Guy H. Broadfoot,
Chief Technical Director,
Verum Consultants.

Introducing Formal Methods into Industry using Cleanroom and CSP

In this paper, we present an overview of our observations and experiences of applying formal methods in an industrial setting and incorporating them into a practical software development process. This work has developed from an ongoing collaboration between the two authors coming from opposite ends of the spectrum and sharing a mutual interest in bridging the gap between academic research in formal methods and their current lack of use in the software industry. We explore a number of observations as to why, despite their need in industry and their strong presence in academia, formal methods are not widely exploited in practice. The problem we are interested in is the use of formal methods to develop software systems of a business-critical and untestable nature, where the software forms an essential part of some core product or service offered by a business. We argue that the successful implementation of such systems needs a more formal approach. We present a brief overview of our use of two formal approaches, Cleanroom Software Engineering Method [MDL87] and CSP [Hoa85, Ros98] (together with its model checker FDR [Ros94]) and how they can be applied in practice, together with a summary of results achieved on a business-critical industrial project so far.

1. INTRODUCTION

For increasing numbers of businesses, software development is a strategic issue. Software is embedded as a key component in an increasing number of products. It has been the case for some time that products aimed at the professional or industrial user depend heavily on embedded software components for their function. Increasingly, products aimed at the general consumer are also becoming reliant on embedded software. This latter trend raises new issues of consumer protection and product liability and challenges the ability of these businesses to compete in their chosen markets.

When asked what they perceive to be the major problems with software development, most business leaders would list just three problems:

- The cost is unpredictable and always much higher than expected;
- The development time is unpredictable and much longer than expected;
- The resulting product quality is unpredictable and generally much too low.

In short, software development, which is an activity of strategic business importance, is not under control. And we know this to be true: 40% - 50% of total development costs are wasted 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 annually, or about 0.6 percent of the gross domestic product, according to a study commissioned by the Department of Commerce's National Institute of Standards and Technology [NIS02].

This lack of predictability and the poor product quality is an immense and growing business problem with increasingly severe business consequences.

The difficulties such businesses are having in developing embedded software and being able to guarantee its quality and correct functioning is a testament to the limitation of current testingcentric software development practices. It is our conviction that in this domain at least, the only solution is to adopt a more formal approach.

In this paper, we present an overview of our observations and experiences of applying formal methods in an industrial setting and incorporating them into a practical software development process. This work has developed from an ongoing collaboration between the two authors coming from opposite ends of the spectrum and sharing a mutual interest in bridging the gap between academic research in formal methods and their current lack of use in the software industry. We explore a number of observations as to why, despite their need in industry and their strong presence in academia, formal methods are not widely exploited in practice.

The problem we are interested in is the use of formal methods to develop software systems of a business-critical and untestable nature, where the software forms an essential part of some core product or service offered by a business. In Section 2, we introduce this problem domain and argue that the successful implementation of such systems needs a more formal approach.

We discuss some of the barriers we encountered when introducing formal techniques into industrial projects in Sections 3. In Section 4, we present an overview of our use of two formal approaches, Cleanroom Software Engineering Method [MDL87] and CSP [Hoa85, Ros98] (together with its model checker FDR [Ros94]) and discuss how their combination can be applied in practice. We give a summary of their application within an industrial case-study in Section 5 and present our conclusions in Section 6.

2. DOMAIN: BUSINESS-CRITICAL AND UNTESTABLE SOFTWARE

The domain in which we have applied formal methods is the development of business-critical and untestable software. Business-critical software has the following characteristics:

1. It forms an essential part of some core product or service provided by a business to its customers;
2. Software failure can have severe commercial, financial and/or legal consequences for the business and its customers;
3. In safety critical domains, such as aerospace, automotive and medical systems, software failure can endanger human life; and
4. 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 example, 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 500,000 lines of new code embedded in it.

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:

1. 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;
2. 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; and
3. Event driven embedded software systems commonly comprise large sets of loosely coupled cooperating processes and threads that communicate via queues. These characteristics often lead to design errors resulting in race conditions, deadlocks, and timing errors, all of which are not easily or predictably reproducible and are hugely expensive to detect and remove.

In this paper, we describe to such software systems as being untestable because there is no 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:

1. Projects experience long and unpredictable integration and test periods;
2. Business cases are damaged or destroyed by delayed product introduction due to delays in getting the software correct and stable;
3. The software fails to comply fully with its specifications;
4. Frequent failures occur after completion and deployment resulting in recall and repair actions and issues of product liability;
5. 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. BARRIERS TO INTRODUCING FORMAL METHODS

There have been numerous formal methods developed over the years in academia, for example: 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 not widely encountered in the industrial workplace for developing software. Furthermore, it is our experience that many of those who have tried formal methods are not keen to repeat the experience.

Given that the need (within our chosen domain) is great, why are formal methods not being exploited more widely in industry? The existing gap between academic research and industry is well known and reasons usually cited for this include: lack of scalability, limited accessibility to non-specialists and immaturity of tools and techniques. However, if one compares the

enthusiasm with which the software industry adopted higher-level languages the late 1950's and 1960's with the apparent disinterest shown towards formal methods, the reasons usually cited fail to convince. The software industry is not reluctant to adopt new tools and techniques and is not reluctant to train its people in new techniques. Early compilers were generally poor, but the economic advantages of high level languages were so compelling that industry adopted them anyway and drove forward the development of better compilers.

So what is the explanation? We highlight some of our observations based on our experiences over the last year of applying formal techniques in industry. We make a distinction between issues arising at a business level involving cost and economic incentives, and those arising at a more technical level regarding the problems faced with integrating a formal methodology into a software development project.

3.1. Change is costly and disruptive

Organisational change of any kind is costly and disruptive. The cost is usually immediate and quantifiable; the potential benefits occur in the future, if at all, and are largely aspirational. Most businesses will consider formal methods only if the problems they face are sufficiently important and unlikely to be solved by existing methods. In short:

The software must be an essential part of some product that is core to the business and vital to its success;

- Past experience suggests that existing, testing-centric methods are unlikely to be successful;
- The promised benefits can be realised within the existing development environment by many of the people currently employed;
- The proposed method does not exclude from the development process those critical project stakeholders who are not formal methods experts, such as business analysts, domain experts and customers.

This last point is crucial. Critical project stakeholders will not understand the difficult mathematical concepts and notations many formal methods use to specify a software system. But because all of the essential product domain knowledge is known by these people, they must be able to participate in verifying the formal specification. If they are excluded from the verification process, this responsibility falls on those who are expert in the formal method being used, but who do not have the product knowledge necessary to perform this task. Also, they would be verifying their own work; independent verification would be impossible.

3.2. Testing-centric development practices

Most current software development practice is testing-centric; that is, it emphasises defect detection and removal. There is a genuine belief in industry that software defects are a fact of life and that the key to improvement is to detect and remove more defects, more quickly and more cheaply. This, of course, ultimately means testing, the economically least efficient way to address product quality and the way guaranteed to maximise avoidable rework. In proposing the use of formal methods, we are asking for a shift in emphasis from defect detection and removal to defect prevention. This is considered normal in just about every other branch of engineering and in manufacturing, but requires a large paradigm shift within software development.

3.3. 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 system. 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 to (i) require that business analysts, domain experts, Requirements Specialists, Customer and end users present their requirements in a formal specification; (ii) present these stakeholders with a formal specification that they cannot understand and expect them to verify it; (iii) expect that such a specification will be accepted as the basis for a contractual agreement; or (iv) require the stakeholders to be trained in formal methods. In short, the only people with the domain knowledge necessary to verify the formal specification cannot do so because they do not understand it.

3.4. 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" (whatever that is defined to be) 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). Similarly, verifying that the the formal models truly reflect the actual system suffers from the same problem.

We have used formal methods to model existing software components causing problems that could not be identified or reproduced using conventional testing. A formal methods expert with little specific product knowledge can reverse engineer a formal model from informal requirements specification, design specifications and program code, plus extensive discussions with the developers.

Depending on the skill and practical experience of the formal methods expert, the model may reveal root causes of known problems and discover new problems that were previously unknown. The following questions then arise: How does one verify the model itself against the actual system design and implementation? Do we have to rely entirely on the interpretation of the formal methods expert? While his expertise lies in the formal model, it is unlikely to extend into the knowledge required and known by the various domain experts. How do the domain and formal methods experts communicate with one another to provide critical input into the abstract model and the actual system? We have encountered this problem repeatedly when bringing in a formal methods expert to analyse a complex component.

This problem has frequently been classed as an "education" issue: if the domain experts were trained in the necessary formal methods (or a suitable background enabling access to the method with relative ease), then this problem would be resolved. In practice, however, this is unrealistic. As projects change and evolve, so do the domain experts (for example, the domain experts required for the development of medical technology will differ from those needed for aircraft design). Furthermore, one may choose to use different formal frameworks depending upon the characteristics of the system being developed. By the very nature of the problem domain we are considering (business-critical and untestable software systems), 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 equates essentially to an additional domain expert within the project team.

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.

4. A PROPOSED SOLUTION

We have started to address the barriers described in Section 3 by combining two complimentary formal methods, Cleanroom Software Engineering [MDL87] and the CSP/FDR framework [Hoa85, Ros98, Ros94]. Cleanroom provides us with a means of translating the informal requirement specification into a formal specification that is nevertheless accessible to the critical project stakeholders. It also provides us with a means for performing systematic stepwise refinement to derive designs and implementations from the formal specifications. CSP/FDR provides us with the means to model the Cleanroom specifications and designs and verify them for correctness. It also enables us to verify automatically every refinement step.

4.1. Cleanroom Software Engineering

The term "Cleanroom" has been taken from the semiconductor manufacturing industry where it is used to describe manufacturing processes that avoid injecting faults into the finished product via chemical and other contamination present in the atmosphere. It was chosen to emphasise the shift from defect detection and removal to defect prevention. As applied to software development, it has come to represent any software development process that incorporates the following principles:

(i) software developers can and should strive to produce software that is nearly error free when entering testing and (ii) the purpose of testing is quality measurement and not an attempt to "test in" quality.

As Dijkstra famously observed, testing demonstrates the presence of errors and not their absence.

The Cleanroom approach was first used in the early 1980s [MDL87] at IBM's Federal Systems Division. This division is responsible for space and defence systems where the cost of software failures are measured in millions of dollars and possibly human lives. It was an initiative of Dr Mills who observed the quality gap between the hardware and software components of a system. Cleanroom Software Development has since become a general term that encompasses a variety of different practices; we use the term Cleanroom to refer specifically to the version described in [PTLP98]. Cleanroom encompasses the following:

(i) Sequence-Based Software Specification [PP98, PP03] to establish a formal specification of the software system to be built; (ii) Box Structure Development Method [Mil88, MLH86] as a vehicle for applying

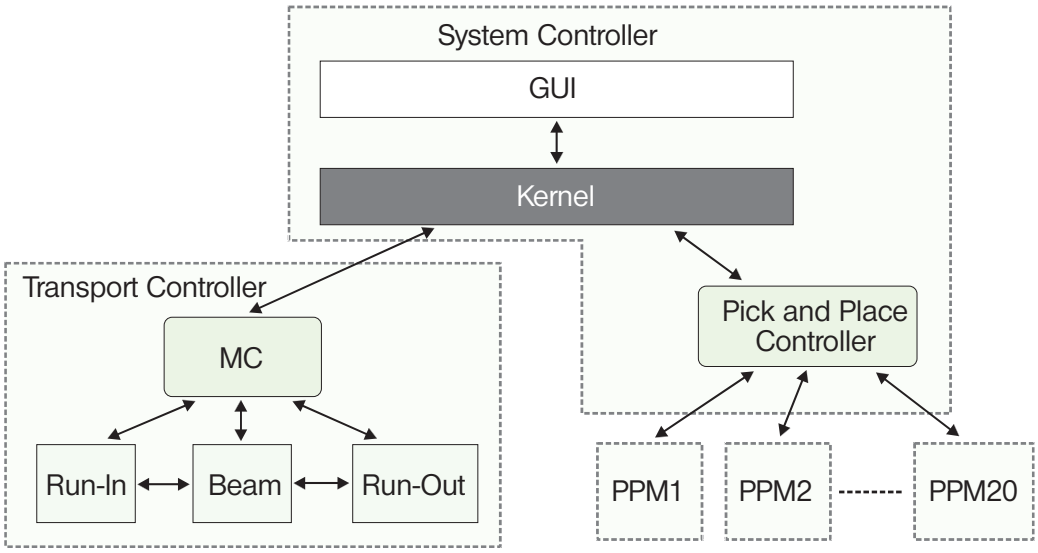

Figure 1. Assembléon AX Series Architecture.

systematic stepwise refinement from the formal specification to implementation; (iii) Statistical Testing based on usage models derived directly from the formal specifications; and (iv) Incremental Development where progress towards completion is accomplished in a series of predefined releases, each of which defines a growing subset of complete functionality towards the final system. In this paper, we concentrate only on the Box Structure Development Method and the Sequence-Based Software Specification Method. For a complete description of the Cleanroom Method, see [PTLP98].

4.1.1. Sequence-Based Specification Method

In Cleanroom, the behaviour of a software system or component is defined by a total function called a Black Box function. The domain is the set of all possible sequences of input stimuli, including illegal sequences, and the range is the set of possible system responses. The method used to derive this function is the Sequence-Based Specification Method [PP03].

The Sequence-Based Specification Method is used to derive the initial Black Box function from an informal specification; it is the vital bridge between formal and informal worlds. The starting point is the informal, natural language requirements specification, written according to current practice in domain specific terms; we use the Sequence-Based Specification Method systematically to develop a formal specification while retaining complete traceability with the original informal specification. The original terms and domain concepts are used and together with the traceability, this enables the critical project stakeholders to establish by inspection that the formal specification specifies the same system as the informal specification. This directly addresses the problem of different starting points, and the need for traceability between the formal specification and the customer's requirements specifications (described in Section 3.4).

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 specifications1. 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 specification problems 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.

1According to current best practice, it helps greatly if the requirements in the informal specification are each identified by a unique tag.

A 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 exhaustively consider the entire state space of the program and determine the correct behaviour in each state. The domain of the Black Box function 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 behaviours; real systems cycle between 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 the Black Box function 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 extensions of them result in exactly the same future system behaviour. Each equivalence class is characterized 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 as a set of Sequence Enumeration Tables: one such table for each equivalence class and thus Canonical Sequence. There is a row in eachtable 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.

4.1.2. 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 externally visible behaviour 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 Black Box must interact. A Black Box has neither state nor control flow in its description. It is a total function relating all possible sequences of input to the set of all possible system responses. 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.

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. Cleanroom 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.

4.1.3. Practical strengths and weaknesses

The Cleanroom Method has a number of significant strengths that lead to defect prevention:

- A Cleanroom specification is still 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 Specification 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 to the Cleanroom specifications.
- The Cleanroom specifications are sufficiently formal and rigorous so as to enable the automatic generation of running code, statistical test cases and other formal models as we shall see later. Cleanroom on its own is not sufficient for the purposes of producing business-critical and untestable software. There are a number of areas which it does not address:
- In common with many development methods, Cleanroom is component-centric. That is,

Cleanroom is very effective in defining the behaviour of individual components but provides little in the way of compositionality. We cannot combine Cleanroom specifications of individual components and makes statements about the behaviour of the system as a whole.

- Cleanroom provides no means of analysing the dynamic behaviour of sets of components for deadlocks, livelocks, race conditions and so on. We cannot animate and analyse Cleanroom models.

The Box Structure Development Method requires each refinement step to be verified but provides no easy means or tool support for doing this. We wish to verify our implementation against our specification before we write code; on each refinement cycle, we must verify the State Boxes against their Black Boxes and the Clear Boxes against their State Boxes. In practice, without tool support, this is so time consuming and labour intensive and error prone that it is not consistently done.

It is to address these shortcomings that we have combined Cleanroom with the CSP/FDR framework described below.

4.2. CSP/FDR: A framework for formal verification

CSP [Hoa85] is a process algebra for describing concurrent systems composed of processes that

interact with one another or their environment through some form of communication. The communication is synchronous and takes the form of events. CSP has a rich expressive language, enabling a broad spectrum of systems to be modelled. It also comprises a collection of semantic models and mathematical techniques for formally reasoning about such systems' behaviours.

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 in a process. 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.

A broad range of properties (both safety and liveness) can be expressed in this framework. We will not go into the details of the CSP language or semantics here; rather we seek to give an intuitive overview of the scope of its functionality from a practitioner's point of view. See [Ros98] for further details.

One of the aspects that makes this framework a good choice for an industrial setting is its extensive tool support. FDR is a mature model checker for CSP; it was developed by and is a commercial product of Formal Systems Ltd [For03]. FDR enables the fully automated analysis of the following properties: refinement checking between two processes in all the semantic models mentioned above and verifying whether a process is deterministic, deadlock free and livelock free.

If such a check fails, then FDR returns corresponding counter-examples. FDR returns the shortest counter-examples (due to a breadth first search of the state space) and also provides extensive support for analysing them: One can view the example from the top system level down to the individual subcomponent behaviour.

4.2.1. Practical strengths and weaknesses

CSP together with FDR have a number of important strengths that Cleanroom alone does not:

- CSP has a rich expressive language for describing concurrent systems that are composed of components communicating with one another, together with a collection of semantic models for formally reasoning about such systems' behaviours.
- Unlike the Cleanroom approach, we can also combine the models of individual components and, using FDR, formally analyse the resulting combined behaviour.
- CSP is able to represent nondeterminism. This is particularly important when modeling 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.
- CSP supports encapsulation. When combining components, events and behaviours internal to components can be hidden.
- Refinement in CSP is transitive and all CSP operators are monotonic with respect to it, which in turn enables compositional development. Transitively also enables stepwise refinement. These properties are common in a software development process; being able to capture them in the CSP framework, therefore, enhances the expressive power of the model.
- 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.

Used on its own, CSP/FDR has a number of shortcomings that make its use in an industrial setting problematic:

- CSP models are abstract and the notation is very mathematical. The gap between an informal requirements specification and its CSP representation is substantial. It is very difficult to verify a CSP model against an informal specification and would require a high degree of expertise in CSP together with a complete familiarity with the product domain and its users. In short, the critical project stakeholders would have to be conversant with CSP and this is unrealistic.
- Traceability between CSP and the informal requirements specification is difficult to achieve. This is not a one-time problem; because requirements are subject to change during the product development, this problem reappears and makes it difficult to keep both CSP model and requirements specifications synchronised.
- Safe use of CSP requires extensive mathematics background. For example, making changes to the CSP model by hand for the purposes of optimising the state space search.

4.3. Combining Cleanroom and CSP

Our approach combines Cleanroom with CSP/FDR to address the weaknesses that either method has when used on its own in an industrial setting. A Cleanroom specification is rigorously specified with clearly defined operational semantics. Therefore, in principle, we should be able to translate it into other formal notations, secure in the knowledge that the original semantics have been preserved. In particular, G. Broadfoot shows in [Bro01] that the Sequence Enumeration Tables, Black Box and State Box specifications can be faithfully translated into CSP in a way that preserves the operational semantics of the original.

We have since learned that we are able to generate statistical test cases, run them and verify the result automatically, all based on the Cleanroom specifications. We are also able to generate state machine control logic code directly from the Cleanroom specifications.

We apply Cleanroom and CSP/FDR in combination in the following way:

- We use the Sequence-Based Specification method to define all interfaces between the system components and their environment. We can generate CSP models of these specifications and check them for divergence and deadlocks.
- When we have defined Black Boxes, we generate the corresponding CSP models from the Black Box specifications and check them for compliance against their specifications, deadlock, divergence and nondeterminism.
- As we derive State Boxes from their corresponding Black Boxes, we generate the CSP models from the State Box specification and use a process equivalence check to ensure that the State Box and corresponding Black Box have exactly the same externally visible behaviour; that is they are trace equivalent within the CSP traces model.
- We can analyse the interactions of sets of components by generating the corresponding CSP models from the Cleanroom specifications and composing them into a single model representing the combined behaviour of (large parts) of the complete system. We can analyse the resulting model for deadlocks, divergence and nondeterminism.

There are two important factors that facilitate the formal link between Cleanroom and CSP. Firstly, the notion of a canonical sequence of visible events within the Cleanroom Method is essentially that of traces within CSP: A trace is simply a sequence of visible events that a given process can perform. This is precisely what a canonical sequence represents and therefore leads to an intuitive mapping to representative CSP processes. CSP is a more abstract representation than Cleanroom because it makes no distinction between input stimuli and responses. Secondly, the notion of refinement in Cleanroom, to move from a high level specification to a more detailed description (for example, from Black Box to State Box), can intuitively be captured by the refinement models in CSP. Cleanroom is designed to create deterministic systems. This can be checked and exploited by FDR, since the notion of determinism corresponds to that of CSP. Due to space restrictions, we do not discuss the formal translation mechanism between Cleanroom and CSP here; details can be found in [Bro01].

By combining Cleanroom with CSP/FDR, we address the weaknesses of Cleanroom by using CSP models generated from the Cleanroom specifications and composing them together, as follows:

- We are able to examine the interactions between components and set of components; the combined method is no longer component-centric.
- We can examine individual components and sets of components for deadlocks, livelocks, race conditions and so on. We can animate and analyse Cleanroom models.
- We can automatically verify the stepwise refinement from Black Box to State Box.

Similarly, by using CSP models generated from the Cleanroom specifications and composing them together, the combined approach addresses the

shortcomings of CSP/FDR used alone:

- CSP models are abstract and the notation is very mathematical. However, by generating the CSP models from the Cleanroom specification automatically, the difficulty of verifying the CSP model against the specifications is no longer an issue.
- Traceability between CSP and informal requirements specification is no longer an issue because the CSP models are generated from the specification. For the same reason, keeping the CSP models synchronised with the informal requirements specification as it evolves and changes, is no longer an issue.
- Safe use of CSP requires extensive mathematics background; this is still an issue to some extent. No CSP knowledge is required to use this approach for verifying refinement steps; this is completely automated. We have seen, however, that as the benefits of CSP modeling are realised, developers want to use the generated CSP models for other purposes, such as verifying that the specifications do not allow forbidden behaviour (the lift must not move if the doors are not confirmed closed and locked.) This leads to writing such safety rules as CSP processes by hand and leads to false results when done by developers who do not have a firm grip on CSP.

5. A PRACTICAL CASE-STUDY

In this section, we present a practical case study where we applied Cleanroom together with CSP/FDR 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. We start by introducing the case study and identifying the software components to which we applied Cleanroom and CSP/FDR. We then summarise how Cleanroom and CSP/FDR was applied, the benefits gained.

5.1. 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 about 500,000 lines of original code.

Software execution 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 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.

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.

5.2. Applying Cleanroom

Cleanroom 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 a high level abstraction of 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.

These interfaces were already specified using a combination of informal specifications to explain the operational semantics and interface definition files to describe the syntax.

When we started this work, the implementation of both the Transport Controller and the Pick and Place Controller was already well advanced and did not use Cleanroom. The corresponding designs and interface specifications had been subject to a peer review process. In spite of this, while applying the Sequence-Based Software Specification technique to formally specify the interfaces between the Kernel and the two Controllers, 186 major specification issues arose on the Transport Controller interface and 119 issues arose on the Pick and Place Controller interface. These were 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 randomoccurrence 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.

We 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 particular 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 the Cleanroom techniques, we were able to make complete formal specifications of the operational semantics of the GUI, Transport Controller, and the Pick and Place Controller. 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 parameters); this leads to incomplete and ambiguous descriptions of the operational semantics. Cleanroom is particularly strong in specifying behaviour with great precision.

From the sequence-based specifications, we derived the corresponding 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 generated by hand.

The formal specifications provide 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.

5.3. Formal analysis using CSP/FDR

The Cleanroom specifications are sufficiently precise to enable us to generate automatically by means of simple PERL scripts both the formal CSP models needed for correctness proving and the control flow code used in the final product. 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 can verify that: (i) All three Kernel Layers are deterministic; (ii) The Machine State Layer behaves as expected by the GUI 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 Sequence Enumeration and a corresponding State Box, we can verify that the State Box has exactly the same operational semantics as the Sequence Enumeration using FDR. It is not necessary to verify the Black Box against the Sequence Enumeration and then verify the State Box against the Black Box; we can verify the State Box against the Sequence enumeration directly.

5.4. 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 Cleanroom 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 Cleanroom tables.

The Kernel consists of 20,000 lines of C++ code, about a third of which was generated automatically from the Cleanroom 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 Cleanroom 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 issues2. 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 errors related to race conditions or timing errors have occurred. The total amount of rework to date is less than 2%.

6. 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 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.

In Section 3, we highlighted a number of barriers encountered when introducing formal methods in practice. The first two were concerned with economic incentives for justifying the need for the introduction of a more formal approach towards software development in certain domains. The third observation was the different starting points between a typical industrial software project and applying formal verification techniques. Formal development requires a formal specification as its starting point. However, the style and notation used in many formal approaches renders them inaccessible to critical project stakeholders who are not experts in formal methods.

This alone is sufficient, in the first author's experience, to prevent their wide scale adoption into industrial software development.

Cleanroom addresses this issue extremely well. 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.

Finally, we discussed the difficulties inherent in the use of formal verification tools and mapping the abstract model to the actual system being developed. Even with sophisticated and fully automated tools, we are exploring and verifying the behaviour of a model of the system to be developed, and not the system itself. Developing such models typically requires a high degree of skill; once such a model is built, the onus lies on the modeller to show that it accurately reflects the system being developed. However, the model builder is typically not a domain expert and the model is frequently so abstract that the critical project stakeholders cannot comment on the model's accuracy. We have started addressing this problem using Cleanroom in combination with the CSP/FDR framework by generating our CSP models automatically from the Cleanroom specifications. Formal verification is performed upon the resulting CSP models using FDR.

We have shown that Cleanroom combined with CSP is one 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 Cleanroom in conjunction with CSP and FDR, 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 Cleanroom 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 sets of Sequence Enumeration Tables. 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 Cleanroom specifications is that we are able to test that the specifications have certain properties. For example, the operational semantics of each of the two 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.

6.1. Future research avenues

The work presented in this paper provides an overview of some initial experiences (from a practical perspective) of introducing formal methods into industrial software development practices. This is very much ongoing research, particularly in the combining of Cleanroom and the CSP framework.

The initial work by G. Broadfoot [Bro01] provides the foundations for this. He showed how one could translate Black Box and State Box descriptions from Cleanroom into equivalent CSP processes. The Cleanroom equivalence between the two is then determined by verifying tracesequivalence of the corresponding CSP processes using FDR. This approach is simplified by the fact that Cleanroom only generates deterministic descriptions (for both black boxes and state boxes) and therefore problems of how to handle nondeterminism within the models are not raised.

Comparing deterministic CSP processes is straightforward and only requires the traces model. In practice, the implementations of the individual components are indeed deterministic and therefore the Cleanroom stepwise refinement for developing them works very well (and similarly combining this with CSP/FDR to verify each step). However, research is ongoing to extend this framework to enable a more systematic approach to handling the nondeterminism arising from reasoning about the behaviour of the system as a whole and verifying it against under-specified specifications. While the CSP/FDR framework is excellent for this purpose, Cleanroom is not and therefore limits the way in which we derive the CSP models.

Another area of ongoing research is in the development and application of optimisation strategies within the CSP models. The models generated from these examples were very large, in some cases the ASCII files of the CSP source exceeded 7 Mb. The models proved to be well within the range of FDR's capabilities, but further research is needed to establish the correct structure for efficient modelling. We are so far not convinced that a single structure is suitable or sufficient for every case. We have learned that certain design patterns are easy to model. For example, the Publisher-Subscriber3 pattern [BMR+96] is easy to model for two reasons: (i) we can dispense with the intermediate queues in the model without degrading the model's fidelity; (ii) we can omit all illegal sequences from the CSP model of the Publisher with the result that FDR will automatically generate input stimuli for every valid sequence in the Publisher thus causing the

Publisher in turn to generate all possible sequences of input to the Subscriber. The CSP model for the Subscriber retains the illegal sequences and we use FDR to determine whether or not the Publisher is able to generate a sequence that the Subscriber regards as illegal. However, more complex Client/Server relationships require more careful consideration concerning queueing issues. We are seeking to classify common design patterns and process relationships according to the modelling techniques appropriate to each; this is part of ongoing research.

ACKNOWLEDGEMENTS

We are grateful to Assembléon 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. We also thank Bill Roscoe and Michael Goldsmith for useful discussions and comments ■

Guy H. Broadfoot, Chief Technical Director, Verum Consultants.

Figure 2. Guy Broadfoot (CTO).

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.
- [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.

Figure 3. Robert Howe (CEO).