Analytical Software Design

Guy H. Broadfoot [1]

Verum Consultants B.V.
The Netherlands

Philippa J. Hopcroft [2]

Oxford University Computing Laboratory
United Kingdom

Abstract

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 a new approach called Analytical Software Design that marries software engineering mathematics developed in academia and practical software development methods developed in industry. This approach enables the power of mathematically based techniques to be applied in a practical way within existing software development organisations. The use of Analytical Software Design is illustrated with an industrial case.

1 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

[1] Email: guy.broadfoot@verum.com
[2] Email: philippa.hopcroft@comlab.ox.ac.uk

time-to-market. For many businesses, software has become *business critical* and software development is a *strategic* business activity.

It has long been the case that products aimed at the professional or industrial user depend heavily on embedded software components for their function. Now, in addition, the consumer is being confronted by products whose safe and reliable operation depends increasingly on embedded software. This trend raises new issues of consumer protection and product liability and challenges the ability of these businesses to compete in their chosen markets.

At the same time, software development continues to suffer from poor predictability. Business managers want 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, recognising this situation, companies have invested heavily in software development process improvements, technology, infrastructure and training. In spite 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 [McG99]; 15% - 25% of software defects are delivered to customers [McG99]; in 2002, software failures cost the U.S. economy an estimated $59.5 billion [NIS02].

Existing development methods appear to have reached a quality ceiling that incremental improvements in process and technology are unlikely to breach. The challenges many businesses experience developing embedded software and being able to guarantee its quality and correct functioning is a testament to the limitation of current testing- centric software development practices. To break through this ceiling, a different approach is needed.

2 Business-critical and untestable software

The domain we are addressing in this paper is the development of *business-critical* and *untestable* software. Business-critical software has the following characteristics:

- It forms an essential part of some core product or service provided by a business to its customers;
- Software failure can have severe commercial, financial and/or legal consequences for the business and its customers;
- In safety critical domains, such as aerospace, automotive and medical systems, software failure can endanger human life;
- The software development becomes the critical path for product development.

It matters greatly to these companies and their customers that the software functions correctly and as intended; if the software fails, the product fails.

Many products and services upon which modern society depends could not exist without their embedded software components. In the manufacturing sector, for ex-

ample, the ASML Twin Scan wafer stepper has 12,500,000 lines of code embedded in it. At one point during its development, the software team was 350 strong. The latest generation AX Series component mounting machine from Assembléon has more than 500,000 lines of new code embedded in it. In the telecommunications sector, 10,000,000 lines of code in a digital trunk exchange is not uncommon.

Not only is software increasingly business-critical, it is also becoming increasingly complex. This is due to due to a number of technical and economic forces such as:

- Cheaper hardware and higher performance requirements are leading to the increasing use of multiple processor configurations, frequently in the form of locally distributed computing platforms. The Assembléon AX Series component mounter has 22 processors when fully configured, interconnected via an internal Ethernet network and a modern car has 30 or more software controlled Electronic Control Units (ECU) networked together via 10 or more different buses;
- Cost pressures are leading to increased use of standard 'off-the-shelf' hardware and software components, instead of specialised components. This frequently requires additional complex software to adapt these components to the product architecture and frequently introduces additional sources of performance problems and error;
- Event driven embedded software systems commonly comprise large sets of loosely coupled cooperating processes and threads that communicate via queues. Frequently, these processes are physically distributed among multiple processors. These characteristics often lead to design errors resulting in race conditions, deadlocks, and timing errors, all of which are not easily reproducible and are hugely expensive to detect and remove;
- Software development is increasingly carried out by development teams in different locations increasing the need for clear, rigorous and precise specifications.

In this paper, we describe such software systems as being *untestable* because there is no amount of testing that can prove correctness and no economically feasible amount of testing that can establish a sufficient level of confidence that the software system is correct.

Embedded software systems are both business-critical and untestable. Some of the business consequences of applying current testing-centric development practices to developing these systems include:

- Projects experience long and unpredictable integration and test periods;
- Business cases are damaged or destroyed by delayed product introduction due to delays in getting the software correct and stable;
- The software fails to comply fully with its specifications;
- Frequent failures occur after completion and deployment resulting in recall and repair actions and issues of product liability;
- Development costs overrun and sustaining costs are high.

Because conventional testing centred software development is failing to deliver correctly functioning business-critical and untestable software on time, or even at all in many cases, we must adopt an alternative and more formal approach based on the following two principles: (1) business-critical and untestable software must be based on architectures and designs that are *verifiably correct* before a single line of code is written; and (ii) software architects and designers must limit themselves to those architectures and designs that can be verified *using the currently available tools*.

3 Engineering and Mathematics

What distinguishes engineering from craftsmanship? One major characteristic is predictability of outcome. All branches of engineering, except software engineering, routinely employ mathematics to verify specifications and designs before starting implementation. Aircraft engineers use mathematical modelling to establish with certainty the structural integrity, flying characteristics and key performance and operating parameters of a new aircraft long before construction of a prototype begins. Structural engineers use mathematical modelling to prove the structural integrity of buildings and bridges before they build them. No architect charged with designing an earthquake proof building would build it and then wait for an earthquake to see if the design was correct; the design would be subject to rigorous and continuous mathematical analysis and modelling throughout the entire design process.

Why is this the case? Simply, the human, social and economic cost of crashing airplanes, collapsing bridges and unsafe buildings are considered unacceptable. These increasingly complex engineering challenges cannot be met in any other way than by applying mathematics. Indeed, the Merriam-Webster dictionary defines engineering as *"The application of science and mathematics by which the properties of matter and sources of energy in nature are made useful to people"*.

4 Software Engineering and Mathematics

Most software is not developed this way; except in safety critical domains where it may be mandated, mathematics are not routinely employed when specifying or designing software systems. As a consequence, we have only informal, review based methods to examine software designs and specifications for completion and correctness before investing in programming. We have no way of verifying designs for correctness in any formal and complete way. Software development methods rely almost exclusively on testing the implementation in order to determine the correctness of specifications and designs. When testing software, we must detect and remove specification errors, designs errors and implementation errors. As a branch of engineering, software engineering is unique in this approach.

Over the past 30 years or so, a number of formal methods have been developed in order to apply mathematical techniques to software development. For exam-

ple: formal state-based specification languages such as Z [Spi92]; process algebras designed to formally reason about concurrent systems from an event-based perspective, such as CSP [Hoa85,Ros98]; formal model checking approaches such as FDR [Ros94], that takes a machine readable version of CSP as input language, and Spin [Hol97], that uses a high level language to specify systems descriptions called PROMELA (a PROcess MEta LAnguage); and formal frameworks such as the B-method [Abr96] that encompass a breadth of formal notation and refinement methods with the aim of defining a formal transition from a specification through to an implementation. Yet, in spite of the need for them and the promise they hold, with the exception of those domains where the use of such methods is mandatory, formal methods are seldom found in the industrial software development organisations. Furthermore, many of those who have tried formal methods are not keen to repeat the experience. In the following sections we review the two principle reasons for this.

4.1 Starting points: Informal requirements versus formal specifications

A formal method must start with a formal specification; however, conventional practice in industry starts with compiling an *informal requirements specification*. This is typically a substantial document, running to hundreds or thousands of pages describing the required behaviour and characteristics of the software. It is prepared by critical project stakeholders such as business analysts, domain experts, requirements specialists and customers and reflects all of the knowledge and experience of the business about its product domain, customer base and competitors. It is a key part of the business case supporting the development and is written in business and domain specific terms. Furthermore, this document is not static; managing changes in requirements during software development is simply an industrial necessity. So the requirements specification is a living document that evolves over time.

To apply a formal method we must start by developing a formal specification from the informal requirements specification and during development we must be able to keep the formal specification synchronised with changes to the informal specification. Having done this, how do we verify that the formal specification describes the same system as the informal requirements specification?

Formal specifications are typically constructed in a specialised mathematical language and require the use of specialists with an extensive mathematical background and expertise in the method. In practice, it is rarely if ever the case that the business analysts and domain experts have that expertise or experience; in our experience, customers and end users never have this expertise. It is unrealistic (i) to require that business analysts, domain experts, requirements specialists, customers and end users present their requirements in a formal specification; (ii) to present these stakeholders with a formal specification that they cannot understand and expect them to validate it; (iii) to expect that such a specification will be accepted as the basis for a contractual agreement; or (iv) to require the stakeholders to be trained in formal methods. In short, the only people with the domain knowledge

necessary to validate the formal specification cannot do so because they do not understand it.

4.2 Disunity between abstract model and software system

Formal verification techniques are typically applied to reason about some abstract representation or model of parts or all of the actual system under development. Abstract models are, by definition, an abstraction of the actual system being developed; as a result, the verification draws conclusions regarding the 'correctness' (however that is defined) of the abstraction as opposed to the actual system's design or implementation.

Regardless of how effective a formal verification technique is, if we cannot define a clear mapping strategy between the abstract formal models being analysed and the actual system being developed, the benefits of the formal verification will remain limited. The lack of such a clearly defined relationship between the two poses a fundamental barrier to introducing formal methods in industry. By the very nature of the applications that need a more formal approach, the systems being developed are too complex to reason about confidently by hand (hence the need for formal models). The task of verifying that the formal models truly reflect the actual system is similarly complex.

By the very nature of the problem domain we are considering (complex business-critical software), the specialist knowledge required by the domain experts is extensive and application specific; to propose to retrain them at the start of a project to reach the necessary level of expertise in the chosen method is not practical. Developing scalable and accurate abstract models within a formal framework (together with the necessary analysis) requires specialist knowledge and experience.

It is necessary to reconcile the gap between the formal analysis being done on the abstract models and the software development process, such that the necessary feedback can flow accurately between the two in both directions.

5 Analytical Software Design

Analytical Software Design is a marriage between software engineering mathematics developed in academia and practical software development methods developed in industry. It overcomes the two problems mentioned in the previous section because (i) ASD specifications avoid complex mathematical notations and remain accessible to critical project stakeholders and (ii) the abstract mathematical models needed for formal verification and analysis are generated automatically from the ASD specifications.

ASD incorporates the following elements:

- It uses function theory and the Sequence-based Specification method [PP98,PP03] to specify functional performance requirements in the form of Black Box Relations and functional designs in the form of Black Box Functions. Although mathematically based, ASD specifications avoid the use of complex mathemati-

cal notations. They are fully 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.

- The Box Structured Development Method [Mil88,MLH86] is used to apply the principles of stepwise refinement to transform the Black Box Function design specification into a State Box Function specification 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 [Hoa85,Ros98] and can be analysed and verified using the model checker FDR [For03]. For example, we can use the model checker to verify (i) whether or not a design correctly complies with its functional requirements; (ii) whether or not the State Box coding specification specifies exactly the same behaviour as the Black Box design; and (iii) whether or not 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 ASD specifications already verified mathematically. Code generation may not be applicable to every project but in those cases where it is, significant development efficiency gains can be realised.
- 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 very large numbers of self-running test cases and analyses their results.

The use of mathematical models enables specifications and designs to be analysed and verified mathematically using the model checker. Because these models are generated automatically from the ASD specifications that are accessible to project stakeholders, ASD overcomes the problems described in the previous section. Mathematical modelling can be applied cost effectively within existing software development organisations without the need for extensive training in the modelling techniques.

5.1 Sequence-Based Specification Method

In ASD, the externally visible functional behaviour of a component is called the functional interface behaviour. This is the functional behaviour visible to the component's clients. At this level of abstraction, design and implementation details are not visible and as a result, the functional interface behaviour is almost always non-deterministic. In ASD, these specifications are in the form of a total mathematical

relation called a Black Box relation.

The functional design of a component is always deterministic; it specifies the component's behaviour in terms of its interactions with its clients and all other components it uses. In ASD, designs are specified in the form of a total mathematical function called a Black Box function.

The domain of the Black Box relation and Black Box function is the set of all possible sequences of input stimuli, including illegal sequences, and the range is the set of all possible system responses. The domain and range of the Black Box relation is restricted to the set of sequences and responses visible to clients via the component's interface. The domain and range of the Black Box function are the unions of the sets of sequences and responses visible at its client interface plus all interfaces via which it interacts with other components.

The method used to derive a Black Box function representing a design is the Sequence-Based Specification Method [PP03]. A generalisation of this method is used to derive the Black Box relation to specify a component's functional interface behaviour.

Both input stimuli and responses are abstractions representing all externally visible input and output actions. They can represent procedure calls, method calls, return values, events, interrupts or messages; in short, any normally used programming constructs to represent interface interactions.

ASD specifications form a vital bridge between the formal and informal worlds. Using the Sequence-based Specification Method, ASD specifications are systematically derived from the informal, natural language specifications written in domain specific terms as usually produced by conventional software development practices. ASD specifications retain complete traceability to the original informal specifications. The original terms and domain concepts are used and complex mathematical notations avoided. This enables the critical project stakeholders to establish by inspection that the ASD specification specifies the same behaviour as the informal specification. It also enables them to remain engaged and in control of the specification process.

The method requires the enumeration in length order of all possible input sequences of stimuli, including the empty sequence, and the assignment of the correct system response to each. Every mapping of an input sequence to a response is justified by explicit reference to the informal specifications [3]. During this process, it frequently occurs that choices are required concerning issues about which the informal specification is silent, ambiguous or inconsistent. In these cases, new requirements must be formulated and validated with the project stakeholders in order to resolve the issue. The informal requirements specifications are updated accordingly and references to these new requirements are used as the justification for the eventual mapping rules.

The importance of this procedure cannot be over emphasised: it forces specifi-

[3] According to current best practice, it helps greatly if the requirements in the informal specification are each identified by a unique tag.

cation issues to be resolved during the specification of the Black Box, rather than leaving it to the individual programmer to resolve during implementation according to his or her taste. It focuses attention on defect avoidance during requirements analysis instead of detecting defects later on. It plays a major role in preventing requirements issues 'leaking' through to subsequent development phases.

A Black Box relation and Black Box function must be total; therefore, we must include input sequences of stimuli that may be illegal or cannot happen. This is deliberate, since the method requires us to consider the entire state space of the program and determine the correct behaviour in each state.

The domain of the Black Box is infinite because the sequences of input stimuli allow stimuli to be repeated and there is no limit on sequence length. However, in most systems encountered in practice, unbounded input sequences do not imply infinite sequences of unique behaviour; real systems cycle between a finite number of externally observable states. This cyclic behaviour suggests that the infinitely long input sequences can be recursively defined, avoiding the need for endless enumerations. This is indeed the case, making Black Box relations and functions total.

The Sequence-Based Specification Method partitions the infinite set of sequences of input stimuli into a finite set of equivalence classes. Two sequences are equivalent if all possible (nonempty) extensions of them result in exactly the same future system behaviour. Each equivalence class is characterised by the sequence with the minimal length, known as the *canonical sequence*; if there is more than one minimal length sequence, we pick the one enumerated first as the canonical sequence. The empty sequence with length zero is by definition a canonical sequence.

The results of the sequence enumeration are presented in tabular form, there being one such table for each equivalence class and thus Canonical Sequence. There is a row in each table for each input stimulus and each row describes a Black Box rule that: (i) defines the new set of sequences formed by extending the Canonical Sequence (and thus every sequence in the equivalence class) by the corresponding stimulus; (ii) maps this newly formed set of sequences onto a system response; and (iii) identifies the equivalence class to which every sequence in the newly formed set belongs. If this equivalence class already exists, then all future extensions of the set of newly formed sequences have already been defined and represent system behaviour already analysed; we do not need to consider them further in our analysis. Otherwise, we have identified a new equivalence class that must be defined by its own table and set of Black Box rules. When all equivalence classes have been identified and defined, the Sequence Enumeration is *complete*.

ASD includes an ASD Editor to support this process and generate the ASD specifications in a machine readable form from which state transition graphs and mathematical models are generated automatically.

5.2 Box Structure Development Method

The Box Structure Development Method defines three views of the software system, referred to as the Black Box, State Box and Clear Box views (and described

below). These views form an abstraction hierarchy that allows for stepwise refinement and verification as each view is derived from the previous. This should not be taken as suggesting that refinement is performed in exactly three steps, regardless of system size or complexity; rather it means that each refinement cycle should be accomplished by defining a Black Box, refining this into a State Box and then refining the State Box into a Clear Box.

Black Box View

A Black Box specifies the deterministic functional design of a system or component in terms of its visible interactions with its environment. By environment, we mean all other systems or components, hardware or software, with which the component must interact. As described above, a Black Box design is specified by its Black Box function. This is a total mathematical function from the set of all possible sequences of input stimuli to the set of responses. A Black Box has neither state nor control flow in its description. According to Mills [MLH86], all systems of every kind demonstrate Black Box behaviour.

State Box View

A State Box is derived from a completed Black Box specification. The sequences of input stimuli are replaced with state variables that capture the information encapsulated in the input sequences. As there is more than one possible State Box for a given Black Box, it is for the software designer to determine the appropriate one and show that this is equivalent to the Black Box specification.

Within ASD, this verification step is performed automatically. The ASD Model Generator is used to translate both the Black Box and the State Box into CSP and the model checker is used to verify that they are equivalent.

Clear Box View

The State Box is refined into a Clear Box, the least abstract view of the system at the current refinement step: logic is added to it to capture the transformation from old to new state and the method by which the system responses are generated. For many cycles through the refinement process, the logic of a Clear Box is expressed as a composition of newly defined Black Boxes. BSDM rules restrict the ways in which Black Boxes can be composed to form the logic of a Clear Box in a manner analogous to the Structured Programming constructs of Mills [LMW79]. The refinement cycle finishes when all Clear Boxes are specified as code. As is the case for the State Box, there are many Clear Boxes that implement a given State Box. It is the task of the software designer to design the appropriate one and show that it correctly refines the State Box.

Within ASD, verifying that program code correctly implements a State Box is addressed in three ways:

- Significant amounts of code can be generated automatically from the State box specifications, reducing the amount of manually programmed code that has to be checked;

- Large numbers of self-running statistically selected test cases are generated automatically from the State Box specifications;
- The State Box forms a very clear coding specification. State Boxes may be optimised and transformed mathematically if needed to make coding more convenient and easier to compare by inspection with the State Box. Each new State Box can be verified automatically by generating mathematical models and applying the model checker.

5.3 CSP and the FDR model checker

CSP [Hoa85] is a process algebra for describing concurrent processes that interact with one another or their environment through some form of communication. CSP has a rich expressive language and a collection of semantic models for reasoning about process behaviour. What follows is a brief overview of the notation and semantic models; see [Ros98] for further details.

A process is defined in terms of synchronous, atomic communications, known as *events*, that it can perform with its environment. For example, process $a \rightarrow P$ can initially perform event a and then act like process P. We write $?x : A \rightarrow P(x)$ (prefix choice construct) to denote the process that is willing to perform any event in A and then behave like process $P(x)$.

Channels carry sets of events; for example, $c.5$ is an event of channel c. The process $c?x \rightarrow P_x$ inputs a value x from channel c and then acts like P_x. The process $d\,.\,x \rightarrow Q$ performs the event $d\,.\,v$, where v is the value assigned to x, and then acts like process Q.

In CSP, there are *external* and *internal* choice operators. $P \,\Box\, Q$ denotes an external choice between P and Q; the initial events of both processes are offered to the environment; when an event is performed, the choice is resolved. $P \sqcap Q$ denotes an internal (or nondeterministic) choice between P and Q; the process can act like either P or Q, with the choice being made according to some criteria that we do not model.

Processes can be placed in parallel, where they synchronise upon (all or specified) common events or interleaved, where they run independently of one another.

CSP defines three semantic models for formally reasoning about the behaviour of such systems. The simplest of these, known as the *traces* model, captures the visible events that a process can communicate: A *trace* of some process P is a sequence of visible events that P can perform and $traces(P)$ is the set of all possible traces of P. The *failures* model extends the traces model to capture nondeterminism. The *failures-divergences* model extends the failures model to capture divergence.

Correctness conditions are typically formulated in terms of *refinement* within the CSP framework: A process *Spec* is refined by another process *Impl*, precisely when all *behaviours* of *Impl* are also defined to be possible behaviours of *Spec*. Hence *Spec* specifies the correctness conditions to be satisfied by *Impl* and *Impl* represents the system being verified. The precise meaning of behaviour here is

dependent upon the semantic model used. For example, in the traces model, one can only capture safety properties: By considering the traces of a process *P* alone, one can only reason about what events *P* is able to perform; no details are captured as to what events *P* can refuse. Hence this model is unable to capture the notion of nondeterminism. The more advanced semantic models, namely the failures and failures-divergences models, capture the refusal information and are therefore able to express liveness properties and nondeterminism, in addition to any safety properties.

Refinement in CSP is transitive and all CSP operators are monotonic with respect to it; this in turn enables compositional development. This means that a design can be verified against models of the external interface behaviour of other, used components, without requiring their designs to be modelled. This is important in practice for a number of reasons such as: (i) The designs of the other components may not be available to us as ASD design specifications, perhaps because the components are bought-in, or were designed without using ASD, or perhaps the deigns have not yet been made. This makes it difficult to produce the necessary CSP models; (ii) CSP models of external interface behaviour are typically many times smaller than the models of a design implementing the interface. This enables components of very large systems to be modelled.

One of the characteristics that makes this framework a good choice within an industrial setting is its extensive tool support. FDR is a mature model checker for CSP and a commercial product of Formal Systems Ltd. [For03]. It provides fully automated verification of refinement in all semantic models, determinism, deadlock freedom and livelock freedom. If such a check fails, then counter-examples are generated with extensive support for analysing them (from top level down to the individual component behaviour). There are optimisation techniques available to enhance scalability in practice.

5.4 Practical strengths of ASD

ASD has a number of significant strengths that lead to defect prevention:

- ASD specifications are understandable to critical project stakeholders who have not been specially trained in the method. Experience shows that critical project stakeholders can and do participate enthusiastically in formal inspections and can participate in verifying them against the informal requirements specification.
- The Sequence-Based Specification Method forces every possible execution scenario to be considered. This technique is particularly effective in defining the response of event driven software components to all possible input events under all possible circumstances and combinations.
- The Sequence-Based Specification Method results in complete traceability between the resulting formal specification and the informal requirements specification.
- Specification issues are addressed early; correct use of the Sequence-Based Spec-

ification Method results in specification issues being addressed before designs and implementations are made and forces them to be resolved by the critical project stakeholders rather than the designers and programmers later in the cycle.

- The Box Structure Development Method applies stepwise refinement in order to move from specifications to implementation in verifiable steps and preserves traceability between program code and ASD specifications.
- The ASD specifications are sufficiently formal and rigorous so as to enable the automatic generation of the mathematical models used for verifying designs and specifications, running code and statistical test cases.
- Mathematical models of a component's design can be combined with models of the interface behaviour of the other components it uses so that using FDR, we can formally analyse the design behaviour in combination with the components it uses. Note that we do not need the designs of the used components in order to do this; models of their external interface behaviour are sufficient. This is a major reason why this approach scales to industrial-sized systems.
- CSP is able to represent nondeterminism. This is particularly important when modelling specifications as opposed to implementations. For example, we are easily able to model that an implementation may or may not generate error events without considering the detailed circumstances in which errors can be detected.
- FDR is a mature and efficient finite state model checker able to handle industrial-sized models. With FDR, we can check specifications for safety and liveness conditions and determine whether a proposed implementation correctly refines the specification. We can also check whether an implementation deadlocks, diverges or is nondeterministic.
- CSP and FDR are both mature and have been the subject of extensive academic research. As a result, many modelling and state space reduction techniques have been (and still are being) developed to optimise the formal verification process.

6 An industrial case

In this section, we present a practical case where ASD was applied within the setting of an existing software development environment. The aim was to develop some key parts of the equipment control software embedded in a complex manufacturing machine.

6.1 Assembléon AX Series Component Mounter

Assembléon is a company in the Netherlands that develops and manufactures machines that place components on PCBs; the AX Series is their latest generation and most technologically advanced machine to date. It makes in excess of 100,000 placements per hour with an accuracy of plus or minus 25 microns. The embedded software consists of more than 500,000 lines of new code. Software execution

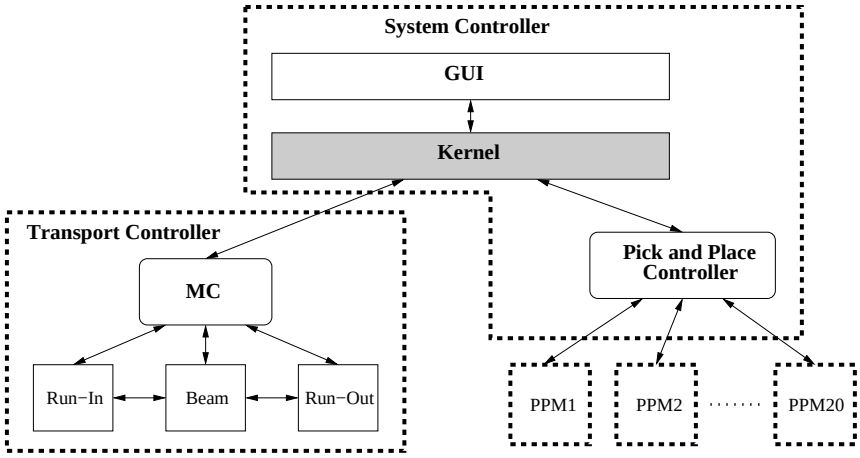

Fig. 1. Assembléon AX Series Architecture.

is distributed between 22 processors, loosely coupled via an internal Ethernet network. Fully configured, the AX has 20 placement modules working in parallel, each of which is controlled by its own processor and consists of a placement robot, component handling hardware, component laser alignment unit and on-board digital vision.

PCBs are moved through the machine by means of the Transport System controlled by its own dedicated processor. The Transport System holds the PCBs and positions them with sufficient accuracy to enable the Placement Modules to place components.

Figure 1 shows an overview of the AX structure. The individual boxes represent processes and the dotted boundary lines represent the grouping of processes onto hardware processors. The System Controller consists of a set of processes, of which the following three are shown: (i) GUI - Graphical User Interface providing the interface to the machine operator; (ii) Kernel - responsible for coordinating the activities of the Transport Controller and the Placement Modules to ensure correct and safe machine behaviour; and (iii) Pick and Place Controller - responsible for supervising the individual Placement Modules. The boxes labelled PPM1 through PPM20 represent the placement modules, each controlled by its own dedicated processor.

The arrows represent the interactions between these processes. Each process receives commands via a FIFO queue (not shown) and passes commands and data to the other processes by placing them into the receiving processes queue.

6.2 Applying ASD

ASD was used to develop the Kernel component; this was seen by the project team to be a critical component whose design had proved to be problematic on a previous generation machines. The Kernel's *environment* consists of the GUI, Transport Controller and the Pick and Place Controller. The first step was to apply the Sequence-Based Specification process to develop the ASD specifications of the ex-

ternal interface behaviour of the Kernel, the Transport Controller and the Pick and Place Controller to serve as a formal specification of the operational semantics of the interfaces between them and the Kernel.

When this work started, these interfaces were already specified using a combination of informal specifications to explain the operational semantics and interface definition files to describe the syntax and the implementation of both the Transport Controller and the Pick and Place Controller was already well advanced, although not using ASD. The corresponding designs and interface specifications had been subject to a peer review process. In spite of this, while analysing the existing interface specifications and making the ASD specifications to formally specify these interfaces in the form of the Black Box relations, a significant number of major specification issues arose. These were interface behaviour issues that were ambiguous, inconsistent or not addressed in the informal specification and most of them were related to race conditions, deadlocks and nondeterminism arising from the use of queues between the processes. The errors resulting from such design flaws are of a type notoriously difficult to find and solve by testing due to their apparent random occurrence and non-repeatability. As these issues were identified and resolved, the informal specifications were updated accordingly and references to these were used to justify the Black Box rules. The Transport Controller and Pick and Place Controller code was also updated.

The Kernel design partitioned the Kernel into three layers, each of which would be implemented as a separate process with its own input queue. The bottom layer called the *Module Interface Adaptor* exists in order to reduce the complexity of the input sequences sent by Transport and Pick and Place to the Kernel. Both Controllers are multi-threaded and therefore streams of events generated by different threads within the Controllers arrive at the Kernel's input queue arbitrarily and unpredictably interleaved. This could result, for example, in an event generated while a Controller was in a given state being seen by the Kernel after the event signalling exit from that state, leading the Kernel to interpret the event in the wrong context. The Module Interface Adaptor simplifies the design of the other two Kernel processes by imposing a deterministic ordering on the stream of events as they are seen by the other processes. It guarantees that events generated by one of the controllers in a specific state are always seen by the other Kernel processes before the event signalling an exit from that state.

The middle layer is called the *Machine Control Layer* and it implements the basic operational behaviour of the machine. Its main function is to guarantee correctly coordinated behaviour between the Transport Controller and the Pick and Place Controller to ensure machine consistency and safety. The top layer, called the *Machine State Layer*, implements the overall machine state behaviour as it appears to the GUI and thus the machine operator.

Using ASD, we made a complete formal specifications of the operational semantics of the GUI, Transport Controller, and the Pick and Place Controller as visible at their interfaces. The Sequence-Based Specification Method exposed all the under-specified, ambiguous and/or inconsistent issues in the original informal

interface specifications. Without a formal approach, such problems arise frequently in practice when dealing with most informal interface specifications, because they tend to concentrate on the syntactic details of the interface (for example, method names and parameter types); this leads to incomplete and ambiguous descriptions of the operational semantics. ASD is particularly strong in specifying behaviour with great precision.

From the sequence-based specifications, we derived the corresponding design Black Boxes and State Boxes according to the approach described in [PTLP98]. We were able to generate all of the control flow code automatically for each of the layers from the State Box tables; the generated code implemented a Mealy machine that responded correctly to each input stimulus and performed the state transition. The code that implemented the responses and other internal logic was programmed by hand.

The ASD specifications provided explicit traceability to the original informal requirements and were completely understandable both by the software engineers from the other development teams and by the domain experts, all of whom participated in the inspection process to validate them. Instead of being excluded from the process, the critical Project Stakeholders played a vital role in verifying the specifications.

6.3 Formal analysis using CSP/FDR

CSP models were generated automatically from the ASD specifications of the ASD design specifications of the Kernel and the ASD interface specifications of the Kernel, Transport Controller and the Pick and Place Controller. The following formal verification was carried out using FDR:

Verifying the Kernel Black Box Behaviour

Using the generated CSP models plus the model checker FDR, we verified that: (i) All three Kernel Layers are deterministic; (ii) The Machine State Layer behaves as expected by the GUI (that is, it complies with its interface specification) and the GUI and Machine State Layer cannot deadlock each other; (iii) The Machine State Layer behaves as expected by the Machine Control Layer. That is, there are no sequences of stimuli from either layer that lead the other to an illegal state; (iv) The Machine State Layer and the Machine Control Layer cannot deadlock each other; (v) The Machine Control Layer behaves as expected by the Module Interface Adaptor and they cannot deadlock each other; (vi) The Module Interface Adaptor behaves as expected by the Transport Controller and the Pick and Place Controller and cannot deadlock with each other.

Verifying the Kernel State Boxes against the Black Box

Given the CSP models of a Black Box Function and a State Box Function, we used FDR to verify that the State Box has exactly the same operational semantics as the Black Box.

6.4 Results

The Kernel Machine State Layer has 2,835 Black Box rules and 47 Canonical Sequences, the longest of which is 11. The Kernel Machine Control Layer has 837 Black Box rules and 23 Canonical Sequences, the longest of which is 6. The ASD analysis and CSP modelling for verifying the design took about 12 man-weeks over a 6 week period. The coding was completed in 4 man weeks by 2 people and was partly done by hand and partly generated directly from the ASD specifications.

The Kernel consists of 20,000 lines of C++ code, about two thirds of which was generated automatically from the ASD specification. On its first test on the machine with the Transport and the Pick and Place Controllers, it executed correctly. The integration took a few hours. This contrasts significantly with the experience of an earlier project to develop a working "proof of concept" model. It took several months to integrate a prototype Kernel with early versions of the Transport and the Pick and Place Controllers.

This dramatic reduction in the time needed to integrate the production Kernel with the other two controllers came from two sources: Firstly, modelling the interfaces to the Transport Controller and the Pick and Place Controller exposed many important interface issues. Secondly, using the CSP models generated from the ASD specifications, we formally verified that the Kernel's design satisfied the following: (i) the Kernel behaves correctly and according to the interfaces with the Transport Controller and the Pick and Place Controller under all possible inputs; (ii) the Kernel behaves according to the interface agreed between it and the GUI, and would react as specified to all possible inputs; and (iii) the interactions of the three internal processes within the Kernel implement the specified behaviour.

Most importantly, using these techniques we are assured that the correctness of the Kernel is independent of timing issues [4] . This assures us that changing to faster hardware in the future will not break the Kernel. It also assures us that running test versions of the system with various logging and tracing tools activated will have no effect on the logical correctness of the Kernel's behaviour due to altering the relative execution speeds of the processes.

In 12 months of intensive use, a total of 8 minor defects were found in the Kernel, all of which were simple coding errors and easy to reproduce and fix. No difficult design errors related to race conditions or timing errors have occurred. The total amount of rework to date is less than 2%.

7 Conclusions

In Section 2, we reflected that conventional testing centred software development, with its emphasis on defect *detection* and *removal*, is failing in practice to deliver

[4] By timing issues, we do not mean hard, real-time response times. In the AX, dedicated programmable controllers handle real-time issues. The timing issues we are referring to are those issues where the logical correctness of the system depends on accidental relative execution speeds of the various processes. Such defects are common in the embedded software world.

correctly functioning business-critical and untestable software on time and with required quality. We stated the need focus on defect *prevention* by adopting an alternative and more formal approach embodying the following two principles: (i) business-critical and untestable software must be based on designs that are *verifiably correct* before a single line of code is written; and (ii) software architects and designers must limit themselves to those designs and patterns that can be verified correct *using the currently available tools*.

ASD is a practical means of addressing the current difficulties. It provides a formal framework in which one can develop organised, accessible and concise specifications that are expressed in domain terms and have explicit traceability to the original informal requirements. Consequently, it is accessible to software engineers without extensive training in the technique and remains understandable to the critical Project Stakeholders. Instead of being excluded by the use of formal methods, they become a key part of the validation and verification process.

We have shown that ASD is a practical way to introduce formal methods into software development on an industrial scale. Additionally, limiting designs to those that are verifiably correct does not prevent the development of industrial scale embedded software systems.

Having seen the effectiveness of ASD as applied to the AX Series Kernel software, the case-study Project Management decided to redesign and re-implement the Exception Handling Component using the same technique. This component had been the source of many defects reported during development and testing and had reached the steady state in which new defects were being discovered at about the same rate as others were being solved. The resulting new version of the Exception Handler has since been defect free.

We have also used ASD to recover designs and specifications for other components already written and proving error prone. Black Box behaviour was recovered by analysing the existing designs and implementation and generating the corresponding ASD specifications. CSP models were generated from these and verified using FDR. This approach proved very useful as a means of tracking down design flaws in existing software.

An important benefit of being able to generate CSP models from ASD specifications is that we are able to test that the specifications have certain properties. For example, the operational semantics of each of the two AX Controllers are described by a minimal Mealy machine with nine states; thus there are 81 possible combinations of states. Machine consistency rules can be formulated in terms of which state combinations are valid and which are not. Furthermore, there are mechanical rules that determine the order in which operations are carried out to start and stop the machine; violating these causes expensive mechanical damage. These safety rules can be defined in terms of allowed sequences of legal state combinations. We can express these rules in CSP and use FDR to perform refinement checks to verify whether they are satisfied by our specifications and designs.

Acknowledgements

We are grateful to Assembléon [5] for allowing us to present the case study and for their support when we applied these techniques to develop critical parts of the AX software.

References

[Abr96] J. R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1*. Wiley, 1996.

[Bro01] G. H. Broadfoot. Using CSP to support the Cleanroom Development Method for software development. Master's thesis, University of Oxford, 2001.

[For03] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 2003. See http://www.fsel.com.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.

[McG99] Thomas McGibbon. A business case for software process improvement revised. Technical report, Data & Analysis Center for Software, 1999.

[MDL87] H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom Software Engineering. *IEEE Software*, pages 19–25, 1987.

[Mil88] H. D. Mills. Stepwise refinement and verification in box structured systems. *Computer*, 21(6):23–26, 1988.

[MLH86] H. D. Mills, R. C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design*. Academic Press, 1986.

[NIS02] The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology NIST, US Department of Commerce, 2002.

[PP98] S. J. Prowell and J. H. Poore. Sequence-based software specification of deterministic systems. *Software - Practice and Experience*, 23(3):329–344, 1998.

[5] Assembléon Netherlands B.V., PO Box 80067, 5600 KA Eindhoven, The Netherlands.

[PP03] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions of Software Engineering*, 29(5):417–429, 2003.

[PTLP98] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering - Technology and Process*. Addison-Wesley, 1998.

[Ros94] A. W. Roscoe. Model-checking csp. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[Spi92] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall International, 1992.