MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering

IST-1999-11435

Project Manager's Handbook for Systems Construction in MATISSE

MATISSE/D10/b/1.0

36 pages

January 2003

Project Information

Project Number	IST-1999-11435
Project Title	Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE)
Website	www.matisse.qinetiq.com
Partners	Aabo Akademi University
	ClearSy
	CNRS-TIMA Laboratory (SC)
	Gemplus
	Siemens Transportation Systems
	University of Southampton
	QinetiQ (C)

Document Information

Document Title	Project Manager's Handbook for System Construction
Workpackage	WP1
Document number	D10 (Part B)
Lead Partner	CNRS
Editor	Traian Muntean
Contributors	All partners
Due date	February 2003

Version 1.0	initial version

Abstract

Methods and aids for the management of system development projects based on the formal approach proposed in the MATISSE project are presented here. Although not initially planed in the original project proposal, it was decided to deliver three storyboards threads for the MATISSE Methodological Handbook. This document is one of those threads, **project manager's storyboard**, and complements the **practitioner's storyboard** and the **board level storyboard** of the Handbook. The recommendations are based on analysis and experiences of the three industrial case studies developed as part of of the MATISSE project.

List of Contents

1 Introduction

The MATISSE Methodological Handbook is intended to be a convenient reference on systems construction and management methods using the formal constructive approach proposed in the MATISSE Project, through proved refinements, and aids. This thread of the storyboard is part of the MATISSE Handbook comprising also a "Practitioners Storyboard" (thread targeting the engineering practice) and a "Management Board" thread (targeting the decisional board for projects development).

The *project manager's storyboard* offers concise and relevant information describing:

- what constructive formal methods like the B-Method can accomplish for systems design;
- when they can be applied in a project lifetime;
- how they are applied;
- where the project manager can find more background material from the MATISSE project.

The management methods and aids included here are those that have proved effective in the experience of the MATISSE Industrial Case Studies, namely, a Smart Card embedded prover (GEMPLUS), an Automatic Transportation System (SIEMENS) and a Health Care Control System (WALLAC). The characteristics of these software and system engineering projects appear in the deliverables and working documents produced in the Work Packages of the MATISSE Project (http://www.matisse.qinetiq.com/).

1.1 Intended Audience

The intended audience of this document is the *project manager*, who, as defined in this storyboard of the handbook, serves as either an *administrative or technical manager*. These positions overlap somewhat in their information needs.

The *administrative manager* has overall responsibility for developing software that meets requirements and is delivered on time and within budget. Typically, this manager is not involved with the day-to-day technical supervision of the programmers and analysts who are developing the software. The administrative manager will be involved in the activities listed below:

- Organizing the project
- Estimating resources and Costs required
- Evaluating documents and deliverables
- Monitoring progress
- Evaluating results of reviews and audits
- Certifying the final product

The *technical manager* is responsible for direct supervision of the developers. This position is frequently filled by the project administrative manager also; although, on some projects, a proper technical manager will fill this role instead. This person shares some of the activities listed for the administrative manager, especially with regard to monitoring development progress. More specifically technical manager's activities are:

- Producing the software/system development/management plan
- Estimating development costs
- Scheduling the project tasks
- Staffing the project tasks
- Directing the production of documents and deliverables
- Monitoring development progress (using automated management aids when required)
- Supervising technical staff
- Ensuring software quality
- Preparing for reviews

A secondary audience for the handbook may consist of those who serve a particular peripheral function but do not act in either of the two managerial capacities. Two examples of such specific functions are participating as an external reviewer at a scheduled review and conducting an audit of the project.

1.2 Development Life Cycle

The process of software development is usually modelled as a set of stages that define the software life cycle. The systems engineering life cycle is generally defined by the following phases:

- Requirements definition
- Requirements analysis
- Preliminary design
- Detailed design
- Implementation
- System testing
- Acceptance testing
- Maintenance and operation

The life cycle phases are important reference points in monitoring a project, the manager may find that the key indicators of project condition at one phase are not available at other phases. Milestones in the progress of a software project are keyed to the reviews, documents, and deliverables that mark the transitions between phases. Management aids and resource estimates can be applied only at certain phases because their use depends on the availability of specific information.

Usually, in the *requirements definition* phase, a working group of analysts and developers identifies previously developed subsystems that can be reused on the current project and submits a reuse proposal. Guided by this proposal, a requirements definition team prepares the *requirements* document and completes a draft of the *functional specifications* for the system.

During the next phase, *requirements analysis*, the development team classifies each specification and performs functional or object-oriented analysis. Working with the requirements definition team, developers resolve ambiguities, discrepancies, and to-be-determined specifications, producing a final version of the *functional specifications* document and possibly a *requirements analysis* report

The baselined functional specifications form a contract between the requirements definition team and the software development team and are the starting point for *preliminary design***.** During this third phase, members of the development team produce a *preliminary design report* in which they define the software system architecture and specify the major subsystems, input/output (I/O) interfaces, and processing modes.

In the fourth phase, *detailed design*, the system architecture defined during the previous phase is elaborated in successively greater detail, to the level of specific implementations details.

During the *implementation* (code, unit testing, and integration) phase, the development team codes the required modules using the detailed design document. The system grows as new modules are coded, tested, and integrated. The developers also revise and test reused modules and integrate them into the evolving system. Implementation is complete when all code is integrated and when supporting documents are written.

The sixth phase, *system testing*, involves the functional testing of the end-to-end system capabilities according to some system test plan. Successful completion of the tests required by the system test plan marks the end of this phase.

During the seventh phase, *acceptance testing*, an acceptance test team that is independent of the software development team examines the completed system to determine if the original requirements have been met.

The eighth and final phase, *maintenance and operation*, begins when acceptance testing ends. The system becomes the responsibility of the maintenance and operation group. The nature and extent of activity during this phase depends on the type of system developed.

1.3 MATISSE Approach

Over the last 30 years, computer scientists have been developing and advocating the use of rigorous mathematically-based software engineering techniques, so-called *formal methods* http://www.fmnet.info/, that support validation throughout the development life-cycle by providing rigorous specification and design notations as well as proof techniques, model-checking techniques and simulation techniques. Formal methods also allow the complexity of systems to be dealt with through abstraction and modularity. Despite this, the use of formal methods is not widespread in the industry because of various managerial, sociological and technological barriers. One of the major objective of the MATISSE project is to overcome some of these barriers by:

- providing methodologies and associated technologies that are integrated with standard working practices;
- providing further evidence through well-founded evaluation plans that the use of formal methods is cost effective
- showing how formal methods can help ensure that products and services meet appropriate standards for safety, security and reliability.

The industrial case studies provided by partners allowed us to identify and understand the most effective way of introducing and using formal methods in a range of industrial environments and to identify, understand and overcome the barriers to their uptake and propose enhancements to the existing tools used in the project. The industrial case studies are:

- An embedded verifier for a multi-applications smartcard system;
- A railway signalling and control system;
- A diagnostic system for healthcare clinicians and researchers.

The approach of the MATISSE project involves exploiting and enhancing existing formal methods developed and used by the partners of the project together with associated technologies that support the correct construction of software-based systems. In particular, a strong emphasis is being placed on the use of the B Method [1] and its associated technology, Atelier B[7], provided by ClearSy. Use of the B Method is being complemented by the use of the Action Systems [6] and CSP [2,3] formal methods. Event B[8], B-action systems[13] and a Distributed-B notation [16], are formalisms developed in MATISSE for supporting the development of complex distributed and communicating systems.

The project proposes a refinement-oriented approach (for details of the methodological approach see Deliverable D4-"Methodological Handbook") as a basis for critical system design. We start from an abstract specification and stepwise refine the system into a more concrete and deterministic system and thus eventually into a proved implementation. This constructive approach has been successfully and extensively used by partners for all the three case studies proposed in the project and also in other application areas (e.g telecommunication network protocols, adaptive routers). The methodology for systems construction by refinement does not rely on a preconceived notion of how an implementation may be constructed from a given specification and specific refinement transformation functions. We also consider as an important methodological issue the compositionality of designs, that is to say how proved subsystems can be further composed for the construction of complex systems.

Some of the impediments for integration of formal methods frequently cited by the formal methods community include such industrial problems as inadequately educated engineers, the not-invented-here syndrome, and greater emphasis on reducing costs than on increasing safety. However, it is commonly accepted that the primary causes for the lack of wide-scale industrial use of formal methods are: inadequate tools, inadequate examples, and a "build it and they will come" expectation.

The case studies above have succeeded because our partners have been able to overcome or avoid these pitfalls in one way or another. As the case studies have been chosen from a large spectrum

of industrial applications, we believe that the results of Matisse can be applied also to other industrial applications and systems.

In order to illustrate the approach proposed in MATISSE the B-Method has been extensively used in the Case Studies together with Action Systems and CSP. The fundamental characteristic is that the software developed with B is **correct by construction**.

It consists of the following elements:

- a modular language that covers the whole cycle of development;
- the phases of specification, design and implementation are thus carried out within a homogeneous framework, a language to express properties and safety critical requirements;
- Reliability, Availability, Maintainability are expressed within a same framework, the mathematical demonstration that the software respects these properties;
- a rigour of development absolutely necessary when developing a large and complex software.

Below is a schematic representation of the activities that are part of a B project:

- Specification: an abstract model of the software is built from its (informal) requirements. Implementation details are not introduced at this level. This model is proved to be consistent (typing is correct, invariant properties are established by the initialisation and preserved by the operations). Tractability between software requirements and abstract model should be check manually.
- Design: the abstract model is refined down to a concrete model, containing all the details of the software. This concrete model is proved to be consistent and not to be contradictory with the abstract model.
- Translation: the concrete model is then automatically translated into target code, which can be C, C++, ADA, Java….

1.4 Classical software development cycle versus B development cycle

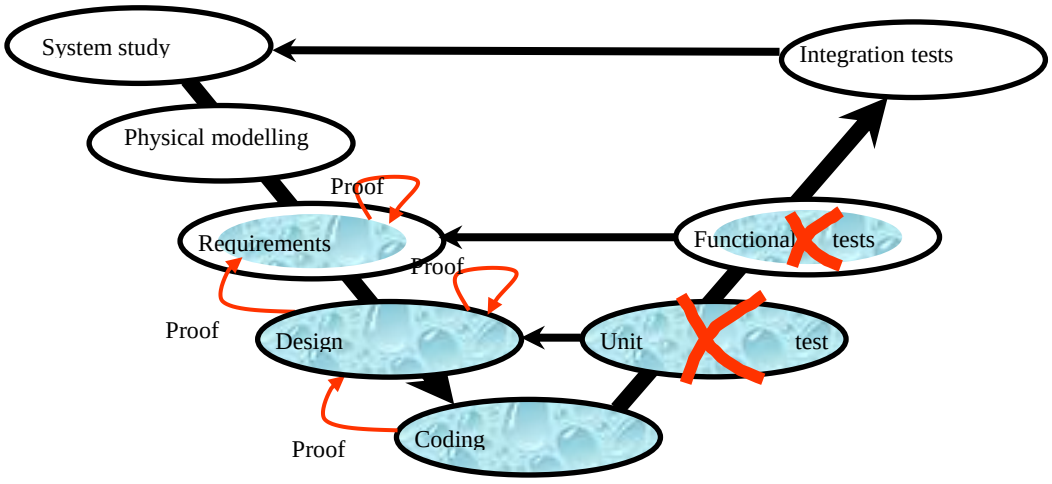
The figure below enables to compare a classical development cycle with a B development cycle[8]. What can be observed is that:

- no unit test is required (apart basic machines testing), since proof ensures that every piece of code complies with its specification.
- no functional validation test is required, as proof ensures that top-level properties hold. Nevertheless, tractability between software requirements and abstract model should be manually verified.

The main principles to used for a B development are: to concentrate energy on the initial phases of development in order to be “as precise as early as possible”, then to make best use of proofs to be sure we stick to our initial aims.

However it is still not easy to evaluate the real cost of using B for a full system design, nevertheless, the contribution of the MATISSE Case Studies is to have produced some evaluation criteria and figures:

- time spent on development, including proof, is highly dependent on team's maturity,
- proof phase cost is closely linked to the chosen modelling solution and thus to team's experience,
- there is a lack of feedback on the reusability of B components.



Typical B development cycle compared to a classical development cycle.

2 Using Formal Methods in the Project Life Cycle

2.1 Structuring a project using a FM approach - a manager's perspective

2.1.1 A Conceptual Model for the Project Managers thread

From a project manager's perspective, how formal methods might be linked into a project development plan is of crucial interest. What threads might be useful in guiding the project manager to find such things as costs, training requirements, risks and benefits can be stated in a conceptual model which focuses on the integration of the use of formal methods into the Project Life Cycle. It shows where the formal method fits and what links there are to that part of the PLC it influences. It also shows those characteristics of a formal method which are likely to offer up the kind of parameters, constraints and properties which will be useful to the project manager. It identifies the threads which must be available. The clarity with which these threads are mapped through the documentation could easily influence the success with which the formal method is (or isn't) employed.

The conceptual model proposed is encapsulated in the two figures below. The model is deemed conceptual because it is rather a novel approach that documentation might be constructed to help project managers and then linked into the management process.

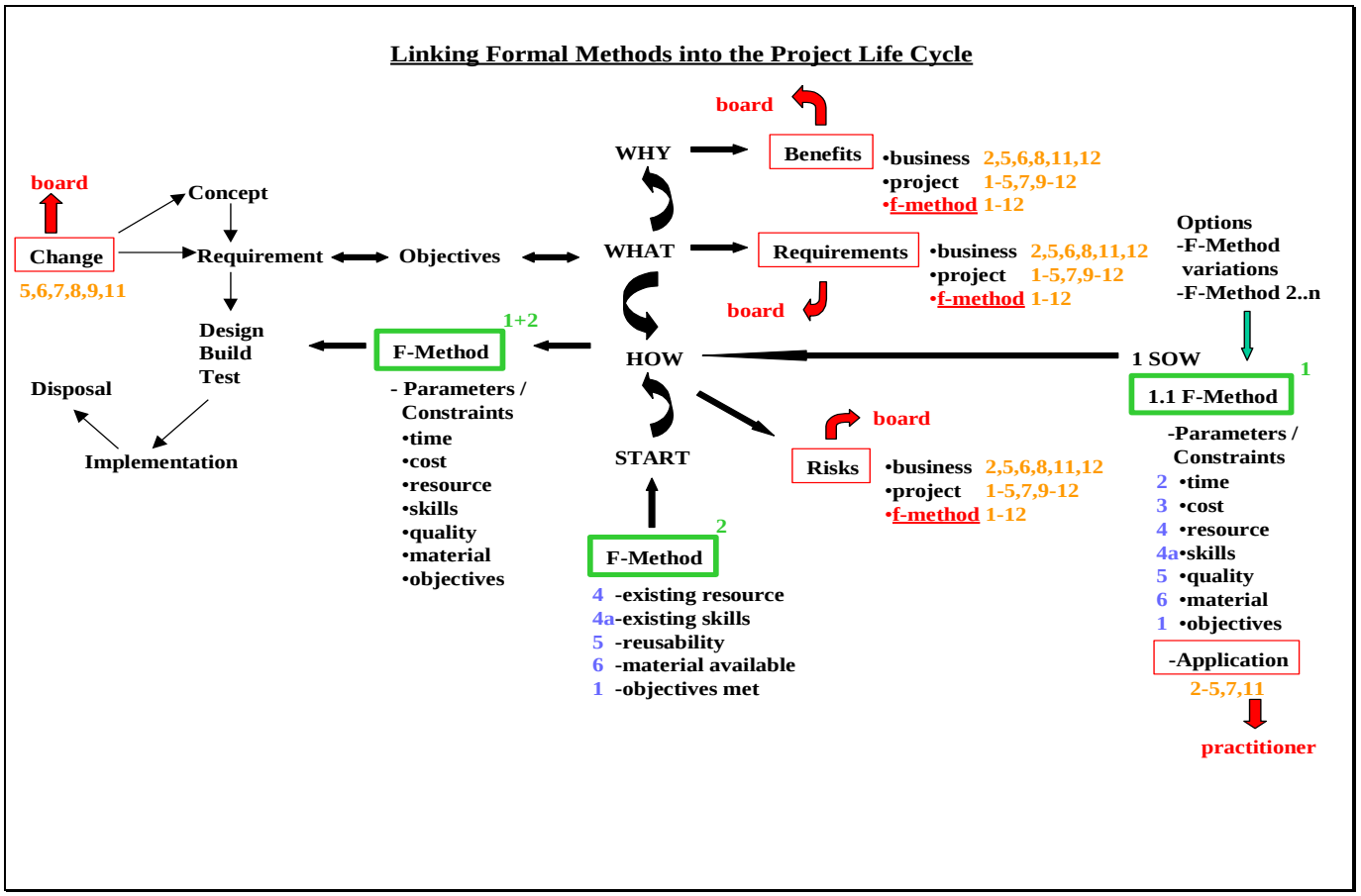


Figure 1

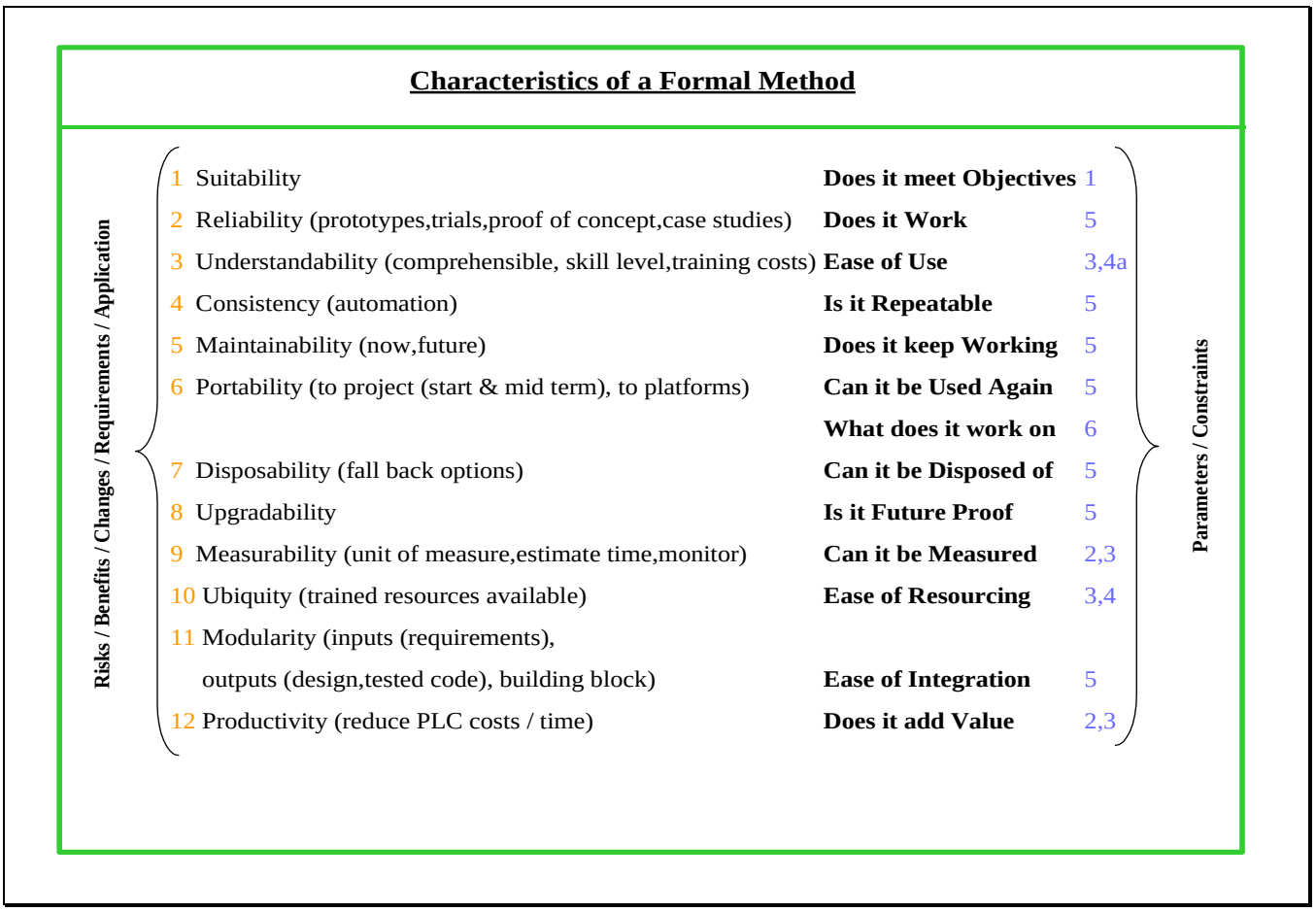


Figure 2

2.1.2 Threads which drill down from the project management process to the formal method.

i) If we consider using a formal method (f-method in the model), we must first look at parameters and constraints which will be of interest to the project manager. If a formal method is to be used it will be referred to in the Statement of Work (SOW), (green) box 1 in the Figure1, which describes 'HOW' the formal method will contribute to doing the job. The parameters/constraints associated with using a formal method must have been populated with values. These values will assume for the moment that we are starting the job from scratch. Alternatives (green arrow) will have been considered.

ii) The values of parameters and constraints must be directly measurable by examination of the characteristics (or properties) of the formal method. We must be able to drill or focus down into the documentation (figure 2) via an entry point and then follow threads to the various characteristics of the formal method in the documentation or handbook. There are 12 major characteristics each of which should give a guide to the population of one or more parameters or

constraints. In the model, these threads are illustrated by the (blue) numbers at the left of the parameter/constraint list in the SOW (box 1.1). They link to the (blue) numbers at the right side of figure 2.

iii) For example we might need to establish a metric to measure translation of descriptive design requirements into formal design notation. This is a time and cost implication and follows threads 2 and 3 to characteristic 9.

iv). Once we have populated the (green) box 1 parameters/constraints with initial 'from scratch' values, we can now consider pre-existing conditions. (Green) box 2 describes how pre-existing conditions might contribute to doing the job. 'START' parameters/constraints can be populated via threads through the documentation as described in ii).

v). Amalgamation of (green) box 1 with (green) box 2 describes how the formal method will contribute to doing the job taking into account pre-existing conditions. This is (green) box 1+2 which completes the link from formal method via the project management process and into the PLC. For example we might already have 1 of the 2 staff we need to train to use the formal method already trained as a result of a previous project. This is a training implication and follows thread 4a to characteristic 3.

2.1.3 Threads which focus up to the risk register.

i) During the drill down process, measures, values, descriptions and comparisons found might offer up risk assessment data. All characteristics of the formal method have the potential to reveal risks associated with using the formal method. Each risk must be directly quantifiable by examination of the characteristics (or properties) of the formal method. We must be able to follow threads through the documentation/ handbook which focus back up to the risk register. In the model these threads are illustrated by the (orange) numbers on the left of figure 2. They link to the (orange) numbers at the right side of the (red) f-method bullet in figure 1 and then into the (red) risk register box.

ii) For example, we might have identified from characteristics 3 and 10 that we have to train a new member of staff from scratch and that it will take x months. There is a risk that if the delay to having trained resource available is not correctly managed, adverse conditions may result. Such a risk is only manageable if we can quantify x. A project manager will try to remove the source of unmanageable risk.

iii) The risk register may be interpreted at various levels. The bullets on the right of the (red) risk box illustrate these levels and the (orange) numbers to the right of the bullets indicate those characteristics which might have an influence at each level. A risk may be significant at one or all of f-method, project and business levels. For example, characteristic 6 (figure 2) might indicate that the formal method is not portable to alternative platforms. This is a risk to the formal method in that it might not be used on this or future projects. It is not a direct risk to the project. It is a business risk in that start-up costs are not reduced in future project bids.

iv) The (red) arrow above the (red) risk box shows an interface to the 'board perspective' view of the formal method. It is a link by which the project manager identifies risks which might be of interest to the board.

v) Risks associated with a formal method are a direct result of its use. This is illustrated by the arrow from 'HOW' to the (red) Risk box.

2.1.4 Other threads which focus up

There are other threads which function in much the same way as the risk register threads described above in 2.1.3. These are the benefits list threads (Benefits box), the change management threads (Change box), the requirements met threads (Requirements box), and the application threads (Application box). For example, characteristic 12 might reveal the benefit of increased productivity attracting some interest at the board level. Characteristic 11 might reveal the change management lever of design/refinement modularity allowing scope to be modified. Characteristic 2 might reveal that the requirement of 'robust design' has been met. Characteristic 4 might reveal that application of the formal method is semi-automated increasing the efficiency of the practitioner.

2.1.5 Steerage nodes which link formal methods, project management process and external interfaces

i) In the model, the (red) steerage boxes are positioned to link threads from the formal method characteristics to the project management process and to external interfaces.

ii) The (red) Risk box is a node for threads gathered from the formal method characteristics and is directly linked to 'HOW' the formal method is employed. It has the board perspective as an external interface.

iii) The (red) Benefits box is a node for threads gathered from the formal method characteristics and is directly linked to 'WHY' the formal method is employed. It has the board perspective as an external interface.

iv) The (red) Change box is a node for threads gathered from the formal method characteristics and is directly linked to the PLC Concept and Requirement phases. It has the board perspective as an external interface.

v) The (red) Requirements box is a node for threads gathered from the formal method characteristics and is directly linked to 'WHAT' the formal method is employed for. It has the board perspective as an external interface.

vi) The (red) Applications box is a node for threads gathered from the formal method characteristics and is directly associated with the application of the formal method as specified in the SOW. The (red) arrow below the (red) application box shows an interface to the 'practitioner perspective' view of the formal method. It is a link by which the project manager identifies features which might be of interest to the practitioner.

2.1.6 Discussion

Although the above conceptual model has not been applied as such within the MATISSE Project Case Studies, we mention the model here in order to be able to refer to in the following sections.

Moreover this model could help project managers to understand dependencies and consequences of using formal methods.

The reader of this thread of the Handbook will easily notice in the following how the model applies to some relevant results of the MATISSE Case Studies. Some metrics obtained in the Industrial Case Studies will also be given in order to help project managers to understand the characteristics of the conceptual model when applied to real industrial projects in MATISSE.

2.2 Threads of the MATISSE Methodology useful to Projects Managers

2.2.1 Integrating a Refinement Strategy in the Project Life Cycle (PLC)

The facility to perform proofs within the B formalism enables direct and powerful validation. A functional property that a system should possess can be demonstrated using an automatic tool prover (e.g. FDR Model Checker for CSP models [3,9], Atelier B Prover for B refinements). The problem is the amount of effort that proof in general requires in the system design process.

The General Approach in MATISSE

We have based the general methodological approach on the B Method developed by one of the MATISSE Project partner, namely ClearSy. The B refinement method produces powerful results; we have a guarantee, through mathematical proof at each refinement step, that the implementation satisfies the original B specification. However, as systems become increasingly complex and distributed it is more likely that mistakes will be made in their design (specification), and that vulnerabilities of such systems will go unnoticed. If a mistake were to be made in the specification of a system, the B development process of refining the specification to code would not be able to identify that mistake.

As a precursor to any B refinement, a technique is needed which verifies that the specification does satisfy the security or safety properties required of it. Such properties are not functional, in that they are specified in terms of what the system shouldn't do, rather than what it should.

For example: "*the system should not reveal the contents of the transaction table to anyone other than the administrator",* is a non-functional security property. It is likely that any specification will be required to satisfy a number of such properties simultaneously. However, it is common that such safety or security properties are not defined either formally or informally. Usually such properties are deemed implicit in the functional specifications. This poses a problem for verifying such specifications, since there are no explicit requirements for the specifications to be checked against. In such cases the requirements have to be derived using hazard/vulnerability analysis [17]. A technique has been developed to facilitates the identification of critical properties of systems, and is designed to be used in conjunction with, and to compliment, the B method.

When developing in B, starting interactive proof too early may have dramatic consequences in term of cost and delay, because models are likely to evolve and proof work would be lost. On the other hand, if no proof is undertaken in the early stages of the development, high level specification may be wrong and, again, models would be corrected. *(cf. characteristics C2 and C9 of the Conceptual Model-Figure 2 – yellow column).*

A good way to proceed, which proved to be efficient on many industrial applications, is to mix automatic proof and proof obligations (PO) visual inspection. The objective is to be quite sure quickly that no false PO remains. The general approach is:

- Write abstract machines according to requirements document,
- Control the correctness of the formal expression of the needs,
- Start automatic proof on this abstract machine,

- If some proof obligations are not automatically demonstrated, check them quickly by reading them and verify that they can be true. A model checker can be used also at this high level of abstraction to verify some critical properties of a system. If some of them are false, the abstract machine should be corrected.
- Derive implementation components,
- Compare implementation versus abstract machine,
- Make the formal demonstration of the remaining PO in the abstract machine and in the implementation, using interactive prover.

Critical Systems High Level Analysis

The aim is to identify the critical properties of complex systems (i.e. the properties that the corresponding specifications must ensure). Such systems can be viewed as a collection of communicating components/objects. The range of behaviours such systems exhibit is a direct result of the variety of communications that components can enter into, the possible interleavings of such events, and the resulting states of the system. Due to the unpredictable interleaving of communications, there can be so many different sequences of communication events that it is impossible for a human to conceive of every one. This level of system complexity makes it likely that pathological system behaviours or critical properties might be missed, ones which can be critical to the correct behaviour of the system. The formalism used to perform such hazard/vulnerability analysis needs to support the debugging, re-programming and refinement of the specification. Further, it needs to support the capture of the non-functional properties that the specification must satisfy.

Tool support for a B development takes the form of theorem proving. Theorem proving tools are unable to provide feedback as to why proofs fail; determining why a proof is impossible can be difficult and time consuming. Further, the B method describes specifications functionally, hence, it is not obvious how B could be used to model non-functional properties. In contrast, model-checkers exhaustively check that every state of an implementation is also a valid state of the specification property. If this is not true, they provide information pertaining to how the system reaches such an invalid state, identifying the trace of events leading up to the property violation. Hence, model checking lends itself naturally to system vulnerability analysis.

In the MATISSE project a heterogeneous approach has been experienced using a model checker for critical systems specification based on CSP models combined with the B refinement using the Atelier B for automatic code generation from correct specifications for components of a communicating system of distributed components.

Once the correct specification has been constructed, using CSP, it can be incorporated into the B model. Using the techniques described elsewhere in the Methodological Handbook, the B model is then refined down to code. Hence, where previously we were only able to use formal techniques in refining our specifications to code, we can now employ formal analysis in the initial construction of such specifications. In this way the use of CSP analysis to discover the critical properties of systems complements the B development method. We gain the benefits of formal analysis much earlier on in our project development cycle. It isn't obvious how one would use only a theorem prover to achieve such results. Undoubtedly, it must be possible to devise an applicable method. However, model-checking is a theory which is mature and well suited to this

type of analysis. The process of specification refinement discussed here fits easily into the body of knowledge already present within the community.

The process algebra CSP [2,3] allows us to model components in isolation, and then to build models of the total system using the composition of its components. This modular approach to modelling reduces the risk of human error: the user is not required to capture the whole system at once, rather just a small part. It also produces models that are easily traceable to the original specification. Tool support for CSP comes in the form of the model-checker FDR [4]. FDR takes a CSP model of a specification and implementation, and automatically checks that the implementation refines the specification.

Before continuing it is worth mentioning that the success of a vulnerability analysis is dependant on the clarity of the conceptual understanding of what each part of the specification (or implementation) is supposed to do (or does). More subtly, the actual context in which the code is going to be used may have a significant impact on the vulnerability analysis; for example, what interfaces within the system that can be externally monitored, intercepted, or changed may be context dependent. Hence, both a clear layered design of the system and the context in which it is going (intended) to be used are required for a reasonable vulnerability analysis.

Supposing that an adequately clear description of the system and its context are available, and the hazards (or critical information) have been identified, then it is possible to perform various vulnerability analysis techniques [17] to identify if, when, and how a hazard could arise.

Here it is assumed that the *design*, the system (and context) to be analysed, can be broken down into a collection of capabilities and *inference* rules, which state what capabilities are required to gain new capabilities. Hence, in principle, given an initial set of capabilities it is possible to use these inference rules to determine the potential set of capabilities that can be obtained. If this potential set of capabilities contains a capability representing a *hazard* or *critical information* then it is concluded that either the *hazard* could occur or the *critical information* could be obtained.

At this point it is worth mentioning, that for any *valid* set of capabilities and inference rules, the CSP/FDR vulnerability analysis is fully automatic, returning a tick or a cross depending on whether the analysis was successful. If the analysis failed, then a minimal trace to the *hazard* that could occur, or the *critical information* that could be obtained, is provided for free by FDR's graphical "debug" facility; this provides a list representation the events (inference rules used) to obtain the failure. The minimally of the trace is guaranteed by FDR's breadth first search strategy.

Consider a CSP/FDR design that has a collection of initial capability sets, which represent different entities views of the system. And for each view associate a set of *misuse* capabilities that should not be available from that perspective (i.e., *critical information*). Then the normal expectation would be that the CSP/FDR vulnerability analysis for each view would demonstrate that it was impossible to use the inference rules to obtain any of that view's *misuse* capabilities. Having achieved this, the analyst might want to check how resilient the design was to component failure or capability compromise. This can be achieved by a technique generally known as *fault injection*. Here capability compromise can easily be modelled by adding new capabilities to the initial capability set; and component failures can be modelled by appropriate modification of the inference rules. Further, it is straightforward to construct an automatic check for all *n*-fault designs (i.e., a design with *n* faults). Though, in practice, space and time limits typically limit *n* to

somewhere between 1 and 3, depending on the precise nature of the design, and limitations placed on the faults to be injected.

A more technical description of the CSP/FDR analysis has been, in the MATISSE project, illustrated by a simplified analysis applied to the Smart-card case study.

Automatic Proof Rate as a Quality Indicator

Automatic proof rate is a good indicator of the suitability of a given B model. Most of the time, if your proof rate is too low, your B model needs to be reshaped. From our experience in MATISSE and also in some other industrial projects, large-scale projects have an automatic proof rate between 70% to 95%. Locally, some components may have a low rate, depending on their complexity and the symbols that are used for the B model.*(Characteristics 2 and 3 above-Figure2).*

Your models may be too complex, because:

- too many details are included in your top-level specification components.

    Hints:

    Try to abstract and to use only variables that are necessary to be present at this level.

    Introduce details in the lower components.

- you make use of heavy sequencing.

    Hint: try to decompose your system in communicating components.

Coming along with Atelier B 3.6, a new package of rules will be available, providing axiom-like rules related to these symbols that were badly manipulated by older versions of the prover.*(Risk thread –Fig1 and C3-Fig2 -Requirements Thread -Fig1)*

Many proof forces are available: 0, 1, 2, 3, Fast and UserPass. Below are the effectiveness figures (mean values) related to these forces:

Force	Proof time per lemma	Performance
0	Less than 10 s	70 %
1	From some seconds to 3 minutes	+ 1 %
2	From some minutes to tens of minutes	+ 3 %
3	From tens of minutes to hours	+ 1 %
Fast	Less than 3 s	30 %

Figures are indicative and come from diverse, large-scale projects. They are likely to vary, depending on the size of your project and the way B models are written. Note that Fast Force is far less efficient than Force 1, and should be only used on large components. UserPass enables you to "program" the demonstrations that would be tried by the automatic prover.

There is one major rule related to manual proof: inspect PO before any manual proof activity. PO inspection (after models reading and Force 0 proof) can be done either with PO viewer or with interactive prover. Difficult POs should be evaluated first.

PO inspection is composed of 5 steps:

- goal reading (interpret and isolate verified constraint)
- justification (use the physical meaning of the component),
- key hypothesis selection (search matching hypothesis),
- intuitive demonstration (reuse justification and examine used rules),
- notes and trials (write down used simplifications and try a quick demonstration).

The average proof rate has been measured between 10 and 20 POs per day, including debugging, for the lifetime of the project *(Measurability C9-Fig2)*.

Particularities of formal developments

Many errors have been found during proof activities. Nevertheless, as development and proof are closely linked, no figures are available. Several errors have been found by the validation team on the mathematical rules added.

It is interesting to notice that they did not correspond to bugs in the models but only bugs in the proof: they were fixed without any change in the model and, consequently, in the code.

Errors Found by Testing

In the transportation case study the following results have been observed:

1. functional validation on host computer: no bug found;
2. integration validation on target computer: no bug found;
3. on-site tests: no bug found;
4. since the line operates: no bug found.

This is clearly a very unusual situation: in comparison with a previous informal development of less complex safety critical software products, the different phases of validation have revealed several tens of errors. Several releases were necessary before the acceptance tests.

Integrating formal development with industrial standards and documentation (Health Care Case Study)

The healthcare case study was performed in MATISSE as a co-operation between Aabo Akademi University and PerkinElmer Life Sciences (in the document generally referred to as Wallac) using an industry-as-laboratory approach. This means that the formal methods expertise is provided by the researchers at the academia and the domain knowledge is brought into the team by the experts at the R&D department of the industrial partner. Wallac mainly manufactures systems that should guarantee extremely high precision and dependability. The company is interested in applying formal methods to enhance dependability of their systems and prepare to meet tightening regulatory requirements. Wallac is new to using formal methods. Therefore, our goal is to propose a methodology for the development of dependable systems which would enable integration of formal methods in the development life cycle of such an industrial partner. We observed that their development cycle is incremental. The company subcontracts a mechanical part of a system under construction and then designs hardware and software to achieve the desired functionality. To cope with the complexity of this task Wallac designs the system incrementally. Namely, they focus on the development of one component at the time adding new components in a stepwise manner. Testing plays a paramount role throughout the whole development process. It is used to identify required hardware, learn about operating and failure modes of hardware components as well as to test software under construction. Designing correct software is an especially difficult task: software should not only perform routine control functions but also guarantee fault tolerance.

Wallac bases its development life cycle on an application of UML. Initially developers have just an idea of the system's basic functions. They apply UML to specify an implementation in detail. The system designers argue that the use of UML facilitates interdisciplinary discussion of the development, improves quality of software and the system in whole as well as streamlines the documentation flow.

From the observations above we derived the following guidelines on integration of formal methods with systems engineering practice in Wallac.

i). A smooth incorporation of UML artifacts in the formal development process. Wallac developers argue that it is cumbersome to use and even review formal specifications. Therefore integration of UML in the formal development can be seen as a way to cope with this prejudice. For instance, by designing a UML representation of formal specifications we facilitate understanding of them.

The role of UML is to act as a graphical interface to the formal methods. We depict the informal requirements with UML diagrams. The services provided by the system are given in use case diagrams and the dynamic behaviour of these services are described in statechart diagrams. The class diagrams show the static behaviour of the system. The specifications of the use cases describe explicitly not only the normal behaviour of the system but also the error situations. We describe an error situation and a corresponding system reaction on it. This is a convenient way to document fault-tolerance design decisions. Already the initial specification should ensure safety and be proved to be consistent. This can be achieved by translating the UML specification (the abstract class diagrams and statechart diagrams) into B-action systems. The B Method and its

associated tool Atelier B provide us with a good mechanised support for the consistency proof of the B-action systems. An example of this translation can be found in deliverable D7. When further developing the system by adding more implementation details to it using superposition refinement, we benefit from the graphical features provided by UML giving class and statechart diagrams for each development step. Hence, the refinement can be performed on the class diagrams by adding new attributes, as well as on the statechart diagrams by adding new states and taking the new attributes into consideration when describing the service. We also represent the error situations and their mitigation in a more concrete way in the statechart diagrams. In order to be able to prove the correctness of the refinement step we then translate the refined UML model with the tool U2B into B-action systems and prove the correctness of the refinement step with the autoprover and the interprover of AtelierB. In this way we have a uniform documentation in UML of the whole development at the same time as we have the system formally proved within B.

ii). Use of stepwise development paradigm. Wallac's development cycle starts with no exact specification of the final implementation, i.e., hardware configuration, possible faults, the way functions are to be implemented etc. The problem with the absence of final implementation details can be dealt with by stepwise refinement. We start from an abstract specification of the intended functionality and try to fit the implementation details as we proceed. Using superposition refinement we stepwise add more functionality to the specification and turn it into a more concrete and deterministic system. In each step, safety properties of the system are preserved. Each step is proved using the provers of Atelier B. The use of abstraction allows us to solve one more acute problem – dealing with frequent changes in design decisions influenced by the results of testing. While developing the initial specification we try to avoid all the implementation details. It gives us freedom to refine the system in many different ways. Obviously, the changes in the design decisions might affect the requirements which are captured already in the preceding refinement steps. It would require to redo some refinement steps. However, while discussing our development with Wallac we usually try to come up with a consensus regarding the requirements to be captured in the forthcoming refinement steps.

In general we can state that design changes that involve changes in the safety properties will have to be incorporated already in the specification and will require refinement steps to be redone. Also changes in the services provided by the system need to be incorporated already in the abstract specification. For example, in the Fillwell development process new safety requirements were introduced on the system under construction. Several components – the signaling lamps — were added and their behaviour was included in the safety requirements. These changes had a significant impact on the formal development and even implied changes of the initial specification. Design changes in the form of more detailed information (new features) on parts of the system can be added in forthcoming steps. As an example of this in the Fillwell case study we can mention the speed of the gantry. Moreover, for the protocol runner it became clear later in the design that (nested) loops and constants would be needed in the protocol. These features could, however, easily be added as a last refinement step of the protocol runner.

iii). Incorporation of safety analysis and fault tolerance. Safety practitioners argue that proper system safety can only be achieved if safety consideration starts from early stages of system development [12]. However, Wallac's as other industrial developer's development practice is to incorporate safety and fault tolerance measures only after faults in the design have been discovered by testing. Such an approach is one of the main causes of frequent changes in the

design decisions. We propose to incorporate fault tolerance in the formal development cycle. By developing a system by stepwise refinement we ensure that the final implementation exhibits the same globally observed behaviour as the initial abstract specification. Therefore, if the final implementation may fail (even in a safe way) we need to reserve that possibility in the abstract specification, too. The failure of a system is a result of the faults of its components. However, the failure modes of the system components are not identified initially. In the initial specification we specify the effect of errors on system's services and give an abstract representation of the classes of errors. At the later refinement steps we try to identify the components faults which manifest themselves as the system errors. Furthermore, we elaborate on the abstract representation of errors and their mitigation by refinement.

As a conclusion of the integration of formal methods within the healthcare case study we can state that the combined use of UML and B in the methodology, providing a graphical interface to the formal method, was appreciated by the team at Wallac. The B Method forces the developers to specify the system in an unambiguous way and, hence, to think carefully about the precondition of each service and which postconditions these services should establish. The stepwise introduction of features and identification of component faults in the system provides a structured way of managing the complexity of the system. The typical course of events (use cases) treats the error situations in a very exact way, which was found to be very useful for the development. Furthermore, UML provides us with consistent documentation means at every stage of the development process. Hence, the methodology facilitates better understanding of the system and leads to better design decisions.

2.2.2 Estimating Project Resources and Costs

2.2.2.1 Comparing conventional and formal developments

In this section we provide metrics concerning conventional and formal development for the Smartcard Case Study. We use these metrics to make some comparisons. The section concerning the formal development contains also metrics about the proof activity.

In the MATISSE Smartcard Case Study we wanted to determine if formal methods can be used in developing smart cards in such a way that gains in quality come at predictable and acceptable cost. By stating this goal we defined the expected effects of using formal methods. The hypotheses to be tested are detailed enough to make clear what measurements are needed to demonstrate the effects. With the following aims, we define five hypotheses:

- H1: The use of formal methods and in particular the B method improve the quality of resulting software. It is of prime importance to demonstrate to managers that a consequent formal development increases the quality.
- H2: The cost overhead of a formal development is acceptable. It is expected that the introduction of formal methods will not raise too much the costs of development. This hypothesis asserts that different treatments have different effects on a project. To draw some conclusions from the case study result, it is necessary to have two developments to determine if there are some differences. The formal development in the case study must have a sister project using the usual procedure for development. The measurements will take into account

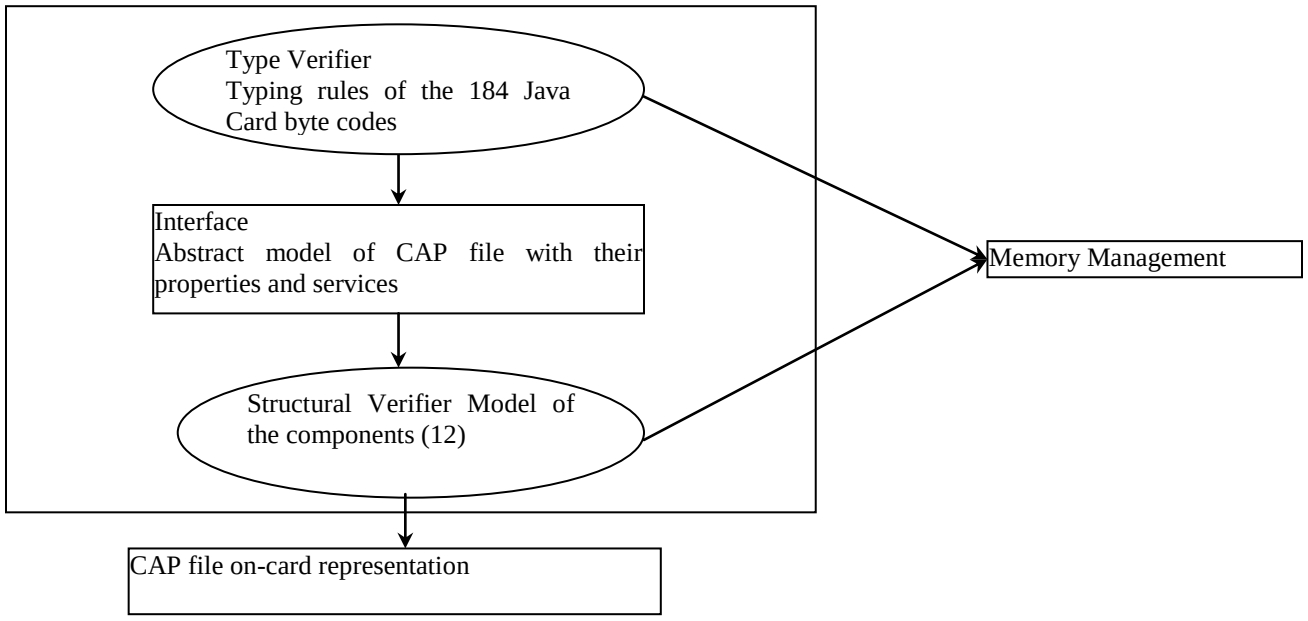
developments, proofs, reviews, tests and documentation for traditional and formal developments.

- H3: Non specialist engineer can use the formal method effectively when guided by an expert. This hypothesis will point out the importance of know-how and difficulties in using formal methods and will help in the definition of the team. It will also help the project manager to evaluate the part of subcontracting. From the Gemplus experience, it seems very useful to subcontract at the early phase of a project and possibly at the end for solving complex proofs. The know-how of an external expert will be necessary until it can be supplied by experienced practitioners within Gemplus. The second part concerns the training of the developer for formal modelling. It is claimed that B models can be developed by beginners. To verify that point a part of the development will be given to trainees and a special attention will be paid to fluctuations in measurements.
- H4: The use of formal methods and B facilitates fulfilling regulatory requirements. The Common Criteria (CC) requires the use of formal methods from the so-called EAL5 level and upwards. At the EAL4 level, only semi-formal documents are required, but to reach the higher levels formal (mathematical) proofs are required and formal methods and associated tools have to be used (*e.g.* for the formal description of security policies, formal proofs of the consistency of the security policies, *etc.*). There are already very professional development models that fulfil the regulatory requirements at Gemplus. However, formal specification and in some cases even direct software code generation from these specifications would help to improve the way of working and the productivity as well as to prepare for future regulations. A special emphasis is put on using the B method to meet these regulatory requirements. Even if formal methods are not required for meeting these regulations, they would be favourable by making it easier to trace the requirements all the way to the final code. This traceability is a very important feature, since it facilitates the reuse of code in future developments.
- H5: Code generated by the use of the B method does not have significantly increased memory requirements or execution time for constrained smartcard design principles. We have verified in previous studies that the produced code from the Atelier B had too large memory footprint to be usable. Moreover we do not know whether using the B method increases the size of the code independently of the code generator. Engineers have a great experience in generating very efficient C code for a smart card. It will be of a great interest to compare the codes because it is often said that constructs used in formal specification may not translate well into the target language leading to either an inefficient implementation, or a substantial amount of re-work to optimise the code design.

The hypotheses H1 and H2 need to have two developments of the same or similar software. In the MATISSE Project we did not have enough time to validate the hypothesis H4. The last hypothesis is needed to generate the code that fits the smart card constraints. Between the formal methods compared for Gemplus needs only, the B method proved possibility to generate acceptable code for smart cards. But the current code translator was not efficient enough for the card, so we had to developed our own translator. It was a prototype, and has not been validated within the MATISSE project.

Modeling a byte code verifier: architecture and models

In this part, we focus on the architecture and the modeling of a byte code verifier for Smartcards. We detail the architecture in a first part, then, we describe the methodology we have used to model the different parts of the byte code verifier. The figure below depicts the general architecture and introduces the two verifiers, *i.e.* the structural verifier and the type verifier. This figure helps identifying four interesting parts. The two firsts concern the structural verifier and the type verifier, *i.e.* the purpose of the modeling. A third part represents the interface. That is, the description allowing the type verifier and the structural verifier to exchange data. In fact, this interface describes the variables, the properties over these variables and the services required by the type verifier in order to execute itself. This interface is then refined by the structural verifier which provides the data required by the type verifier. The last part concerns elements that are not modeled in B. In particular, the memory management and the representation of CAP file on-card are not modeled in B.



General architecture of the formal byte code verifier

Two development teams

As we focus on a comparison of a byte code verifier, we have settled two different teams working in parallel: a team in charge of the conventional development and a team in charge of a formal development. We detail in the next section the skills of the different teams. But, we want to note here that the experiment with these two teams may have some limits. In fact, the two teams, if they do not share developers, they share often the same office and also same technical information. So, when an error is encountered or an issue is raised by one team, the other one is immediately informed.

Some times, it happened that both teams needed to work together in order to understand some of the language subtleties. In fact, both teams had a different interpretation of the smartcard specification provided by Sun. Discussion between teams, as their knowledge on Java Card is important, leads to a better comprehension of the informal specification. This helped both developments. This kind of discussion allowed both teams to confront their ideas and their interpretation of the specification. It could have avoided many errors but it is not measurable.

Apart from these two development teams, we have also settled two other teams. The first one is in charge of writing the test cases to test the two different verifiers. The second one is in charge of integrating the code produced by the two developments into a smart card. These two last teams have worked independently of the two first teams and for the two first teams by testing and integrating the code.

As indicated above, we have dedicated a team to write test cases and to develop test applets for both developments. These test cases are grouped in two different categories: the first category aims to test the type verifier and the second category aims to test the structural verifier. The documents describing the test cases are the same for both developments. However, the test applets are not the same.

In fact, to test the conventional verifier, we have developed hundreds of test applets that check the functional conformity of each byte code of the conventional type verifier. Note that the conventional verifier does not include a structural verifier. These test applets cannot be used directly to test the formal type verifier. The reason is that the algorithm implemented to perform the type verification is not the same. In the case of the formal verifier, it needs an off-card pre-computation that performs type unification. The problem is that if we generate the pre-computation on test applets, the obtained test applets are generally refused by the structural verifier. So to produce test applets that test the formal type verifier, we have to generate a correct applet with the pre-computation and then to modify it without modifying its structure. It is a very long task and we only developed few test applets. Therefore, we test the correct normal execution of the formal type verifier by using standard applets that are known correct. We have some test applets on few byte codes but not as much as for the conventional verifier. However, we have developed the test applets for the structural verifier on which we check its functional conformity.

The main point is that the testing phase of the two type verifiers is not entirely equivalent and we are aware of it when we compare the time spent on this specific step.

Generating test suites for the formal type verifier is very long, in particular computing new offset and new size in order to make these test applets pass the structural verifier to test the type verifier. We are thinking about generating tests from the B specification. This work is under investigation in the future.

Estimating resources and cost required

As defined in the evaluation plan of MATISSE we collected several metrics that could help manager to chose or not formal methods based technology for their development. Not all the case studies of MATISSE collected the same metrics. We will try here to convince project manager that the overhead in resources must be fairly balanced with the quality finally obtained. To obtain this metrics the MATISSE Smartcard case study conducted two developments one conventional

and one using the B method. With the latest development, two components have been developed, namely a structural verifier and a type verifier.

The conventional development takes into account the verification technique chosen for this development as well as the skill of the developer. The idea is to provide the context of the development in order to ease the comparison and to identify the key point of the comparison. In particular, we clearly identify the verification technique used in the conventional development and the skills of the developer. The developer of the conventional verifier is already familiar with C development, as the conventional development is performed in C. He is also aware of developing code for smart card. Developing for smart card is quite different than developing for standard targets. It is mainly due to the smart card constraints, both in terms of memory footprint and in terms of program complexity. As a consequence, there is no need to train the developer on the programming language (C code), on the smart card special programming constraints, nor on the verification techniques. However, as the developer is not familiar with Java Card, the language on which is based the type verifier, time is needed to learn the language. Moreover, time is required to learn the byte code file format in which information mandatory to the verification is stored.

The developer of the formal verifier is already familiar with the B method. Building models and proving them do not require any training for him. Hence, one can say that he is an expert. However, he was nor familiar with smart card neither with byte code verification. So, for instance, he had no idea of code optimisations necessary to fit smart card constraints. This may not be a real issue since formal implementations are automatically translated into C code by using tools. Therefore, optimisation may occur during this translation phase. The fact that the formal developer is not aware of byte code verification is not a major inconvenient. In fact, the modelling activities help understanding the problem while constructing the formal specification. Moreover, the formal developer has a direct access to the knowledge of Java Card experts who help him understanding tricky problems. This particular point allows the formal developer to reduce the time needed to understand the Java Card language.

Table 1 synthesises metrics related to the development. In particular, we can note that the structural verifier is bigger than the type verifier. The reason is that the structural verifier contains a lot of tests which require specifications and implementation for each. The type verifier can be seen as a single machine including the typing rules enforced by Java Card. Moreover, the structural verifier contains services on which the type verifier relies. This explains the difference in the number of components as services are organised in different sets.

There are two other results that are remarkable: the first one concerns the number of generated Proof Obligations (POs). The results shows that the type verifier generates many more POs than the structural verifier. The reason is that there are many more properties in the type verifier than in the structural verifier.

The second results concern the number of C code lines. This number is far smaller than that the of corresponding B code. The reason is that in the code translation, only implementations are taken into account. Moreover, INVARIANT clauses within implementations are not translated. This drastically reduces the number of lines translated from B to C.

	Structural Verifier	Type Verifier	Total
Number of lines of B	35000	20000	55000
Number of components	116	34	150
Number of generated POs	11700	18600	30300
POs automatically proved (%)	81 %	72 %	75 %
Project status	90 %	99.9%	95 %
Number of Basic machines	6	0	6
Number of lines of C code	7540	4250	11790
Workload (men months)	8	4	12

Table 1. *Metrics on the formal development of the byte code verifier*

Two developments were completed concerning only the type verifier: one used formal techniques and the other used traditional techniques. Each development was done by a different person. They both had the same starting point, i.e. an internal document emphasising the requirements of a Java Card type verifier, written in natural language. For each development, we provided a test phase. This step allowed us to check the correspondence between the informal requirements and the code embedded into the smart card. The following tables describe the elements of the comparison.

Table 2 summarises the number of errors found and the step of the development where they were found. The first conclusion from this comparison is that the formal development produces fewer errors than the traditional development: 56 errors compared to 95. Moreover, only 14 errors were found during the testing step. This is in accordance with the fact that only one week was used to perform the test of the verifier.

Compared to the 95 errors of the conventional development and the 3 weeks of testing, there is a significant difference. Unfortunately for the formal development, the proof is very long and costly. However, we believe that this can be decreased thanks to Atelier B improvements and to the development of particular rules and proof tactics.

If we manage to capitalise on experience gained in initial developments, we should decrease the time needed by the proof. Moreover, by proposing a methodology adapted to the smart card, the development time required to build models can also be decreased.

Using formal methods could then be a real advantage as it is no more costly than conventional development while providing high-quality code.

	Formal development	Conventional development
Number of errors discovered by reading	13	24
Number of errors discovered by proof	29	Not applicable
Number of errors discovered by testing related to the type verifier	14	71
Total Number of errors	56	95

Table. 2. *Comparing number of errors for formal and conventional development*

Finally, if the number of errors discovered by reviewing for the formal development is smaller than for the conventional one that is because the modelling activity requires a good understanding of the informal specification and a lot of work required by the refinement method. Errors still exist but the modelling activity helps to clarify the specification by going deeper into the meanings of the specification and thus, reduces the risk of introducing errors.

Table. 2 helps us to state that the code produced through a formal process contains fewer errors which indicates a better quality in terms of compliance with the original requirements.

	Formal development	Conventional development
Main development (weeks)	12	12
Proof activity (weeks)	6	Not applicable
Testing (weeks)	1	3
Integration (weeks)	1	2
Number of weeks for the development	20	17

Table. 3. *Comparing development time for formal and conventional development*

One of the main results of this case study and of this comparison is that it does not appear unreasonable to use formal methods to develop parts of a smart card operating system (Table 3). Moreover, even if in this study the time needed for the formal development is greater than that for a conventional development, it is only a difference of three weeks. With a strong involvement in tool improvement and in methodologies, it is possible to be competitive using formal methods. We demonstrate the possibility and the feasibility of developing parts of the operating system or parts of the Java Card Virtual Machine. Finally, we have also shown that we can control the development time of the formal verifier. Thus, with this experiment, it is now possible to be more accurate about the time required for a given development. Finally, Table 2 & 3 allows us to conclude that it is possible to develop a realistic application with an acceptable overhead, i.e., one induced by using formal method is acceptable compared to a conventional development. This conclusion takes into account the fact that the tools and the methodologies are not yet optimised

for the smart cards. Hence, we expect to reduce in particular the cost of the proof activity, by developing tools and proof rules to speed up the proof process. Experience gained in the first formal developments should improve our knowledge and speed up future developments.

The last comparison that we can make concerns the efficiency of the produced code, both in terms of the size of the code and in terms of time required to verify an applet.

Table 4 and Table 5 contain the results of the comparison. The first comment is that the code obtained and translated from the B is acceptable. Concerning the type verifier, we note that the two sizes are similar. This table also shows that the RAM usage is acceptable for a verification algorithm. Note that there is a large range of RAM usage for the conventional verifier as RAM usage is adaptable for this verifier. We also provide the size of the structural verifier but we cannot compare it as it has not been developed yet for the conventional part. The difference that we can note on the total size regarding the other sizes is that in the total size, we include libraries and APIs necessary for both verifiers inside the card. It includes notably the loader, the memory management and the communication.

	Formal development	**Conventional development**
Type verifier ROM size (kb)	18	16
Structural Verifier ROM size (kb)	24	Not Yet Implemented
Total ROM size (kb)	45	24
RAM usage (bytes)	140	128-756
Applet code overhead (%)	10-20	0

Table 4. *Comparing code size for both developments*

Table 5 proposes a comparison of execution time for a set of applets. Note that the two implementations, the formal and the conventional ones, are not actually done on the same chip. The formal verifier is implemented on an ATMEL SC 6464 C and the conventional one on an ATMEL SC 3232. The main difference between those two chips is the free memory size (greater in the case of the formal development). Note also that the conventional verifier does not include a structural verifier. Hence, we cannot compare the time for this particular part. However, we provide the information in order to compare the structural verifier's complexity with that of the type verifier.

The main observation about the execution time is that the conventional type verifier is twice as fast as the formal one. There can be several reasons to explain this difference. The first is the difference of memory management between the two developments. The conventional one uses a pointer to access to the memory. In the formal one, the pointer is a translation from the one in B. Therefore, each time we access the memory, there is a translation which costs some time in the execution. Another reason is that the developments were done to optimise the size of the code, not its efficiency. So, when the code is compiled, the compilation directives that are used aimed to optimise the size. The conventional development already takes into account some efficiency

optimisations that the formal one does not. Hence the difference of execution time. We have performed some optimisations on the memory management of the formal type verifier. The obtained results show that, with these optimisations, the execution times on the different applets are now similar.

Table 4 and Table 5 help us conclude that the code generated with the C code translator from the formal implementation fits the smart card constraints.

	Formal development		**Conventional development**
	Type Time (ms)	Type Time after optimisation(ms)	Type Time(ms)
Wallet	811	460	318
Utils	2794	1422	1463
Pacap Interface	241	110	61
Tic Tac Toe	3555	1372	1102

Table. 5. *Comparing verification time for a set of example applets*

Conclusion

Comparing these two developments is not so easy. In fact, both developments concern a type verifier but at different level of abstraction. However, it appears that these two developments are equivalent in terms of size and complexity and it is the reason why we aimed to compare them. Therefore, this comparison is encouraging for future formal developments. Of course the data collected concern only a single development. To be more accurate, this kind of comparison should be repeated on other applications. But, this is a realistic application and we think that the comparison is reasonable.

Concerning the hypothesis made in the evaluation plan, it appears that for this Case Study the hypothesis H3 cannot be validated. In fact, the formal developer was already considered as an expert. The possibility of having non specialist engineers use formal methods has not be studied. However, we have studied the impact of having an expert as developer: the development is in-line with the development schedule. The hypothesis H4 cannot be validated as well, as the formal development has not been yet proposed for certification.

The hypothesis H1 has been validated. The hypothesis H2 has been also validated.

The hypothesis H5 has been validated by our experience. The code generated from B0 implementation has been compiled and integrated into a smart card chip. The memory footprint required by the code can be improved as the translator we use is a very basic one. One can imagine some improvement and optimisation of the code translator in order to reduce the memory footprint.

2.2.2.2 PLC resources following the Methodology in the Transportation Case Study

The development and validation activities of the safety critical systems are carried out at STS [20, 15] by three teams: the development teams, the support team, the validation team. *(Resources, Requirements and Risk Threads in Fig1)*

Development Teams

Each safety critical software product (for wayside, on-board, control-center equipment) is developed by one development team. A development team is in charge of the whole development of their software from the *Software Requirements* document to the software/hardware integration. A team includes about two third of engineers trained in B (Costs, Skills, Resources in Fig1).

Concerning the formal model, the teams carry out the following activities:

1. Production of the abstract and concrete models.

Two main documents are used for this:

(a) the *Software Requirements* document, to produce the abstract model,
(b) the *B Book*, in order to use B efficiently.

The models are elaborated in a rigorous compliance with these documents.

2. Cross-reading of the abstract model.

Every component of the abstract model written by a developer is cross-read by another developer of the same team in order to verify its conformity with regards to the software requirements.

3. Automatic and interactive proof of these models.

A developer makes the complete proofs of his models. The interactive proof begins only when the model is mature enough in order to avoid to make proof several times. During the interactive proof, the developer may add new mathematical rules. (Training)

4. Documentation of the models.

Two documents are produced. The Abstract Model document gives the traceability of the abstract model with the Software Requirements document and justifies the modelisation choices. The Concrete Model document mainly justifies the design decisions taken in the refinement steps.

Validation Team.

Validation activities are realized by a software validation team, strictly independent from development teams, the general mission of which is to guarantee the dependability (reliability, availability, maintainability and safety) of the systems.

This team carries out the following activities:

1. Verification of consistency and completeness of the abstract models with regards to the Software Requirements document.
2. Verification of the interfaces between B and non-B subsystems.
3. Validation of the added proof rules.
   Each rule that is not proven by the automatic rules prover is proven by hand.
4. Verification of the proof.
   All the proof generation process is validated. The proof activity is automatically run again in order to verify that all the lemmas are proven.

Support Team.

This team is composed of engineers expert on the B method. It carries out the following activities:

1. Support of the development and validation teams.
2. Reviews of the formal models.
   Each component produced by the developers is reviewed by the support team in order to verify its conformity to the rules of the B Method. It will guarantee in particular that the proof will keep feasible and as easy as possible.
3. Reviews of the added mathematical rules.
   Each rule added by a developer is reviewed in order to verify that it is mathematically correct. At this step the verification is not complete: the reviewer just read the rule and convinces himself that the rule seems right. If the rule is considered as wrong, the developer will have to remove or change the rule and make another proof.
4. Cross-reading the abstract model.
   Each component of the model written by a developer is cross-read by the support team in order to verify that it is provable and easy to read.
5. Reviews of the verification analysis.
   Each analysis produced by the validation team during verification activities is reviewed by the support team in order to control its conformity to the rules of the B Validation Book.

Links Between the Teams.

During the development of the software the support team works together with the development team. The development is considered as finished when:

1. The system and software requirements specification are considered as fully mature.
2. All the reviews have been passed.
3. All the proof have been done.
4. All the nominal functional tests have been passed successfully.

The link between the support and the development or validation team is strong. On the other hand, development and validation teams work independently.

Three courses are usually provided by the support team.

1. Basic common course.
   The objective is to make people familiar with B models, in order for them to be able to read and understand B models, on the basis of the B-Book. People from development and validation teams are attending this one-week course.
2. Development course.

The objective of this (one-week) course is to teach how to elaborate large abstract models in B, to prove them and to derive proven concrete models.

3. Validation course.
   The objective of this (one-week) course is to teach how to validate a development with B.

2.2.3 Certifying Developments (SmartCard Case Study)

2.2.3.1 Security certification requirements

Application security can be enforced using rigorous processes. In order to enforce even more this security, smart-card manufacturers submit more and more often their products and processes to evaluation. The Common Criteria for Information Technology Security Evaluation (CC) standard define a set of criteria to evaluate the security properties of a product in term of confidentiality, integrity and availability. The CC focus mainly on the first part of the lifecycle: requirements, specifications, design, development and test. The CC take into account the security requirement documents, which are a complement of the general requirement document. Indeed, the CC are only concerned by the security aspects of the system.

The CC present the security requirements considering distinct categories of functional requirements (*e.g.*, requirements for identification, authentication, non-repudiation…) and assurance requirements (*e.g.*, constraints on the development process rigor, impacts of potential security vulnerabilities…). The Target Of Evaluation (TOE) is the part of the product or the system that is subject to evaluation. The assurance that the security objectives are achieved is linked to:

- The confidence in the correctness of the security functions implementation, *i.e.*, the assessment whether they are correctly implemented,
- The confidence in the effectiveness of the security functions, *i.e.*, the assessment whether they actually satisfy the stated security objectives.

The Evaluation Assurance Levels (EAL1 to EAL7) form an ordered set to allow simple comparison between TOEs of the same kind. At EAL5 level the assurance is gained through a **formal** model of the TOE **security policy** and a **semiformal** presentation of the functional specification and high-level design and a **semiformal** demonstration of *correspondence* between them. Note that the analysis must include validation of the developer's covert channel analysis and a strong vulnerability analysis. The last EAL levels require a formal in-depth and exhaustive analysis.

Three types of specification styles are mandated by the CC: informal, semiformal and formal. An informal specification is written in natural language and is not subject to any notational restriction but it requires defining the meanings of the used terms. A **semiformal** notation is written with a restricted syntax language and may be diagrammatic (data-flow diagrams, state transition diagrams, entity-relationship diagrams, etc). A **formal** description is written in a notation based upon well-established mathematical concepts. These concepts define the syntax and the semantics of the notation and the proof rules that support logical reasoning. A *correspondence* can take the form of an informal demonstration, a **semiformal** demonstration or a **formal** proof. A semiformal demonstration of correspondence requires a structured approach at the analysis of the *correspondence*. A formal proof requires well-established mathematical concepts and the ability to express the security properties in the formal specification language.

At the highest level (EAL7) formal methods are needed at several stages but only for one part of the requirements: the development. For the test and vulnerability assessment formal methods are not mandatory. But they can provide a useful help by reducing the cost of the certification process.

2.2.3.2 Use of formal method for development

The high-level design (HLD) is a refinement of the whole functional specification in a modular way. Thus, the set of the modular TSF models can represent a convenient base for the high level formal model. The same remark can be expressed about the low-level formal model (LLD) that shall also be provided. The HLD, LLD and FSP formal model must be consistent, and show evidence of the complete instantiation of the TOE requirements. The proof or the formal demonstration of the model, according to the security formal properties is mandatory. The representation correspondence (RCR) between two formal descriptions shall be formal provided by the proof process. The correspondence between the LLD and HLD shall be semiformal. This semiformal correspondence is based on the identification of a correspondence between the concrete data of both formal and semiformal models. The correspondence reveals a similar evolution of all the data and states of the TSF. The HLD formal model is obtained by refinement of the security function specifications, which is an abstraction of the function descriptions. The properties of the security function mechanisms must be inherited from their corresponding security policy models.

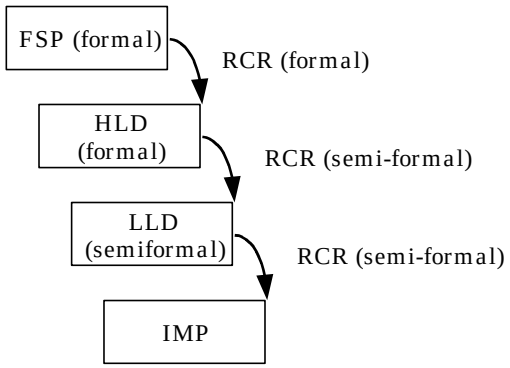


Fig. 1. Use of formal method for development requirements for EAL5

A formal model designed for development is not usable as is for certification but can provide a skeleton. The process is based on the refinement of the model and thus can take benefit of the design.

3 References

1. The B Book: *Assigning programs to meanings*, Abrial, 1996, Prentice-Hall
2. Communicating Sequential Processes, Hoare 1985, Prentice-Hall.
3. The Theory and Practice of Concurrency, Roscoe 1998, Prentice-Hall.
4. FDR user manual, 1997, Formal Systems (Europe) Ltd
5. Using encryption for authentication in large networks of computers, Needham and Schroeder 1978, Communications of the ACM, 21(12):120-126.
6. R.J.R. Back and K. Sere. *From modular systems to action systems.* Software - Concepts and Tools 17, pp. 26-39, 1996.
7. Stéria Méditerranée. *Atelier B.* France, 1996.
8. *Event B Reference Manual* (Draft) v1. ClearSy, 2001.
9. *FDR user manual*, Formal Systems (Europe) Ltd, 1997.
10. Fillwell™2002 – Features Guide. Via http://lifesciences.perkinelmer.com/.
11. L. Petre and K. Sere. Developing Control Systems Components. In *Proceedings of IFM'2000* - Second International Conference on Integrated Formal Methods, Germany, November 2000. LNCS 1945, pp. 156-175, Springer-Verlag.
12. E. Troubitsyna. *Stepwise Development of Dependable Systems*.Turku Centre for Computer Science, TUCS, Ph.D. thesis No.29. June 2000.
13. M. Walden and K. Sere. *Reasoning About Action Systems Using the B-Method. Formal Methods in Systems Design 13(5-35),* Kluwer Academic Publisher, 1998.
14. J.-R. Abrial. Extending B without changing it (for developing Distributed Systems). In Henri Abrias, editor, *Proceedings of the 1st Conference on the B Method*, Putting into practice methods and tools for information system design, pages 169-191, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de Recherche en Informatique de Nantes.
15. P. Behm, P. Benoit, A. Faivre and J.-M. Meynadier. METEOR: A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, 1999.
16. T. Muntean, O. Rolland "Distributed Refinement in the B Method", Workshop RCS'02, http://www.esil.univ-mrs.fr/~spc/rcs02/rcs02.html
17. Trusted system construction, O'Halloran, 1999, in Proceedings of IEEE Security Foundations Workshop.
18. L. Casset, *Construction correcte de Logiciels Critiques pour Cartes à Puces avec la Methode B*, PhD Thesis, Marseille, France, October 2002.
19. L. Casset, L. Burdy, A. Requet, Formal Development for an Embedded Verifier for Java Card Byte Code, Proc. DSN Conf., Washington, 2002.
20. J. Falampin, STS, From an event-based B model to a B software, RCS'02 Workshop http://www.esil.univ-mrs.fr/~spc/rcs02/rcs02.html