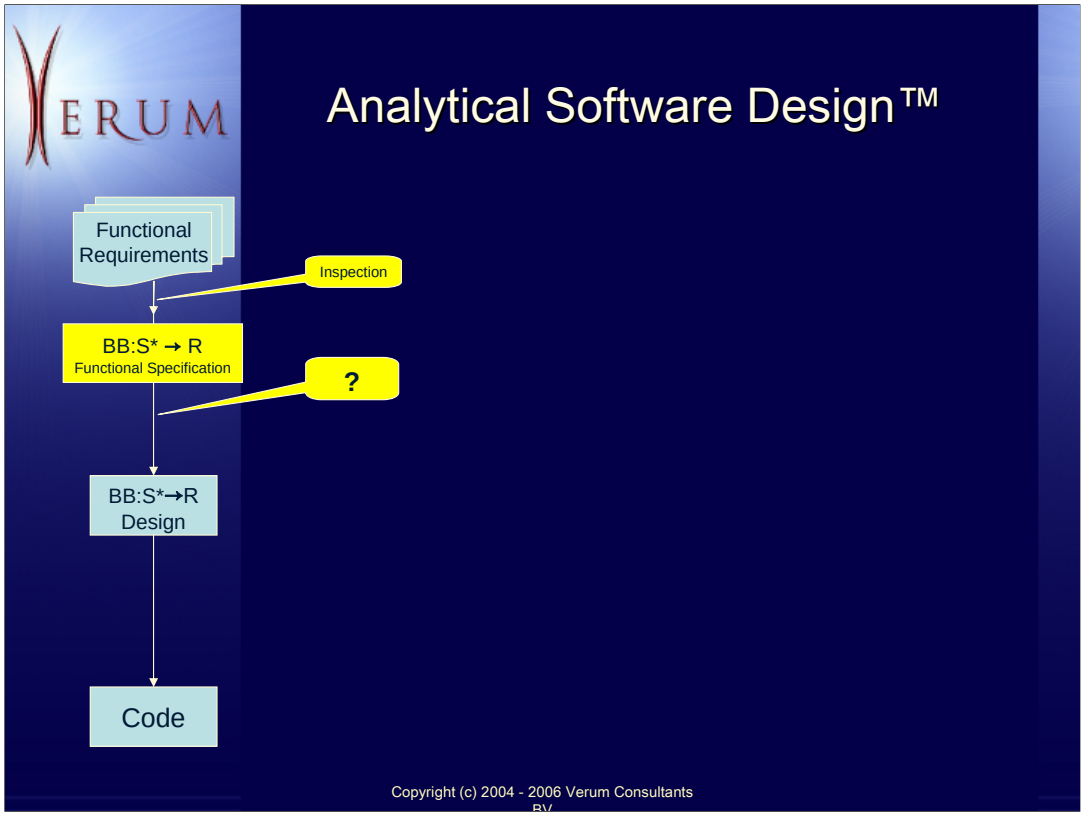
When we have completed the ASD specification, we must establish 1) that it matches the original specification 2) that the design fully implements it and 3) that the code fully implements the design.

The first we do by inspection. This is possible because although the ASD specifications are based on mathematical principles, they do not use difficult mathematical notations. They are easily accessible to stakeholders and fully traceable to the original specifications. The other questions are answered next – starting with the design.

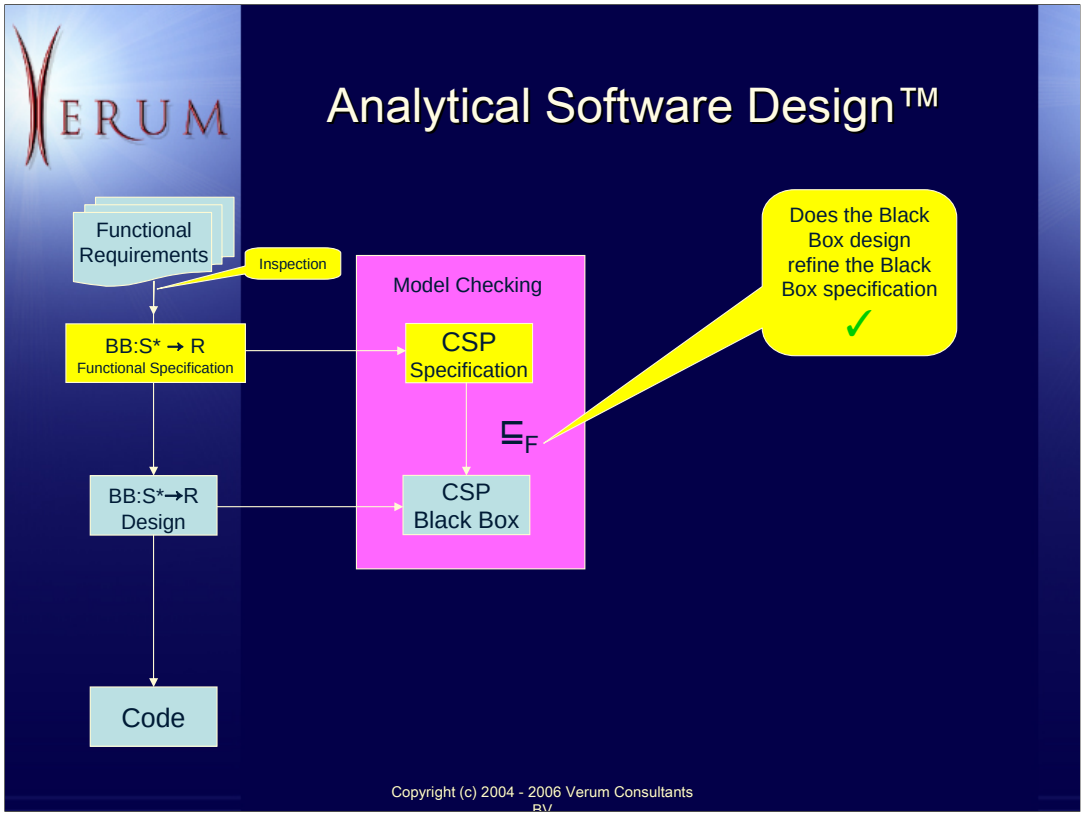


We make the design following generally accepted, conventional approaches, the big differences being 1) the emphasis we place on *precision* and 2) the way in which we document the design. Function behaviour is captured using SBS in the form of a design BB. Again, the ASD specifications allow full participation of other engineers because they do not rely on much visible mathematics. Most software engineers learn this technique quite quickly and like it.

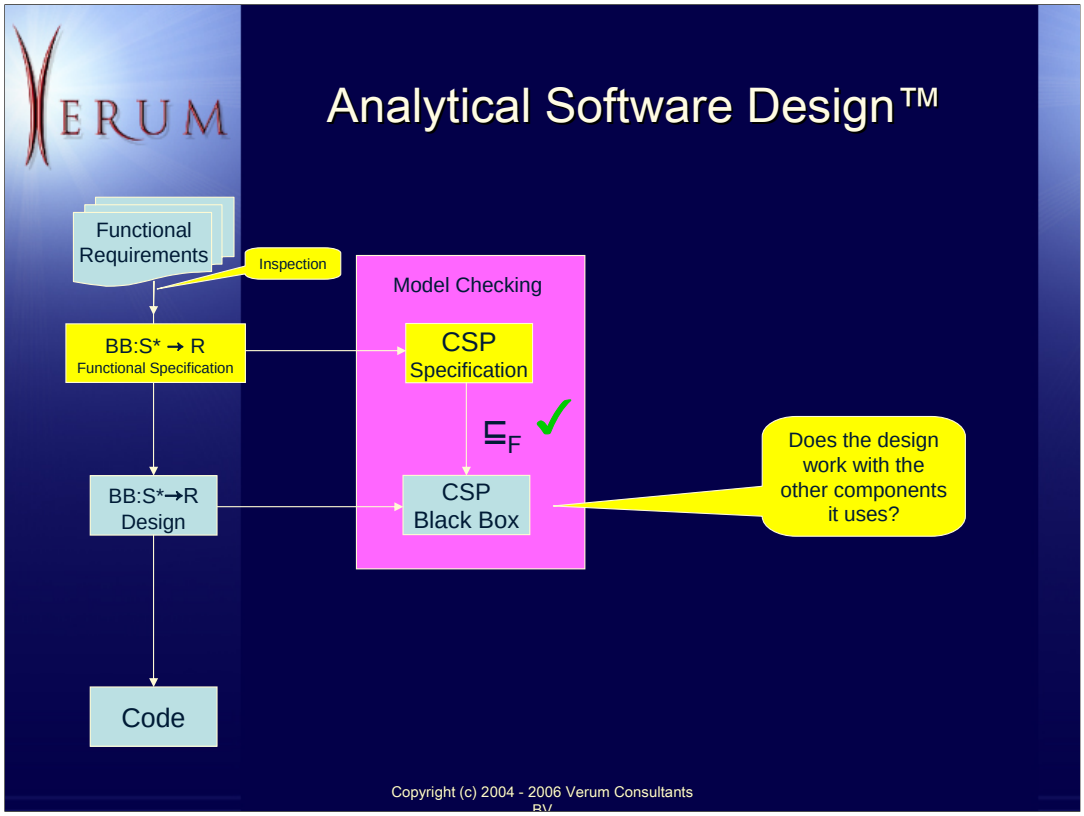
If we are reengineering and existing component, then during the design we may reverse engineer much of the design from the existing code.



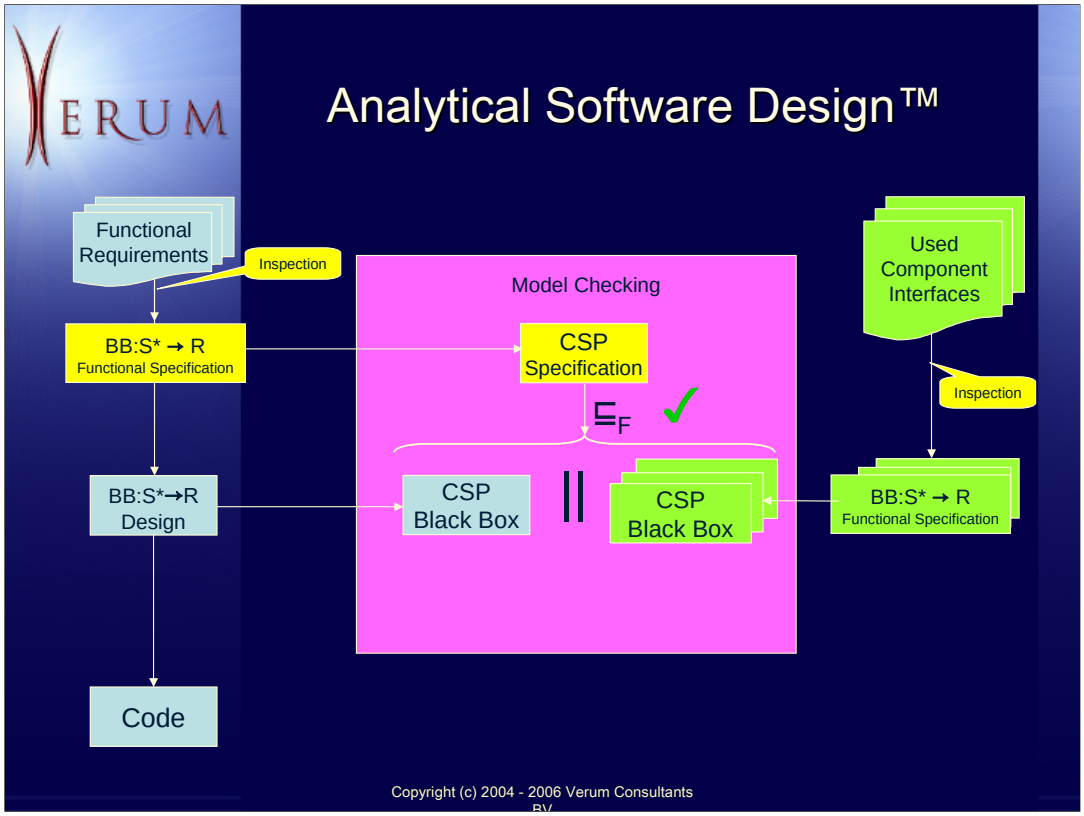
Having done this, we have a “proof” obligation to discharge; namely verifying the BB function of the design against the BB we made from the requirements. How do we know the design implements everything in the requirements and nothing else? How do we know it will behave according to its functional requirements?



We translate the BB specifications of the requirements and the design automatically to CSP models and we usea tool called a model checker to establish that the BB design exactly complies with it. The way we apply SBS to specifications enables nondeterminism to be captured properly, essential when describing externally visible behaviour. CSP algebra also captures nondeterminism and the refinement principles used in CSP are able to compare deterministic design models mathematically to nondeterministic specification models. The mathematical verification we use in this case is called Failures Refinement. With this, we can verify whether or not the design (i) specifies all required behaviour in the correct way; (ii) does not specify any behaviour not specified in the specification and (iii) if optional behaviour is specified in the design, it is designed according to the specification. These are not inspections or tests; these are mathematical proofs so they hold for all possible execution scenarios. We could never establish this by testing.



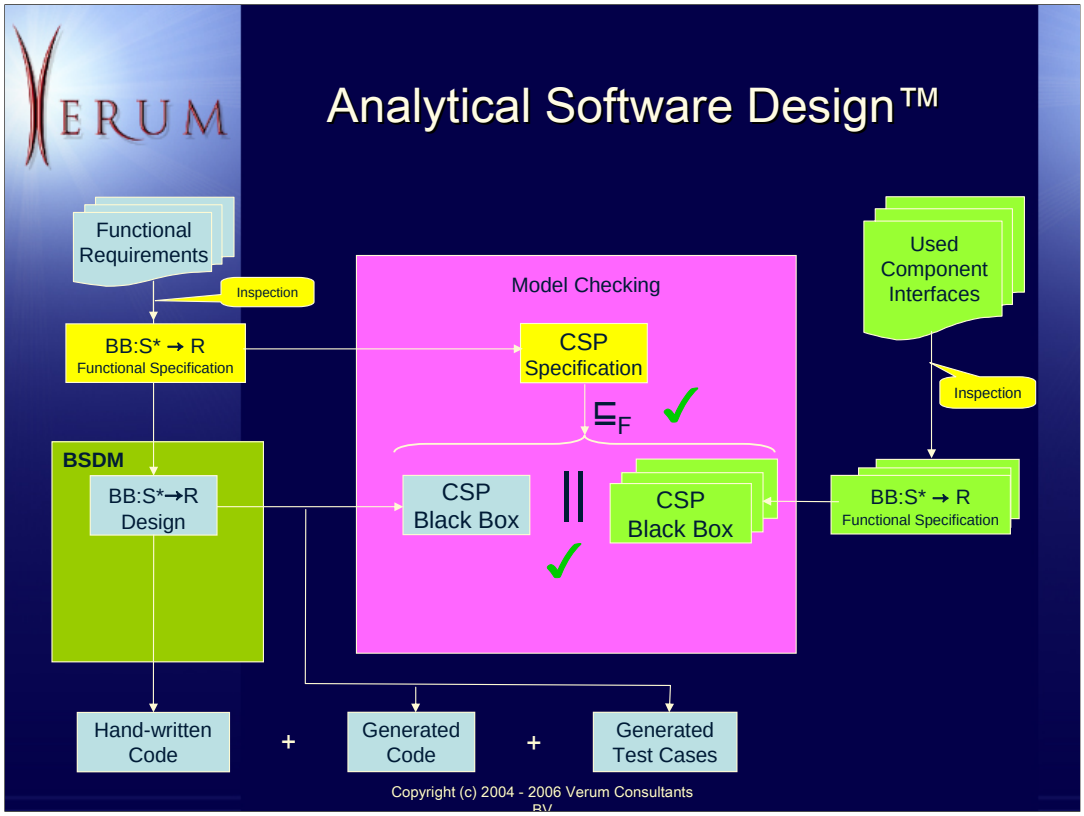
But of course, in reality, we cannot establish that a design behaves correctly without considering how it interacts with the other components it uses. Indeed, the way in which the design will interact with other components, HW or SW, is a key part of establishing that the design is correct. Particularly in event driven, reactive systems with concurrent behaviour, this cannot be done by inspecting static design specifications individually. We need some way of exploring the dynamic behaviour of the design as it will behave together with its runtime environment when it executes. And of course, we wish to do this **before** we implement our designs in code. How do we do this?



We apply SBS to analyse the externally visible behaviour of these other components and make BB function specifications of them. This is a valuable exercise in itself; it leads to a more complete and deeper understanding of the behaviour of these other components; it focuses on interface behaviour and frequently raises important questions not clearly addressed in the conventional interface specifications. It looks like new work, but it is not; we have to do this analysis and understanding anyway in order to successfully program against these interfaces even in a conventional development process. The new work is just capturing this knowledge as a BB function and we get a huge payoff for this little extra effort. We verify this work by inspection and discussion with “experts”.

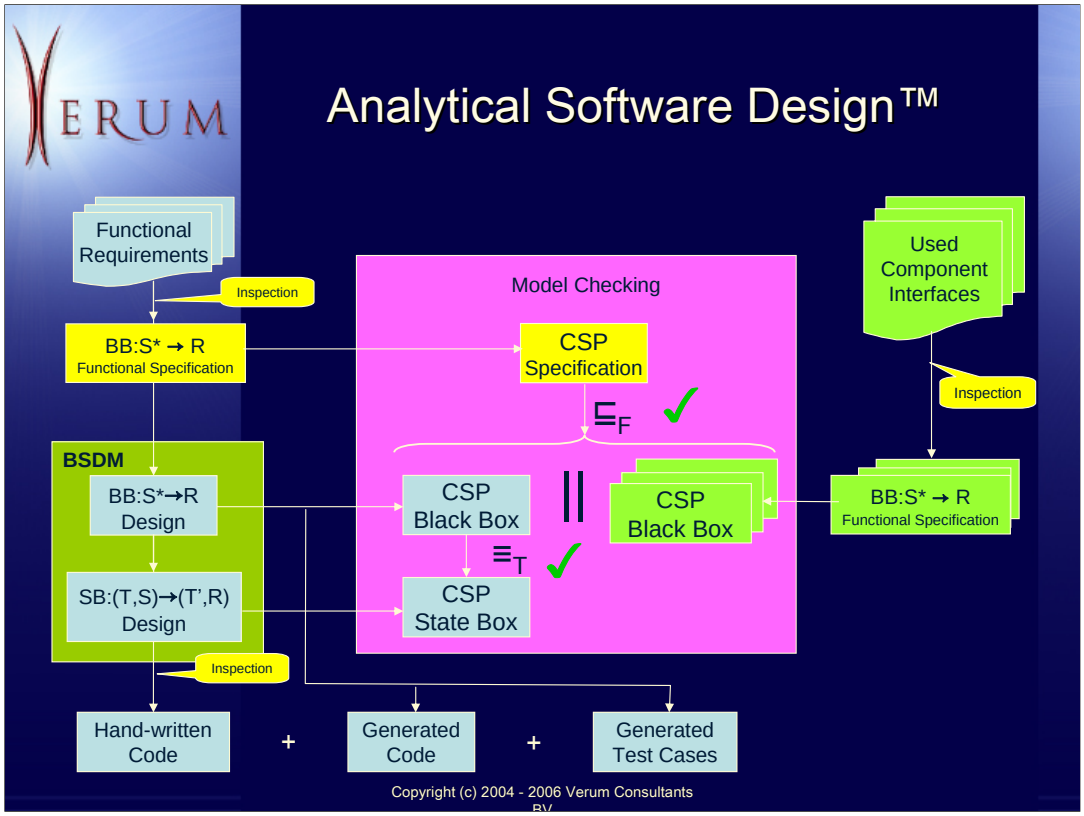
When implementing new software components that are to be a part of an existing legacy system, it is frequently the case that the current implementation of the legacy software no longer behaves according to the existing specifications. In these situations, the ASD specifications will be made with frequent reference to the existing legacy code base, “recovering” the current specifications from the existing implementation.

Having done this, we generate the CSP models of these interfaces and check our design together with these interface models.



At this point, we have a design which is verified against the functional requirements. We now have to implement this and verify the implementation against the design. The BB specification of the design is not a good programming specification – it uses abstractions such as infinite sequences of abstract events that are difficult to represent in most programming languages. The “abstraction” step is too big to expect a programmer to move directly from the BB specification to code. These abstractions have to be made more concrete before we can program them. This is done using the Box Structured Development Method (BSDM). This gives us a mathematically sound way to transform the BB into a State Box (SB) in which all these difficult abstractions are replaced by state data and state data update rules. We can program directly from this and we can check the code against this by inspection.

But first, we must establish that we made no mistakes and the SB exactly refines the BB.



This we do by automatically generating the corresponding CSP model of the SB and using a mathematical refinement called traces refinement to establish that the SB describes exactly the same behaviour as the BB. This is checked using the model checker.

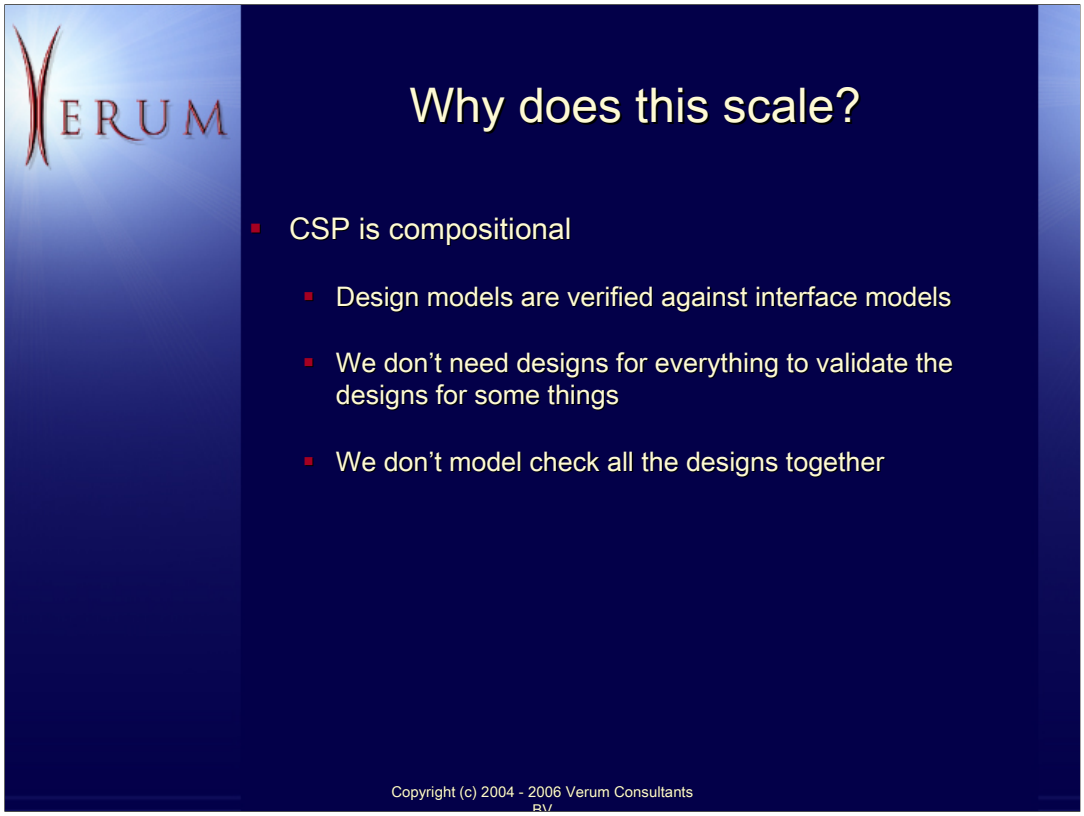
We address the issue of programming compliance with the design in three ways:

1. Some code (it depends on each project as to how much) can be generated automatically and we do not need to check this at all;

2. Some code still has to be hand written and checking this against verified designs in the form of SB specifications is straight forward using inspection;

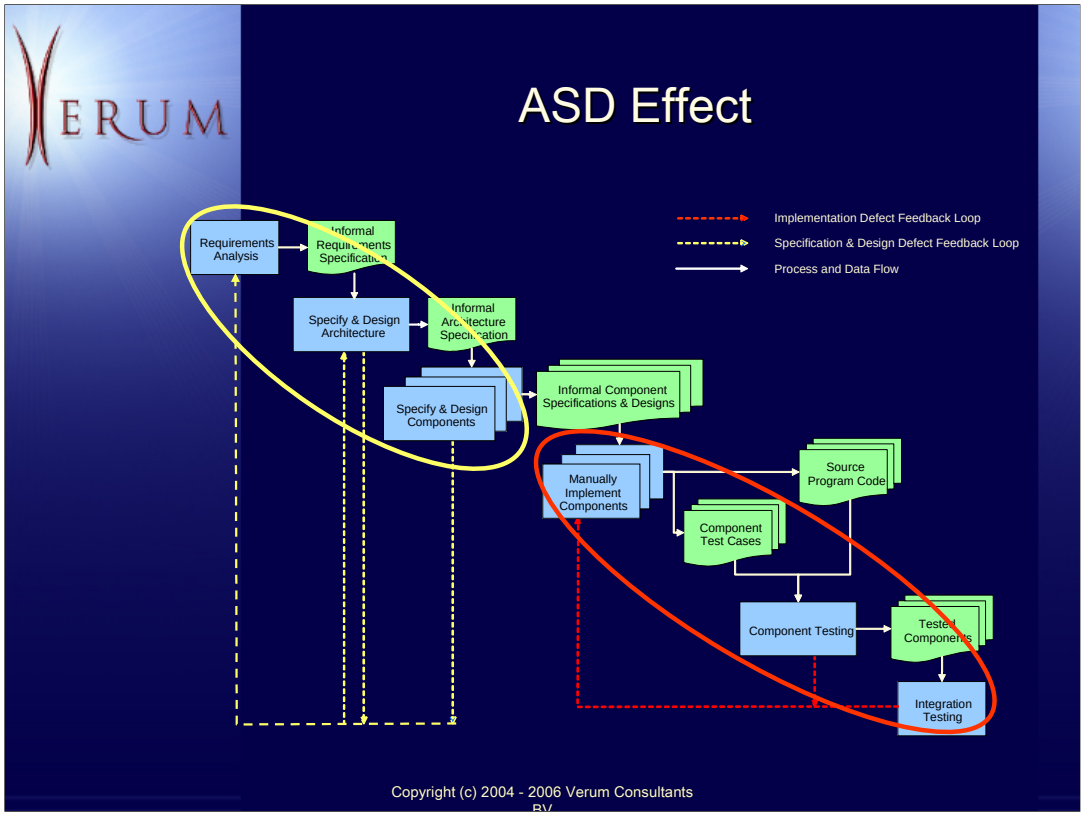
3. We can generate large numbers of test cases in the form of self running test programs, execute the tests and analyse the results automatically. This testing is based on statistical concepts and is very cost efficient and effective.

By applying these techniques in this manner, components should enter integration testing with far fewer defects than is usual. Also, because we are able to analyse dyanmic behaviour between components before investing in programming, there should be far fewer difficult integration defects to detect and remove.

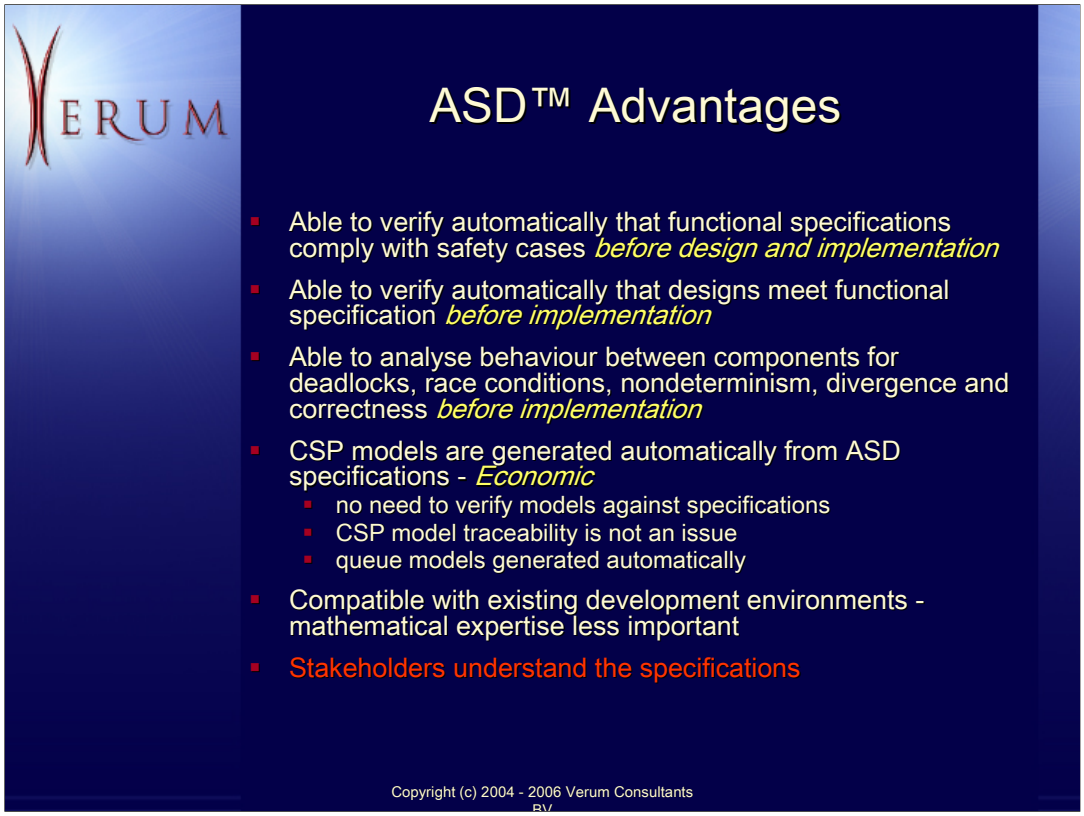


How can we be sure that this scales to industrial sized projects? Scalability, or the lack thereof, is one of the traditional arguments against this approach.

CSP is *compositional;* all CSP operators are monotonic with respect to refinement and refinement is transitive. This is what allows us to check designs against interface models and means we do not have to model all designs together at once. Indeed, when using third-party components, we may never have the designs available to us. Any safety or liveness properties we can describe in terms of refinement can be performed in this compositional way. This is very important; not all process algebras and modelling checking techniques can verify liveness properties in this compositional manner.



The big advantage of ASD is that it enables us to break the feedback loops we spoke about in the beginning into two separate cycles. What you saw in the demo was how these feedbacks loops can be separated and what the effect of doing this can be. We detected and removed several difficult classes of errors before investing in programming. Not only this, but the types of defects we detected are the very types most difficult to find by testing the implementation because they are very hard to reproduce.



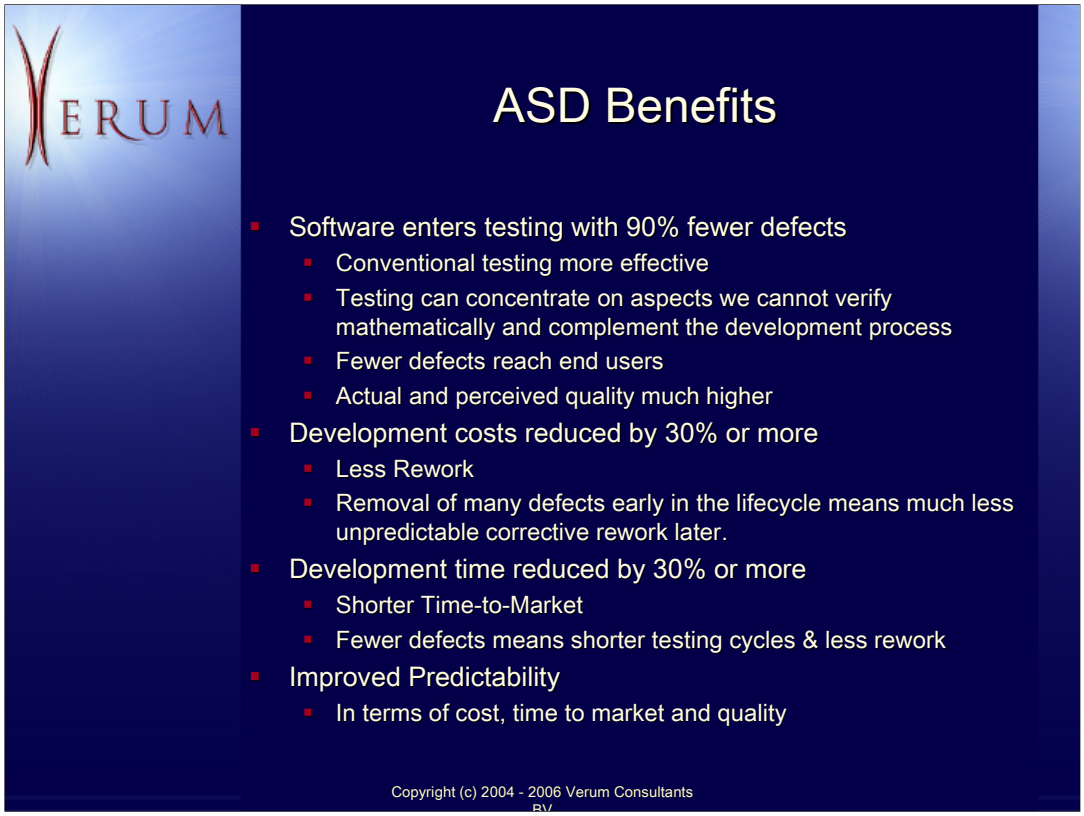
This gives us a number of important advantages.

(i) We can verify specifications and designs before we invest in implementation. This is both cheaper and more certain than testing; it is also much quicker.

(ii) We can analyse the dynamic behaviour of designs before implementation; including behaviour between components as well as within individual components. Because models are generated automatically, we don't need to verify models against specifications and we have no traceability issues.

(iii) In safety critical areas, we can work with domain safety engineers to analyse safety cases and formulate them as safety specifications to be verified by refinement. This means we can verify designs mathematically and ensure that such safety case hold. Again, this is not inspection or testing, but mathematical proof, providing a degree of certainty not achievable any other way.

(iv) Most importantly, ASD can be added to existing project teams in existing environments with minimum disruption and stakeholders retain control over specifications because they can understand and verify ASD specifications.



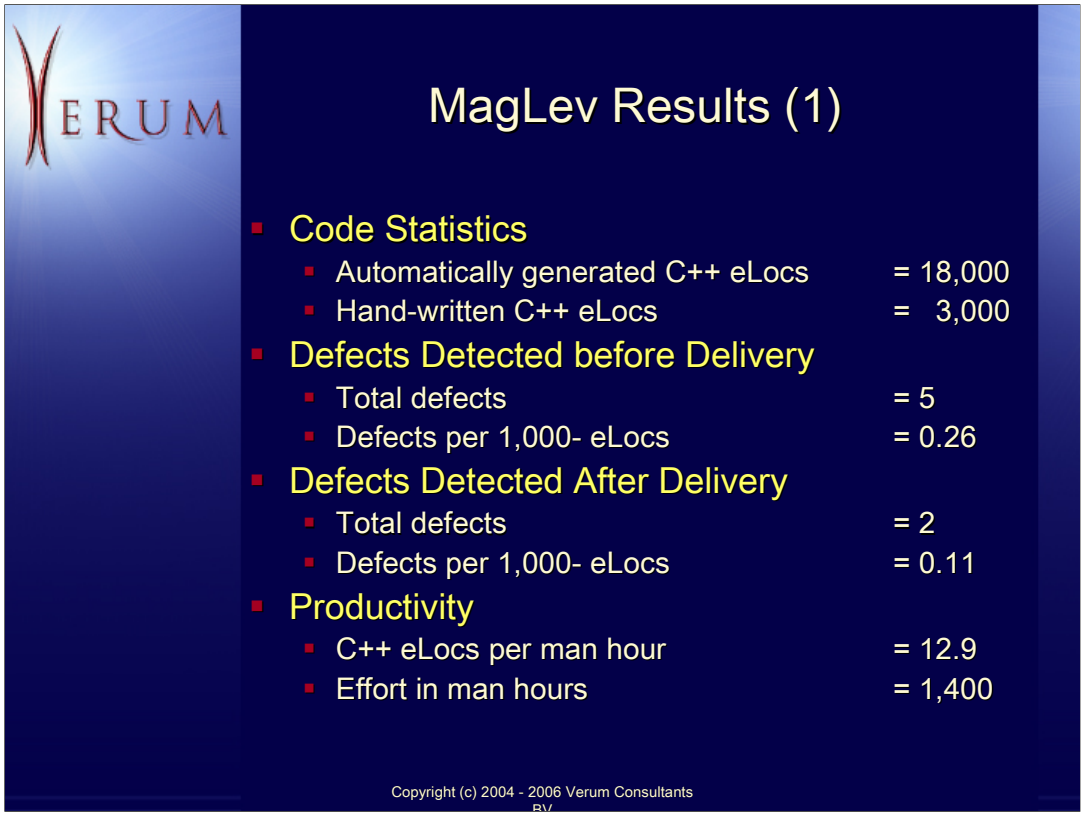
This is the connection to the “bottom line” business goals of the organisation. This is our experience and that of our Customers based on the projects we have completed so far. Software development by ASD is cheaper, quicker and results in fewer defects reaching end users.

All of this translates to bottom line profit increase and competitive advantage.

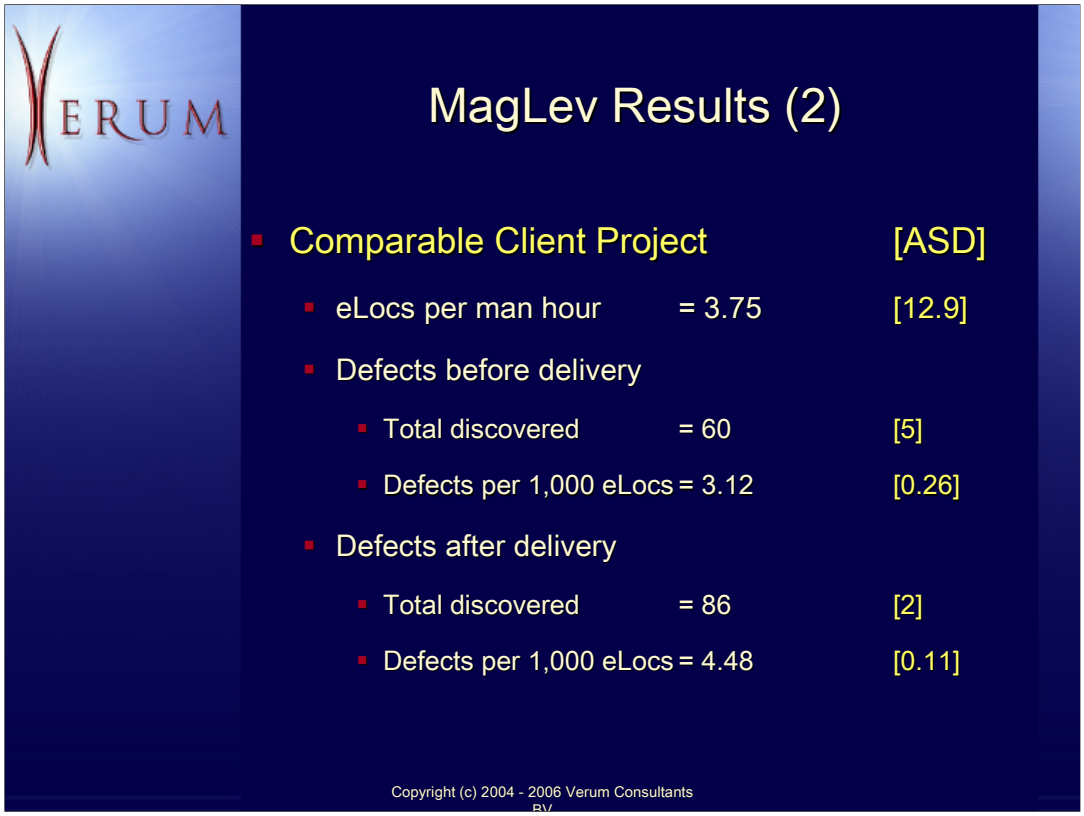
Because software enters testing with far fewer specification and design errors, testing can concentrate on detecting construction errors and those defects that we cannot easily verify mathematically.

Because we have eliminated the difficult, nondeterministic design errors such as deadlocks and race conditions before construction, the errors that remain will be more easily reproducible, quicker to detect by testing and quicker and cheaper to repair.

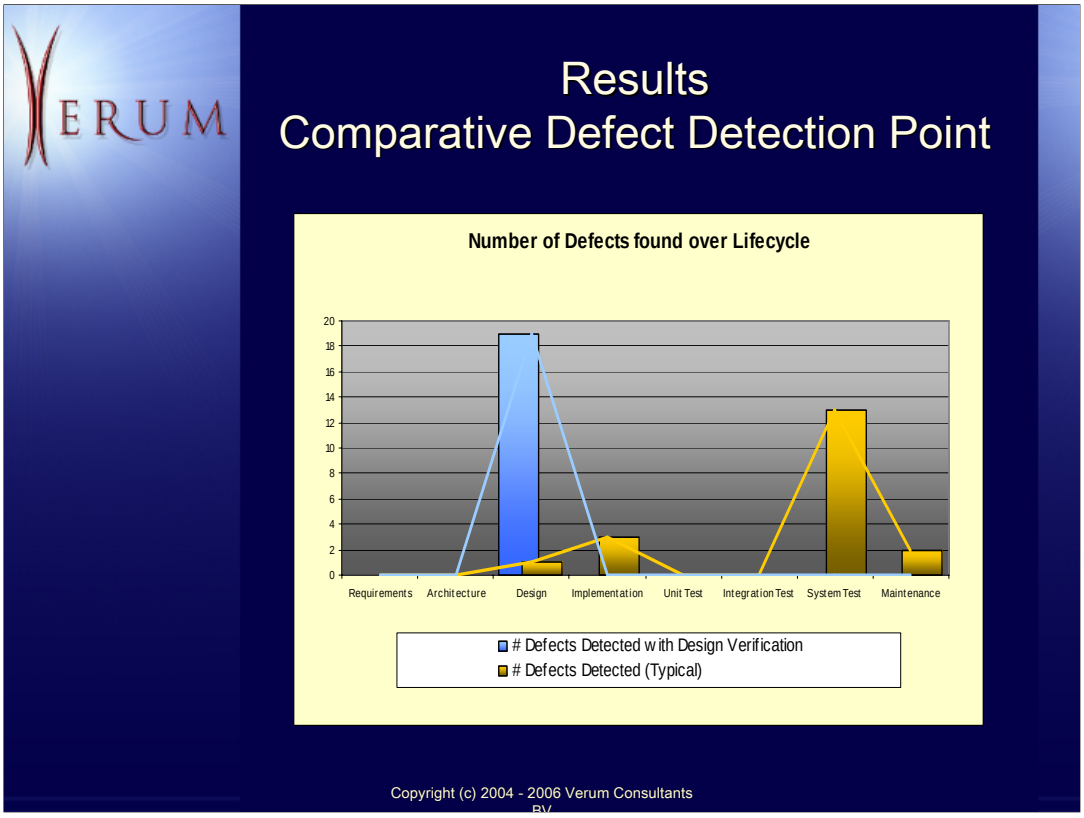
These are some of our customers. What follows are two case examples.



In this project, a joint project team from Verum and Philips Applied Technologies (Mechatronics) developed the control software for a new “stage”.



This is the comparison between the project performance and a comparable project selected by Philips Applied Technologies.



In this project, the goal was to verify a new software design being made by the customer's design team. After the project, the customer carried out an evaluation to answer the following question: assuming all the errors found by Verum would also have been found by the customer's own development process, where in the life-cycle would these faults have been found? This graphic shows the results of their analysis. Verum found all the faults during the design phase, before implementation. The customer would have founf some during that phase, a few more during implementation but most would have been found during system testing – a lengthy and expensive part of the life-cycle. Significantly, some major errors would not have been detected until after the product was delivered to the end users.

