MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering

IST-1999-11435

MATISSE Handbook for Correct Systems Construction

Traian MUNTEAN (Ed)- CNRS& Université de la Méditerranée-Marseille

April 2003 - muntean@imag.fr
### Project Information

<table>
<thead>
<tr>
<th>Project Number</th>
<th>IST-1999-11435</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title</td>
<td>Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE)</td>
</tr>
<tr>
<td>Website</td>
<td><a href="http://www.matisse.qinetiq.com">www.matisse.qinetiq.com</a></td>
</tr>
<tr>
<td>Partners</td>
<td>Aabo Akademi University</td>
</tr>
<tr>
<td></td>
<td>ClearSy</td>
</tr>
<tr>
<td></td>
<td>CNRS-TIMA Laboratory Grenoble (SC)</td>
</tr>
<tr>
<td></td>
<td>Gemplus</td>
</tr>
<tr>
<td></td>
<td>Siemens Transportation Systems</td>
</tr>
<tr>
<td></td>
<td>University of Southampton</td>
</tr>
<tr>
<td></td>
<td>QinetiQ (C) &amp; Formal Systems-Subcontractor</td>
</tr>
</tbody>
</table>

### Document Information

<table>
<thead>
<tr>
<th>Document Title</th>
<th>MATISSE Handbook for Correct Systems Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workpackage</td>
<td>WP1</td>
</tr>
<tr>
<td>Document number</td>
<td>D10</td>
</tr>
<tr>
<td>Lead Partner</td>
<td>CNRS</td>
</tr>
<tr>
<td>Editor</td>
<td>Traian Muntean</td>
</tr>
<tr>
<td>Contributors</td>
<td>All partners</td>
</tr>
<tr>
<td>Due date</td>
<td>M34 (end of project)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Matisse project final version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 2
PREFACE

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.

Methods, tools 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 therefore composed of three parts: the board level storyboard, project manager's storyboard, and practitioner's 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:

- automatic transportation system
- multi-application smartcard embedded byte-code verifier
- health care control system

This document is composed of the above three level storyboards.

Each level addresses different audiences and therefore has been written in order to be read independently if required. Nevertheless the three storyboards are composing a unique handbook which is intended to be yet improved before the publication of this project deliverable (which will be accessible at http://www.esil.univ-mrs.fr/~spc/matisse/Handbook).
1 Introduction

The core objective of the MATISSE project is the development of industrial-strength methodologies and associated technologies for the engineering of software-based critical systems. These methodologies and technologies will support industry in providing essential services that are highly dependable and therefore lead to increased public confidence and trust in the services.

This document briefly describes the benefits of using the MATISSE methodology and what are its impact on the organisation structure. Data presented in this handbook was obtained from the experience gained in developing the project case studies and from external projects using / extending techniques and methods elaborated in parallel.

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, that support validation throughout the development lifecycle 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 industry because of various managerial, sociological and technological barriers.

2 The 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 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, integrated with standard working practices, for correct systems construction through formal refinement;
- 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.
IST-1999-11435

MATISSE Handbook

Part I: Board-level Storyboard
<table>
<thead>
<tr>
<th>Document Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Title</td>
<td>MATISSE Handbook, Board-level Storyboard</td>
</tr>
<tr>
<td>Workpackage</td>
<td>WP1</td>
</tr>
<tr>
<td>Document number</td>
<td>D10 (Part A)</td>
</tr>
<tr>
<td>Editor</td>
<td>Thierry Lecomte (ClearSy)</td>
</tr>
<tr>
<td>Contributors</td>
<td>All Matisse Partners</td>
</tr>
</tbody>
</table>
1 Keys to success

This chapter draws up an outline of the benefits that are induced by the use of the MATISSE methodology, combining formal methods and existing industrial (best) practices.

1.1 Product quality

| The MATISSE methodology enables the development of systems correct by construction |

by proposing a refinement-oriented approach as a basis for critical system design. Starting from an abstract specification, the system is stepwise refined into a more concrete and deterministic system, leading to a final proved implementation. This constructive approach has been successfully and extensively used by the 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 pre-conceived notion of how an implementation may be constructed from a given specification and specific refinement transformation functions.

| The MATISSE methodology allows one to obtain clear requirements and specifications |

The main principles to be used for a B development are: to concentrate efforts on the initial phases of development in order to be “as precise as possible, as early as possible”, then to make best use of proofs to ensure the initial aims are adhered to. A formal development\(^1\) puts the stress on the specification phase, which is usually longer but leads to documents that are self-sufficient, clear and will not be subject to error-based modifications. Producing quasi-final documents for the first shot reduces the delay of acceptance and allows one to minimise the overriding of two successive phases. Moreover, the delivery of high quality documents would have a positive impact on the final customer and would increase his confidence on the ability of ones organisation to provide good products in time.

The use of formal methods can be restricted to the writing of the documents (statement of work, specification, preliminary design, detailed design), providing a good technical basis for the rest of the project and ensuring a suitable communication among the different partners involved in the development, especially in the case of sub-contractors.

---

\(^1\) By development, we mean: writing of the specification document, design and code generation.
1.2 Cost/benefit issues

The MATISSE methodology does not imply a significant overhead when compared with conventional development, if a dedicated methodology has been previously experimented and adapted to manufacturer’s needs. Developing a product using formal methods is comparable, in terms of cost, with developing a safety critical product, as heavy verifications have to be performed alongside the ordinary development process. The MATISSE methodology is model-based, so for similar products, the related models may be reused from one product to another with only slight modifications.

In the case of a software development:

- 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.

1.3 Regulation compliance

The MATISSE methodology allows the development of regulatory compliant equipment. The use of formal methods is recommended, even highly recommended, when building safety or security-related products, for example EAL7 for information systems and SIL4\(^3\) for railway systems [4], automotive, chemical systems [3], etc. Certification\(^4\) is eased because:

- Formalism helps one to think about the system and its properties. Formal proof allows one to verify that the behaviour of the system is correct in all the cases, not only in those one has in mind. So bad surprises are unlikely to occur during evaluation and multiple iterations are prevented.
- Formal justification of the design choices allow the evaluators to come more quickly to a conclusion, as they do not need to informally prove them. The

\(^2\) Product line for example.

\(^3\) Safety Integrity Level: range 0 to 4, where 4 is the highest level. The Mean Time To Failure of a SIL4 product is about 100 000 years.

\(^4\) Verification of the conformance of a product with its referential.
number of questions is minimised and the duration of the evaluation process is reduced.

- The MATISSE methodology is model-based, so for similar products\(^5\), the related models may be reused from one product to another with slight modifications. In this case, differential evaluation of those models can be performed at minimal cost, as the impact of those modifications can be easily computed and formally verified.

### 1.4 Competition

<table>
<thead>
<tr>
<th>The MATISSE methodology helps to maintain competitiveness.</th>
</tr>
</thead>
</table>

Formal methods are not required by the market, as most of the time, final consumers are not aware of the use/embedding of such methods: application fields are mainly driven by cost reduction. However, depending on the constraints of each domain, the use of formal methods may be positively considered by regulation authorities, for example:

- in the railway domain, most competitors have integrated formal methods into their development process (B at Siemens Transportation Systems and Alstom Transportation, SCADE at CSEE, SDL at RATP, etc.), as safety issues are now of paramount importance.
- in the smartcard domain, many competitors (like Gemplus and Schlumberger) have investigated the use of formal methods (B, coq, PVS, etc.). Their common objective is the building of highest security level products (up to EAL7), for which

### 1.5 Teams

<table>
<thead>
<tr>
<th>The MATISSE methodology does not require development teams to be changed.</th>
</tr>
</thead>
</table>

Setting up a formal methods activity does not require the complete modification of the existing structure. Functional experts (ako local gurus) need to be inserted into this structure, to advise project leaders, from a strategic point of view (metrics, identification of technological locks, …), and practitioners, from a tactical point of view (modelling, proof, …). Such experts would have at least one year experience in using formal methods. Light training (up to three weeks), immediately followed by full-time practice,

---

\(^5\) Product line for example.
is sufficient to have a junior practitioner in a formal methods project. External consultants could provide assistance on some particular aspects of the development (modelling, proof, etc.).

As formal development life cycles are similar to non-formal ones, current management teams need to be lightly trained in order to understand the vocabulary and gain some insights into the specific aspects of formal projects.

1.6 Technology

| The MATISSE methodology comes along with existing, mature tools. |

Atelier B [5] and FDR [2] tools, supporting the MATISSE methodology, have been applied on industrial case-studies, up to 100,000 lines for the former. The automatic prover of Atelier B has been designed to demonstrate one proof obligation in 10 seconds or less (mean time), allowing one to prove an industrial-size project in less than one day of computation. Training sessions and support are available for both Atelier B and FDR.
2 References

4. EN 50129 : *electronic safety critical signalling systems*
## Contents for Part II

1 Introduction  
   1.1 Intended Audience  
   1.2 Development Life Cycle  
   1.3 MATISSE Approach  
   1.4 Classical software development cycle versus B 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  
      2.1.2 Threads which drill down from the project management process to the formal method  
      2.1.3 Threads which focus up to the risk register  
      2.1.4 Other threads which focus up  
      2.1.5 Steerage nodes which link formal methods, project management process and external interfaces  
      2.1.6 Discussion  
   2.2 Threads of the MATISSE Methodology useful to Projects Managers  
      2.2.1 Integrating a Refinement Strategy in the Project Life Cycle (PLC)  
      2.2.2 Estimating Project Resources and Costs  
      2.2.3 Certifying Developments (SmartCard Case Study)  
3 References
1 Introduction

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
### Characteristics of a Formal Method

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Does it meet Objectives</th>
<th>Does it Work</th>
<th>Ease of Use</th>
<th>Is it Repeatable</th>
<th>Does it keep Working</th>
<th>Can it be Used Again</th>
<th>What does it work on</th>
<th>Can it be Disposed of</th>
<th>Is it Future Proof</th>
<th>Can it be Measured</th>
<th>Ease of Resourcing</th>
<th>Ease of Integration</th>
<th>Does it add Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Suitability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Reliability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Understandability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Consistency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Maintainability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 Portability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 Disposability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 Upgradability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 Measurability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 Ubiquity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 Modularity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 Productivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 that 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:

<table>
<thead>
<tr>
<th>Force</th>
<th>Proof time per lemma</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less than 10 s</td>
<td>70 %</td>
</tr>
<tr>
<td>1</td>
<td>From some seconds to 3 minutes</td>
<td>+ 1 %</td>
</tr>
<tr>
<td>2</td>
<td>From some minutes to tens of minutes</td>
<td>+ 3 %</td>
</tr>
<tr>
<td>3</td>
<td>From tens of minutes to hours</td>
<td>+ 1 %</td>
</tr>
<tr>
<td>Fast</td>
<td>Less than 3 s</td>
<td>30 %</td>
</tr>
</tbody>
</table>

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 develop 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.

Sometimes, 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 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.

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

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.

<table>
<thead>
<tr>
<th></th>
<th>Formal development</th>
<th>Conventional development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of errors discovered by reading</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>Number of errors discovered by proof</td>
<td>29</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Number of errors discovered by testing related to the type verifier</td>
<td>14</td>
<td>71</td>
</tr>
<tr>
<td>Total Number of errors</td>
<td>56</td>
<td>95</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Development Activity</th>
<th>Formal Development (weeks)</th>
<th>Conventional Development (weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main development</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Proof activity</td>
<td>6</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Testing</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Integration</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of weeks for the development</td>
<td>20</td>
<td>17</td>
</tr>
</tbody>
</table>

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.
Table 4. Comparing code size for both developments

<table>
<thead>
<tr>
<th></th>
<th>Formal development</th>
<th>Conventional development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type verifier ROM size (kb)</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>Structural Verifier ROM size  (kb)</td>
<td>24</td>
<td>Not Yet Implemented</td>
</tr>
<tr>
<td>Total ROM size (kb)</td>
<td>45</td>
<td>24</td>
</tr>
<tr>
<td>RAM usage (bytes)</td>
<td>140</td>
<td>128-756</td>
</tr>
<tr>
<td>Applet code overhead (%)</td>
<td>10-20</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Formal development</th>
<th>Conventional development</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type (ms)</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>Wallet</td>
<td>811</td>
<td>460</td>
</tr>
<tr>
<td>Utils</td>
<td>2794</td>
<td>1422</td>
</tr>
<tr>
<td>Pacap Interface</td>
<td>241</td>
<td>110</td>
</tr>
<tr>
<td>Tic Tac Toe</td>
<td>3555</td>
<td>1372</td>
</tr>
</tbody>
</table>

### 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.

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, \textit{i.e.}, the assessment whether they are correctly implemented,

• The confidence in the effectiveness of the security functions, \textit{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 \textbf{formal} model of the TOE security policy and a \textbf{semiformal} presentation of the functional specification and high-level design and a \textbf{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 \textbf{semiformal} notation is written with a restricted syntax language and may be diagrammatic (data-flow diagrams, state transition diagrams, entity-relationship diagrams, etc). A \textbf{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 \textit{correspondence} can take the form of an informal demonstration, a \textbf{semiformal} demonstration or a \textbf{formal} proof. A semiformal demonstration of correspondence requires a structured approach at the analysis of the \textit{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.

\subsection*{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.

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

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
3.2.2.5 Creating a B Model from UML ................................................................. 132
3.3 Computation Models .......................................................................................... 137
  3.3.1 Modelling a Byte Code Verifier ..................................................................... 137
    3.3.1.1 Modelling a Byte Code Instruction ......................................................... 137
    3.3.1.2 Modelling a Structural Test ................................................................. 143
4 Analysing Models .................................................................................................. 151
  4.1 Verifying Models in B ...................................................................................... 151
  4.2 Expert Validation ............................................................................................ 152
    4.2.1 Transportation Case Study ....................................................................... 152
      4.2.1.1 Validation of the Properties, Events and Automatons Document .......... 153
      4.2.1.2 Completeness and Correctness of the System B Model ...................... 154
    4.2.2 Smart Card Case Study ............................................................................ 154
  4.3 Safety Analysis ............................................................................................... 155
    4.3.1 Hazard Analysis ....................................................................................... 155
    4.3.2 Software Development ............................................................................ 157
    4.3.3 Dependability Impairment ....................................................................... 158
    4.3.4 Formal Specification and Safety Analysis ................................................. 159
  4.4 Proof ................................................................................................................ 160
    4.4.1 An Automated Proof Strategy ................................................................. 160
      4.4.1.1 The General Approach ....................................................................... 160
      4.4.1.2 Automatic Proof Rate as a Quality Indicator ....................................... 161
      4.4.1.3 Playing with Forces ............................................................................ 164
    4.4.2 Manual Proof ........................................................................................... 165
      4.4.2.1 PO Inspection ..................................................................................... 166
      4.4.2.2 Proving Manually ............................................................................. 166
      4.4.2.3 Example of Manual Proof ............................................................... 167
    4.4.3 Added Rules Management ........................................................................ 178
5 Refining Models .................................................................................................... 180
  5.1 Introduction .................................................................................................... 180
    5.1.1 B Refinement ......................................................................................... 181
5.1.1.1 Example ........................................................................................................... 181
5.1.2 Event B Refinement .......................................................................................... 184
5.1.3 B-Action System Refinement .......................................................................... 186
5.2 Iterative Construction of Gluing Invariants for Refinement ............................... 187
5.2.1 Refined Machine ............................................................................................. 188
5.2.2 Gluing Invariant .............................................................................................. 189
5.2.3 Further Proof Obligations ................................................................................. 194
5.2.4 Constant Properties .......................................................................................... 199
5.3 Guidance on Proving Loops ................................................................................. 200
5.3.1 Overview .......................................................................................................... 200
5.3.2 Loop Development ........................................................................................... 201
5.3.3 Loop Invariants ................................................................................................. 201
5.4 Refining from System Level to Software Level Leading into Decomposition ...... 202
5.4.1 Healthcare Case Study .................................................................................... 205
5.4.1.1 Determining Controller and Plant ................................................................. 207
5.4.1.2 Determining Sensors and Actuators ............................................................... 208
5.4.2 Transportation Case Study .............................................................................. 210
5.5 Distributed Refinement ....................................................................................... 215
5.5.1 Definition .......................................................................................................... 215
5.5.2 Creating, Splitting and Grouping Processes ....................................................... 216
5.5.2.1 Process Creation ............................................................................................ 216
5.5.2.2 Grouping by sequence .................................................................................. 217
5.5.2.3 Grouping by loop ......................................................................................... 218
5.5.2.4 Grouping/Splitting by selection .................................................................... 218
5.5.2.5 Grouping/Splitting by Condition ................................................................. 220
5.5.2.6 Splitting by Iteration .................................................................................... 221
5.5.2.7 Variables suppression ................................................................................... 221
5.5.3 Methodology ..................................................................................................... 222
5.5.4 Case study................................................................................................................223
  5.5.4.1 Average Bit Rate Protocol...............................................................................223
  5.5.4.2 Initial specification ..........................................................................................224
  5.5.4.3 First refinement ...............................................................................................225
  5.5.4.4 Second refinement ...........................................................................................226
  5.5.4.5 Third refinement ..............................................................................................228
  5.5.4.6 Fourth refinement ...........................................................................................229
  5.5.4.7 Fifth refinement ...............................................................................................231
  5.5.4.8 Sixth refinement ..............................................................................................233
  5.5.4.9 Seventh refinement ..........................................................................................235
  5.5.4.10 Last refinement: implementation.....................................................................236

5.5.5 Conclusion...............................................................................................................237

5.6 Safety Requirements in Refinement............................................................................238
  5.6.1 Safety Issues During the Refinement ..................................................................238
  5.6.2 Proving the Correctness of the Refinement.......................................................240

5.7 Refinement with UML+B ...........................................................................................242
  5.7.1 Refinement in Class Diagrams ...........................................................................243
  5.7.2 Refinement using Statecharts ..........................................................................246
    5.7.2.1 Transition Refinement .................................................................................246
    5.7.2.2 State Refinement .........................................................................................249
    5.7.2.3 Hierarchical State Refinement .....................................................................250

6 Implementing Models........................................................................................................256
  6.1 Producing Efficient Code Automatically ..............................................................256
    6.1.1 Machine Conversion .......................................................................................256
    6.1.2 Operation Conversion ....................................................................................257
  6.2 Timing Constraints in Code Generation.................................................................257
  6.3 Resource Constraints in Code Generation .............................................................258
    6.3.1 Smart Card Constraints ..................................................................................258
    6.3.2 Machine Instances .........................................................................................259
    6.3.3 Variables Storage .........................................................................................259
6.3.4 Initialisation Translation .............................................................. 261
6.3.5 Operation Inlining ........................................................................... 261
6.4 Integrating Formal and Non-Formal Code ......................................... 262
6.5 Integrating Testing and Formal Development ....................................... 262
   6.5.1 Test Generation Automation ...................................................... 264
      6.5.1.1 Example of Flaws ................................................................. 264
      6.5.1.2 Testing a Verifier: A Manual Test Strategy ......................... 265
7 Emerging Complementary Developments .............................................. 266
   7.1 Event Based Model Decomposition ............................................... 266
      7.1.1 Introduction ........................................................................... 266
      7.1.2 Informal Definition ................................................................. 266
      7.1.3 Outcome and Constraints of Decomposition ......................... 267
      7.1.4 The Main Difficulty: Variable Splitting .................................... 267
      7.1.5 The Solution: Variable Sharing .............................................. 268
         7.1.5.1 Shared Variables Replication ............................................ 268
         7.1.5.2 A Notion of External Variable ........................................... 268
         7.1.5.3 A Notion of External Event ............................................... 268
         7.1.5.4 Final Recomposition .......................................................... 269
   7.2 Refining Iterative Programs in the B Method .................................... 269
      7.2.1 Introduction ........................................................................... 269
      7.2.2 Problem under Consideration in the Framework of the B Method ........ 270
      7.2.3 Proof Obligations for the Loop Refinement ............................ 272
      7.2.4 Case Study ............................................................................ 273
   7.3 Vulnerability Analysis ................................................................. 276
      7.3.1 Smart Card ............................................................................ 276
      7.3.2 Motivation ............................................................................. 278
      7.3.3 Requirements of the Specification ......................................... 279
      7.3.4 Vulnerability Analysis ............................................................ 279
1 Introduction

The practitioners handbook intends to provide a guide to the formal development of critical systems. This handbook covers all the phase of a typical development lifecycle: from requirements analysis to code generation. Each chapter of the handbook focuses on a single phase of development lifecycle, containing several approaches to that specific phase. For example, Chapter 4 describes different ways for analyse models: expert validation, safety analysis, and proof.

The practitioners handbook was composed of contributions from all the MATISSE partners. The next table shows for each section of the handbook which partners contributed to it:

<table>
<thead>
<tr>
<th>Section</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>ClearSy</td>
</tr>
<tr>
<td>2.1</td>
<td>ClearSy and University of Southampton</td>
</tr>
<tr>
<td>2.2</td>
<td>ClearSy</td>
</tr>
<tr>
<td>2.3</td>
<td>University of Southampton and Aabo Akademi University</td>
</tr>
<tr>
<td>3.1 – 3.2</td>
<td>University of Southampton, Siemens Transportation Systems, and Aabo Akademi University</td>
</tr>
<tr>
<td>3.3</td>
<td>Gemplus</td>
</tr>
<tr>
<td>4.1</td>
<td>ClearSy</td>
</tr>
<tr>
<td>4.2</td>
<td>Siemens Transportation Systems and Gemplus</td>
</tr>
<tr>
<td>4.3</td>
<td>Aabo Akademi University</td>
</tr>
<tr>
<td>4.4</td>
<td>ClearSy and University of Southampton</td>
</tr>
<tr>
<td>5.1</td>
<td>ClearSy, Aabo Akademi University, and University of Southampton</td>
</tr>
<tr>
<td>5.2</td>
<td>University of Southampton</td>
</tr>
<tr>
<td>5.3</td>
<td>ClearSy and Gemplus</td>
</tr>
<tr>
<td>5.4</td>
<td>Aabo Akademi University and University of Southampton</td>
</tr>
<tr>
<td>5.5</td>
<td>Centre National de la Recherche Scientifique</td>
</tr>
<tr>
<td>5.6</td>
<td>Aabo Akademi University</td>
</tr>
<tr>
<td>5.7</td>
<td>University of Southampton and Aabo Akademi University</td>
</tr>
<tr>
<td>6</td>
<td>Gemplus and ClearSy</td>
</tr>
<tr>
<td>7.1</td>
<td>ClearSy</td>
</tr>
<tr>
<td>7.2</td>
<td>Centre National de la Recherche Scientifique</td>
</tr>
<tr>
<td>7.3</td>
<td>QinetiQ and Gemplus</td>
</tr>
</tbody>
</table>
This document proposes work practices and guidelines to software practitioners to be used through the lifetime of a project. The guidelines presented in this handbook were obtained from experience gained in developing the project case studies. The development of the case studies 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.

Within the MATISSE project we consider techniques suitable for overall system design rather than for isolated pieces of software. The overall trajectory of system design generally ranges from user needs, representing the first informal ideas about the system to be designed, to a real system implementation in a specific environment of use. The chapters of this handbook follow the trajectory of a typical development lifecycle, so each chapter corresponds to a phase of the lifecycle.

The remaining of this chapter describes the development life cycle of a B application, and compares it with a classical development life cycle. Chapter 1 concludes by presenting three industrial case studies developed within the MATISSE project: transportation, healthcare, and smart card case studies. These case studies are used through this handbook to illustrate guidelines of use of formal methods in industrial settings.

Chapter 2 presents the modelling notations used in MATISSE. Because B is the central notation in MATISSE this chapter starts by given an introduction to its main concepts, followed by a brief description of the B notation. To illustrate the use of B an example of a railway is presented, which shows how to model time properties in B. Another notation presented in this chapter is event B that extends B to deal with the development of distributed systems. The last notation presented is UML+B that allows the use of UML class diagrams and state charts to define a B specification.

Chapter 3 provides guidelines on how to construct formal models. This chapter is divided into system-level models and computational models. A system-based model includes the overall system including its environment, while a computational model only describes the part of the system under specification. Within the system-level models (Section 3.2.1) the transportation case study is used to illustrate how informal properties may be translated into an abstract B model. Based on the healthcare case study, Section 3.2.2 describes how UML may be used model the safety aspects of a system starting with the use cases, followed by constructing class and statechart diagrams, and ending with the automatic generation of B models. In Section 3.3.1 the smart card case study is used to illustrate the construction of computational models, more specifically it shows how to model a byte code verifier in B.

Chapter 4 describes different methods for analysing formal models. This chapter starts by presenting practical recommendations on how to build a B model. In Section 4.2 we describe the procedures used for reviewing a B model by domain experts, which were based on the transportation and smart card case studies. Section 4.3 shows how a preliminary hazard analysis
may contribute to the construction of the UML+B model, and conversely how the UML+B model informs the hazard analysis. Last, Section 4.4 describes how to use Atelier-B to generate and prove the proof obligations of a B model.

Chapter 5 discusses the refinement of formal models. The introduction describes the B notion of system refinement, illustrated with a simplified version of the transportation case study. Refinement in event B and B-action systems is also presented. Section 5.2 exemplifies how to use Atelier-B to construct gluing invariants for refinement. Section 5.3 provides guidance on defining and proving loop invariants, as these are one of the most difficult tasks of formal development. In Section 5.4 we discuss several approaches to the decomposition of a system, and show how decomposition is done in B. To illustrate this we present the decompositions done on both the transportation and healthcare case studies. Section 5.5 proposes a new refinement operator for the refinement of distributed processes, providing a set of splitting and grouping rules. This approach is applied to the Average Bit Rate (ABR) flow control protocol. Section 5.6 uses the healthcare case study to show how to deal with safety aspects in refinement. Last, Section 5.7 describes refinement of UML+B models, by refining class and statechart diagrams.

Chapter 6 discusses the automatic code generation from B implementations. Section 6.1 describes a more efficient code generator than Atelier-B, which includes several optimisations. Sections 6.2 and 6.3 discuss how to deal with timing and resources in code generation. Section 6.5 discusses experiences from the smart card case study over integrating testing and formal development, leading to the definition of some guidelines for the generation of test cases from formal specifications.

Chapter 7 describes emerging complementary developments. Section 7.1 proposes to extend event B to support model decomposition, with this approach each sub-model could be independently developed. Section 7.2 proposes a methodology that supports the introduction of while loops at any refinement level, instead of only at the last refinement. This section also presents proof obligations patterns to be used in the loop refinement process. Section 7.3.1 discusses the use of formal methods in the certification of smart cards. Section 7.3 shows how CSP may be used to detect critical properties for complex and distributed systems, complementing the B development.

The chapters of this handbook are organised according to the usual application life cycle, where each chapter contains several approaches to a specific phase of the life cycle. Here are some reading suggestions that emphasise different aspects of the handbook:

- **UML development**: Sections 1.2.2, 2.1, 2.3, 3.2.2, 5.1.1, 5.7.
- **Formal development with safety aspects**: Sections 3.2.1, 3.2.2, 3.3.1, 4.3, 5.6, 7.3.
- **Formal development with Atelier-B**: Sections 1.1, 2.1, 3.2.1, 3.3.1, 3.1, 4.4, 5.1.1, 5.2, 5.3.
- **Transportation case study**: Sections 1.2.1, 2.1, 2.2, 3.2.1, 4.2.1, 5.1.1, 5.1.2, 5.4.2.
- **Healthcare case study**: Sections 1.2.2, Chapter 2, Sections 3.2.2, 4.3, 5.1, 5.4.1, 5.6, 5.7.
1.1 B Development Life Cycle

1.1.1 Overview

The aim of our approach is to build a B [Abrial96] application from an initial informal statement which is enriched, structured, and formalised progressively. This demands that, while preserving its own nature and goals, each phase of the construction process is carried out according to the needs and constraints of the following phase. Thus, the results of the each phase can be effectively reused by the next one, which optimizes the process and assures a good traceability across the phases. The role and the principles of the phases of the process are the following.

Figure 1.1: A classical V development life cycle.

1.1.2 Comparison with Standard Development Life Cycle

Standard development life cycle (Figure 1.1) and formal development life cycle (Figure 1.2) are quite similar in their form, except that the former contains less steps: most unit, integration and
validation tests\textsuperscript{6} are suppressed as they are replaced by proof, performed during all the development process. On the other hand, formal specification phase is usually than its classical counterpart, as specification is verified in depth by proof during its writing.

The phases of the formal V lifecycle are:

- **The requirements specification phase**, that states the capabilities of the software to be developed and the operational and implementation constraints on it.

- **The software specification phase**, that defines abstractly which functions the software must perform in order to have the desired capabilities.

- **The design phase**, that defines the program implementing the software functions, and satisfying it’s operational and implementation constraints. The design phase is divided in two sub-phases:
  - **The preliminary design phase**, that defines the program components (sub-programs), their interfaces, and their organization.
  - **The detailed design phase**, that completes the sub-programs definition, and produces the representation of the eventual executable program.

- **The programming phase**, that translates the program representation in the implementation programming language, and produces the actual executable program.

- **The tests phase**, that is divided in three sub-phases:
  - The unit tests phase, that checks each procedure of each sub-program.
  - The integration tests phase, that checks the cooperation of sub-programs.
  - The validation tests phase, that checks the adequacy of the program with respect to requirements.

### 1.1.3 Process Description

#### 1.1.3.1 Requirements Specification

The requirements specification phase poses the problem to be solved by the software to be developed. Although informal, this first statement classifies and orders the requirements according to their nature and their safe level.

\textsuperscript{6} Integration of third party libraries or non formally developed software should there be tested.
1.1.3.2 Software Specification

The software specification phase analyses the problem, and defines an abstract solution that meets requirements. The solution is defined in terms of functions and data, hierarchically organized from the most global and abstract description to the most detailed and concrete. The result of this phase is a semiformal model made of a functional model, a data model, and a collection of formal comments. The models describe the dependencies, causalities, and decompositions of functions and data. The formal comments are mathematical statements modelling functional requirements, safety requirements, and integrity properties of data of the model.

1.1.3.3 Preliminary Design

Preliminary Design transforms the semiformal model produced so far in a partial AMN model. This model introduces all the software components (abstract machines), defines their interfaces (constants, variables, and operations), and sets the implementation architecture (imports relations). The transformation principles are the following:

- Functions are transformed into AMN operations.
- Data is transformed into AMN variables or operation parameters.
- **Levels of functional decomposition** are transformed into layers of abstract machines: *i.e.*, the variables and the operations corresponding to data and functions of a decomposition level are grouped into one or several abstract machines. The criteria to group them are dictated by the constraints of AMN and the data model.

- **Function decomposition** is transformed into operation refinement: each operation associated with a decomposed function is implemented with the operations associated with the decomposition functions. In other words, the abstract machines associated with the non terminal levels of functional decomposition are implemented with the abstract machines associated with the next lower level of functional decomposition.

- **Formal statements** are transformed into invariants or operation definitions either in abstract machines or implementations. The result of preliminary design is then a layered architecture of abstract machines, where each layer represents a level of the functional decomposition, and is implemented, if it is not the bottom layer, with the abstract machines of the next lower layer. We shall see that in this kind of model, the safety and functional requirements are distributed at the different layers of abstract machines. Note that this model contains neither the refinements of abstract machines at intermediate layers, nor the refinements and the implementations of the abstract machines at the bottom - given implementations must be added to the partial AMN model.

### 1.1.3.4 Detailed Design

Detailed design finalizes the partial AMN model produced at the previous phase. It creates the refinements of the abstract machines of the intermediate layers and the refinements and the implementations of the abstract machines of the bottom layer which are developed in B. Therefore, the result of the detailed design is the complete AMN model of the software.

Some abstract machines are neither refined, nor implemented. They are supposed to formalize the interface of a component that is not formally developed, either because it is an already developed component or library, or because it is an interface with the underlying operating system, or because it is a low level component whose formal development would be too expensive compared with the expected benefits.

### 1.1.3.5 Programming

In the present approach the executable code of the software is obtained by translating, either automatically or manually, the implementations of the AMN model and, if necessary, by developing conventionally the code of the abstract machines that have not been formally refined and implemented.
1.1.3.6 Tests
Tests phase is divided into unit tests, integration tests, and validation tests. Unitary tests concern exclusively operations of abstract machines not formally refined and implemented and operations that could not be proved completely.

Integration tests concern the integration of formally and not formally developed modules. Functional tests are carried out as in conventional developments. Unitary, integration, and non intensive validation tests are performed by the development team. Then, the software is delivered to a validation team, independent from the development team, which performs intensive validation tests on the target computer.

1.1.3.7 Proof Validation and Program Proof
Proof validation is carried out at preliminary and detailed design phases by the validation team. It ensures that the specific proof rules added to discharge proof obligations generated by AMN modules are correct mathematical statements.

1.2 Case Studies
The driver for the research and development in MATISSE are three major industrial case studies representing a spectrum of essential services for the information society. The industrial case studies are:

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

These three case studies are used to describe the guidelines and work practices through this handbook.

1.2.1 Transportation

1.2.1.1 Formal Methods in the French Railway Industry
The use of formal methods in the French railway industry was introduced by the SACEM system, an automatic train control system for RATP, the transit authority in Paris, in the beginning of 1980. The consortium of manufacturers in charge of this project decided to develop non-diverse software using a new technology to secure it, the Vital Coded Processor (VCP) technique, instead of diversified software using the concept of redundancy. The VCP technique consists in protecting each software information by a redundant code, checked at run-time. Because the
software was not duplicated, a *zero default* design was required. Moreover, since validation by testing was considered insufficient by RATP, they asked manufacturers to use a new approach based on formal methods. The process was as follows:

1. The functional software requirements were re-written in a formal language.
2. The software source code, written in the language Modula 2, was completed with *pre-assertions* and *post-assertions*, and checked with a partially automated program proof activity.
3. A check between formal specification and code assertions was manually performed.

Despite the heaviness of this process and its weak automation, the confidence achieved by this first application had convinced RATP of the advantages of this approach. For the subsequent tender, the Meteor line, they required the use of formal method for safety critical software development. MATRA Transport International was chosen in 1992 to develop the automatic train operation system for the Meteor line, using the B method for the safety critical software.

### 1.2.1.2 Transportation Case Study

The railway application case for MATISSE concerns a generic product originally designed for the protection of trains. It is an Automatic Train Protection (ATP) transmission-based system using a movement authority principle.

The ATP gives the movement authority to trains to enter and travel through a specific section of track in a given direction. The movement authority used by the limitation of movement is located in fixed places. The ATP supervises and enforces the authority for movement to maintain a safe train separation and provides protection based on interlocking information to the train movement.

In addition to ATP, the Automatic Train Operation function (ATO) starts and stops the train, controls the train doors, and regulates the speed of the train as it travels over the railway network in accordance with the Automatic Train Supervision system inputs. The ATO maintains the train speed below the speed limits imposed by the ATP.

The intelligent part of ATP and the safety decision-making process is placed on-board the trains. As messages on a section are broadcast to all trains located in an area, the on-board equipment calculates the position of the train on the track and filters out the relevant data. The on-board equipment, programmed with the parameters of the train, checks the train movement against the safety constraints and, in automatic mode, achieves the ATO functions.
Description of the Automatic Train Control Functions
The main functions provided by the SACEM\textsuperscript{7} Automatic Train Control (ATC) are:

- Checking train movement and passenger safety (ATP) consisting of:
  - Compliance with protection points (this ensures spacing, routing, end of track and floodgate protection),
  - Compliance with speed limits,
  - Rollback monitoring,
  - Run authorization by zone for passenger safety (Platform Emergency Stop and Platform Screen Doors status supervision), staff protection (KeySwitch) and train movement protection (Flank and Fouling protection).
  - Movement or speed conflicting with the above results in irreversible emergency braking until the train is completely stopped.
  - Authorisation to open the train and platform screen doors at the platform.

- Train operation:
  - In Manual mode:
    - Speed instructions delivered to the Train Operator complying with protection points, speed restrictions, and other conditions.
    - Control of the opening of Platform Screen Doors.
  - In Automatic mode (Automatic Train Operation):
    - Optimum train movement between stations, complying with protection points and arrival/departure timetables.
    - Stopping of trains in stations (the actual departure is controlled by the Train Operator).
    - Stopping of trains in change of end zones (according to the Automatic Train Supervision orders).
    - Control of the opening of Platform Screen Doors.
    - Control of the opening of Train Doors (only in Driverless Turnaround mode).

\textsuperscript{7} The SACEM system is a automatic train control developed by Siemens Transportation Systems in the beginning of 1980.
Train start in automatic mode between stations (with no action by the Train Operator).

- Interface with Operations and Maintenance personnel:
  - SACEM ATC delivers reports used by Operation or Maintenance personnel.

### 1.2.2 Healthcare

PerkinElmer Life Sciences (later in the document referred to as Wallac) supplies complete analytical systems. The systems are used to provide researchers and clinicians with reliable determinations of substances found in blood or other biological sample materials that are difficult to measure. Typically, the substance to be measured is tagged with a radionuclide or a fluorescent or luminescent label. The amount of the substance is then determined from the *signal*, the amount of radiation or light given off by the label. The manufactured analytical systems include reagents, sample handling and measuring instrumentation, as well as computer software.

The team of the Computer Science Department at Aabo Akademii University (later in the document referred to as Aabo Akadem) conducted a parallel development of a new product of Wallac’s – Fillwell™, a workstation preparing samples [PerkinElmer01]. The workstation is shown in Figure 1.3. The system belongs to the class of products for drug discovery and bioresearch. The Fillwell microplate liquid handling workstation offers significantly advanced features in the line of the sample preparation systems. The Fillwell base unit consists of a dispense head dispensing liquid into microplates on a processing table. A gantry moves the dispense head with high precision and speed from one plate to another.

![Figure 1.3: The Fillwell microplate liquid handling workstation.](image)

The Fillwell workstation is the first liquid handling system specifically designed for high density microplates. The system is modular and can therefore be customised into a variety of configurations. The dispense head can have up to 384 tips attached. With the tips the head can...
perform automated pipetting into plates with 96, 384 or 1536 wells. The head provides a precise dispensing with volumes from 0.5 to 300 µl. The processing table contains up to 6 plate positions. An extension with three plate positions is easily removable. In Figure 1.3 this processing table extension has been removed. In order for the dispensing head to be able to reach all the positions on the processing table it is mounted on a gantry that can move in XYZ-directions. The precision of the gantry is very high with an accuracy of 100 µm. The system can function as a standalone workstation or be integrated into a robot.

The main application of the Fillwell workstation is drug discovery. Within this application area the system can be used for microplate replication, for dilution, transfer and addition of the liquid in the plates, for reformatting of plates with different densities (number of wells), as well as for rapid plate filling to homogenous and cell based systems. The FillWell system is a safety-critical system. Safety is the property of a system, which ensures that it will not harm humans, environment or equipment. The direct harm to the humans using the drug discovery systems is quite moderate according to the classification for normal safety-critical systems. However, the indirect harm caused by the results of incorrectly performed experiments might be catastrophic. Furthermore, the system can be used to handle extremely expensive substances (valued up to a billion EURO per kilogram) as well as serve as a part of an expensive production chain. Hence, failures of such a system might result in significant economical losses and the system can also be considered as money critical.

1.2.2.1 The Setting of the Case Study

This case study was organised using an industry-as-laboratory approach. This means that the researchers in academia provided the formal methods and B expertise while the experts at the R&D department of the industrial partner brought the domain knowledge to the team.

In order to come up with a good methodology for the development we started with a smaller part of the case study concentrating on modelling the dispense head and its movement up and down as well as its dispensing of liquid. However, the branch of development of medical and pharmaceutical equipment is highly competitive. Because of that, the requirement to preserve confidentiality of the equipment under construction was very strict. As the product was confidential for a long period of the project, we developed a mirror case study, called the Robot analyser, in order to have a public case study as an example of the methodology. We have chosen a mirror case study that models very well the significant and critical aspects of the Fillwell case study concerning the precision of movement and liquid handling. Both the case studies belongs to the same class of systems. The development of the Fillwell case study will be documented elsewhere [Bostroem03].

In the Fillwell case study we have a workstation dispensing to and aspirating from plates. Our mirror case study is a robot analyser that analyses samples on a plate that is placed on a movable operating table. The movement of the operating table corresponds to the movement of the dispense head, while the analysing corresponds to the dispensing. A first extension of the systems...
would be to take into consideration the XY-movement of the dispense head and, hence, the corresponding rotation of the robot to be able to work with more than one plate.

1.2.2.2 The Requirements of the Case Study

The Robot analyser consists of two interacting parts, a Robot and an Analyser. We concentrate here on the Analyser. The task of the Robot is to place a plate to be analysed into the Analyser, as well as to remove an analysed plate from it.

The Analyser has an operating table – a horizontal surface on which a plate is placed by the Robot. The Analyser can move its operating table to pick up a plate from the Robot, move the table to its analysing component, analyse the plate, as well as move the table to a position from which the Robot can remove the plate. During the analysis it is checked that there is enough liquid on the plate before the analysis and that the right amount of liquid was used during the analysis. The quality of the analysis should be kept at a constantly high level.

The Robot has two arms, one for placing a plate in the Analyser and the other for removing this plate. These arms are at different heights. Therefore, the table has to be at different positions for placing and removing plates:

- **Upper end position:** A plate is analysed by the analysing component.
- **Middle position:** Arm 1 of the Robot places a plate into the Analyser.
- **Lower end position:** Arm 2 of the Robot removes a plate from the Analyser.

Initially the Analyser is at its middle position and it is empty.

Due to the fact that the operating table and the Robot arms may collide we define safety requirements that the Robot analyser has to fulfil. These safety requirements are expressed in English. Later during the development they will be translated to predicates in B:

- When the Robot is ready to load the Analyser, arm 1 may extend only if the table is in its middle position.

- When the Robot is ready to unload the Analyser, arm 2 may extend only if the operating table is in its lowest position.

- The table may move only when the respective Robot arm has retracted.

- The table must not move beyond its upper and lower end position.

During a typical course of events the following sequence of actions happens with a plate. Here, we have not considered the possible failures.

1) The Analyser moves the operating table to the middle position ($move(zmid)$).

2) The Analyser waits to receive a plate from the first Robot arm after the Robot has picked up a plate and rotated to the Analyser ($receive$).
3) The Analyser moves the operating table to the upper end position \((move(z_{max}))\).

4) The Analyser analyses the plate \((analyse)\).

5) The Analyser moves the operating table to the lower end position \((move(z_{min}))\).

6) The Analyser waits to deliver an analysed plate to the second Robot arm and the Robot rotates to pass the plate on to another device \((deliver)\).

We can note that other courses of events are possible depending on the order in which the commands \(move\), \(receive\), \(analyse\) and \(deliver\) are given. In order to prohibit the Analyser from analysing a plate and moving an infinite number of times, only a certain number of moves are allowed to be performed in a row before some other action needs to be taken by the Analyser. Similarly, only a certain number of analyses on one plate are allowed before the plate has to be delivered.

1.2.3 Smart Card

The smart card case study was developed by Gemplus using the B method and the Atelier-B toolkit. This case study investigates how B can be used efficiently in the specific domain of the smart card. Gemplus has concentrated its effort on applying B at the software level, with the aim of producing more reliable and certifiable products.

Security in Java

Security is a critical aspect of downloadable code that originates from a remote source. It is not possible to know \(a\ priori\) the behaviour of the program. By executing the mobile code, we grant it the right to perform operations on our machine and we provide it access to our local resources: files, network access, local information and devices. Each piece of mobile code that is allowed to execute must be tightly controlled in order to ensure that it will not destroy our data, divulge confidential information, waste our resources, or use our machine as an intermediary to launch an attack on someone else. For the smart card it is important that an applet cannot have access to the data of other applets except by using the sharing mechanism, or accessing the code of the operating system.

The security in Java is partially based on language features like strong typing. By disallowing programmers to manipulate or forge pointers, Java controls the access a program can perform. But once the program is compiled in byte code, those properties can be violated by manipulating the byte code itself. The verifier is a key component of the Java security architecture. It examines incoming code in order to ensure that it is valid. It checks that the code respects the syntax of the byte code language and that it respects the language rules. Other components are responsible for protecting system resources from abuse but they depend on the verifier as they rely on language features such as access restrictions \((private, protected, final, etc)\).
As opposed to ordinary native machine languages, the Java byte code language has been defined so that Java byte code programs can be statically verified and validated. The verifier is able to examine a compiled class and decide whether or not:

- it is syntactically correct and well-formed,
- it respects the language rules, and
- references from the current class to other classes are consistent.

The verifier performs this validation statically, and thus does not hinder the performance of the execution. Equivalent validation cannot be performed on programs compiled into native machine code, nor can it be performed on typical assembler source. It cannot even be performed on source code for many higher-level languages such as C or C++. Java byte code is probably the most feature-full language for which such verification is possible.

1.2.3.1 The Verifier

The verifier examines loaded classes to ensure that they are syntactically correct and well-formed, that they are consistent with other classes and that the byte code they contain respects the language rules. The verifier's purpose is to prevent the Java Virtual Machine (JVM) process from being subverted or corrupted by inconsistent or malformed CAP files, and to enforce the language's security mechanisms. The verifier checks the integrity of the constant pool and the syntax of all method and field declarations. It checks the consistency of the current class with its super classes (or super interfaces) and its implemented interfaces. For instance, it is checked that:

- references to the constant pool entries target entries of appropriate type for the referring entity,
- modifier flags (also called access flags) are consistent for the class and all its methods and fields,
- final classes are not subclasses and final methods are not overridden,
- classes really implement the interfaces they claim to implement,
- non-abstract classes do not have abstract methods.

The verifier also inspects the byte code for each method, checking that all opcodes and instruction operands are valid. For instance, it is checked that:

- branches target valid instructions,

---

8 The CAP file is produced by the converter and is the standard file format for the Java Card platform. This file contains the executable binary representation of the classes of a Java package.
arrays of more than one dimension are not created.

The verifier keeps track of the types of all values on the stack and in the registers. It ensures that all instructions take appropriately typed arguments from the stack and registers. The property known as type-safety implies that:

- values of primitive types are never taken to be references, nor the reverse,
- instances of a given class are used only where that class or a super class of that class is expected,
- an instance of a class can be used where a given interface is expected only if the class does implement the interface,
- objects are not used until they are initialised and that they are properly initialised.

Finally, whenever a reference to another class, method or field is about to be used by a running method, it is checked that the target exists, that it is accessible to the referring class and that it is used properly.

1.2.3.2 Java Card Verifier

Java and Java Card restriction

In the Java architecture, the class file verifier is used prior to execution. The code is loaded by the Java Virtual Machine, and then checked by the verifier before its execution by the interpreter. The Java verifier performs static checks and ensures some properties:

- there are no violations of memory management and no stack underflows or overflows,
- access restrictions are enforced,
- methods are called with appropriate arguments of the appropriate type,
- fields are modified with values of the appropriate type,
- objects are accessed with the appropriate type,
- no pointers are forged,
- no illegal data conversions are performed, and
- binary compatibility rules are enforced.

However, the Java Card language differs from the Java language. First the execution architecture is different: the classes of an applet correspond to one or several packages. A converter is used in order to convert the classes of a package into a CAP file. In addition to the CAP file creation, an export file representing the public APIs of the package being converted is generated. Therefore, if one wants to verify a CAP file of a particular package, the CAP file, export file of the package
and export files of the imported packages are needed as depicted in Figure 1.4. During verification, the verifier must ensure that the CAP file is internally consistent, consistent with the export files it imports and consistent with the export file that represents the API.

![Diagram showing CAP file verification process]

**Figure 1.4: The CAP file verification.**

Furthermore, the Java Card language is a subset of the Java language. In particular, several features of Java are not present: dynamic class loading, security manager, garbage collection and finalization, threads, cloning and access control in Java packages are not supported. The basic types `char`, `double`, `float` and `long` are not supported. Arrays of more than one dimension are also not supported by Java Card. These restrictions are imposed to check that:

- only supported data types are used and data of type `int` are used only if the Java Card interpreter supports them;
- no unsupported Java language features are used;
- usage of certain Java operations are within limited ranges; and
- no potential overflow or underflow can occur that might cause arithmetic results to be computed differently than they would be on the Java platform.

Most of these restrictions are checked by the converter and do not need to be explicitly performed by the verifier. In fact, if any data is of type `float`, the verifier would reject the program not because the type `float` is not supported but because the type is unknown. So the verification appears to be implicit. However, during verification, the CAP file verifier must ensure that a CAP file has the correct format. It also ensures that the byte codes within the CAP file fit to the set of structural constraints. Some of these constraints are the same as for Java and some others are particular to the CAP file structure such as the following:

- The package and each applet defined in the package must have a valid AID (Applet IDentifier) that is between 5 and 16 bytes long. The package AID and the applet AIDs must share the same RID (Resource IDentifier) number.
- An applet must define an `install` method with the correct signature so that instances of the applet can be appropriately created on the card.
• The order of class and interface definitions in a CAP file must follow the rules that interfaces appear before classes, and superclasses appear ahead of subclasses.

• The int flag is set if the int type is used in the CAP file.
2 Modelling Notations

In the MATISSE project, B is the core method; most of the other notations used either evolved from B or integrate B with an existing formal or informal notation. The exception is the CSP notation that will be used to complement B. This section briefly describes the notations that will be used through the handbook.

2.1 B Method

B is a formal specification method that allows, the properties required in a schedule of conditions to be rigorously expressed.

We can prove in an automated way that these properties are unambiguous, coherent and non-contradictory. Then, we guarantee by mathematical proof that these properties are satisfied in every stage of design.

This method and the associated proof that allow:

- to obtain technical specifications and system schedules of conditions that are clear, structured, coherent and without ambiguity,
- to develop software guaranteed by contract to be without defect.

The B method has been applied in large industrial projects in fields such as real time, process controls, communications protocols, cryptographic protocols, and embedded systems.

2.1.1 Synopsis

Created by Jean-Raymond Abrial, from the research of E.W. Dijkstra and C.A.R Hoare, the B method is described in [Abrial96]. It is based on the mathematical concept of set theory.

Generally, the initial expression of requirements is captured using natural language, or by combination of methods such as SADT or SA-RT or descriptions: such as graphs, automata, logic tables or Petri networks.

A B development begins with the construction of a model from the requirements, that describe

- the main state variables of the system,
- the properties (or invariant) which these variables will have to meet constantly, and
- the transformations of these variables by services (or operations).

The B model obtained constitutes a specification of what the system will have to realise (the what). The B model is then refined, i.e., specialised, until a complete implementation of the software system is obtained (the how). Several refinements can fulfil a specification, the choice of solutions depending on various criteria such as the simplicity of proofs, and the architecture of the system. The refinement can also be used as a specification technique. In this case, the
refinement enables progressive inclusion, of the problem's details in the formal development. The formal specification is then realised gradually rather than directly.

Using B in the development of a system is therefore about:

- Clearing all ambiguity right from the initial interpretation of the requirement,
- Constructing a specification that is both coherent and conforms to the requirement (the model),
- Elaborating the software system that realises the specification, in successive stages.

The coherence of the model and then the conformity of the final program to this model are guaranteed by mathematical proofs. The demonstration of these proofs can be carried out by using automatic proof tools, such as those provided by Atelier-B [AtelierB98].

### 2.1.1.1 Abstract Machines

The abstract machine is the basic mechanism of the B method. It is a concept very close to well known programming notions, such as modules, classes or abstract data types.

A machine contains variables and operations. It encapsulates variables. The operations enable access to the variables and their manipulation.

![Abstract Machine Diagram](image)

**Figure 2.1: Abstract machine.**

The elements of an abstract machine are described through expressions using mathematical concepts such as sets, relations, functions or sequences. The static laws, which the variables must obey, are defined by predicates and constitute the invariant of the abstract machine.

The specification of the behaviour of the operations uses a pseudo-code made of generalised substitutions. The substitutions are defined formally in the way they transform predicates. A given operation of an abstract machine can contain a pre-condition: it is a predicate expressing the conditions necessary to invoke the operation. An operation also contains an action: it is a substitution describing how the variables of a machine are manipulated. As an abstract machine is defined in the specification phase, some concrete substitutions such as sequencing or loops are
forbidden. It is supposed to describe what the operation must do, not how. The parallel substitution, does not prescribe the application order of the individual substitutions. The nondeterminism present in the substitutions leave open choices for further development.

To ensure that an operation call preserves the static properties of the abstract machine (the invariant), a Proof Obligation has to be shown to hold. This Proof Obligation is automatically constructed from the formal definition of the substitutions in the abstract machine.

2.1.1.2 Refinements and Implementations

The mechanism of refinement consists of successively reformulating the variables and operations of the abstract machine, so that a computer program finally results. The intermediary stages of reformulation are called refinements, and the final level of refinement, the implementation.

![Refinement mechanism diagram]

The refinements must redefine the operations of the abstract machine. Each B component (abstract machine, refinement or implementation) is defined using the B Language.

During each refinement, the behaviour of an operation has to be proven to be compatible with the operation it refines. This guarantees that the code of an implementation will conform to its specification in the abstract machine.

The production of a refinement:

- progressively transforms the abstract data types (sets, relations, functions, sequences) into concrete data types (scalar variables, arrays);
- gradually clears the level of non determinism of the substitutions; and
• replaces abstract substitutions (parallelism, choice) with concrete substitutions (sequencing, conditionals, loops).

2.1.1.3 Project

A B project enables the development of a piece of software realising a particular system. To realise a complete project, we use the mechanisms of composition and decomposition of abstract machines. As soon as the level of complexity of the refinement of an abstract machine is too high, we decompose it into several, more simple parts. The implementation (the final level of refinement of an abstract machine) can then be implemented on the specifications of one or several abstracts machines, which are themselves refinable. This is achieved with call to operations of the machines involved. The user of an abstract machine is therefore always an implementation. In this way, a project is constructed gradually, according to an architecture made from layers of abstraction.

Figure 2.3: A B project.

Abstract machines of a lower level in the layer architecture of a project can pre-exist the rest of the development. Indeed, there is a library of abstract machines available. These machines encapsulate the more classic data structures, and are guaranteed, checked and proved.
2.1.2 The B Language

The B language consists of the following elements:

- Predicates
- Expressions
- Substitutions
- Components (abstract machines, refinements, implementations)

2.1.2.1 Predicates

The predicates used in the B language are a subset of the logic of first order predicates. They are formulae allowing the expression of properties relating to data.

Predicates are used directly in the B language to express properties relating to all the data of a component. For example, they express the invariant properties of variables and the pre-conditions under which an operation can be called (these conditions relate amongst others to input parameters of the operation).

They are used to express the Proof Obligations. The goal of a Proof Obligation is a predicate, which can either be proved or refuted. Hypotheses under which a proof is made are also predicates.

The predicates of the B language are:

- simple propositions (conjunctions, negation, disjunction, implication, equivalence);
- quantified predicates (universal, existential); and
- comparison predicates between expressions.

2.1.2.2 Expressions

An expression is a formula describing data. Every datum has a type and a value. The categories of expression are:

- basic expressions;
- boolean expressions;
- arithmetic expressions;
- maplet expressions;
- set expressions (empty set, set of integers, boolean,...);
- set constructions (set of sub-sets, union, intersection,...);
• relation expressions (identity, inverse, projection, composition, iteration, domain, ...);
• function expressions (injections, surjections, bijections, ...); and
• function constructions (constant functions, lambda expressions, ...).

2.1.2.3 Substitutions

Generalised substitutions are mathematical notations that describe the transformation of predicates. They are used to describe the dynamic aspect of B components: their operations.

They describe the behaviour of operations at the abstract machine level as well as the refinement and implementation levels. The specification substitutions can be non-deterministic and non-executable, whereas the implementation substitutions correspond to instructions of a classic computing language.

They also enable the Proof of Obligations to be built automatically from the B components. For example, the Proof Obligation corresponding to the preservation of the invariant during the call of an operation is constructed by taking the invariant as hypothesis and the proof of substitution of the operation applied to the invariant as the goal.

The substitutions of the B language are:

• substitution becomes equal, substitution becomes such as, substitution becomes element of:
  \[ x := E; \]

• pre-condition substitution, to express pre-conditions of operation calls:
  \[ \text{PRE } G \text{ THEN } S \text{ END;} \]

• bounded choice substitution, \text{SELECT} substitution:
  \[ \text{SELECT } G \text{ THEN } S \text{ END;} \]

• substitutions \text{ANY} and \text{LET} which introduce data verifying certain properties:
  \[ \text{ANY } v \text{ WHERE } P \text{ THEN } S \text{ END,} \]
  \[ \text{LET } v \text{ BE } P \text{ IN } S \text{ END;} \]

• \text{VAR} substitution which introduces local variables:
  \[ \text{VAR } v \text{ IN } S \text{ END;} \]

• conditional \text{IF} and \text{CASE} substitutions:
  \[ \text{IF } P \text{ THEN } S \text{ ELSE } Q \text{ END,} \]
  \[ \text{CASE } E \text{ OF} \]
  \[ \text{EITHER } 1 \text{ THEN } S \]
OR $m$ THEN $T$

... OR $n$ THEN $U$ END

END

- simultaneous substitution:
  $$x, ..., y := E, ..., F,$$
  $$x := E || ... || y := F;$$

- loop substitution; and

### 2.1.2.4 Components

A B component can be an abstract machine, a refinement or an implementation. Components possess clauses incorporating the static and dynamic description of behaviour.

The main clauses are given in Table 1:

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>declaration of the abstract machine's name and the list of eventual parameters of the machine</td>
</tr>
<tr>
<td>REFINEMENT</td>
<td>declaration of the name of a refinement</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td>declaration of the name of an implementation</td>
</tr>
<tr>
<td>REFINES</td>
<td>declaration of the refined component</td>
</tr>
<tr>
<td>IMPORTS</td>
<td>when an implementation imports an abstract machine, it can use freely the latter's operations (but not its data)</td>
</tr>
<tr>
<td>SEES</td>
<td>when a component sees an abstract machine, it refers to this machine and can consult its data and use the operations which do not modify these data</td>
</tr>
<tr>
<td>INCLUDES</td>
<td>when an abstract machine includes another abstract machine it integrates the data of the included machine</td>
</tr>
<tr>
<td>DEFINITIONS</td>
<td>declaration of textual definitions which will be expanded in the component before any further analysis</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>declaration of the properties of the abstract machine's parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>declaration of deferred and enumerated sets</td>
</tr>
<tr>
<td>CONCRETE_CONSTANTS</td>
<td>declaration of constants, concrete and implementable, which will be kept during successive refinements</td>
</tr>
<tr>
<td>ABSTRACT_CONSTANTS</td>
<td>declaration of abstract constants, which are non-implementable</td>
</tr>
</tbody>
</table>
and must therefore be refined

**PROPERTIES**
declaration of the constants’ properties

**CONCRETE_VARIABLES**
declaration of concrete variables, which are implementable and will be kept during the successive refinements

**ABSTRACT_VARIABLES**
declaration of the abstract variables, which are non-implementable and must therefore be refined

**INVARIANT**
declaration of invariant properties of the variables

**INITIALISATION**
initialisation of variables

**OPERATIONS**
declaration of the operations in the form of a header and body

Table 1: Clauses of a B component.

### 2.1.3 Example: Modelling Time Properties in B

This section illustrates how to model time constraints in a specification [Butler02b], by using a simplified version of the case study presented in Section 1.2.1. The approach taken to modelling timing constraints is to include a clock variable representing the current time and an operation which advances this variable. The timing constraints are ensured to be satisfied by preventing the clock variable from progressing to a point at which the required properties would be violated. It would seem that one could always satisfy the properties by preventing time from progressing. However, one assumes that in the real system time cannot be prevented from progressing and thus it is an obligation on the final implementation to ensure that the timing properties are always satisfied in time. This is a fairly standard approach to dealing with time in formalisms. For example, this is similar to the approach taken by Abadi and Lamport [Abadi94]. Unlike Abadi and Lamport, this approach uses a discrete model of time rather than a continuous one. A discrete model is sufficient for our purposes since we are interested in ensuring that certain properties hold within fixed time bounds and since the control of the real system is based on fixed time cycles. When we say that something happens at a certain time $t$, what we mean is that it happened within time period $t$.

Consider the timing property on the emergency brakes again: *within $X$ milliseconds of a section becoming restrictive, the emergency brakes of a train on that section are applied*. To represent this constraint in B, we include a variable which records the most recent time at which a section went restrictive, $rtime$. Then the requirement may be formalised in the following form ($t$ is the current time):

$$\text{Prop}(t) = t \geq rtime + X \Rightarrow \text{braking} = \text{true}$$

The operation which progresses time is then guarded by the condition that the property holds in the new time period:
NextTime = \textbf{SELECT} Prop(t+1) \textbf{THEN} t := t+1 \textbf{END}

Ultimately the real system will have to ensure that $\text{Prop}(t+1)$ does indeed hold by the time $t+1$ is reached.

\begin{verbatim}
MACHINE RailwayT
SETS SECTION, TRAIN, BLOCK
CONSTANTS block, next, XX
PROPERTIES
  block $\in$ SECTION 3 BLOCK \wedge
  next $\in$ BLOCK 1 BLOCK \wedge
  XX $\in$ NAT \wedge XX > 0
VARIABLES occ, res, pos, cur, rtime, atime, braking
INVARIANT
  occ $\subseteq$ SECTION \wedge /* set of occupied sections */
  res $\subseteq$ SECTION \wedge /* set of restrictive sections */
  pos $\in$ TRAIN $\rightarrow$ SECTION \wedge /* each train has a
  single position */
  cur $\in$ NAT \wedge /* The current time interval */
  rtime $\in$ SECTION $\rightarrow$ (0..cur) \wedge /* time at which a section
  goes restrictive */
  atime $\in$ TRAIN $\rightarrow$ (0..cur) \wedge /* time at which a
  train enters current position */
  braking $\subseteq$ TRAIN \wedge /* set of braking trains */
  brakingProperty(cur)
\end{verbatim}

Figure 2.4: RailwayT machine.

Figure 2.4 presents part of the system specification. The property $\text{BrakingProperty}(cur)$ in the invariant, where $\text{BrakingProperty}(i)$ is defined as follows:

$$\text{BrakingProperty}(i) = \forall tt. ( tt \in \text{TRAIN} \wedge pos(tt) \in res \wedge$$

$$\text{atime}(tt) < \text{rtime}(pos(tt)) \wedge \text{rtime}(pos(tt)) + XX \leq i \Rightarrow tt \in \text{braking} ) \wedge$$

$$\forall tt. ( tt \in \text{TRAIN} \wedge pos(tt) \in res \wedge$$

$$\text{atime}(tt) \geq \text{rtime}(pos(tt)) \wedge \text{atime}(tt) + XX \leq i \Rightarrow tt \in \text{braking} )$$

The braking property has two cases. If the train has arrived in the section before the section went restrictive, then the time bound is relative to the time at which the section went restrictive. If the
train has arrived in the section after the section went restrictive, then the time bound is relative to
the time at which the train arrived.

<table>
<thead>
<tr>
<th>Figure 2.5: Railway\textit{T} operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Arrive}(t,s) = \hspace{1cm} &amp; \hspace{1cm} \textbf{PRE} t \in \text{TRAIN} \land s \in \text{SECTION}\hspace{1cm} &amp; \hspace{1cm} \text{THEN} \hspace{1cm} pos(t) := s \hspace{1cm} &amp; \hspace{1cm} \text{</td>
</tr>
<tr>
<td>\textbf{Restrict}(s) = \hspace{1cm} &amp; \hspace{1cm} \textbf{PRE} s \in \text{SECTION}\hspace{1cm} &amp; \hspace{1cm} \text{THEN} \hspace{1cm} \text{res} := \text{res} \cup {s} \hspace{1cm} \text{END};</td>
</tr>
<tr>
<td>\textbf{NextTime} = \hspace{1cm} &amp; \hspace{1cm} \textbf{SELECT} \hspace{1cm} \text{BrakingProperty}((\text{cur}+1)) \hspace{1cm} &amp; \hspace{1cm} \text{THEN} \hspace{1cm} \text{cur} := \text{cur}+1 \hspace{1cm} \text{END};</td>
</tr>
<tr>
<td>\textbf{Brake}(t) = \hspace{1cm} &amp; \hspace{1cm} \textbf{PRE} t \in \text{TRAIN}\hspace{1cm} &amp; \hspace{1cm} \text{THEN} \hspace{1cm} \text{braking} := \text{braking} \cup {t} \hspace{1cm} \text{END}</td>
</tr>
</tbody>
</table>

Train arrival and sections going restrictive are modelled by the operations presented in Figure
2.5. Note that have simplified things considerably here. In our larger model, sections are
restrictive when certain other sections are occupied so that the \textit{Arrival} operation causes relevant
other sections to go restrictive. Also in the larger model, a train cannot arrive at an arbitrary
section.

The operation which increases time is guarded by the braking property holding on the next time
interval. In case the braking property would not hold in the next time interval, it can always be
made to hold by some invocations of the operation which models application of the brakes.

\section*{2.2 Event B}

The event B [Abrial98] approach is based on B notations of the B method and extends the
methodological scope of initial concepts such as set-theoretical notations and generalised
substitutions to take into account abstract systems. An abstract system is characterised by a
(finite) list of variables possibly modified by a (finite) list of events; an invariant establishes
properties satisfied by variables and maintained by the activation of events (with true guards)
reacting to the environment. Abstract systems are close to action systems. A system model may
be decomposed into sub-systems, representing both hardware and software components. At the
lowest level, the model provides event based software level specification that can be used for a
traditional software development in B or using any other language.

In event B, a model of a system has two parts:

- a \textit{static} part – which describes the state of the system; and
- a \textit{dynamic} part – which describes the events that may occur in the system.

Properties of the state are related to \textit{safety}, whereas properties of events relate to \textit{liveness}.

Besides its state, a model contains a number of events that show the way it may evolve. These
events are only supposed to be observable; they are in no way actions that can be “called”.
Indeed, we are not describing the realisation here, all be it abstract, of a system. Rather, we are
making a mathematical simulation, which allows us to reason about the future system we are going to construct. This reasoning is precisely what will allow us to analyse very early on the behaviour of our future system and to draw up a possible architecture.

Each event is composed of a guard and an action. The guard is the necessary condition under which the event may occur. In other words, once its guard holds, the occurrence of the event may be observed at any time (but it may also never be observed). As soon as the guard ceases to hold however, the event cannot be observed. The action, as its name indicates, determines the way in which the state variables will evolve when the event does occur.

It is possible for the guards of several events to hold simultaneously. From this point of view, the model presents a certain external non-determinism. Note that when several guards hold simultaneously, no two events cannot be observed to occur “together”: events are atomic.

Finally, we must observe that the events are, \textit{a priori}, asynchronous. Possible synchronisms are only the consequence of the actions of some on the guards of others.

\textbf{Structure of an Event}

An event is expressed in the following form:

\[ e = \{ \forall x, y, \ldots \text{WHERE } P(x, y, \ldots, v, w, \ldots) \text{ THEN } S(x, y, \ldots, v, w, \ldots) \} \]

For such an event, its guard corresponds to the existential predicate:

\[ \exists (x, y, \ldots) . P(x, y, \ldots, v, w, \ldots) \]

In other words, the necessary (but insufficient) condition for the event \( e \) to take place with the current value of the state variables or constants \( v, w, \ldots \) of the model, is that it be possible to assign to the local variables \( x, y, \ldots \) of the event \( e \), some values making the predicate \( P(x, y, \ldots, v, w, \ldots) \) true. As can be seen \( e \) presents a certain latitude in the choice of possible values for the local variables \( x, y, \ldots \). We can speak here about internal nondeterminism.

The action \( S(x, y, \ldots, v, w, \ldots) \) is presented in the form of the simultaneous assignment of certain state variables \( x, y, \ldots \) to certain expressions \( E, F, \ldots \) depending upon the state of the system and the local variables \( x, y, \ldots \) of the event. Note that any variables not mentioned in the list \( x, y, \ldots \) do not change:

\[ x, y, \ldots = E, F, \ldots \]

Sometimes, the event can have the simpler form:
In this case, there are no variables local to the event, and the guard just corresponds to the condition \( P(v, w, \ldots) \) holding on the state variables of the model.

**Consistency: Preservation of the Invariant**

Once a model has been built, it has to be proven to be consistent. Each event of the model has to be proven to preserve the invariant. More precisely, it must be proved that the action associated with each event modifies the state variables in such a way that the corresponding new invariant holds under the hypothesis of the former invariant and of the guard of the event. For a model with state variable \( v \) and invariant \( I(v) \), and an event of the form:

\[
\text{ANY } x \text{ WHERE } P(x, v) \text{ THEN } v := E(x, v) \text{ END}
\]

the statement to be proved is:

\[
I(v) \land P(x, v) \Rightarrow I(E(x, v))
\]

### 2.2.1 Example: Dealing with Events

The following example shows how to add/group/split events.

We consider here a simple system \((M0)\). Its state is represented by a variable \( xx \) and one event may occur, \( evol \), which led variable \( xx \) to evolve according to its definition domain, namely \( \mathbb{N} \).

```plaintext
SYSTEM M0
VARIABLES
    xx
INVARIANT
    xx \in \mathbb{NAT}
INITIALISATION
    xx := xx
EVENTS
    evol = BEGIN xx \in (xx \in \mathbb{NAT}) END
END
```
In the next refinement ($M_1$), we express that our variable $xx$ evolves according to the value of another state variable, $yy$, which can be considered as a stimulus.

A new event is created, $evoly$, related to the evolution of the variable $yy$. In order to ensure that this new event may not take forever the control of the system and prevent event $evol$ to occur, the number of occurrences of that event should be finite. We introduce in this case a finite set $zz$. Values for $yy$ are chosen in this set and removed from it. When this set is empty, event $evoly$ cannot be fired any more and variable $yy$ keeps its last value. The VARIANT clause is introduced, containing an expression that should be decreased each time the new event $evoly$ is fired and should reach zero in a finite number of steps. The VARIANT expression is the cardinal of the set $zz$.

Event $evol$ is split into 3 events, each one describing the evolution of $xx$ in different cases, depending on the value of $yy$. In fact, $xx$ is modified in order to reach the value of $yy$.

The event refinement diagram is given below:

![Event Refinement Diagram]

**Figure 2.6: Events location in components $M_0$ and $M_1$.**

In the B model (component $M_1$), $evol$ is explicitly replaced by $evol1$, $evol2$ and $evol3$. Each one is declared to refine $evol$. On the other hand, there is no need to express that $evoly$ is new event since it can be deduced from previous component.

We obtain the event B model below.

```plaintext
REFINEMENT M1
REFINES M0
VARIABLES
   xx, yy, zz
INVARIANT
   xx ∈ NAT &
   yy ∈ NAT &
   zz ⊆ NAT &
   zz ∈ FIN(zz)
INITIALISATION
   xx :∈ NAT ||
   yy, zz ∈ (yy ∈ NAT ∧ zz ⊆ NAT ∧ zz ≠ ∅ ∧ yy ∈ zz ∧ zz ∈ FIN(zz))
VARIANT
   card(zz)
EVENTS
   evoll ref evol = SELECT xx=yy THEN skip END;
```
evol2 ref evol = SELECT xx>yy THEN xx:=xx-1 END;
evol3 ref evol = SELECT xx<yy THEN xx:=xx+1 END;
evol =
   SELECT zz ≠ ∅ THEN
      ANY val WHERE val ∈ zz THEN
         zz := zz - {val} ||
      END
      yy := val
   END
END

In the second refinement, evoly remains unchanged. evol1, evol2 and evol3 are grouped into one event to form an IF THEN ELSE substitution. This new event, evol4, is explicitly declared to refine evol1, evol2 and evol3.

Figure 2.7: Events location in components M0, M1 and M2.

### 2.2.2 Example: Using Modalities

The following example shows how to use modalities.

We consider here a simple system (S0). Its state is represented by two variables xx and input. xx is linked with input such as, when input evolves (event ev1), then xx is likely to evolve too (events ev2 to ev7). We can consider that ev1 represents a kind a stimulus (from the environment) and ev2 to ev7 represent the response to this stimulus. Modalities can be used to write invariant properties restricted to some events (i.e., ev2, ev3, ... ev7). In this case, we would like to verify that the response complies with some general law, exhibited in the ESTABLISH clause.

```plaintext
SYSTEM
   S0
SETS
   EE = {e0, e1, e2, e3, e4};
   INPUTS = {v1, v2, v3}
VARIABLES
```
\[ xx, \]
\[ \text{input} \]
\textbf{INVARIANT}
\[ xx \in \text{EE} \land \]
\[ \text{input} \in \text{INPUTS} \]
\textbf{INITIALISATION}
\[ xx := e_0 \mid | \mid \]
\[ \text{input} \in \text{INPUTS} \]
\textbf{EVENTS}
\[ \text{ev1} = \]
\[ \text{ANY} \text{ in WHERE in} \in \text{INPUTS} \]
\[ \text{THEN} \]
\[ \text{input} := \text{in} \]
\[ \text{END}; \]
\[ \text{ev2} = \text{SELECT} \text{ input=}v_1 \land xx = e_0 \text{ THEN xx} := e_2 \text{ END}; \]
\[ \text{ev3} = \text{SELECT} \text{ input=}v_1 \land xx \neq e_0 \text{ THEN xx} := e_0 \text{ END}; \]
\[ \text{ev4} = \text{SELECT} \text{ input=}v_2 \land xx = e_0 \text{ THEN xx} := e_2 \text{ END}; \]
\[ \text{ev5} = \text{SELECT} \text{ input=}v_2 \land xx \neq e_0 \text{ THEN xx} := e_3 \text{ END}; \]
\[ \text{ev6} = \text{SELECT} \text{ input=}v_3 \land xx = e_1 \text{ THEN xx} := e_4 \text{ END}; \]
\[ \text{ev7} = \text{SELECT} \text{ input=}v_3 \land xx \neq e_1 \text{ THEN xx} := e_1 \text{ END} \]
\textbf{MODALITIES}
\textbf{BEGIN}
\[ \text{ev2, ev3, ev4, ev5, ev6, ev7} \]
\textbf{ESTABLISH}
\[ \text{input m xx} \in \{ \]
\[ (v_1 \text{ m e0}), (v_1 \text{ m e2}), (v_1 \text{ m e3}), \]
\[ (v_2 \text{ m e2}), (v_2 \text{ m e3}), \]
\[ (v_3 \text{ m e1}), (v_3 \text{ m e3}), (v_3 \text{ m e4}) \}
\[ \} \]
\textbf{END}

\subsection*{2.3 UML+B Notation}

The U2B tool is a prototype tool to convert adapted forms of UML class diagrams and state chart diagrams into specifications in the B language. The aim is to use some of the features of UML diagrams to make the process of writing formal specifications easier, or at least more to the average programmer. The translation relies on the precise expression of additional behavioural constraints in the specification of class diagram components and in state charts attached to the classes. These constraints are described in an adapted form of the B abstract machine notation. The type of class diagrams that can be converted is restricted in order to comply with constraints of the B-method without making the resultant B unnatural. The resulting UML model is a precise formal specification but in a form which is more friendly to the average programmer, particularly if they use the same UML notation for their program design work.
2.3.1 U2B Class Diagram Translator

The U2B translator converts Rational Rose\(^9\) UML Class diagrams [Rational00a], including attached state charts, into the B notation. U2B is a script file that runs within Rational Rose and converts the currently open model to B. It is written in the Rational Rose Scripting language, which is an extended version of the Summit Basic Script language [Rational00a, Rational00b]. U2B is configured as a menu option in Rose. U2B uses the object-oriented libraries of the Rose Extensibility Interface to extract information about the classes in the logical diagram of the currently open model. The object model representation of the UML diagram means that information is easily retrieved and the program structure can be based around the logical information in the class rather than a particular textual format. U2B uses Microsoft Word\(^10\) to generate the B Machine files.

2.3.1.1 Translation of Structure and Static Properties

The translation of classes, attributes and operations is derived from proposals for converting OMT (Object Modelling Technique) to B [Meyer99]. However, since our aims are primarily to assist in the creation of a B specification rather than to generate a formal equivalent of a UML specification, our translation simplifies that proposed by [Meyer99]. This is achieved by restricting the translation to a suitable subset of UML models.

A separate machine is created for each class and this contains a set of all possible instances of the class and a variable that represents the subset of current instances of the class. Attributes and (unidirectional) associations are translated into variables whose type is defined as a function from the current instance to the attribute type (as defined in the class diagram) or associated class.

For example consider the following class diagram with classes \(A\) and \(B\), where \(A\) has an attribute \(x\) and there is a unidirectional association from \(A\) to \(B\) with role \(y\) and 0 or 1 multiplicity at the target end. A second association, \(w\), has a ‘many’ multiplicity:

\[
\begin{align*}
A & \quad x : X \\
+ y & \quad 0..1 \\
+ w & \quad 1..n \\
B &
\end{align*}
\]

This will result in the following machine representing all instances of \(A\):

\(^9\)Rational Rose is a trademark of the Rational Software Corporation

\(^{10}\)Microsoft Word97 is a trademark of the Microsoft Corporation.
Note that the multiplicity of the association \( w \) is handled as a function from instances of class \( A \) to sets of instances of class \( B \) using the power set operator (\( \text{POW} \) and \( \text{POW1} \)). The machine is initialised with no instances and hence all attribute and association functions are empty. A separate machine will be generated for class \( B \).

**Association multiplicities** In UML, multiplicity ranges constrain associations. The multiplicities are equivalent to the usual mathematical categorisations of functions: partial, total, injective, surjective and their combinations. Note that the multiplicity at the target end of the association (class \( B \) in the example above) specifies the number of instances of \( B \) that instances of the source end, class \( A \), can map to. This can be confusing when thinking in terms of functions because the constraint is at the opposite end of the association to the set it is constraining. The multiplicity of an association determines its modelling as shown in Table 2. We use functions to sets of the target class instances (e.g., \( \text{POW}(B) \)) to avoid non-functions. Note that \( 0..n \) is assumed unless otherwise specified in UML.
Association Representations in B for Different Multiplicities

$Ai$ and $Bi$ are the current instances sets of class A and B respectively (i.e., $A\text{instances}$ and $B\text{instances}$) and $f$ is a function representing the association (i.e., the role name of the association with respect to the source class, $A$).

<table>
<thead>
<tr>
<th>UML association multiplicity</th>
<th>Informal description of B representation</th>
<th>B invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0..n \rightarrow 0..1$</td>
<td>partial function to $Bi$</td>
<td>$Ai$ 2 $Bi$</td>
</tr>
<tr>
<td>$0..n \rightarrow 1..1$</td>
<td>total function to $Bi$</td>
<td>$Ai$ 3 $Bi$</td>
</tr>
<tr>
<td>$0..n \rightarrow 0..n$</td>
<td>total function to subsets of $Bi$</td>
<td>$Ai$ 3 $\text{P}ower(Bi)$</td>
</tr>
<tr>
<td>$0..n \rightarrow 1..n$</td>
<td>total function to non-empty subsets of $Bi$</td>
<td>$Ai$ 3 $\text{P}ower1(Bi)$</td>
</tr>
<tr>
<td>$0..1 \rightarrow 0..1$</td>
<td>partial injection to $Bi$</td>
<td>$Ai$ 4 $Bi$</td>
</tr>
<tr>
<td>$0..1 \rightarrow 1..1$</td>
<td>total injection to $Bi$</td>
<td>$Ai$ 5 $Bi$</td>
</tr>
<tr>
<td>$0..1 \rightarrow 0..n$</td>
<td>total function to subsets of $Bi$ which do not intersect</td>
<td>$Ai$ 3 $\text{P}ower(Bi)$ $&amp;$ $\text{inter(ran}(f))=0$</td>
</tr>
<tr>
<td>$0..1 \rightarrow 1..n$</td>
<td>total function to non-empty subsets of $Bi$ which do not intersect</td>
<td>$Ai$ 3 $\text{P}ower1(Bi)$ $&amp;$ $\text{inter(ran}(f))=0$</td>
</tr>
<tr>
<td>$1..n \rightarrow 0..1$</td>
<td>partial surjection to $Bi$</td>
<td>$Ai$ 2 $Bi$</td>
</tr>
<tr>
<td>$1..n \rightarrow 1..1$</td>
<td>total surjection to $Bi$</td>
<td>$Ai$ 7 $Bi$</td>
</tr>
<tr>
<td>$1..n \rightarrow 0..n$</td>
<td>total function to subsets of $Bi$ which cover $Bi$</td>
<td>$Ai$ 3 $\text{P}ower(Bi)$ $&amp;$ $\text{union(ran}(f))=Bi$</td>
</tr>
<tr>
<td>$1..n \rightarrow 1..n$</td>
<td>total function to non-empty subsets of $Bi$ which cover $Bi$</td>
<td>$Ai$ 3 $\text{P}ower1(Bi)$ $&amp;$ $\text{union(ran}(f))=Bi$</td>
</tr>
<tr>
<td>$1..1 \rightarrow 0..1$</td>
<td>partial bijection to $Bi$</td>
<td>$Ai$ 4 $Bi$</td>
</tr>
<tr>
<td>$1..1 \rightarrow 1..1$</td>
<td>total bijection to $Bi$</td>
<td>$Ai$ 5 $Bi$</td>
</tr>
<tr>
<td>$1..1 \rightarrow 0..n$</td>
<td>total function to subsets of $Bi$ which cover $Bi$ without intersecting</td>
<td>$Ai$ 3 $\text{P}ower(Bi)$ $&amp;$ $\text{union(ran}(f))=Bi$ $&amp;$ $\text{inter(ran}(f))=0$</td>
</tr>
<tr>
<td>$1..1 \rightarrow 1..n$</td>
<td>total function to non-empty subsets of $Bi$ which cover $Bi$ without intersecting</td>
<td>$Ai$ 3 $\text{P}ower1(Bi)$ $&amp;$ $\text{union(ran}(f))=Bi$ $&amp;$ $\text{inter(ran}(f))=0$</td>
</tr>
</tbody>
</table>
Table 2: How associations are represented in B for each possible multiplicity constraint.

In Figure 2.8 a mapping represents an association between the classes A and B with multiplicity $0..n \rightarrow 0..1$. The representation in the B notation is a partial function. It is not a total function because $a_4$ doesn’t map to anything in B (as indicated by the 0 at the right hand end of $0..n \rightarrow 0..1$). It is not injective because $b_2$ is mapped to by both $a_2$ and $a_3$ (as indicated by the n at the left hand end of $0..n \rightarrow 0..1$). It is not surjective because $b_3$ is not mapped to by anything in $A$ (as indicated by the 0 at the left hand end of $0..n \rightarrow 0..1$).

![Figure 2.8: Mapping representing a $0..n \rightarrow 0..1$ association.](image)

**Attribute types** Attribute types may be any valid B expression that defines a set. This includes predefined types such as `NAT`, `NAT1`, `boolean` and `string` (if translating to B-Core B, the appropriate B library machines must be referenced via a `SEES` clause in the class’s specification documentation window) functions, sequences, power sets, instances of another class (referenced by the class name) or enumerated or deferred sets defined in the class specification documentation window. If the type involves another class (and there is no unidirectional path of associations to that class) the machine for that class will be referenced in a `USES` clause so that its current instances set can be read. If there is a path of unidirectional associations to the class it will be extended (`EXTENDS`) by this machine in order to represent the association and this will provide access to the instances set. (Note that only unidirectional associations are interpreted as associations. Unspecified or bi-directional associations are ignored and can therefore be used to indicate type dependencies diagrammatically if required). Any references to the class in type definitions of variables or operation arguments will be changed to the current instances set for that class.

For example, the following shows a class that has an attribute $x$ of type, non-empty finite subset of natural numbers. It has an attribute $y$ that is of type, non-empty sequence of booleans. The library machine `Bool_TYPE` has been referenced via a `SEES` clause in the class’s documentation window (this would not be necessary for Atelier-B). It has an attribute $z$ that has type, total
injection from $Y$ to permutations of $Z$. A \texttt{SETS} clause has been added to the class’s documentation window that defines $Y$ as a deferred set and $Z$ as an enumerated set.

Note that \texttt{Export Control} settings in the class specification are not used in the U2B translation. The corresponding B machine for class $A$ is shown below.

\begin{verbatim}
MACHINE A
SEES
  Bool_TYPE
USES
  B
SETS
  ASET ;
  Y ;
  Z = \{blue, yellow, green, red\}
VARIABLES
  Ainstances, 
  x, 
  y, 
  z, 
  w
INVARIANT
  Ainstances ( ASET &
               x : Ainstances 3 FIN1(NAT) &
               y : Ainstances 3 seq1(BOOL) &
               z : Ainstances 3 Y 5 perm(Z) &
               w : Ainstances 3 POW(Binstances)
\end{verbatim}
Global Definitions It is often useful to define types as enumerated or deferred sets for use in many machines. We use class utilities for this. In UML, a class utility is a class that has no instances, only static (class-wide) operations and attributes. The U2B translator creates a machine for each class utility and copies any text in the specification documentation window of its class specification into the machine. Hence definitions, sets and constants can be described in B clauses in the documentation window. Any machines that reference items defined in this way must have an association to the class utility. This association will not be interpreted as an association to an ordinary class). In the following example a class utility \texttt{DEFS} is used to define a set \(X\) that is used as a type by two other classes.

The corresponding machine for class utility \texttt{DEFS} is:

```
MACHINE DEFS
SETS X
END
```

The machines for classes \(A\) and \(B\) will reference \texttt{DEFS} via a \texttt{SEES} clause:

```
SEES DEFS
```
Local Definitions Sets can also be defined locally to a class in the class's specification documentation window. In fact, any valid B clause can be added in this window. For example, we use this method to specify invariants for the class. Each clause must be headed by its B clause name in capitals and starting at the beginning of a line, the text that follows that clause, up until the next clause title (if any) will be added to the appropriate clause in the machine. Any text before the first clause is treated as comment and added as such at the top of the machine.

Instance Creation and Initialisation of Attributes and Associations. A create operation is automatically provided for each class machine so that new instances can be created. This picks any instance that is not already in use, adds it to the current instances set, and adds a maplet to each of the attribute/association relations mapping the new instance to the appropriate initial value. Note that, according to our definition (via translation) of class diagrams, association means that the source class is able to invoke the methods of the target class. The example below is similar to the first example but class $A$ has an additional attribute $z$, that has an initial value $k$.

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) {\textbf{A}};
  \node (B) at (2,0) {\textbf{B}};
  \draw[->] (A) -- (B) node [midway, above] {$+y$};
  \draw[->] (A) -- (B) node [midway, below] {$+w$};
  \draw[->] (B) -- (A) node [midway, above] {$0.1$};
  \draw[->] (B) -- (A) node [midway, below] {$1..n$};
  \node at (A.225) {\texttt{x : X}};
  \node at (A.315) {\texttt{z : X = k}};
  \node at (B.45) {0.1};
  \node at (B.135) {$1..n$};
\end{tikzpicture}
\end{center}

\begin{verbatim}
Return <- Acreate =
  PRE
    Ainstances != ASET
  THEN
    ANY new
    WHERE
      new ∈ ASET - Ainstances
    THEN
      Ainstances := Ainstances ∪ \{new\} ||
      x(new):∈ X ||
      z(new) := k ||
      w(new):∈ POW1(Binstances) ||
      Return := new
  END
END
\end{verbatim}

Note that, because $x$ has no initial value specified, it is initialised non-deterministically to any value of the type $x$ ($\in$ means any value belonging to). Similarly the association $w$ must be
initialised to a non empty set because its multiplicity may be greater than one but is definitely
greater than zero. (Currently, we have no means of specifying initial values for associations). It is
initialised non-deterministically to any non-empty subset of B instances. The association y is not
initialised because its multiplicity is 0 or 1 and it may, therefore, be undefined. Initially the new
instance will have no maplet representing the association y.

**Singular Classes** Often, a B machine models a single generic instance of an entity, rather than an
explicit set of instances (in the same way that a class in UML leaves instance referencing
implicit). The resulting specification is simpler and clearer for not modelling multiple instances.
The U2B translator creates a single-instance machine if the class multiplicity (cardinality) is set
to 1..1 in the UML class specification. Note that this can only be done at the top level of a
structure since at lower levels the instance set is used for referencing by the higher level. Below is
shown the machine representing class A from the first example above if the class’s multiplicity is
set to 1..1. Note that there is no modelling of instances, the type of attributes is simpler because
it is no longer necessary to map from instances to the attribute type. There is no instance creation
operation; attributes are initialised in the machine initialisation clause.

```
MACHINE A
EXTENDS B
VARIABLES
   x, w, y
INVARIANT
   x ∈ X ∧
   w ∈ POW1(Binstances) ∧
   y ∈ Binstances
INITIALISATION
   X := X ||
   w := POW1(Binstances) ||
   y := ∅
END
```

**Restrictions** The B method imposes some restrictions on the way machines can be composed.
These restrictions ensure compositionality of proof. Their impact is that no write sharing is
allowed at machine level (*i.e.*, a machine may only be included or extended by one other
machine). Also, the inclusion mechanism of B is hierarchical so that, if $M_1$ includes $M_2$, then $M_2$
cannot, directly or transitively, include $M_1$. We reflect these restrictions in the UML form of the
specification, which must therefore be tree like in terms of unidirectionally related classes.
Non-navigable (and bi-directional) associations are ignored but may be used to illustrate the use of
another class as a type (i.e., read access only). However, multiple, parallel associations between the same pair of classes are permitted.

Although we would like to adhere to the UML class diagram rules as much as possible, since our aim is to make B specification more approachable rather than to formalise the UML we are relatively happy to impose restrictions on the diagrams that can be drawn. That is, we only define translations for a subset of UML class diagrams. Other authors [Facon96, Meyer99, Meyer00, Nagui94, Shore96] have suggested ways of dealing with the translation of more general forms of class diagrams. However, the structures of B machines that result from these more general translations can be cumbersome. If the specification were written directly in B, it would be highly unlikely that the resulting B would have this form. Since we also desire a usable B specification we prefer to restrict the types of diagrams that can be drawn.

2.3.1.2 Dynamic Behaviour

The dynamic behaviour modelled on a class diagram that is converted to B by U2B is embodied in the behaviour specification of classes operations and in invariants specified for the classes. UML does not impose any particular notation for these operation and invariant constraint definitions; they could be described in natural language or using UML's Object Constraint Language (OCL). However since we wish to end up with a B specification it makes sense to use bits of B notation to specify these constraints. The constraints are specified in a notation that is close to B notation but has to observe a few conventions in order for it to become valid B within the context of the machine produced by U2B. When writing these portions of B the writer should not need to consider how the translation would represent the features (associations, attributes and operations) of the classes. Also we felt we should follow the more object-oriented conventions of implicit self-referencing and use of the dot notation for explicit instance references. Therefore, when writing the constraints, a dot notation is used to reference the ownership of features. This is illustrated in examples below.

Invariant Unfortunately there is no dedicated text box for a class invariant in Rational Rose. One suggestion is to put invariant constraints in a note attached to the class [Warmer99], but notes are treated as an annotation on a particular view in Rational Rose and not part of the model. This makes them difficult to access from the translation program and unreliable should we extend the conversion to look at other views. Therefore we include the invariants as a clause in the documentation text box of the class' specification window. The invariants are generally of two kinds, instance invariants (describing properties that hold between the attributes and relationships within a single instance) and class invariants (describing properties that hold between different instances). To deal with instance invariants, and keeping with the implicit self-reference style of UML, we chose to allow the explicit reference to this instance to be omitted. U2B will add the universal quantification over all instances of the class automatically. For class invariants, the
quantification over instances is an integral part of the property and must be given explicitly. Hence, U2B will not need to add instance references.

For example, if $bx \in \text{NAT}$ is an attribute of class $B$ then the following invariant could be defined in the documentation box for class $B$:

\[
\begin{align*}
& bx < 100 \land \\
& \forall (b1, b2). ( (b1 \in B \land b2 \in B \land b1 \neq b2) \Rightarrow (b1.bx \neq b2.bx) 
\end{align*}
\]

This would be translated to:

\[
\begin{align*}
& \forall (thisB). (thisB \in B \text{instances} \Rightarrow \\
& \quad bx(thisB) < 100 \land \\
& \quad \forall (b1, b2). ( (b1 \in B \text{instances} \land b2 \in B \text{instances} \land b1 \neq b2) \\
& \quad \Rightarrow (bx(b1) \neq bx(b2)) 
\end{align*}
\]

The translation has added a universal quantification, $thisB$, over all instances of $B$ and this is used in the first part of the invariant. It is not used in the second part where the invariant already references instances of class $B$. (Note that currently the translator adds one universal quantification for the entire invariant whether or not it is needed).

**Operation Semantics** Operation preconditions are specified in a textual format attached to the operation within the class. Details of operation behaviour are specified either in a textual format attached to the operation, or in a state chart attached to the class. Operation behaviour may be specified completely by textual annotation, completely by state chart transitions, or by a combination of both composed as simultaneous specification.

**Operation Textual behaviour specification** In Rational Rose, *Specifications* are provided for operations (as well as many other elements) and these provide text boxes dedicated to writing pre-conditions and semantics for the operation. Although Rational Rose also provides a *post-condition* text box, this is not currently used, as the *semantics* box suit the pseudo-operational style of B better.

Operations need to know which instance of the class they are to work on. This is implicit in the class diagram. The translation adds a parameter $thisCLASS$ of type $CLASS\text{instances}$ to each operation. This is used as the instance parameter in each reference to an attribute or association of the class.
In the above example, \textit{set}_y might have the following precondition:

\begin{verbatim}
i > y.bx
\end{verbatim}

and semantics

\begin{verbatim}
y.b_op(i) || IF y.bx < 100 THEN out := FALSE ELSE out := TRUE END
\end{verbatim}

which would be translated to

\begin{verbatim}
i > bx(y(thisA))
\end{verbatim}

and

\begin{verbatim}
b_op(y(thisA)) || IF bx(y(thisA)) < 100 THEN out := FALSE ELSE out := TRUE END
\end{verbatim}

**Operation Return Type** UML operation signatures contain a provision for specifying the type for a value returned by the operation. Since B infers this from the body of the operation we use it instead to name the identifiers that represent operation return values. The string entered in the return type field for the operation will be used as the operation return signature in the B machine representing the class. For example, the \textit{set}_y operation in the above class diagram has its return field set to \textit{out}. The operation signature for \textit{set}_y in the B machine A will be:

\begin{verbatim}
out c set_y (thisA,i) =
\end{verbatim}
State chart Behavioural Specification  For classes that have a strong concept of state change, a
state chart representation of behaviour is appropriate. In UML a state chart model can be attached
to a class to describe its behaviour. A state chart model consists of a set of states and transitions
that represent the state changes that are allowed. If a state chart model is attached to a class the
U2B translator combines the behaviour it describes with any operation semantics described in the
operation specification semantics windows. Hence operation behaviour can be defined either in
the operation semantics window or in a state chart model for the class or in a combination of
both.

The name of the state chart model is used to define a state variable. (Note that this is not the name
of a state chart diagram, several diagrams could be used to draw the state chart model of a class).
The collection of states in the state chart model is used to define an enumerated set that is used in
the type invariant of the state variable. The state variable is equivalent to an attribute of the class
and may be referenced elsewhere in the class and by other classes. State chart transitions define
which operation call causes the state variable to change from the source state to the target state,
\textit{i.e.}, an operation is only allowed when the state variable equals a state from which there is a
transition associated with that operation. To associate a transition with an operation, the
transition’s name must be given the same name as the operation. Additional guard conditions can
be attached to a transition to further constrain when it can take place. All transitions cause the
implicit action of changing the state variable from the source state to the target state. (The source
and target state may be the same). Additional actions (defined in B) can also be attached to
transitions. The translator finds all transitions associated with an operation and compiles a
SELECT substitution of the following form:

\begin{verbatim}
SELECT statevar=sourcestate1 \land sourcestate1\_guards
THEN statevar:=targetstate1 || targetstate1\_actions
WHEN statevar=sourcestate2 \land sourcestate2\_guards
THEN statevar:=targetstate2 || targetstate2\_actions
<etc>
END ||
\end{verbatim}

This is composed with the operation pre-condition and body (if any) from the textual
specification in the operation's \textit{pre-condition and semantics} windows:

Let \textit{Popw} be the precondition in the operation \textit{precondition window}, \textit{Sosw} be the operation
body from the \textit{operation semantics window} and \textit{Gstc} the SELECT substitution for this
operation composed from the state chart. Then the translator will produce the following
operation:
This can be represented more succinctly in B as:

\[ \text{Popw} \mid (Gstc \lor \text{Sosw}) \]

Hence the precondition, Popw, has precedence and, if false, the operation will abort. If an event B style systems simulation is desired, the specifier should take care not to define pre-conditions that conflict with the transition guards. (For example, if an event only occurs if an attribute bx is positive, and this is modelled by a guarded transition; adding the pre-condition \( bx > 0 \) would change the meaning of the model to represent a system where if the event occurs the operation aborts).

Note that it would be entirely valid (although somewhat obtuse) to write a pre-condition within the operation semantics window: \( \text{Sosw} = \text{Posw} \lor \text{Slosw} \). However, preconditions take precedence in simultaneous substitutions, so

\[ (Gstc \lor (\text{Posw} \lor \text{Slosw})) = \text{Posw} \lor (Gstc \lor \text{Slosw}). \]

Hence, writing the precondition in the operation semantics window is equivalent to writing it in the precondition window. It has the same precedence and possible conflicts with the operation guards derived from the state chart. We feel that writing the precondition in the operation semantics window should be discouraged because the precedence may not be obvious to readers of the specification.

If the pre-condition \((\text{Popw} \land \text{Posw})\) is true, then the guard from \( \text{Gstc} \) takes precedence over the simultaneous substitution \( \text{Sosw} \). This means that the textual operation body from the operation semantics window, although defined separately from the state chart and not associated with any particular state transition, is only enabled when at least one of the state transitions is enabled. That is, if

\[ \text{Gstc} = (G1 \Rightarrow T1) \lor \ldots \lor (Gn \Rightarrow Tn) \]

then,

\[ (\text{Gstc} \lor \text{Sosw}) = (G1 \Rightarrow (T1 \lor \text{Sosw})) \lor \ldots \lor (G \Rightarrow (Tn \lor \text{Sosw})) \]

where \( \lor \) represents choice.

Actions should be specified on state transitions when the action is specific to that state transition. Where the action is the same for all that operation's state transitions, it may be specified in the
operation semantics window in order to avoid repetition. The chart below illustrates how a state chart can be used to guard operations and define their actions. It also shows how common actions can be defined in the operation semantics window and how a pre-condition could upset the constraints imposed by the state chart.

The chart has two states, zero and non_zero. The implicit state variable, \( b_{\text{state}} \) (the name of the state chart model) is treated like an attribute of type \( \text{B\_STATE} = \{ \text{zero, non\_zero} \} \). An invariant defines the correspondence between the value of the attribute \( bx \) and the state zero. When an instance is created its \( b_{\text{state}} \) is initialised to zero because there is a transition from an initial state to zero.
MACHINE B

SETS

BSET;
B_STATE={zero,non_zero}

VARIABLES

Binstances,
b_state,
bx

INVARIANT

Binstances ⊆ BSET ∧
b_state ∈ Binstances → B_STATE ∧
bx ∈ Binstances → NAT ∧
∀(thisB).(thisB ∈ Binstances ⇒
(b_state(thisB)=zero ⇔ (bx(thisB)=0))
)

INITIALISATION

Binstances := ∅ ||
b_state := ∅ ||
bx := ∅

OPERATIONS

Return ← Bcreate =

PRE

Binstances ≠ BSET

THEN

ANY new

WHERE

new ∈ BSET - Binstances

THEN

Binstances := Binstances ∪ {new} ||
b_state(new):=zero ||
bx(new) ∈ NAT ||
Return := new

END

END;

Operation inc can occur in either state. Its action is different depending on the starting state and so actions have been defined on the transitions and are combined with the change state action.
Operation `inc` has two guarded alternatives when in state `non_zero` but does not occur while in state `zero`. Since the action is the same for both transitions it has been defined in the *operation semantics* window.

```
inc (thisB) =
  PRE
    thisB ∈ Binstances
  THEN
    SELECT b_state(thisB)=zero
      THEN b_state(thisB) := non_zero ||
          bx(thisB):=bx(thisB)+10
      WHEN b_state(thisB)=non_zero
      THEN bx(thisB) := bx(thisB)+1
  END
END
```

Operation `dec` has two guarded alternatives when in state `non_zero` but does not occur while in state `zero`. Since the action is the same for both transitions it has been defined in the *operation semantics* window.

```
dec (thisB) =
  PRE
    thisB ∈ Binstances
  THEN
    SELECT b_state(thisB)=non_zero ∧
      bx(thisB)=1
      THEN b_state(thisB) :=zero
      WHEN b_state(thisB)=non_zero ∧
      THEN bx(thisB):=bx(thisB)+1
      WHEN bx(thisB)>1
      THEN skip
      END ||
      bx(thisB):=bx(thisB)-1
  END
END
```

If we had put the pre-condition `bx > 0` in the *operation specification precondition* window (or even in the *operation semantics* window), the guard would no longer function since the precondition would fail resulting in an abort when `bx = 0`. 
dec (thisB) =
  \textbf{PRE} \\
  \quad \text{thisB} \in \text{Binstances} \land \\
  \quad bx(thisB)>0 \\
  \textbf{THEN} \\
  \text{SELECT} \ b\_state(thisB)=\text{non}_\text{zero} \land \\
  \quad bx(thisB)=1 \\
  \text{THEN} \ b\_state(thisB):=\text{zero} \\
  \text{WHEN} \ b\_state(thisB)=\text{non}_\text{zero} \land \\
  \quad bx(thisB)>1 \\
  \text{THEN} \ \text{skip} \\
  \textbf{END} \ \| \\
  \quad bx(thisB):=bx(thisB)-1 \\
\textbf{END}

This could be avoided by repeating the precondition and decrement substitution in the action field of each \textit{dec} transition on the state chart in which case the guard would take precedence.
3 Constructing Models

In this chapter we start by presenting in more detail each of the case studies developed within the MATISSE project. We divide the rest of the chapter into two different types of models: system-level models and computation models.

3.1 Modelling in B

3.1.1 Quick Rules

The following commandments have been successively used on significant projects:

1. Before constructing the AMN model, discover and arrange its most important entities within a not too constraining framework.
2. Adopt a strict top-down approach to analyze the problem and construct the AMN model.
3. Use abstract machines, but also fairly simple implementations, to define an abstract, hierarchical, and modular AMN model of the system.
4. Introduce critical properties in invariants or operations as high as possible in the hierarchical AMN model.
5. Construct small invariants and operations. To do it, introduce intermediate abstract machines that break down large invariants and operations.
6. Before proving and refining a software model, check the relevance and the tractability of the abstract machines, of their interfaces, and of the architecture. To do it, construct the complete architecture of the AMN model before proving and refining.
7. Avoid under-specification, even if it seems redundant, first define completely the desired feature in abstract terms, and then implement it concretely.

3.1.2 Modelling

This chapter aims at providing some useful hints on the modelling with B. These hints are a mix of organizational and technical recommendations.

3.1.2.1 Determining Properties

Method

With a clear and complete statement of requirements, it is possible to start the construction of a system. This is the role of the system specification phase. The designer analyses the problem, and elaborates an abstract model of it, not an abstract solution to it. Later on, s/he identifies the functions and the data that fulfil the requirements, determines their arrangement, and gives a structured description adequate for the forthcoming formalization.
It is a top-down process which goes, by successive refinement and decompositions, from the most general and abstract description of the system to the most detailed and concrete. The process often implies iterations, since information discovered at some stages of analysis may call into question the stages carried out before.

Some of the basic principles of the method are:

- To delimit precisely the system to be analysed, and to state all its links with the environment.
- To deal simultaneously with a reduced amount of information in order to overcome the complexity of the analysed system (divide and conquer approach).
- To decide of a particular point of view and to stick to it all along the analysis. Actually, systems can be analysed from different points of view (usage, safety, design, maintenance, etc.) and it is important to give a homogeneous description.
- To describe the analysed system in terms of activities and data arranged in a hierarchical structure that exhibits the flow of information and the dependence, causality, and decomposition relationships.
- To give a more and more detailed description as and when the analysis progresses until all the required features are captured.
- To adopt the adequate level of abstraction for the type of description built. Depending on this level, the description may tell either what the analysed system does, which is adequate for software specification, or how it does it, which is adequate for design.

According to the level of decomposition the introduced data must be global and abstract or detailed and concrete.

In the top-level component, two kinds of data are good candidates to be introduced:

- The high level data strictly necessary to determine the most important properties of the system.
- The high level data strictly necessary to formulate a major global requirement.

Properties

Statements concerning functions may be predicates defining an activation condition, predicates formulating an input/output relationship, or automata describing a behaviour. The physics of the monitored phenomena is a good source of predicates. Statements concerning data may be predicates describing invariant properties or dependence properties. The range of values of data and the equations that govern the safety conditions of the controlled system are examples of invariant properties. The relations between data of a given level of decomposition and data of a lower level which is needed to calculate the former are examples of dependence properties.
It is not necessary to formalise all the properties of the data and the functions of the system right from the beginning. This would introduce too concrete data structures, and lead to models hard to read and to prove.

**Invariants**

The place of invariants influences the architecture of the AMN model, for two reasons. First, since invariants usually relate variables, and since operations must preserve invariants, an operation updating a variable of an invariant must update the other variables in order to maintain the balance. Consequently, the same checks and the same updates may appear in all these operations, and they will have to be refined, implemented, and proved several times. Therefore, in order to reduce redundancy, the abstract machines that set invariants relating many variables should have few and global operations. Second, in a top-down constructed AMN model, it is the abstract machines at the upper levels of the hierarchy that have few and global operations. Therefore, it is recommended to place invariants relating several variables as high as possible in the hierarchy, in abstract machines containing few and global operations.

It may happen that systems are concerned with global safety requirements that relate two consecutive states of the system. Usually, it is painful to formalise these requirements with invariants, as it is necessary to introduce extra variables that increase the complexity of the model. On the other hand, when an abstract machine declares a single operation, it is equivalent to formalise requirements with the invariant or with the operation, since in the former case the operation preserves the invariant, while in the latter it establishes that invariant after each invocation. Moreover, the definition of an operation allows to reference previous values of the variables. So, global safety requirements that relate two consecutive states of the system, or more generally, dynamic invariants of the system, can be formalised in the definition of the single operation of the topmost abstract machine.

Invariants of an enrichment refinement must state the properties relating the introduced variables to variables inherited from the refined module.

Besides typing the variables introduced, invariants of data refinements set the relation that permits to retrieve the information stored in the replaced variables from the information stored in the replacing variables. There are distinct sorts of retrieve relations.

Sometimes, a value of the replaced variables may be represented by distinct values of the replacing variables. This is the case for example when a set of a refined module is replaced (refined) by a function: The retrieve relation indicates that the replaced set is exactly the range of the introduced function therefore, the indices of the elements of the set in the array are irrelevant, and two distinct permutations of the elements. Some other times, a value of the replaced variables is represented by one and only one value of the replacing variables.

More rarely, some values of the replaced variables are not represented by values of the replacing variables. This is possible only when the operations of the abstract module do not use effectively the non represented abstract values.
Invariants of some implementations produced at preliminary design must be completed,
particularly those of the implementations that are the first and unique refinement of an abstract
machine. All what has been said earlier concerning enrichment and data refinement invariants is
still applicable, the concrete variables being the locally declared variables and the variables of the
imported abstract machines.

Furthermore, the invariants of implementations may be used to relate the results of the operations
of one imported abstract machine to the pre-conditions of the operations of another imported
abstract machine. Usually, these invariants are formulated as implication predicates.

3.1.2.2 Building

Design
Design aims at identifying the different components/sub-systems and define their interfaces and
their organization, that is, the implementation architecture of the system.

Completeness, simplicity, and modularity are three major concerns of design. The former deals
with safety requirements capture. It determines the relevance of the model and, consequently, that
of the proof, as the proof of a model which does not formalize the safety requirements of the
system does not guarantee the absence of critical errors in the final system. The second concern
determines the feasibility of the proof, as we have noticed that proof complexity is closely related
to model complexity. Finally, the latter concern determines the organisation and the balanced
distribution of the work between the development team members.

All design techniques used effectively in industry encourage the hierarchical and modular
description of systems, since it is difficult, or even impossible, to give a flat description of a
complex system which is clear and correct. It is necessary to introduce progressively the
information and organise the processing in order to give a clear and accurate description of the
system: it is difficult to construct the AMN model of an industrial application simply in terms of
abstract machines. Thus, to introduce progressively the information and organise the processing
the refinement technique and the programming substitutions.

The architecture of an AMN model determines the organisation of the development and the
feasibility of the proof. This is why it is so important to construct a modular model with loosely
coupled components which may be developed and proved separately.

Operation bodies
When operations are not defined by simple or multiple substitutions, they must be defined with
“before/after” substitutions. A before/after substitution is either:
- a declaration substitution;
- or an unbounded choice substitution
Although it may be painful to define operations with “before/after” predicates, we think it is worth doing it for two reasons. First, it avoids thinking in programming terms, which is one of the most common mistakes made by B novices. Second, it forces designers to formulate operations in two different styles, and our experience tends to show that this helps to reveal either definition or implementation mistakes.

Bodies should define the final observable effects of operations on variables, they should not describe the actions carried out to obtain them. For global operations, which carry out many actions, non-determinism is of great help to fulfil this requirement.

Non-determinism is also useful to define operations that input external data in the application, as it permits to state abstractly the properties on which the application relies.

Operations of implementations should be small, they should not exceed forty lines, their average size being ten to fifteen lines.

Operations of implementations should be simple, they should avoid too many nested conditional substitutions.

The operations of enrichment refinements and data refinements are defined in the before/after substitution style, as the operations of abstract machines. The operations of algorithmic refinements are progressively constructed.

Finalising preliminary design implementations means writing the implementation invariants and the loop invariants that were deliberately ignored at that phase.

**Abstract Machines**

It may happen that the model obtained contains too many layers involving too simple abstract machines and implementations. This makes heavy the construction and the maintenance of the AMN model, and in the worst case it penalises the performance of the eventual programs, since it generates too many nested operation calls. To avoid this situation, it is recommended to eliminate the useless intermediate abstract machines and implementations, and introduce in place data and operations of the next lower layer. In other words, it is recommended to skip over the useless abstract machines, and use in place the abstract machines that implement them.

The opposite situation may also arise, when a layer comprises too big or too complex abstract machines and implementations. Then, it is necessary either to introduce intermediate abstract machines to encapsulate and abstract parts of the data and operations of the complex components or to create local operations (operations specified and implemented in the same implementation component, that cannot be called from the outside). Local operations are also useful when building loops, as operations called within the loop do not need anymore to be replicated in imported components.

Abstract machines at the leaves of the import tree of the AMN model encapsulate the data and services on which the rest of the application is implemented. Some of these abstract machines are
neither refined nor implemented formally, and their code has to be developed conventionally. They are called basic abstract machines\textsuperscript{11}.

It is good practice to group in distinct abstract machines interfaces concerning distinct devices. However, if there are not too many interfaces, it may be practical to group them all in one single basic abstract machine.

Basic abstract machines encapsulate also low level data manipulations, like data compression and decompression, interfaces of reused software components, \textit{etc}. As any other abstract machine, basic abstract machines may declare variables and should define completely their operations. However, they should not access variables of the other abstract machines of the application. That means in particular, that the exchange of data between the application and the runtime operating system must be carried out exclusively through parameter passing. Input operations return input values in their output parameters, and output operations accept output values in their input parameters. A basic abstract machine should be imported in the lowest implementation (in the hierarchy) that dominates all the abstract machines whose implementation needs that basic abstract machine, and should be seen by all the other abstract machines imported by that implementation. Thus, following our previous recommendations concerning the sees links, all the implementations that need that basic abstract machine can see it.

3.2 System-Level Models

A system-level model starts with an abstract formal model of the overall system to be controlled, including relevant control behaviour. Then this high-level model is refined, adding detail about interactions between sub-components of the system, so that formal models may be decomposed into the system under control and the embedded controllers. Thus, rather than starting with formal models of the embedded controllers and verifying that they interact with the system under control, it starts with a model of the desirable behaviour of the overall system and use refinement to derive models of the embedded controllers. The refinement is typically performed in several steps, and at each step one has to verify that any behaviour of the refined model is allowed by the previous model, thus ensuring that the final detailed model is correct with respect to the original system-level model. The advantage of this approach is that is makes it easier to deal with the complexity of the designs. Starting with a simple system-level model and introducing extra complexity one-step at a time. The system-level model is particularly suited to systems that consist of multiple distributed controllers. The system level model abstracts away from any distribution, and only introduces it in refinement steps.

3.2.1 Translating Informal Properties into B

\textsuperscript{11} Typically, the interfaces of the runtime operating system are formally defined in one or several basic abstract machines.
The railway example presented here [Butler02a] is a simplified version of the transportation case study (see Section 1.2.1). We will use this example to show how to translate informal properties into B. Figure 2.1 describes an abstraction of the railway states in B notation. SECTION and TRAIN are types used to identify track sections and trains respectively (1). A railway switch has two incoming and one outgoing sections (convergent) or vice-versa (divergent). Railway switches are regarded as special cases of track sections. The constant switch (2) represents the subset of sections that are switches, while the constant net is a graph representing the static connectivity between sections (i.e., the connectivity regardless of the switch positions). In the properties clause, switch is declared to be a subset of SECTION (3), while net is declared to be a relation between SECTION (4). Here, we have omitted other properties of net such as the fact that a section cannot be connected to itself.

```
MACHINE Railway
SETS SECTION, TRAIN
CONSTANTS switch, net
PROPERTIES
  switch ⊆ SECTION ∧ net ∈ SECTION ↔ SECTION
VARIABLES next, occp, front, back, braking, checked
INVARIANT
  next ⊆ net ∧ next ∈ SECTION 4 SECTION ∧ occp ∈ TRAIN ↔ SECTION ∧ front ∈ TRAIN → SECTION ∧ front ⊆ occp ∧ back ∈ TRAIN → SECTION ∧ back ⊆ occp ∧ braking ∈ TRAIN → Bool ∧ checked ∈ TRAIN → Bool ∧ ∀ t1, t2 . (t1 ∈ TRAIN ∧ t2 ∈ TRAIN ∧ front(t1) = back(t2) ⇒ braking(t1) = TRUE
```

Figure 3.1 Variables and invariant of abstract railway.

The variables of the machine (5) are typed in the invariant clause. Variable next represents the dynamic connectivity between sections based on the positions of switches. Note that we have abstracted away from the actual positions of switches and concentrated on connectivity. This makes it easier to specify and analyse the system-level model. Variable next will always be a
subgraph of net (6) and be an injective function (7) meaning a section can have at most one predecessor and at most one successor under next.

Each train may occupy several sections and this is modelled by the relation occp (8). Note that more than one train may be related to the same section by occp which is a potential hazard since trains occupying the same section may collide. Although the railway strives to avoid multiple occupancy of the same section, we feel that a pure model which absolutely prevented it would be unrealistic. The invariant does have a weaker safety constraint saying that if the front of \( t_1 \) is on the same section as the back of \( t_2 \), then \( t_2 \) should be braking (13). This may be insufficient to avoid collisions since it doesn't take account of stopping distance, but it gives a flavour of a safety constraint for the purposes of this paper. The functions \( \text{front} \) and \( \text{back} \) map each train to the section occupied by its front and back respectively (9), (10). The function \( \text{braking} \) maps each train to a boolean representing whether or not it is braking.

Properties of the abstract railway

At this level, the system just contains trains, sections, and switches. Safety is not ensured at this level, since collision and derailment might occur. These are the properties:

P1 Trains are running on several sections. This property assures that both front and back sections of a train \( t \) are in the occupancy function of \( t \). Furthermore, the set of sections occupied by a train should be continuous under the next function, i.e., there are no gaps in the train and all the switches occupied by the train are in the correct position.

\[
\forall t . (t \in \text{TRAIN} \Rightarrow \text{front}(t) \in \text{occp}^{-1}([t])) \land \\
\forall t . (t \in \text{TRAIN} \Rightarrow \text{back}(t) \in \text{occp}^{-1}([t])) \land \\
\forall t . (t \in \text{TRAIN} \Rightarrow \text{IsContiguous}(\text{back}(t), \text{occp}^{-1}([t]), \text{next}))
\]

P2 If a train occupies more than one section, then the front and back sections occupied by the train must be different.

\[
\forall t . (t \in \text{TRAIN} \land \text{card}(\text{occp}^{-1}([t])) > 1 \Rightarrow \text{front}(t) \neq \text{back}(t))
\]

P3 Each switch is located on a section, which is represented by property (3) in the railway machine.

\[
\text{switch} \subseteq \text{SECTION}
\]

P4 No section is connected to each other.

\[
\text{net} \cap (\text{SECTION} \times \text{SECTION}) = \emptyset
\]

P5 Switches are not connected to each other.

\[
\text{net} \cap (\text{switch} \times \text{switch}) = \emptyset
\]
A switch has at most two predecessors and one successor or one predecessor and two successors.

\[ \forall s . \ ( s \in \text{switch} \Rightarrow ( ( \text{card}(\text{net}[\{s\}]) \leq 2 \land \text{card}(\text{net}^{-1}[\{s\}]) \leq 1 ) \lor ( \text{card}(\text{net}[\{s\}]) \leq 1 \land \text{card}(\text{net}^{-1}[\{s\}]) \leq 2 ) ) ) \]

Non-switch sections have at most one predecessor and one successor.

\[ \forall s . \ ( s \in (\text{SECTION} - \text{switch}) \Rightarrow \text{card}(\text{net}[\{s\}]) \leq 1 \land \text{card}(\text{net}^{-1}[\{s\}]) \leq 1 ) ) \]

Operations the abstract railway

Figure 3.2 describes some operations of the system-level model. The Check operation checks whether the brakes need to be applied to a train \( t \). The brakes are applied if the front of \( t \) has no successor or its successor is already occupied\(^{12}\). The brakes are applied by setting \( \text{braking}(t) \) to \text{TRUE}. Regardless of whether the brakes need to be applied, the Check operation sets the \( \text{checked}(t) \) flag to \text{true}.

\[
\text{Check}(t) = \quad \begin{cases} 
\text{PRE} & t \in \text{TRAIN} \\
\text{THEN} & \\
& \text{braking}(t) := \text{bool} (\text{front}(t) \notin \text{dom}(\text{next}) \lor \text{next}(\text{front}(t)) \in \text{ran}(\text{occp})) \\
|| & \text{checked}(t) := \text{TRUE} \\
\text{END;}
\end{cases}
\]

\[
\text{EnterSection}(t,s) = \quad \begin{cases} 
\text{PRE} & t \in \text{TRAIN} \land s \in \text{SECTION} \\
\text{THEN} & \\
& \text{SELECT} \quad \text{checked}(t) \land \text{front}(t) \in \text{dom}(\text{next}) \land s = \text{next}(\text{front}(t)) \\
& \text{THEN} \quad \text{front}(t) := s || \text{occp} := \text{occp} \cup \{t \_ s\} || \text{checked}(t) := \text{FALSE} \\
\text{END}
\end{cases}
\]

\[
\text{LeaveSection}(t,s) = \quad \begin{cases} 
\text{PRE} & \\
\text{THEN} & \\
& \text{SELECT} \quad s \in \text{SECTION} \\
& \text{THEN} \quad \text{next}(s) := \text{FALSE} \\
\text{END}
\end{cases}
\]

\[
\text{SwitchChangeDiv}(sw,s1,s2) = \quad \begin{cases} 
\text{PRE} & sw, s1, s2 \in \text{SECTION} \\
\text{THEN} & \\
& \text{SELECT} \quad sw \in \text{switch} \land sw \notin \text{ran}(\text{occp}) \land \\
& (sw \_ s1) \in \text{next} \land (sw \_ s2) \in \text{net} \land s1 \neq s2 \\
& \text{THEN} \quad \text{next}(sw) := s2 \\
\text{END}
\end{cases}
\]

\[
\text{SwitchChangeDiv}(sw,s1,s2) = \quad \begin{cases} 
\text{PRE} & \\
\text{THEN} & \\
& \text{SELECT} \quad sw \in \text{switch} \land sw \notin \text{ran}(\text{occp}) \land \\
& (sw \_ s1) \in \text{next} \land (sw \_ s2) \in \text{net} \land s1 \neq s2 \\
& \text{THEN} \quad \text{next}(sw) := s2 \\
\text{END}
\end{cases}
\]

\(^{12}\) \( t \notin \text{dom}(\text{next}) \) means that \( \text{next}(t) \) is undefined. \( \text{ran}(\text{occp}) \) is the set of sections that are occupied.
Figure 3.2: Some operations of abstract railway.

The `EnterSection` operation models the front of train \( t \) entering section \( s \). Train \( t \) can enter section \( s \) provided \( t \) has been checked (since it last entered a section) and \( t \) has a successor section under \( \text{next} \). The effect of the operation is to change the front section of the train to be \( s \), add a mapping from \( t \) to \( s \) to the relation \( \text{occp} \) and reset the \( \text{checked}(t) \) flag.

Use of the `checked` flag means that the check operation must be applied to each train \( t \) at least once in between each `EnterSection` for \( t \). In the real system, the train cannot be directly prevented from entering a new section and the only way of satisfying this ordering constraint is to ensure that the `Check` operation is performed before the train enters a new section. So this represents a hard real-time constraint.

The `SwitchChangeDiv` operation represents a switch change for a divergent switch and is modelled by the effect the switch change has on the \( \text{next} \) function. Parameter \( \text{sw} \) must be a switch which is currently connected to successor section \( s_1 \). The effect of the operation is to update the \( \text{next} \) function so that \( \text{sw} \) is connected to \( s_2 \) instead. There is a further constraint on the occurrence of the switch change saying that the \( \text{sw} \) should not be currently occupied since changing an occupied switch has the potential to cause a derailment.

### 3.2.2 UML Development Incorporating Safety Aspects

The starting point of a system development is an informal specification describing the services desirable from the eventual product, with no or few hints about the various software and hardware parts. We consider this to be the most general starting point for the development process of a control system. The most important role is played by UML [UML1.4] in the first phase of the process, when a system specification is determined from the initial requirements. We define a method for using the UML diagrams for designing a control system specification [Petre00]. An essential contribution at this level is the embedding of the safety analysis within the UML method.

We depict the informal requirements of the healthcare case study with UML diagrams. The functional requirements of the system are captured together with their relationships in a use case diagram. The reliability and safety issues are given in the specification of the use cases as structured English text. Then the logically related use cases are determined, and grouped together into a control system component in component and class diagrams. The behaviour of the component is specified with a statechart diagram. The informal specification of the eventual system is thus gradually captured and made more and more precise throughout these diagrams.
3.2.2.1 Functional Requirements Capturing Safety Issues

The *functional requirements* of the system are depicted, together with their relationships in a use case diagram as presented in Figure 3.3. Each use case expresses a service that the system will eventually provide to a user. In Figure 3.3 we have a system with three use cases.

![Use case diagram](image)

*Figure 3.3: A use case diagram.*

To exemplify the integrated methodology we create the controlling software for an analysing device, here called the Analyser. The Analyser is restricted to move vertically and to analyse blanks/plates. The simplified operations form the protocol of the analysing device. This protocol is described below. A typical protocol is:

1) Move to middle position.
2) Receive plate.
3) Move to upper end position.
4) Analyse the plate.
5) Move to lower end position.
6) Deliver plate.
7) Repeat from Step 1.

Hence, the functional requirements of the Analyser are to analyse plates, to receive plates from and deliver plates to the Robot and to move the operating table to its three possible positions. A use case diagram of the Analyser is given in Figure 3.4.
The Robot and the Analyser interact by the Robot loading a blank to the Analyser receiving it, as well as by the Robot unloading a blank from the Analyser delivering this blank. The Robot is the active partner while the Analyser is the passive partner during this interaction.

Besides these functional services we also capture, at a reasonable level of detail, the reliability and safety issues related to the determined functional services. These issues are not captured in the diagram, but in the specification of the use cases, most commonly expressed as structured English text. Hence, at this level the vocabulary of the whole system is fixed, an overview statement is defined, and the goals, attributes and basic functions, as well as their possible deviations are all captured by careful systematic analysis. The Analyser in our example is seen as a small part of the whole system.

**Goals**

We consider the following goals for the operator and user (as in Figure 3.3) of the whole system in our case study.

1) Easy operation of the whole assembly by humans.
2) Fast and accurate control of the mechanical parts.
3) Fast and accurate analysis of the protocol results.
4) Automatic inventory of the processing operations (perhaps stored in a log).
5) Careful functional and safety analysis for simulating the protocol of the Analyser operation.

The first four goals are common to the whole project and refer to the eventual software. The fifth goal refers essentially to the process of producing the eventual software. We apply a combination of formal methods (the B Method and B-action systems), UML and safety analysis for checking
the quality of the eventual software and for producing this software, which we could then claim to be correct. This fifth goal is the most important to us.

**Basic Functions**
The basic functions for the eventual software of the healthcare case study are as follows:

1) Capture the protocol information when entered.
2) Communicate the protocol information with the mechanical parts.
3) Record the running protocol.
4) Inform about failures and the failure points.

These are the functions we have identified for the controlling software of the system. In this case study we are only interested in functions 1, 2 and 4.

We assume that the position detection of the operating table is absolutely reliable in the Analyser. The risks and dependencies of the system are listed during the development process.

**Use Cases Incorporating Possible Failures**
We have identified four use cases, move, receive, analyse as well as deliver for the Analyser as shown in Figure 3.4. We consider these use cases as subroutines, since it is more convenient with subroutines for the purpose of safety and reliability analyses.

**Subroutine Move** captures the moving of the operating table to the position pos.

**Typical course of events:**

1. Analyser receives the command to move to position pos from protocol.
2. System checks if pos is a valid position; if not, then MF1.
3. System checks if it is safe to move; if not then MF2.
4. System checks if current_pos does not equal pos and further moves are allowed; if not then MF3.
5. Analyser moves to position pos.
6. System checks if current_pos = pos; if not then MF4.
7. Signal move_ok.
Failure reports:
- MF1: Input parameter pos is outside the valid range.
  - Remedy: User changes parameter pos, identifies the cause and resumes or aborts calling protocol execution.
- MF2: Robot is not safe.
  - Remedy: Wait a while for the Robot to become safe.
  - Remedy: Check and change protocol or abort.
- MF4: Analyser has not reached position pos.
  - Remedy: Calibrate and resume or abort calling protocol execution.

Subroutine Receive captures the receiving of a plate from the Robot, provided the operating table is at its middle position and there is no plate in the Analyser.

Typical course of events:
1. Analyser receives command receive from protocol.
2. System checks if Robot has a plate available; if not then RF1.
3. System checks if current_pos = middle position to Receive; if not RF2.
4. System checks if the Analyser is empty; if not then RF3.
5. Robot loads the Analyser.
6. System checks if plate has been received; if not then RF4.
7. Signal receive_ok.

Failure reports:
- RF1: Robot is not ready to load the Analyser.
  - Remedy: Wait a while for the robot to become ready.
- RF2: Current_pos is not the middle position to Receive.
  - Remedy: User identifies the cause and moves the Analyser manually.
- RF3: Plate is present.
  - Remedy: User identifies the cause, and unloads the Analyser manually or aborts.
• RF4. The Analyser has not been loaded.
  – Remedy: User identifies the cause, loads the Analyser manually and resumes or aborts.

Subroutine Analyse captures the analysing of a plate, provided the operating table is at its upper end position and there is a plate in the Analyser.

Typical course of events:
1. Analyser receives command analyse from protocol.
2. System checks if a plate is present in the Analyser; if not then AF1.
3. System checks if current_pos = upper end position to Analyse; if not then AF2.
4. System reads the amount of liquid on the plate; if insufficient liquid to perform the analysis then AF3.
5. Analyser performs the analysis if still allowed to analyse; if not then AF4.
6. System reads the amount of liquid on the plate; if the analysis was not performed satisfactorily then AF5.
7. Signal analysis_ok.

Failure reports:
• AF1: No plate is present in the Analyser.
  – Remedy: User loads the Analyser manually and resumes or aborts.

• AF2: Current_pos is not the upper end position to Analyse.
  – Remedy: User moves the Analyser manually to upper end position and resumes or aborts.

• AF3: There is insufficient liquid on the plate.
  – Remedy: User determines the cause of the situation and resumes or aborts the calling protocol. In case of resuming: user manually loads liquid and resumes protocol.

• AF4: Erroneous analysis command. Protocol failure.
  – Remedy: Check and change protocol or abort.

• AF5: Failure in analysis, the analysis did not use the right amount of liquid.
  – Remedy: User initiates a maintenance procedure to fix the pump precision. After maintenance user resumes or aborts protocol.
**Subroutine Deliver** captures the delivering of a plate to the Robot, provided the operating table is at its lower end position and there is a plate in the Analyser.

**Typical course of events:**
1. Analyser receives command *deliver* from protocol.
2. System checks if Robot is ready for plate; if not then DF1.
3. System checks if current_pos = lower end position to Deliver; if not then DF2.
4. System checks if a plate is present; if not then DF3.
5. Robot unloads the Analyser.
6. System checks if plate has been delivered; if not then DF4.
7. Signal delivery_ok.

**Failure reports:**
- **DF1:** Robot is not ready to unload the Analyser.
  - Remedy: Wait a while for the Robot to become ready.
- **DF2:** Current_pos is not the lower end position to Deliver.
  - Remedy: User identifies the cause and moves the Analyser manually.
- **DF3:** Plate is not present.
  - Remedy: User identifies the cause, and loads the Analyser manually or aborts
- **DF4:** The Analyser has not been unloaded.
  - Remedy: User identifies the cause, unloads the Analyser manually and resumes or aborts.

**Failures:**
From the alternative courses of events above we can derive the following failures:
- a) Analyser move failure
- b) No plate present
- c) Plate already present
- d) Analysis failure
- e) Robot rotate failure
f) Robot arms move failure

g) Robot arms gripper failure

h) Protocol failure

Upon having this reasonably grained use case based specification, the next step is to determine the logically related use cases, and group them together. Each group of related use cases defines (informally still) a control system component.

### 3.2.2.2 Component-Oriented Development

Component-oriented development consists of reusing existing components, developing new components, and assembling new systems from them. A component is usually defined as a black box implementing a set of services, provided that the required set of context dependencies are available. A component is thus an abstraction over the space of services of a system. The services that a system has to perform, captured by the use case diagram, are partitioned into groups, each group of services being eventually implemented by a component. Components interact with each other by using the provided services. Thus, an important issue regarding a component-based system is the interactions among the participating components as shown in Figure 3.5.

![Component-Based System Diagram](image)

*Figure 3.5: Typical component-based system.*

By defining a set of components of the final system we distribute the computation already in this early development stage. The strategy of separating computation and communication from the beginning provides good support for reuse and scalability. A subset of the eventual system components may already exist and can be reused. Each of the unimplemented components is a self-contained system of finer granularity than the initial system. Therefore, the process of specifying a system is reduced to specifying each defined, but not yet implemented component, encompassing the communication means with other components. The resulting specification models the necessary conceptual model of the component required for providing the component services.
The component diagram is deduced from the use case diagram, and the component interactions are deduced from the use case relationships. Each use case can be mapped to a component service. While the use case diagram shows only the required services and their relationships, the component diagram distributes the services to a set of components that will implement them. The conceptual model of each component, required for providing these services, is typically specified in a precise manner, using a certain specification language. We use the B-action systems to express the component specifications. Each component is modelled as one or more machines within the B Method. The services of the component are operations in the machine.

In our case study the functional requirements of the Analyser are to receive plates which are being analysed and then delivered to an external device, the Robot. From the use case diagram in Figure 3.4 we can construct an associated component diagram as given in Figure 3.6. We are only interested in the Robot loading and unloading the Analyser. The remaining services of the Robot are only given as ‘other services’.

![Figure 3.6: A component diagram for the analysing device.](image)

### 3.2.2.3 Class Diagrams

At this level, the conceptual model of the component is shown as active classes, *i.e.*, classes whose instances have autonomous behaviour. For each class we give the attributes and their types as well as the methods. Hence, the classes model the static behaviour of the component. In the most abstract class diagram we merely consider the attributes modelling the state and the command of the component. The methods consist of the main functionality of the system and the abstract representation of the classes of errors as well as the possibility to fix these errors.

An abstract class diagram for the Analyser is given in Figure 3.7. The methods of the Analyser are the services *move*, *receive*, *analyse* and *deliver* in the use case diagram together with their possible errors and fixes. The class *ANALYSER* is the main class representing the actions of the Analyser component. The classes, *A_PROC* and *R_PROC*, represent encapsulations of the interface variables for the analyser component and the robot component. They contain procedures that can be called within the guards and actions of the analyser to set and determine the state of
the interfaces between it and the robot component. The class, \textit{GLOBALVAR_PLATE} performs a similar role to encapsulate the state concerning the presence of a plate to be analysed. The role of these classes is elaborated later in this document when discussing the corresponding B components. \textit{DEF0} defines the set of commands (\textit{ACOMMAND}) in a way that can be used by several classes/components.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{classDiagram.png}
\caption{An abstract class diagram of the Analyser.}
\end{figure}
3.2.2.4 Statechart Diagrams

The autonomous behaviour of each class is specified using statechart diagrams. At least three such statechart diagrams are developed.

![Statechart Diagram](image)

Figure 3.8: A basic statechart diagram for a component.

The first primitive statechart diagram is shown in Figure 3.8. The services in this statechart diagram are derived from the services in the use case diagrams. Here we merely model state transitions and events causing these transitions. We go from state *Idle* to a service *n* or to state *Suspension*. After service *n* is performed we return to state *Idle* or go to state *Suspension*. From state *Suspension* we may return to service *n* or to state *Idle*, or go to the state *Abort*.

The value of state is analogous to an additional attribute of the class whose behaviour is being described.
Figure 3.9: Basic statechart diagram for the Analyser.

The basic statechart diagram of the Analyser in
Figure 3.9 can be derived from the use case diagram in Figure 3.4. The services move, receive, analyse and deliver in the use case diagram form the states in this basic statechart diagram together with the states idle, suspended and abort. The methods of the class diagram are given as the transitions between these states. This basic statechart diagram and the class diagram in Figure 3.7 are translated to B with the U2B-tool [Snook00].

3.2.2.5 Creating a B Model from UML

In order to be able to ensure the correctness of the system we need a formal analysis tool. A formal method that comes with such tools is the B Method. In order to be able to reason about distributed systems within the B Method we use B-action systems [Walden98a]. Event B also supports the development of distributed systems and indeed the extensions of event B are built on action systems. However, some features for action systems like procedure handling are not supported in event B yet. Procedures are important in distributed systems, since they provide more structuring to the systems, as well as a general communication mechanism between interacting systems. Hence, we apply action systems within B to model the distributed systems within the healthcare case study.

MACHINE Component
VARIABLES
state, cmd
INVARIANT
state \in \{Idle,Service1,Service2,Service3,Suspension,Abort\} \land
cmd \in \{serv1,serv2,serv3\}
INITIALISATION
state := Idle \lor cmd \in \{serv1,serv2,serv3\}
OPERATIONS
New_command(ss) =
  \text{PRE} \ ss \in \{serv1,serv2,serv3\} \ \text{THEN}
  \text{SELECT} \ state = Idle \ \text{THEN} \ cmd := ss \ \text{END}
END;
Service1 =
  \text{SELECT} \ cmd = serv1 \land state = Idle \ \text{THEN} \ state :=
Service1 END;
Service1_fail =
    SELECT cmd = serv1 ∧ state = Idle THEN state := Suspension END;
Service1_ok =
    SELECT state = Service1 THEN state := Idle END;
Service1_notok =
    SELECT state = Service1 THEN state := Suspension END;
... Service_notready =
    SELECT state = Idle THEN state := Suspension END;
Remedy =
    SELECT state = Suspension
    THEN state := \{Idle, Service1, Service2, Service3\} END;
Failure =
    SELECT state = Suspension THEN state := Abort END

Figure 3.10: An abstract B-action system.

The first step in our formal development is to create an abstract B-action system from the basic statechart diagram in Figure 3.8. The tool U2B [Snook00] supports this translation. The event triggering state transitions (names assigned to arrows) correspond to operation names in the abstract machine specification in Figure 3.10. Each service of the basic statechart diagram will have four operations: the first for checking its precondition, the second for taking care of a false precondition, the third for executing the proper result of an operation and the fourth for taking care of a failed computation. We also have an operation Remedy for fixing an exception and continuing the computation, and an operation Failure taking the system to the state Abort from where no further execution is possible. The attribute cmd corresponding to ServiceN in the statechart diagram as well as the attribute state form the variables of the abstract machine Component.

As an example of the U2B translation we show the part of specification of the Analyser in Figure 3.11 (the complete machine ANALYSER is presented in Appendix ).

MACHINE ANALYSER
SEES DEF0
INCLUDES
    GLOBALVAR_PLATE,
    R_PROC,
    A_PROC
SETS
   ASTATE = {aidle, amove, aanalyse, areceive, adeliver, aabort, asuspended}

VARIABLES
   astate,
   acmd

INVARIANT
   astate ∈ ASTATE ∧
   acmd ∈ ACOMMAND

INITIALISATION
   astate := idle ||
   acmd := receive

OPERATIONS
   NewCommand (cc) =
      PRE
      cc ∈ ACOMMAND
      THEN
      SELECT astate=aidle THEN acmd:=cc END
      END;

   MoveA (pos) =
      PRE
      pos ∈ NAT
      THEN
      SELECT astate=aidle ∧ acmd=move
      THEN astate:=amove || Safe
      END
      END;

   MoveAFail =
      BEGIN
      SELECT astate=aidle ∧ acmd=move
      THEN astate:=asuspended || NotSafe
      END
      END;
The operations in a B-action system consist of guards and actions as in event based B. We can also have global variables that can be read and updated by more than one system. The global variable is declared in a separate machine `Global_z` which is included in the communicating systems `A_comp` and `B_comp`. By including the machine `Global_z` in `A_comp` and `B_comp`, they are both allowed to assign a value `v` to the variable `z` via the operation `assign_z(v)` [Butler96].

Previously developed machines that have been shown correct can be used in the development by including them in the component using `SEES`, `INCLUDES` or `EXTENDS` [Abrial96]. The procedures declared in a B-action system can be local or global. The local procedures are declared and referenced locally within the same system. The global procedures, on the other hand, may be referenced by other B-action systems as well. The global procedures are services provided/claimed between systems (components) and can be derived from the component diagram. Let us consider the exported procedure `P` declared in system `B_comp` and called from system `A_comp` as shown below.
We note that the B method requires the calling operation and the procedure to be in separate machines. Hence, in case an operation in \texttt{B\_comp} also calls the procedure \texttt{P}, we have to outsource \texttt{P} into a separate machine and incorporate them via the \texttt{INCLUDES} clause. When operation \texttt{A1} calls procedure \texttt{P}, \texttt{A1} is enabled only if procedure \texttt{P} is also enabled, \textit{i.e.}, \( Q_1 \land Q \) holds. The procedure \texttt{P} and the operation \texttt{A1} are executed as a single atomic entity. The procedures are discussed in more detail elsewhere [Sere00, Walden98b].

**Formal Specification of the Analyser**

From the class and the statechart diagrams in Figure 3.7 and Figure 3.9. We can get the abstract B-action system \texttt{ANALYSER.mch} automatically with the tool \texttt{U2B}. The Analyser has four services/subroutines \textit{move}, \textit{receive}, \textit{analyse} and \textit{deliver}. The only variables in this first machine are the variables modelling the states \texttt{astate} and the current command \texttt{acmd}, as well as a global variable \texttt{plate} stating whether there is a plate in the Analyser or not. The global variable is given in the machine \texttt{GLOBALVAR\_PLATE.mch} in Appendix .

The global procedures of the Analyser are given in the machine \texttt{A\_PROC.mch} in Appendix . The global procedures of the Robot are the imported by the Analyser. The Analyser has the global procedures \texttt{ReadyToDeliver}, stating if the Analyser is ready to deliver a plate or not, \texttt{ReadyToReceive}, stating if the Analyser is ready to receive a plate or not, as well as \texttt{BlankInAnalyser} stating whether the Analyser senses a plate or not. In order for the Analyser to be able to function correctly it checks via the imported procedures whether the Robot is safe or not, \texttt{Safe}, whether the Robot has a blank available or not, \texttt{Available\_blank}, and whether the Robot is ready to receive a blank or not, \texttt{ReadyFor\_blank}. The global procedures are only modelled as \textit{skip} statements here, but they will be refined later in the development.
The Robot has to be in a safe state when the Analyser is moving and the plate has to be in the Analyser during an analysis. During the delivery operation the Robot should also be ready to receive the plate that should exist in the Analyser, while the receive operation requires the Robot to have a plate available for the empty Analyser. A new command can be given each time the Analyser is in state \textit{idle}.

3.3 Computation Models

3.3.1 Modelling a Byte Code Verifier

In this section, we describe the methodology used to develop models of the type verifier and structural verifier. It is important to note that a formal specification cannot be constructed out of nothing; the goals of the verifier need to be established, before it can be modelled. Hence, the development of an informal specification is an important mandatory step that simplifies the model construction.

This section focus on the methodology used to translate informal specifications into formal ones. We distinguish between two cases: the type verifier and the structural verifier, because they require two different modelling techniques.

3.3.1.1 Modelling a Byte Code Instruction

Despite the modelling of the two nested loops, the one over the methods of the package being verified and the one over the instructions contained within a particular method, the heart of a type verifier is the modelling of each byte code instruction. Note that the Java Card language contains 184 different byte codes. In this section, we present the methodology that we have chosen in order to construct the formal specification of each byte code instructions. We can identify three steps to model a byte code instruction as shown in Figure 3.12. First, pertinent information is extracted from the requirements laid down by Sun. Then, information is gathered into an informal document describing the typing behaviour of each byte code instruction. Finally, this informal description into a formal specification.

```
Requirements
  Extract information

Informal Description
  Produce model

Formal Specification
```
**Extracting Information from the Requirements**

The first step in modelling an instruction is to understand the Java Card language. [Sun00] contains a description of the behaviour of each instruction of the Java Card Virtual Machine (JCVM). The main problem with this informal specification is that typing and operational information is mixed. The goal of the first step is to determine the principles of the Java Card and moreover the language itself. This comprehension helps to distinguish between static and dynamic semantics.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaload</td>
<td>Load reference from array</td>
</tr>
</tbody>
</table>

**Stack**

,..., arrayref, index ,
,..., value

**Description**

The arrayref must be of type reference and must refer to an array whose components are of type reference. The index must be of type short. Other arrayref and index are popped from the operand stack. The reference value in the component of the array at index is retrieved and pushed onto the top of the operand stack.

**Runtime Exceptions**

If arrayref is null, aaload throws a NullPointerException. Otherwise, if index is not within the bounds of the array referenced by arrayref, the aaload instruction throws an ArrayIndexOutOfBoundsException.

The example provided in Figure 3.13 gives the semantics of a JCVM instruction called aaload. In the Description paragraph, the two first few sentences describe type verification and the two last sentences describe the dynamic JCVM execution. The Stack paragraph describes the stack evolution. The Exceptions paragraph is not really useful, one just has to know that this byte code can throw exceptions. It is important to note that in the stack paragraph, Sun provides values whereas we want, for the type verifier, the type of elements in the stack. So there needs to be some interpretation of the specification.

**Gathering Information into an Informal Description Document**

From the specification provided by Sun, given in Figure 3.13, the person who writes the informal specification of the type verifier needs to extract information and to format it in a suitable way. To do so, we have proposed a format containing 5 different parts:

- The first part describes the stack from a typing point of view. In fact, it corresponds to what is expected by the instruction to execute itself and how the stack is transformed. In Figure 3.14,
it says that there are at least two elements in the stack. If the stack contains a pointer to an array of objects and a short, then the instruction returns the contents of the array at the index specified by the short. Otherwise, if the stack contains a null pointer and a short, the instruction returns a null pointer.

- The second part of the specification concerns the pre-modification tests. In fact, it consists in giving the tests that have to be verified before executing the instruction. In the example in Figure 3.14, these tests check that there are at least two elements in the stack and that these two elements have a correct type, according to the static semantics.

- The third part describes the modifications performed by the instruction. The modifications concern the evolution of the stack or of the local variables. In the example provided in Figure 3.14, the two elements at the top of the stack are removed and an element is pushed onto the stack. The type of this latter element is function of the type of the second element in the stack before the modification. Note that in this case, local variables are not modified but others instructions modify them.

- The fourth part contains tests that allow one to ensure properties of the modification performed by the instruction. In the example, there is are post-modification tests. Finally, the last part indicates the categories of exceptions that can be thrown by the instruction. It is necessary to know if the instruction can throw an exception or not for the modelling.

```
aaload
[... , refarray class , short ] => [... , ref class ]
[... , null , short ] => [... , null ]

Pre-modification tests:
1. The stack must contain at least two elements
   \( (1) \)
2. The two topmost elements of the stack have to be of types compatibles with refarray class and short.
   \( (2) \)

Modifications:
\( (3) \)
The two topmost elements of the stack are removed.
If the second element was a refarray type, then a reference of the same class is pushed onto the stack. Otherwise a type null is pushed.

Post-modification tests:
None

Throws
- NullPointerException
- ArrayOutOfBoundException
```
Each instruction of the byte code is described following this template in an internal document. This document represents the requirements that help to define the type verifier. This internal document is then the only reference used by the formal developer to translate an informal description into a formal specification of the type verifier. Note that the writer of this informal description is not the same person as the formal developer. The main reason is that in this way, the formal developer, when modelling the type verifier, asks the writer of the informal description for more information. Thus, the formal developer performs a review of the informal description. It enables ambiguities to be removed from within the informal document and the production of a clearer informal specification and a more precise formal specification.

**Translating Informal Description into a Formal Specification**

The last step in the modelling of the type verifier is the translation from the informal description to a formal B specification. This stage is critical, as the final code and the proof are constructed from this formal specification. Therefore, if there is a mistake in the translation, the proof will not be able to detect it. Moreover, the proof may be useless since the model will not reflect the correct behaviour of a type verifier.
Figure 3.15: Steps for building a formal model.

```
bb ← verify_aaload =
BEGIN
  SELECT
  2 ≤ size(stack) \land
  last(stack) = c_short \land
  last(front(stack)) = c_refarray \land
  (pc ∈ dom(exception_handler))
  \Rightarrow
  \forall label. (label ∈ exception_handler(pc)
  \Rightarrow COMPATIBLE(loc_var, loc_var_descriptor(label))) \land
  c_uref \notin ran(loc_var))
THEN
  bb := TRUE \land
  stack := front(front(stack)) ← c_ref
```

WHEN

2 ≤ size(stack) ∧

(1)

last(stack) = c_short ∧

last(front(stack)) = c_null ∧ (pc ∈ dom(exception_handler))

(2)

(4) ⇒

∀label.(label ∈ exception_handler(pc)

⇒ COMPATIBLE(loc_var, loc_var_descriptor(label))) ∧

ran(loc_var))

THEN

bb := TRUE ||

stack := front(front(stack)) ← c_null

(3) v

ELSE

bb := FALSE

END

Figure 3.16: Formal specification for aaload instruction verification.

Figure 3.16 concludes the example of the aaload instruction that we have chosen. In fact, we use this example to study the correspondence between the informal description and its B abstraction. We use italic bold numbers on the figures to show the correspondence between sentences of Figure 3.14 and the B specification in Figure 3.16. Hence, (1) indicates that there must be at least two elements on the top of the stack. (2) shows that the element on top of the stack must be a short and that the second element is either a reference to an array (c_refarray) or a null pointer.

As shown in Figure 3.14, we have to distinguish these two cases because the aaload byte code instruction has two potential treatments, the SELECT clause is the most suitable. This clause allows us to specify an action guarded by a condition. In this example and all the other instructions, all the SELECT clauses are deterministic and the global specification is deterministic. The operation returns a result, a value is given to the variable containing the result in each branch and the stack is modified in the two correct branches (two elements are popped and one is pushed).

In the SELECT and WHEN clauses, the developer indicates the pre-conditions that allow the execution of the instruction. In our example, the pre-conditions state that there must be at least two elements in the stack and that the top element is a short and the second element is either a reference to an array of objects or a null pointer. We also add the test for the exception handler (4). In fact it is an additional condition which checks that if the current program counter pc is in an exception handler, we have to check that for the proof label given for each handler: the local
variable is compatible with the local variable descriptor associated with the label, and that there are no uninitialised references in the local variables. This test is factored out in the internal document describing the requirements for the type verifier.

Finally, in the **THEN** and **ELSE** clauses, we indicate the behaviour of the instruction. In both cases, when dealing with non-exception behaviour, the return variable is set to **TRUE** and the stack is modified consequently to the test. Two elements are popped from the stack and a reference is pushed onto the stack. In the case of exceptional behaviour, which is caused by there being insufficient elements on the stack, the typing of the elements is incorrect or the exception handler checks are incorrect. Here, the instruction returns **FALSE** and does not modify the stack or the local variables.

If an exception is raised, it is propagated within the program and the verification process is stopped. Otherwise, the verification process proceeds with the next instruction.

3.3.1.2 Modelling a Structural Test

If the type verifier appears to be a single, complex machine that ensures the enforcement of the typing rules, the structural verifier appears more as a collection of tests that have to be checked in order to accept data. In this section, we describe how we construct these tests by providing the methodology used, the templates and some examples. The structural verifier ensures that data received by the card is in the correct format, *i.e.*, the format defined by Sun in [Sun00]. To do so, we have to model each component of a CAP file by representing their properties and checking that data which is supposed to correspond to a component does indeed do so.

As for the type verifier, we have developed a methodology to model each component. We describe this methodology in this section and discuss the obtained result.

**Extracting Information from Requirements**

In [Sun00], Sun informally specifies the CAP file format. As the goal of a structural verifier is to ensure that a given set of data is well formed, this informal specification is the basis. However, information strictly related to structural verification has to be extracted from this informal specification. In fact, Sun’s documentation not only includes an explanation of what elements can be found in a specific component, but also information not related to it. Figure 3.17 provides an example of Sun’s specification. It concerns the definition of the **Export** component, one of the eleven components of the CAP file defined by Sun. In this description, Sun provides a description in a C-like notation for the structure of the component and then a description in a natural language for each of the elements. In particular, the C-like notation includes the size of each element given with **u1** or **u2** indicating whether the element is a byte or word, respectively. In this definition of a component, the size of each element is needed for the structural verification.

The export component is represented by the following structure:

```plaintext
export_component {

```
The tag item has the value COMPONENT_Export (10).

The size item indicates the number of bytes in the export_component structure, excluding the tag and size items. The value of the size item must be greater than zero.

The class_count item represents the number of entries in the class_exports table. The value of the class_count item must be greater than zero.

The class_exports item represents a variable-length table of class_export_info structures. If this package is a library package, the table contains an entry for each of the public classes and public interfaces defined in this package. If this package is an applet package, the table contains an entry for each of the public shareable interfaces defined in this package. An index into the table to a particular class or interface is equal to the token value of that class or interface (§4.3.7.2). The token value is published in the Export file (§5.7) of this package.

The items in the class_export_info structure are:

The class_offset item represents a byte offset into the info item of the Class Component (§6.8). If this package defines a library package, the item at that offset must be either an interface_info or a class_info structure. The interface_info or class_info structure at that offset must represent the exported class or interface. If this package defines an applet package, the item at the class_offset in the info item of the Class Component must be an interface_info structure. The interface_info structure at that offset must represent the exported, shareable interface. In particular, the ACC_SHAREABLE flag of the interface_info structure must be equal to 1.

The static_field_count item represents the number of elements in the static_field_offsets array. This value indicates the number of public and protected static fields defined in this class, excluding final static fields of primitive types. If the class_offset item represents an offset to an interface_info structure, the value of the static_field_count item must be zero.

The static_method_count item represents the number of elements in the static_method_offsets array. This value indicates the number of public and protected static methods and constructors defined in this class. If the class_offset item represents an offset to an interface_info structure, the value of the static_method_count item must be zero.
The static_field_offsets item represents an array of 2-byte offsets into the static field image defined by the Static Field Component (§6.10). Each offset must be to the beginning of the representation of the exported static field. An index into the static_field_offsets array must be equal to the token value of the field represented by that entry. The token value is published in the Export file (§5.9) of this package.

The static_method_offsets item represents a table of 2-byte offsets into the info item of the Method Component (§6.9). Each offset must be to the beginning of a method_info structure. The method_info structure must represent the exported static method or constructor. An index into the static_method_offsets array must be equal to the token value of the method represented by that entry.

In this description, there is also information about other components. For example, in Figure 3.17, the class_offset element represents a memory offset into the Class component. Therefore, interactions between components are possible. The remaining information that appears in this description concerns information not related with verification.

This description contains most of the information necessary to perform internal and external structural tests. However, as we would like to differentiate between internal and external tests as in the architecture, we have decided to distinguish them. Therefore, we have to write a specific internal document, as for the type verifier, containing the different tests by component.

### Gathering Information into an Informal Description Document

From the description provided in Figure 3.17, the person writing the informal requirements of the structural verifier can extract tests related to structural verification. This document is written in natural language but tries to classify the tests into two categories per component: the internal tests containing the format of the component, and the external tests checking information shared by several components.

Figure 3.18 and Figure 3.19 provide examples of tests extracted from Sun’s specification. For example, to check that the Export component is well formed, one has to verify that the tag is equal to 10, that the table static_field_offsets has static_field_count entries and the table static_method_offsets has static_method_count entries.

- Check that Tag = 10
- Check that the number of entries in the static_field_offsets is equal to static_field_count.
- Check that the number of entries in the static_method_offsets is equal to static_method_count.

Figure 3.18. Informal specification of internal tests related to Export component.

- Check that the value of component_size[tag-1] of the Directory component is equal to the size item: size = component_size[tag-1].

Figure 3.19. Informal specification of external tests related to Export component.
• Check that the ACC_Export flag of the flags item in the Header component is equal to 1.

**Figure 3.19:** Informal specification of external tests related to Export component.

For all the components specified by Sun, and an additional one for the proof component for the PCC (Proof Carrying Code) technique, there is a correspondence in the internal document that we have written. In fact, the internal document specifying the informal requirements for the structural verifier is organised as depicted in Figure 3.18 and Figure 3.19. The document produced can now be used by the developer of the structural verifier.

**Modelling a Component**

The first step in the structural verifier is to provide a B specification of each CAP file component. This first B abstraction, describes what information needs to be by each component. As the CAP file is composed of eleven standard components, we aim to model each component in a similar way in order to split the development between different developers. Therefore, we propose the template in Figure 3.20. This template represents the abstract B machine. The goal of this template is to simplify the B development. It also includes a naming convention for the machines, the variables and the operations of the machine. It is important as the proof of each component can be made independently; changing the name of a variable or an operation generally implies that the proof must be restarted, almost from the beginning.

In the VARIABLES clause of the specification, one adds the variables needed to describe the components. In the case of the Export component, this clause includes `exp_class_count`, `exp_class_offsets` etc. that correspond to elements of the corresponding informal specification (Figure 3.17). Moreover, we define a specific variable named `component_verified`, i.e., `exp_verified` in the case of the Export component, that indicates whether this component has been successfully verified or not. At the beginning, this variable is set to FALSE.

In the INVARIANT clause of the specification, one indicates the properties of the variables defined in the VARIABLES clause. It contains the typing of each variable and the properties they must respect. For example, in the case of the Export component, one property is that the `exp_class_count` is a byte (`exp_class_count ∈ t_byte`) and that its value is strictly positive (`exp_class_count > 0`). In the INVARIANT, the variable indicating that the verification of the component has been performed is typed as a boolean. This clause also includes the property that is enforced when the component is verified.

In the INITIALISATION clause, variables are initialised for the first time. Note that `component_verified` is set to FALSE, indicating that, at the beginning, the component is not verified.

```plaintext
MACHINE cpn_component
VARIABLES
<Set of variables used to describe the component>,
```
Finally, the last clause is that of **OPERATIONS**. This clause contains all the services of the machine. We distinguish two kinds of services; first an operation is declared called `component_internal_verif`. This operation performs the internal verification of the component. In order to be called only once, this operation is pre-conditioned by the `component_verified`. That is, if this variable is **FALSE**, then the operation can be executed otherwise, it has already been executed (and the verification is a success) and there is no need to execute it again. This particular operation contains the properties which ensure the correctness of the component. The refinement of this operation is performed by implementing the list of internal tests proposed in the internal document written using the Sun specification and of which some examples are provided in Figure 3.18. The second kind of service concerns operations allowing access to the content of the component. These operations are pre-conditioned with the same variables but they can be executed only if the component has been verified, i.e., `component_verified` is **TRUE**. These pre-conditions ensure that data is not accessed unless the component has been verified and that it then can be considered as a valid component, from a syntactic point of view.

This template provides a general framework for writing B specifications of each component, and makes communication between developers easier. The abstract machines obtained are then refined in order to map this abstraction with the real data made from zeros and ones. Moreover, this template helps formal developers to relatively easily translate the informal description as this formal specification is not low-level and represents the component. In fact, this B specification provides a view of each component that is used to model external tests and the type verifier.
Modelling an External Test

Modelling an external test is easier than modelling a CAP file component. The main reason is that we abstract each component to build higher level tests. In fact, external tests involve information, and so variables, that reference or represent the same value. Therefore, these tests rely on the CAP file model, on the variables defined in each of these model and their properties. So it is relatively straightforward to translate the informal specifications into the formal ones.

For example, the formal specification of the two tests provided in Figure 3.19 is depicted in Figure 3.21. The first test specifies that the size of the Export component is the same as the one stored in the dir_component_sizes table of the Directory component. Therefore, in B, one has just to write the test in a specific operation. Moreover, we have chosen to write one test per operation. Of course some tests can be factorised into a single operation, but allowing one test per operation makes the B machine clearer to read, write and easier to maintain. Each operation is pre-conditioned by tests. These tests, in our case exp_verified and dir_verified for the first precondition, ensure that both the Export and the Directory component have been verified before any external test can be made on them. The body of the operations, contain the properties that need to be checked. The operation returns the result of the test: TRUE if the test passes, FALSE otherwise.

```b
res ← exp_ext_1 =
  PRE
  exp_verified = TRUE ∧
  dir_verified = TRUE
  THEN
  res := bool(exp_size = dir_component_sizes(c_tag_export - 1))
  END;
res ← exp_ext_2 =
  PRE
  exp_verified = TRUE ∧
  hdr_verified = TRUE
  THEN
  res := bool(hdr_has_export = TRUE)
  END;
```

Figure 3.21: B specification of external tests for the Export component.

The second test modelled involves the Export and the Header components. The property checks that, an element indicating the presence of the Export component in the Header component is set to TRUE. Note that it is still a simple property to express as the developer of the external tests does not have to model the whole problem. In fact, most of work has been done whilst modelling the
component. The developer now has access to a formal description of each component and just has to express the relationships between the variables.

**Embedding External Tests into a B Component**

As we have just seen, the specification of external tests is simple. It is a translation of the informal specification into a formal specification. In order to simplify the B development and to have the same format for the B specification for each external test, Figure 3.22 gives a template for writing external tests. First, we consider that each component `cpn_component` has a machine `cpn_component_ext` that contains the external tests of this component on the other components.

```
MACHINE cpn_component_ext
SEES <All cpn_components concerned by the consistency of the component>
OPERATIONS
    Res ← test1 =
        PRE component_verified = TRUE ∧
            component1_verified = TRUE ∧
            ...
        THEN
        Res := bool( <Description of the property> )
    END

    Res ← test2 =
        PRE ...
        THEN ...
    END;

END
```

**Figure 3.22:** Template for modelling the external tests for the structural verifier.

The `SEES` clause of the B machine lists all the components involved in tests described within this machine. It contains the names of components interacting with the component being tested, and some context machines that provide definitions. There is no need to add any invariants since the proof relies on checking the consistency of component invariants and their properties.

There is no need to have an `INITIALISATION` clause either, since all variables are already initialised, as they belong to other machines.

So, the final B clause is that of the `OPERATIONS`. Each operation describes one or more tests. The pre-conditions statement ensures that internal verification of each component affected by the test has been performed. This allows the B prover to obtain more properties and hypotheses. The body of the operation contains the formal specification of the test, *i.e.*, the abstract representation
of what is performed by this test. In this representation, abstract variables are used to express the property of the specified test. That is, the description of the relationship between abstract variables of the different components affected by the test (as shown in Figure 3.21). The next step in the refinement process is the implementation. In the implementation, all abstract variables are replaced by concrete ones and access to data is performed by the different operations of each component.
4 Analysing Models

This chapter describes different ways in which formal methods may be analysed for validity and soundness.

4.1 Verifying Models in B

Evaluation

The evaluation of the eventual AMN model obtained at preliminary design should be carried out before interactive proof\(^{13}\). As one of its aims is to check that model is well adapted for proof. The AMN model must fulfil several properties:

- It must be complete, as far as possible. Proof reading and peer review (models, variables and events dictionary) would ensure it.

- It must be relatively simple. This can be checked by looking at the number of proof obligations discharged automatically.

- Operations of implementations should be neither too big nor too complex. This can be checked also by looking at the number of proof obligations generated by the operations of the implementations.

- All the static and dynamic safety properties have been completely defined abstractly before being implemented concretely.

- Enough intermediate enrichment, data, and algorithmic refinements have been introduced. It is not recommended

- to have simple abstract machines and either a single refinement or implementation that concentrates all relevant information and produces too many or too complex proof obligations. Whenever possible, this information should be distributed among several intermediate refinements, each concerned with a particular topic.

- On the opposite, it must be checked that no superfluous refinements have been introduced that increase either the complexity or the number of proof obligations. This can be checked by suppressing the suspected refinements and comparing the complexity and the number of generated proof obligations before and after the suppression.

- Invariants of enrichment and data refinements must include all the necessary properties linking concrete variables to abstract variables. There should not be enrichment and data refinements that generate abnormally few proof obligations considering the variables of the refined module and the variables of the refinement.

\(^{13}\) Automatic proof is required as this phase provides some metrics concerning the quality of the model (in term of provability).
**Validation**
In addition to conventional activities, validation of a B model consists in:
- Assessing the functional and non-functional requirements that have been taken into account in the AMN model.
- Checking that all the AMN modules have been successfully type checked.
- Checking that all the AMN modules generate proof obligations.
- Checking that all proof obligations generated by abstract machines and refinements have been discharged,
- Assessing the mathematical correctness of the specific proof rules added to discharge proof obligations.

**4.2 Expert Validation**
In section we describe the procedures used in the railway and smart card case studies for reviewing B models by domain experts. The review by domain experts verifies if the model describes the requirements and physical properties properly.

The reviewing by domain experts is an important task as many errors can be rapidly discovered. And discovering errors as soon as possible is one of the aims of formal methods. Therefore, reviewing the specification and source code is mandatory for good development. However, some precautions need to be taken in order to have a good review. For example, the review must be performed by a domain expert that has not participated neither in the development of the system nor in the production of informal specification in order to avoid any bias.

**4.2.1 Transportation Case Study**
In the railway case study the expert validation performed several verifications through all the phases of the system model development:
- review of the semiformal properties taking in account the informal documents;
- conformity analysis between B model and semiformal properties;
- proof based upon B model;
- safety analysis based upon semiformal document.
Figure 4.1 shows all the validations done by the domain experts. The two higher validation activities, the review of the properties, events and automatons document, and the conformity analysis, are detailed below.

4.2.1.1 Validation of the Properties, Events and Automatons Document

The entire system was modelled using Event-based B. This first modelisation will be proved, as far as possible since some predicates will not be provable: these non-provable predicates will lead to discover some mistakes and/or incompleteness in the system specification and/or in the modelisation.

However, the system specification must be complete and correct enough to write a first “nearly correct” model, because if not, we may have a “completely false” model with a large number of false proof obligations, and we will not be able to correct the model by correcting all mistakes successively.

This means that the system specification must be validated before the modelisation task (the proof of the system model cannot be considered as a sufficient validation of the system specification).
This document is reviewed and then approved by system experts, with two points of view: correctness of the documents regarding previous documents and safety of the system described in the properties, events and automatons document.

4.2.1.2 Completeness and Correctness of the System B Model

The system model must be strictly equivalent to the system properties, events and automatons. This means that all properties, events and automatons must be correctly described in the system model. Moreover, all pieces of information of the model must be equivalent of the connected properties, events and automatons. This task is performed by reviewing.

In the properties, events and automatons document, each property, event is associated to a unique identification number. Each automaton transition and state is identified with also with a unique identification number. This enables a rigorous traceability from the document to the B model.

The classification, with several level of abstraction, of the system properties, events and automatons cannot be validated in itself, since there is not only one solution, but an infinite number of « more or less relevant » solutions. However, the reviewer may give advice if it is obvious that some classification choices may cause difficulties in the modelisation task.

If the classification of properties, events and automatons is relevant, each level of abstraction is connected with one refinement of the system model. In that case, the rereading task is performed step by step, for each refinement (which allows detecting errors as soon as possible). The task is complete when the last refinement is revised.

4.2.2 Smart Card Case Study

The review of the smart card case study has been done by experts of the application field without any prior experience in reading B specification. They compared the informal rewriting of the specification we have done previously with the B specification. The domain experts understood rapidly the B syntax without any training. The experts wrote three reports on the flaws discovered at the most abstract level of the specification. At the time the reports were delivered some of those errors have been already discovered during the refinement and proof process. But some of the remaining undetected errors were relevant. In particular, one of those undetected errors has affected 12 instructions. It concerns an incorrect transformation of the stack by those instructions. This error was not due to the translation of informal description to the formal modal, but to the flawed informal specification written for the type verifier. So besides correcting the formal model, we also corrected the informal specification. Another error has been discovered during the review of the B model concerning a badly modelled instruction (checkcast instruction).
4.3 Safety Analysis

While developing safety-critical systems, it is necessary to satisfy not only functional requirements defining the set of tasks to be performed by the system, but also safety requirements describing which characteristics the system should possess in order to ensure safety.

4.3.1 Hazard Analysis

The safety requirements result from safety analysis. A paramount role in safety analysis plays hazard analysis. A hazard is a situation in which there is actual or potential danger to people or to the environment the system operates in [Leveson95, Storey96]. Hazard analysis allows the designers to identify the potential danger associated with a system.
Hazard analysis is a set of techniques, which provide the mechanism for identification of safety requirements. Each of techniques gives a different insight into the characteristics of the system under construction. One of the most widely used techniques is Fault Tree Analysis (FTA) [Leveson95, Storey96]. FTA is a deductive safety analysis technique. It is a top-down approach applied during the design stage. As an input FTA takes information about functions of the system and possible dangers associated with them. The result of the FTA is an identification of component faults that result in different hazardous situations. Each fault tree has a root representing a hazardous situation. The tree traces the system to the lower component level to reveal the possible causes of the hazard. Moreover, each fault tree provides a logical representation of a hazard in terms of faults of the components.

Below we illustrate how FTA allows us to formulate safety requirements, which the controlling software should satisfy. Consider the Robot Analyser described in Section 3.2.2. Preliminary hazard identification establishes that the main danger associated with the system is a damage of the equipment. This hazard forms the root of the fault tree given in Figure 4.2. Each of two events – “Collision of Robot with Table” and “Breaching boundaries of safe areas” might cause this hazard. These events are connected by disjunction, i.e., logical gate “or” with the root of the fault tree. Next we analyse a
logical connection the events, which in their turn lead to “Collision of Robot with Table”. The analysis continues until a
desired level of formulation of safety requirements is reached. For instance, safety requirements obtained using the fault
tree given in

Figure 4.2 are as follows:

- When the Robot is ready to load the Analyser, arm 1 may extend only if the table is in its middle position.
- When the Robot is ready to unload the Analyser, arm 2 may extend only if the operating table is in its lowest position.
- The table may move only when the respective Robot arm has retracted.
- The table must not move beyond its upper and lower end position
- The robot must not extend its arm beyond their limit positions.

### 4.3.2 Software Development

While designing software for safety-critical systems, it is necessary to consider the possible safety implications of the actual development, *i.e.*, to ensure that the suggested design does not introduce additional hazards. Moreover, we should ensure that controlling software reacts promptly on hazardous situations by trying to force the system back to a safe state. Clearly, this goal can be achieved only if the information provided by safety analysis is taken into account in the process of software development.

We propose an approach for conducting software development hand-in-hand with safety analysis as presented in Figure 4.3. Observe, that both the safety analysis and stepwise program development conduct reasoning from an abstract level to a concrete level. Safety analysis starts by identifying hazards that are potentially dangerous, and proceeds by producing detailed descriptions of them together with finding the means to cope with these hazards. The refinement process starts by producing an abstract specification describing what should be done, rather than how. Each refinement step allows us to incorporate more implementation details in the abstract specification, so that we end up with a program. Therefore, we can incorporate the information that becomes available at each stage of the safety process by performing corresponding refinements of the initial specification.
We, as other researchers [Leveson96], argue that only if safety and reliability attributes are considered from the early stages of the system development, can the required dependability of the system be achieved.

### 4.3.3 Dependability Impairment

A safety-critical system is typically a control system with a computer-based controller managing a plant. There are two main entities involved: an environment and a controller. While developing a specification for a control system, we model the behaviour of the overall system, that is, the physical environment and the controller together. This allows us to state explicitly the assumptions that we make about how the environment behaves. By specifying the overall system we can abstract away from means of interaction between the controller and the plant (via sensors and actuators) and, as a result, obtain a more succinct initial specification. In the later stages of the development process, when the desired level of detail has been achieved, we separate the controller and the plant, thus, obtaining the specification of the controller.

The plant behaviour evolves according to the physical processes involved and the control signals provided by the controller. The controller observes the behaviour of the plant and adjusts it to guarantee the intended plant functionality. While observing the plant behaviour the controller reacts to certain events requiring its intervention. Thus, this kind of system is a typical example of a reactive system.

The controller bases its decision on the information that it obtains from the sensors. Due to faults of the system components and imprecision in the sensors, the controller obtains an approximation of the equipment state rather than a real state. Failure occurrences deteriorate this approximation and potentially lead to dependability impairment by:

- loss of precision in the function execution, *i.e.*, a decreased reliability of the system,
- system failure as a result of violation of safety, complication in synchronizing the involved machines (which has implications for both safety and reliability).

Therefore, failure handling constitutes an important aspect of safety-critical system development. In order to build a controller able to withstand component failures the following actions are required:

- to understand the nature of faults and their impact on the overall system,
- to decide on the detection procedures, repair procedures, functioning of the system in presence of failures, and
- to introduce an effective maintenance procedure.

These actions are performed within safety analysis. To incorporate the results of safety analysis in the refinement process the following methodological aspects should be addressed:
• modelling fault occurrence;
• modelling fault detection;
• modelling system behaviour in the presence of faults;
• modelling different operating modes and transition between modes including the system failure; and
• the design and specification of a maintenance procedure.

Figure 4.3: Safety analysis and stepwise system development.

4.3.4 Formal Specification and Safety Analysis
In the healthcare case study the initial specification of the system (see Figure 3.8) is rather abstract – it merely models transition between states. However, already in the initial specification we reserve a possibility of fault occurrence and system failure. The actions Service1_fail and Service1_notok model failure of execution of command.

\[
\begin{align*}
\text{Service1_fail} & = \text{SELECT cmd = serv1} \land \text{state = Idle} \text{ THEN state := Suspension} \text{ END} \\
\text{Service1_notok} & = \text{SELECT state = Service1} \text{ THEN state := Suspension} \text{ END}
\end{align*}
\]

There is also possibility of spontaneous fault occurrence even while the service is not requested, as modelled by the action Service_notready:

\[
\begin{align*}
\text{Service_notready} & = \text{SELECT state = Idle} \text{ THEN state := Suspension} \text{ END}
\end{align*}
\]

In all the actions presented above, the system reacts on fault occurrence by entering the state Suspension. From that state the system tries to execute a recovery procedure and continue functioning as specified by the action Remedy:

\[
\begin{align*}
\text{Remedy} & = \text{SELECT state = Suspension} \text{ THEN state :\in \{Idle,Service1,Service2,Service3\}} \text{ END}
\end{align*}
\]

We model failure of the system by action Failure:

\[
\begin{align*}
\text{Failure} & = \text{SELECT state = Suspension} \text{ THEN state := Abort} \text{ END}
\end{align*}
\]

This action represents transition of the system in a fail-safe state. The failure of the system occurs if the fault tolerance limit has been reached and the system cannot carry out its functions anymore.

### 4.4 Proof

This section shows how to use Atelier-B to validate a B model; it describes strategies for automated proof, interactive proof, and rule management.

#### 4.4.1 An Automated Proof Strategy

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 proof. The disadvantage of performing proofs is the amount of effort it generally requires.

#### 4.4.1.1 The General Approach

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. A maxim is:

<table>
<thead>
<tr>
<th>Do not start interactive formal proof too soon, but</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Page 160 MATISSE/WP1-D10/April03
One good way\textsuperscript{14} to proceed is to mix automatic proof and visual inspection of the Proof Obligations (PO). The objective is to achieve a sensible balance between the confidence that no false PO remains, and the time required to achieve this confidence.

The general approach is follows the flowchart of Figure 4.4:

- Write abstract machine 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 seem to be true. If any are false, the abstract machine should be corrected.
- Write implementation component,
- Compare implementation versus abstract machine,
- Start automatic proof on this implementation,
- If some proof obligations are not automatically demonstrated, check them quickly by reading them and verify that they seem to be true. If some of them are false, the abstract machine or the implementation should be corrected.
- Make the formal demonstration of the remaining PO in the abstract machine and in the implementation, using interactive prover.

\textbf{4.4.1.2 Automatic Proof Rate as a Quality Indicator}

Automatic proof rate is a good indicator of the suitability of a B model. Most of the time, if the proof rate is too low, the B model needs to be reshaped. From our experience, large-scale projects have an automatic proof rate between 70\% to 95\%. Locally, some components may have a low rate (a small percentage automatically proven), depending on their complexity and the operations that are used to construct the B model.

\textit{Reducing model complexity}

A model may be too complex, because:

- Too many details are included in the top-level specification components; this may be overcome by:
  - identifying and using only those variables that need to be present at this level; and

\textsuperscript{14} It has proved to be efficient on many industrial-size projects.
– introducing details in the lower components.

• Too heavy use has been made of sequencing; this may be overcome by:
  – decomposing operations (the LOCAL_OPERATIONS clause may be helpful).
Figure 4.4: Automated proof strategy.
**Constructing models that can be easily manipulated by the automated prover**

Some operations may confuse the automatic prover and lead the user to perform interactive proofs. These operations are:

\[ \text{closure, union, inter.} \]

There are two main reasons for this:

- These operations are difficult to manipulate in a general way. Many proofs need to add very particular rules, that are likely to be difficult to establish and demonstrate, which have little chance to be reused for other projects.
- The prover rules database has been built mainly from railway developments which have not used these operations. So there are none or few rules regarding the manipulation of these operations.

There are three general techniques for dealing with this problem.

- Use quantified predicates instead of these operations.
- Use specifically created rewriting rules to rewrite these operations into more manageable operations, expressions or predicates (if applicable).

For other operations, particular rules should be added for arithmetic operators as general sum, general product, and modulo.

In Atelier-B 3.6, a new package of formally validated rules will be available, providing axiom-like rules related to these symbols arithmetic operators that were badly manipulated by older versions of the prover.

Finally, if the extended prover rules database still does not contain required rules, to simplify expressions or predicates containing these operations, they can be manually added.

### 4.4.1.3 Playing with Forces

The automatic prover is composed of:

- a **rule database**, containing about 2800 mathematical rules. These rules are grouped in packets called *theories* and can each *theory* be activated independently from the others.
- a **proof engine**, whose rôle is to:
  - process hypotheses (simplification, stack loading),
  - simplify the goal, by
- applying heuristics. Those heuristics are general (deduction, skolem transformation, …) and don’t address very specific goal patterns.

- using some rule theories from the rule database. Rule theories are triggered by the proof engine if the rule contained in them match with the current goal.

The efficiency of the automatic prover can be parameterised by using different forces. Forces are indexed according to the complexity of the mechanisms involved in the proof process (see Table 3).

<table>
<thead>
<tr>
<th>Force</th>
<th>Proof time per lemma</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less than 10 seconds</td>
<td>70 %</td>
</tr>
<tr>
<td>1</td>
<td>From some seconds to 3 minutes</td>
<td>+ 1 %</td>
</tr>
<tr>
<td>2</td>
<td>From some minutes to tens of minutes</td>
<td>+ 3 %</td>
</tr>
<tr>
<td>3</td>
<td>From tens of minutes to hours</td>
<td>+ 1 %</td>
</tr>
<tr>
<td>Fast</td>
<td>Less than 3 seconds</td>
<td>30 %</td>
</tr>
</tbody>
</table>

Table 3: Efficiency of the automatic prover for different forces.

Force 0 is the most efficient force: its original specification is to have a maximum proof duration of 10 s. Hypotheses are processed differently from higher forces: each hypothesis is simplified, by using a dedicated part of the rule database only used for that goal, then loaded in the stack. The proof engine then try to apply rules from rule theories that can match with the current goal. In case of success, the proof engine starts a new iteration. In case of failure, some heuristics are tried (skolem transform, simplification of negated predicates, simplification of the last hypotheses loaded in the stack, …). In case of failure (nothing new has been obtained), the prover stops. Otherwise, the proof engine starts a new iteration.

Forces 1, 2 and 3 do not process hypotheses the same way: the proof engine tries to demonstrate each hypothesis. If it succeeds, the hypothesis is obviously redundant with hypotheses previously loaded and therefore, is deleted. In case of failure, the simplified hypothesis (the proof engine always try to simplify incoming goals) is loaded in the stack. That way, the stack is supposed to contain a minimal set of hypotheses. The drawback of this approach is the time required to process each hypothesis: about 1 minute for 100 hypotheses. The heuristics used by the proof engine are also different: some of them introduced in forces 1, 2 and 3 allow the proof engine to try to demonstrate missing information that would “probably” help proof (for example, a missing hypothesis that would enable to fire a rule related to the goal). This approach is obviously time-consuming and should never be used when trying to prove a B component for the first time.

4.4.2 Manual Proof

There is one major rule related to manual proof:

Inspect PO before performing any manual proof activity.
4.4.2.1 PO Inspection

PO inspection can be performed either with the PO viewer or the interactive prover.

Difficult POs should be evaluated first. So inspection should begin from the end of the list of all unproved POs, as POs are ordered by their degree of difficulty starting with the simplest.

PO inspection consists of five 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).

Quick demonstrations should be performed within a time limit (no more than 10 minutes for example). First try predicate prover. Do not use more than 5 commands. Try to generalise a demonstration to other PO.

If a false PO is found, the B model should be corrected before continuing. It should be verified that this PO really is false, as this may not be obvious: a good method is to exhibit a counter-example. Introducing this counter-example in the B model allows the source of errors to be localised.

![Diagram of proof obligations generation](image)

**Figure 4.5: Proof rate for the automatic prover.**

4.4.2.2 Proving Manually

Proving is part of the development process, just as much as the creation of models. Indeed, having written a model will help the user to prove and modify it. In a typical project, the automatic prover helps to locate errors in the B model, which represent about 10% of the total POs. Another reason is that model complexity is tightly coupled to proof complexity: two models may be semantically equivalent, but their formulations (in particular the operations that are used),
may mean that one is easier to prove. Efficiency in a B development is achieved through the
construction of easily (most of the time) proved models.

The average proof rate that has been measured is between 10 and 20 POs per day, including
debugging, for the lifetime of the project.

We can consider the following rules as good axioms:

- Use no more than 10 steps in an interactive demonstration. (There are exceptions; the
  maximum known is about 550 steps.)
- Add a user rule if the situation is not covered well by the prover rule database or proof
  mechanisms.
- If a predicate is demonstrated for more than one PO, add an assertion in the B model.

A new assertion should be added after all the other assertions. If this is done, the proof status is
not modified and demonstrations are not lost. Indeed, when a model is modified, the merger (part
of the PO generator) intelligently updates the proof status.

If previously, a PO was $H_1 \land H_2 \Rightarrow G$, the proof status is:

- kept if the new PO is $H_1 \land H_2 \land H_3 \Rightarrow G$ (because it can be logically deduced),
- initialised if the new PO is $H_1 \land H_3 \Rightarrow G$.

Similarly, when an invariant is corrected, by adding a variable or a property, the new predicate
should be added at the end of the clause. This limits the risk of losing interactive
demonstrations, since the proof tree has been modified and it is unlikely that the demonstration
will need to be replayed.

4.4.2.3 Example of Manual Proof

To illustrate the use of interactive prover we will describe the steps necessary to prove the POs of
the RailwayT machine presented in Figure 2.4 and Figure 2.5.

We start by presenting the *Occupy* operation, omitted in Figure 2.5 for simplicity. This operation
describes a train $t_1$ entering section $c_1$. The first LET expression declares the local variable
newocc that holds the set of sections currently occupied by trains. The variables occ, pos, and
atime are updated to reflect train $t_1$ entering a new section. The second LET declares two local
variables, newres and diff: variable newres holds the new restrictive sections; and diff holds
the sections that have become restrictive with the occurrence of *Occupy*. The variable rtime
(that describes the time at which a section goes restrictive) is updated for the new restrictive
sections.

```
Occupy(t1,c1) =

PRE

    t1 \in TRAIN \land c1 \in SECTION
```
THEN

LET new occ BE new occ = ran(pos a {t1 m c1})

IN

occ := new occ ||
pos(t1) := c1 ||
ati me(t1) := cur ||

LET newres, diff BE /* Several sections may become restrictive: */

Restrictive(cc,new occ) } \^ newres = { cc \mid cc \in SECTION \^

diff = newres-res

IN

r time := r time a (diff \times \{cur\}) ||
res:=newres

END

END

END

After applying the automatic prover to the RailwayT machine 41 POs are generated, but only 2 POs remained unproved. These two POs are related to the Occupy operation, and they were proved using the interactive prover of Atelier-B (also called manual prover). In here, we will describe in detail the interactive proof of the following PO:

\[(\text{ati me}+\{t1\} \rightarrow \text{cur})(tt) + XX \leq \text{cur} \&

```
`Check that the invariant (!tt. (tt: TRAIN \& pos(tt): res \& ati me(tt) >= r time(pos(tt)) \& ati me(tt) + XX \leq \text{cur} \Rightarrow tt: braking)) is preserved by the operation - ref 3.4"` &
```

=>

\(tt: braking\)

This PO states that if a train \(tt\) is in a restrictive section, and the train has arrived in that section after the section went restrictive, then it must be braking.

This PO verifies if the Occupy operation preserves the second case of the braking property:

\(\text{BrakingProperty}(i) = \)

...
∀tt.( tt ∈ TRAIN ∧ pos(tt) ∈ res ∧

iferes (pos(tt)) + XX ≤ i

⇒ tt ∈ braking )

We start the by splitting the proof in two cases. First, we consider the case where \( t_1 = tt \), i.e., \( tt \) is the train that has just entered section \( c_1 \) by invocation of \( \text{Occupy}(t_1,c_1) \). After that we start the deduction by invoking the command \( \text{dd} \).

PRI > \textbf{dc}(t_1 = tt)
Starting Do Cases
\[ t_1 = tt \Rightarrow ("\text{`Local hypotheses'}" & newocc = ran(pos<+{t_1|->c_1}) \&

\text{newres} = \{cc \mid cc: CDV \& not(block\{-[next\_block(block(cc))]/\text{newocc} = \{\}}) \&

diff = newres-res \& tt: TRAIN \& (pos<+{t_1|->c_1})(tt): newres \&

(atime<+{t_1|->cur})(tt)

)\Rightarrow

(\text{atime<+{t_1|->cur})(tt)+XX<=cur \Rightarrow tt: braking)

is preserved by the operation - ref 3.4"

⇒

\text{tt: braking}

PRI > \textbf{dd}
Starting Deduction
"`Local hypotheses'" &

newocc = ran(pos<+{t_1|->c_1}) \&

\text{newres} = \{cc \mid cc: CDV \& not(block\{-[next\_block(block(cc))]/\text{newocc} = \{\}}) \&

diff = newres-res \&

\text{tt: TRAIN} \&

(pos<+{t_1|->c_1})(tt): newres \&

(atime<+{t_1|->cur})(tt)

)\Rightarrow

(\text{atime<+{t_1|->cur})(tt)+XX<=cur \Rightarrow tt: braking)

is preserved by the operation - ref 3.4"

⇒

\text{tt: braking}
Next, as we are dealing with the case where \( t_1 = t_t \), the variable \( t_t \) can be replaced by \( t_1 \), which is done by the command \texttt{eh}. After this substitution the consequent of the goal becomes \( t_1: \text{braking} \), and a set simplification is applied to the goal (command \texttt{ss}).

\[
PRI > \texttt{eh}(t_t,t_1,\text{Goal})
\]

Starting use Equality in Hypothesis
"'Local hypotheses'" &
newocc = ran(pos<+(t_1)->c_1) &
newres = {cc | cc: CDV & not(block-[[next_block(block(cc))]])/\newocc = {}}) &
diff = newres-res &
t_1: \text{TRAIN} &
(pos<+(t_1)->c_1))\{t_1\}: newres &
(rtime<+diff*{cur})((pos<+(t_1)->c_1))\{t_1\}<=(atime<+(t_1)->cur))\{t_1\} &
(atime<+(t_1)->cur))\{t_1\}+XX<=cur &
"'Check that the invariant (\( !t_t.(t_t: \text{TRAIN} \& pos(t_t): \text{res} \&
\quad \text{atime}(t_t)>=\text{rtime}(pos(t_t)) \& \text{atime}(t_t)+XX<=cur \Rightarrow t_t: \text{braking}) \)

\[
PRI > \texttt{ss}
\]

Begin SimplifySet
"'Local hypotheses'" &
newocc = ran(t_1)<<|pos)/(c_1) &
newres = {cc | cc: CDV & not(block-[[next_block(block(cc))]])/\newocc = {}}) &
diff = newres-res &
t_1: \text{TRAIN} &
c_1: newres &
(rtime<+diff*{cur})(c_1)<=cur &
XX<=0 &
"'Check that the invariant (\( !t_t.(t_t: \text{TRAIN} \& pos(t_t): \text{res} \&
\quad \text{atime}(t_t)>=\text{rtime}(pos(t_t)) \& \text{atime}(t_t)+XX<=cur \Rightarrow t_t: \text{braking}) \)

\[
We start a deduction by calling \texttt{dd}, which adds the goal antecedents as new hypotheses. To prove the new goal we add two hypotheses: the first one was already in the antecedent of the goal.
above; the second one is part of the definition of constant \(XX\). After calling the auto-prover the
proof for the case where \(t_1 = tt\) is concluded, and now we have to prove the other case where
\(t_1 \neq tt\).

PRI > dd
Starting Deduction
\(t_1: \text{braking}\)

PRI > ah(XX<=0)
Starting Add Hypothesis
\(XX<=0 \Rightarrow t_1: \text{braking}\)

PRI > ah(XX>=1)
Starting Add Hypothesis
\(1<XX \Rightarrow (XX<0 \Rightarrow t_1: \text{braking})\)

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
\[
\begin{align*}
\text{not}(t_1 - tt) & \Rightarrow (\text{"Local hypotheses"} \ & \text{newocc} = \text{ran}(pos<+{t_1}<>c_1)) \ & \\
\text{newres} & = \{\text{cc} \mid \text{cc}: \text{CDV} \ & \text{not}(\text{block}^{-}\text{next}(\text{block}(\text{cc})))\}/\text{newocc} = ()\} \ & \\
\text{diff} & = \text{newres} - \text{res} \ & \text{tt}: \text{TRAIN} \ & (pos<+{t_1}<>c_1)(tt) \ : \ \text{newres} \ & \\
\{\text{rtime}<+\text{diff}\}/\text{cur}(\text{pos}<+{t_1}<>c_1)(tt)) \ : \ \{\text{atime}<+{t_1}<>\text{cur}(tt)) \ & \\
(\text{atime}<+{t_1}<>\text{cur}(tt)) \ : \ \text{XX}<\text{cur} \ & \\
\text{"Check that the invariant } (tt.(tt: \text{TRAIN} \ & \text{pos}(tt)) : \text{res} \ & \\
\text{atime}(tt) = \text{rtime}(\text{pos}(tt)) \ & \text{atime}(tt) + XX < cur \Rightarrow tt: \text{braking}) \ & \\
is \text{preserved by the operation - ref 3.4"} \\
\Rightarrow \\
\text{tt: braking}\)
\end{align*}
\]

To add the antecedents to the hypotheses we invoke the twice the command dd (this is necessary,
because the goal has the form \(A \Rightarrow A' \Rightarrow C\), the first invocation just adds the antecedent \(\text{not}(t_1 = tt)\), while the second invocation adds the remaining antecedents.

PRI > dd
Starting Deduction
\[
\begin{align*}
\text{"Local hypotheses"} \ & \\
\text{newocc} & = \text{ran}(pos<+{t_1}<>c_1)) \ & \\
\text{newres} & = \{\text{cc} \mid \text{cc}: \text{CDV} \ & \text{not}(\text{block}^{-}\text{next}(\text{block}(\text{cc})))\}/\text{newocc} = ()\} \ & \\
\text{diff} & = \text{newres} - \text{res} \ & \\
\text{tt}: \text{TRAIN} \ & \\
\end{align*}
\]
(pos<+{t1|->c1})(tt): newres &
(rtime<+diff*(cur))((pos<+{t1|->c1})(tt))<=(atime<+{t1|->cur})(tt) &
(atime<+{t1|->cur})(tt)+XX<=cur &
"Check that the invariant (tt: TRAIN & pos(tt): res & atime(tt)>=rtime(pos(tt)) & atime(tt)+XX<=cur => tt: braking)
is preserved by the operation - ref 3.4" &

->

 tt: braking

PRI > dd
Starting Deduction

 tt: braking

Now the current goal is tt: braking. To prove that goal several hypotheses are added. These three hypotheses were already given in the antecedent of the implication above.

PRI > ah((pos<+{t1|->c1})(tt): newres)
Starting Add Hypothesis

 (pos<+{t1|->c1})(tt): newres => tt: braking

PRI > ah((rtime<+diff*(cur))((pos<+{t1|->c1})(tt))<=(atime<+{t1|->cur})(tt))
Starting Add Hypothesis

 (rtime<+diff*(cur))((pos<+{t1|->c1})(tt))<=(atime<+{t1|->cur})(tt) =>
 (pos<+{t1|->c1})(tt): newres => tt: braking

PRI > ah((atime<+{t1|->cur})(tt)+XX<=cur)
Starting Add Hypothesis

 (atime<+{t1|->cur})(tt)+XX<=cur =>
 (rtime<+diff*(cur))((pos<+{t1|->c1})(tt))<=(atime<+{t1|->cur})(tt) =>
 (pos<+{t1|->c1})(tt): newres => tt: braking

As we know that tt ≠ t1, all the hypotheses we just added could be simplified. For instance, the expression (pos<+{t1|->c1})(tt): newres can be simplified to pos(tt): newres, which implies that tt is a restrictive section. To be able to do those simplifications, first we have to prove that tt (before the occurrence of Occupy) was already defined for both functions pos and atime (the time a train has entered its current position).

PRI > ah(tt: dom(pos))
Starting Add Hypothesis

 tt: dom(pos)
PRI > **pr**
Starting Prover Call

\[
\begin{align*}
tt: \text{dom}(\text{pos}) & \Rightarrow \\
((\text{atime} + \{t1|\rightarrow cur\})(tt) + XX < cur \Rightarrow \\
((\text{rtime} + \text{diff} \times \{cur\})((\text{pos} + \{t1|\rightarrow c1\})(tt)) \leq (\text{atime} + \{t1|\rightarrow cur\})(tt) & \Rightarrow \\
((\text{pos} + \{t1|\rightarrow c1\})(tt): \text{newres} \Rightarrow tt: \text{braking}))
\end{align*}
\]

PRI > **ah(**tt: dom(atime)**))
Starting Add Hypothesis

\[
tt: \text{dom}(\text{atime})
\]

PRI > **pr**
Starting Prover Call

\[
\begin{align*}
tt: \text{dom}(\text{atime}) & \Rightarrow \\
(tt: \text{dom}(\text{pos}) & \Rightarrow ((\text{atime} + \{t1|\rightarrow cur\})(tt) + XX < cur \Rightarrow \\
((\text{rtime} + \text{diff} \times \{cur\})((\text{pos} + \{t1|\rightarrow c1\})(tt)) \leq (\text{atime} + \{t1|\rightarrow cur\})(tt) & \Rightarrow \\
((\text{pos} + \{t1|\rightarrow c1\})(tt): \text{newres} \Rightarrow tt: \text{braking}))
\end{align*}
\]

PRI > **dd**
Starting Deduction

\[
\begin{align*}
tt: \text{dom}(\text{pos}) & \Rightarrow ((\text{atime} + \{t1|\rightarrow cur\})(tt) + XX < cur \Rightarrow \\
((\text{rtime} + \text{diff} \times \{cur\})((\text{pos} + \{t1|\rightarrow c1\})(tt)) \leq (\text{atime} + \{t1|\rightarrow cur\})(tt) & \Rightarrow \\
((\text{pos} + \{t1|\rightarrow c1\})(tt): \text{newres} \Rightarrow tt: \text{braking}))
\end{align*}
\]

PRI > **dd**
Starting Deduction

\[
\begin{align*}
(atime + \{t1|\rightarrow cur\})(tt) + XX < cur \Rightarrow \\
((\text{rtime} + \text{diff} \times \{cur\})((\text{pos} + \{t1|\rightarrow c1\})(tt)) \leq (\text{atime} + \{t1|\rightarrow cur\})(tt) & \Rightarrow \\
((\text{pos} + \{t1|\rightarrow c1\})(tt): \text{newres} \Rightarrow tt: \text{braking})
\end{align*}
\]

We have now proved the two above hypotheses, so we can simplify our goal by applying directly rule SimplifyRelFonXY.9. This rule states that the substitution \((f + \{a|\rightarrow b\})(c) = f(c)\) can be applied if \(c \neq a\) and \(c \in \text{dom}(f)\).

PRI > **ar**(SimplifyRelFonXY.9,Goal)
Starting Apply Rule

\[
\begin{align*}
\text{atime}(tt) + XX < cur \Rightarrow ((\text{rtime} + \text{diff} \times \{cur\})(tt) \leq \text{atime}(tt) & \Rightarrow (\text{pos}(tt): \text{newres} \Rightarrow tt: \text{braking}))
\end{align*}
\]
Looking at our current goal we can see that the expression \(((\text{rtime}<+\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))\) is still too complex to be dealt by the prover. So we are going to divide the current goal in two cases, depending on whether the train \(\text{tt}\) was already in a restrictive section or not. We start with the case where the section that \(\text{tt}\) is occupying has just become restrictive \(\text{(dc}(\text{pos}(\text{tt}):(\text{diff}))\). After having proved that \(\text{pos}(\text{tt}): \text{dom}(\text{diff}\{\text{cur}\})\) we can then simplify the goal using rule \text{SimplifyRelFonXY.8}. This rule states that \((r<s)(a) == s(a), \) provided \(a \in \text{dom}(s)\).

\[
PRI > \text{dc}(\text{pos}(\text{tt}):(\text{diff}))
\]

Starting Do Cases

\[
\text{pos}(\text{tt}): \text{diff} \Rightarrow (\text{atime}(\text{tt})+XX<=\text{cur} \Rightarrow
\quad ((\text{rtime}<+\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))<\text{atime}(\text{tt}) \Rightarrow (\text{pos}(\text{tt}): \text{newres} \Rightarrow \text{tt}: \text{braking}))
\]

\[
PRI > \text{dd}
\]

Starting Deduction

\[
\text{atime}(\text{tt})+XX<=\text{cur} \Rightarrow
\quad ((\text{rtime}<+\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))<\text{atime}(\text{tt}) \Rightarrow (\text{pos}(\text{tt}): \text{newres} \Rightarrow \text{tt}: \text{braking}))
\]

\[
PRI > \text{ah}(\text{pos}(\text{tt}): \text{dom}(\text{diff}\{\text{cur}\}))
\]

Starting Add Hypothesis

\[
\text{pos}(\text{tt}): \text{dom}(\text{diff}\{\text{cur}\})
\]

\[
PRI > \text{ss}
\]

Begin SimplifySet

\[
\text{pos}(\text{tt}): \text{diff}
\]

\[
PRI > \text{pr}
\]

Starting Prover Call

\[
\text{pos}(\text{tt}): \text{dom}(\text{diff}\{\text{cur}\}) \Rightarrow (\text{atime}(\text{tt})+XX<=\text{cur} \Rightarrow
\quad ((\text{rtime}<+\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))<\text{atime}(\text{tt}) \Rightarrow (\text{pos}(\text{tt}): \text{newres} \Rightarrow \text{tt}: \text{braking}))
\]

\[
PRI > \text{dd}
\]

Starting Deduction

\[
\text{atime}(\text{tt})+XX<=\text{cur} \Rightarrow
\quad ((\text{rtime}<+\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))<\text{atime}(\text{tt}) \Rightarrow
\quad \text{pos}(\text{tt}): \text{newres} \Rightarrow \text{tt}: \text{braking})
\]

\[
PRI > \text{ar}(\text{SimplifyRelFonXY.8},\text{Goal})
\]

Starting Apply Rule

\[
\text{atime}(\text{tt})+XX<=\text{cur} \Rightarrow
\quad ((\text{diff}\{\text{cur}\})(\text{pos}(\text{tt}))<\text{atime}(\text{tt}) \Rightarrow (\text{pos}(\text{tt}): \text{newres} \Rightarrow \text{tt}: \text{braking}))
\]

After simplifying the set expression we obtain an implication with several antecedents. We can see that the conjunction of the inequalities \(0<=\text{cur}-\text{atime}(\text{tt})-XX\) and \(\text{cur}<=\text{atime}(\text{tt})\) is false, if \(XX\) is a number greater than 1.
PRI > ss
Begin SimplifySet
0<=cur-atime(tt)-XX &
cur<atime(tt) &
pos(tt): newres &
=>
tt: braking

PRI > ah(XX>=1)
Starting Add Hypothesis
1<=XX => (0<=cur-atime(tt)-XX & cur<atime(tt) & pos(tt): newres => tt: braking)

PRI > pp(rp_0)
Starting Predicate Prover Call
Proved by the Predicate Prover
not(pos(tt): diff) => (atime(tt)+XX<=cur => ((rtime<+diff*{cur})(pos(tt))<=atime(tt) => (pos(tt): newres => tt: braking)))

Now we still have to prove the case where train tt was already in a restricted section before the occurrence of Occupy, i.e., not(pos(tt): diff). Again, the aim of the next commands is to simplify the expression ((rtime<+diff*{cur})(pos(tt)).

PRI > dd
Starting Deduction
atime(tt)+XX<=cur => ((rtime<+diff*{cur})(pos(tt))<=atime(tt) => (pos(tt): newres => tt: braking))

PRI > ah(pos(tt)/:dom(diff*{cur}))
Starting Add Hypothesis
not(pos(tt): dom(diff*{cur}))

PRI > ss
Begin SimplifySet
not(pos(tt): diff)

PRI > pr
Starting Prover Call
not(pos(tt): dom(diff*{cur})) =>
(atime(tt)+XX<=cur => ((rtime<+diff*{cur})(pos(tt))<=atime(tt) =>
(pos(tt): newres => tt: braking)))
Starting Deduction
atime(tt)+XX<=cur =>
((rtime<+diff*(cur))(pos(tt))<=atime(tt) => (pos(tt): newres => tt: braking))

PRI > ss
Begin SimplifySet
0<=cur-atime(tt)-XX &
rtime(pos(tt))<=atime(tt) &
pos(tt): newres &
=>
tt: braking

Because the above goal is very similar to the second case of the invariant property BrakingProperty, we will use the command ph to particularise that property to train tt.

PRI > dd
Starting Deduction
   tt: braking

PRI > ph(tt,!tt.(tt: TRAIN & pos(tt): res & rtime(pos(tt))<=atime(tt) & (0<=cur-atime(tt)-XX & atime(tt)+XX<=cur) => tt: braking))
Starting Particularize Hypothesis
   tt: TRAIN

In the next steps we prove that the train tt satisfies all the antecedents of the implication within BrakingProperty.

PRI > pr
Starting Prover Call
   pos(tt): res

PRI > ah(pos(tt): newres)
Starting Add Hypothesis
   pos(tt): newres => pos(tt): res

PRI > ah(not(pos(tt): diff))
Starting Add Hypothesis
   not(pos(tt): diff) => (pos(tt): newres => pos(tt): res)
PRI > **eh(diff,newres-res,Goal)**
Starting use Equality in Hypothesis
not(pos(tt): newres-res) => (pos(tt): newres => pos(tt): res)

PRI > **pp(rp.0)**
Starting Predicate Prover Call
Proved by the Predicate Prover
rtime(pos(tt))<=atime(tt)

PRI > **pr**
Starting Prover Call
0<=cur-atime(tt)-XX

PRI > **pr**
Starting Prover Call
atime(tt)+XX<=cur

PRI > **pr**
Starting Prover Call
tt: braking => tt: braking

The above goal is trivially proved by the auto-prover, which concludes the interactive proof for PO Occupy.10.

PRI > **pr**
Starting Prover Call
"'Local hypotheses'" &
newocc = ran(pos<+{t1|->c1}) &
newres = {cc | cc: CDV & not(block~[next_block(block(cc))]/newocc = {})) &
diff = newres-res &
tt: TRAIN &
(pos<+{t1|->c1}))(tt): newres &
(rtime<+diff*(cur))((pos<+{t1|->c1})){tt})<=(atime<+{t1|->cur})){tt} &
(atime<+{t1|->cur})(tt)+XX<=cur &
"'Check that the invariant (!tt.(tt: TRAIN & pos(tt): res &
atime(tt)>rtime(pos(tt)) & atime(tt)+XX<=cur -> tt: braking)) is preserved by the operation - ref 3.4'" &
=>
tt: braking
4.4.3 Added Rules Management

Although the prover has become more powerful, the rules database has been extended and proof mechanisms have been improved, but in some situations, rules addition is still necessary because:

- the prover cannot do anything interesting with the goal,
- using rules may save hours spent in performing interactive proofs.

Rules can be added to be used by:

- all the components of the project, by locating them in the PatchProver file, or
- just one component, by adding them to the associated pmm file.

Rules are expressed in the theory language (see Mathematical Rule Writing Guide [ClearSy D] for more details).

With the exception of Rewriting rules, for which class can be easily induced, there is no difference between Backward and Forward rules. So in order to keep track of their type and use, theory names should be non-ambiguous and the prover trace system should be used. This last feature allows the application of rules to be accurately traced.

All the rules that are added should be demonstrated in order to ensure that a project really is fully proven. The Rules Proof Tools should be used for this task. It performs the following controls:

- syntax checking,
- type-checking,
- jokers instantiation correctness,
- guards translation,
- rules translation (depending on their class: Backward, Forward, Rewriting), and
- demonstration.

In addition, some verification should be done by hand for all rules:

- correct calls to prover mechanisms,
- correct use of comma/maplet,
- well-definedness.

Some metrics regarding rules validation are:

- automatic rule validation (using Rules Proof Tools): from 30 to 85%,
- manual rule validation: approximately 20 rules per day,
• manual rule verification: approximately 50 rules per day.

Projects tend to use less user rules than before, thanks to new commands, mechanisms and extensions of the prover.

For example, the Automatic Train Protection software used more than 1200 rules in 1994, but only 22 rules in 2000.
5 Refining Models

The MATISSE project follows a strategy based on principles of top-down stepwise tree-structured refinements and distributed design. Top-down implies that one starts with an abstract description – the architectural model – and will end up through refinement of abstract specifications and proved transformations, with a correct system implementation. This is to be contrasted with bottom-up design, when one conceptually starts with a concrete realisation and by constructing higher-level functionality from building blocks tries to achieve and a posteriori prove properties of an abstract behaviour specified in the architecture. The methods used in this project belong to the former approach.

Stepwise refinements means that there are many proved intermediate design steps in the process of deriving a correct implementation for a system starting from its architectural specifications. The design step, or transformation \( i \) results in a new level of design \( i + 1 \) more concrete (less non-deterministic in terms of the B Method) than the higher level(s). For instance, a given design step may concern the behaviour of just a single component of the system. At the very end, results a series of designs for each component of a system starting with its specifications and ending with one of its implementations. The complete system design will be composed of a set of related and interconnected designs for all (concurrent) components of a whole system.

The tree-structure results because in every transformation step one has the choice (non-deterministic behaviour) between several different lower-level designs corresponding to possible implementation choices. In particular, each transformation step can lead to several concurrent designs at lower-level which can include distribution or locality dependent properties (communication/synchronisation constraints), or environment generated events (e.g., reactive or timed constraints) for the implementation.

5.1 Introduction

It would be impossible to formalise a complete real system by means of a single model, because the state would be far too complicated and the number of events far too large. In order to master the modelling process, we use two complementary techniques:

- refinement, and
- decomposition.

Refining a model consists in refining its state and its events. A concrete model (with respect to a more abstract one) has a state that should be related to that of the abstraction through a, gluing invariant. Sometimes the concrete state is a simple extension of the abstract one so that the abstraction relation is then just a projection. But in general the abstraction relation can be any relation that is defined on all the concrete states.
5.1.1 B Refinement

An abstract machine is refined by applying the standard technique of data refinement to its state: an abstraction invariant is used to relate the state variables of the abstract system to those of the refined system and data refinement should hold between correspondingly-named operations in the abstract and refined systems (see, e.g., [Abrial96,Butler96]). If $S$ is a statement that acts on abstract variables, $T$ is a statement that acts on concrete variables, and $AI$ is an abstraction invariant, then we write

$$S \quad Q_{AI} \quad T$$

for $S$ is data-refined by $T$ under abstraction invariant $AI$. The formulation of $S \Downarrow_{AI} T$ is omitted here. Details may be found in [Abrial96].

A system $M$ is refined by system $N$ under abstraction invariant $AI$ if $N$ has an operation $N.a$ corresponding to each operation $M.a$, and for each such $a$:

$$M.a \quad Q_{AI} \quad N.a.$$  

As well as having an operation corresponding to each of the abstract operations, a refined system may also have some auxiliary internal operations. These are operations that can change the state of the refined machine but whose occurrence is not observable at the more abstract level. Viewed in terms of the abstract state, such operations should have no effect, that is, they should be data-refinements of the skip statement on the abstract state. Each auxiliary operation $N.h$ of refined machine $N$ must satisfy:

$$\text{skip} \quad Q_{AI} \quad N.a.$$  

The tool support available for B can be used to verify refinement between two models. Proof obligations are generated based on the abstraction invariant and these are discharged automatically and semi-automatically.

5.1.1.1 Example

To illustrate the B notion of system refinement we will use the railway example presented in Section 3.2.1, which is a simplification of the transportation case study.

**First Refinement**

The first refinement of machine Railway (see Figure 3.1 and Figure 3.2) is described in Figure 5.1. The Check operation of the system-level model (Figure 3.2) reads the values of next and occp directly to determine whether to apply the brakes. An individual train cannot read these values directly, but instead receives signals from the trackside controllers. Our first refinement introduces this signalling by introducing an auxiliary operation SendTrainMsg. The variables of the refined model are all of the variables of the abstract model plus an extra variable tmsgs mapping each train to a boolean signal being sent from a trackside controller to the train. BOOL$\perp$ represents BOOL augmented with the value $\perp$, where $\perp$ models the absence of a signal. When sending a signal to train $t$, the SendTrainMsg operation sets the value of the signal to TRUE if it is
ok for the train to continue into the next section and FALSE otherwise. The refined Check operation (also shown in Figure 5.1) reads the signal when it is available and sets the brake accordingly. The abstraction invariant used to verify that this model is a valid refinement of the original model is shown in the invariant clause of the Railway1 refinement.

\[
\text{REFINEMENT} \quad \text{Railway1} \quad \text{REFINES} \quad \text{Railway} \\
\text{INVARIANT} \\
\quad \forall t. (t \in \text{TRAIN} \land tmsg(t) \neq \perp \Rightarrow \\
\quad \quad \text{tmsg}(t) = \text{bool}(\text{front}(t) \in \text{dom}(\text{next}) \land \text{next}(\text{front}(t) \notin \text{ran}(\text{occp})))
\]

\[
\text{OPERATIONS} \\
\text{SendTrainMsg}(t) = \\
\quad \text{PRE } t \in \text{TRAIN} \\
\quad \text{THEN} \\
\quad \quad \text{SELECT } tmsg(t) = \perp \\
\quad \quad \quad \text{THEN} \\
\quad \quad \quad \quad \text{tmsg}(t) := \text{bool}(\text{front}(t) \in \text{dom}(\text{next}) \land \text{next}(\text{front}(t) \notin \text{ran}(\text{occp})) \\
\quad \quad \text{END} \\
\text{END}; \\
\text{Check}(t) = \\
\quad \text{PRE } t \in \text{TRAIN} \\
\quad \text{THEN} \\
\quad \quad \text{SELECT } tmsg(t) \neq \perp \\
\quad \quad \quad \text{THEN} \\
\quad \quad \quad \quad \text{braking}(t) := \neg tmsg(t) \lor tmsg(t) := \perp \lor \text{checked}(t) := \text{TRUE} \\
\quad \quad \text{END} \\
\text{END};
\]

Figure 5.1: First refinement of Railway.

**Decomposition to Parallel Sub-Systems**

[Butler97] describes a technique for composing B machines in a way that corresponds to parallel composition. The interaction between the parallel machines is based on synchronisation (with value-passing) between corresponding operations of the respective machines. There are no state variables shared by the machines. Here, we apply this composition in reverse to break our model into several subsystems. The decomposition structure is illustrated in Figure 5.2. In this figure, each of the boxes represents a separate machine. The three machines are TRACK, representing the trackside behaviour, TRAINS representing the train behaviour, and COMMS representing the
communications layer used to send signals between the trackside and the trains. A line joining two boxes represents a synchronisation between machine operations. For example, the Check operation of TRAINS is synchronised with the DeliverMsg operation of COMMS. An arrowhead indicates directed value-passing. The lines attached to just one box represent operations that are not involved in synchronisation.

Use of the communications layer models asynchronous signalling between the trackside and the trains. The trackside and the trains also synchronise directly through the EnterSection and LeaveSection operations. This models the fact that the trackside constrains train behaviour by determining precisely which section a train enters because the connectivity between sections is contained in TRACK. The synchronisation also models the fact that the trackside detects a train entering or leaving a section immediately when it happens. The guard of the composite EnterSection operation is the conjunction of the guards of the EnterSection operation of TRACK and the EnterSection operation of TRAINS, so that the composite operation is constrained by the operation in both machines. Occurrence of EnterSection in the TRAINS machine models a train actually entering a section, while occurrence of EnterSection in the TRACK machine models the detection of this by the trackside.

Figure 5.2: Decomposition structure.

The variables of the three machines are formed by partitioning the variables of the previous refinement. For example, the tmsg variable is placed in the COMMS machine, while the braking variable is placed in the TRAINS machine. A copy of the occupancy variables is placed in both TRAINS and TRACKS. These are not shared variables but instead they are separate variables which will always contain the same value since they are updated simultaneously with the same values by the shared EnterSection and LeaveSection operations. The values of the occupancy variables in the TRAINS machine models the actual positions of trains, while the values in the TRACK machines models the trackside's awareness of the positions of the trains.
THEN

VAR b ∈ BOOL WHERE

    b ← comms.DeliverMsg(t);
    trains.Check(t, b)

END

END;

Figure 5.3: Gluing the components together.

The decomposition of the railway is modelled in B by defining a B machine which glues together the three machines TRACK, TRAINS and COMMS. This gluing machine is declared to be a refinement of the previous model. This is partly shown in Figure 5.3 where the B INCLUDES clause is used to include each of the three machines in refinement Railway3. Figure 5.3 shows how the composite Check operation is formed by sequentially composing a call to the DeliverMsg operation of COMMS with the Check operation of TRAINS. The call to DeliverMsg returns a boolean value which is assigned to local variable b, and b is then passed on as an input parameter for the call to Check in TRAINS. The semantics of the B notation mean that the guard of the composite Check operation is the conjunction of the guard of comms.AcceptMsg and trains.Check, so, as well as modelling value passing, this construction also models synchronisation.

5.1.2 Event B Refinement

In event B each event of the abstract model is refined into a corresponding event of the concrete one. Informally speaking, a concrete event is said to refine its abstraction when

1) the guard of the former is stronger than that of the latter (guard strengthening), and

2) the gluing invariant is preserved by the conjoined action of both events.

Another frequent way of refining an event system consists in adding new events. This corresponds to observing the system with a finer granularity than in the abstraction. Such new events have an implicit (hidden) counterpart in the abstraction, namely, the event that does nothing. The new events that are introduced at some level must obey a specific constraint: they must not monopoly the “control” forever. In other words, they should not have the possibility to be fired indefinitely without letting the old events be executed from time to time. In practical terms, this means that, should the control be given exclusively (guards permitting) to the new events, then, at some point, the disjunction of their guards should become false.

A global constraint of a refined model with regards to its abstraction deals with deadlock. Since a system should normally run forever, then so must its refinement. But more generally, we require that a refined model should not deadlock more frequently than its corresponding abstraction.

Proofs of Correct Refinement
Suppose we have an abstract model with state $v$ and invariant $I(v)$, and also a corresponding concrete model with state $w$ and gluing invariant $J(v, w)$. If an abstract and corresponding concrete events are as follows:

\[
\begin{align*}
\text{ANY } x & \text{ WHERE } \\
& P(x, v) \\
\text{THEN } \\
& v := E(x, v)
\end{align*}
\quad
\begin{align*}
\text{ANY } y & \text{ WHERE } \\
& Q(y, w) \\
\text{THEN } \\
& w := F(y, w)
\end{align*}
\]

then the refinement is correct if:

\[
I(v) \land J(v, w) \land Q(y, w) \Rightarrow \exists x. ( P(x, v) \land J(E(x, v), F(y, w)) )
\]

This states that for each possible choice of the local variables of the concrete event there is a choice of the local variables of the abstraction that makes the gluing invariant (as modified by both events) true: the concrete event refines its abstraction. As can be seen the concrete guard is stronger than its abstract counterpart. In case of a new event of the form:

\[
\begin{align*}
\text{ANY } y & \text{ WHERE } \\
& Q(y, w) \\
\text{THEN } \\
& w := F(y, w)
\end{align*}
\]

the correctness of refinement statement is simpler since that new event is only supposed to refine the non-event that does nothing. Formally:

\[
I(v) \land J(v, w) \land Q(y, w) \Rightarrow J(E(v, F(y, w))
\]

**Proofs of the Impossibility of Monopoly of New Events**

Given a new event of the form

\[
\begin{align*}
\text{ANY } y & \text{ WHERE } \\
& Q(y, w) \\
\text{THEN } \\
& w := F(y, w)
\end{align*}
\]

we must prove that, under the invariant, the event decreases a certain (natural number) variant expression $V(w)$ that has to be exhibited. Notice that this variant expression must be the same for all the new events. This decreasing is thus a global property of the new events because we want to
prevent their entire population from taking control forever. Should the variant be distinct for, say, two events, then they could very well stop individually after some steps, but the other could then take control in an endless ping-pong: this is clearly something we want to avoid. Formally:

\[ I(v) \land J(v, w) \land Q(y, w) \Rightarrow V(F(y, w)) < V(w) \]

**Proofs of the Limitation of Deadlocks**

For each abstract event of the form:

```
ANY y WHERE
  P(y, w)
THEN
  w := E(y, w)
END
```

and for an abstract invariant \( I(v) \) and a gluing invariant \( J(v, w) \), the following statement must be proved:

\[ I(v) \land J(v, w) \land P(x, v) \Rightarrow \text{disjunction of the concrete guards} \]

where the right hand side of the implication denotes the disjunction of the concrete guards. In other words, whenever an abstract event can be fired, then it must also be the case for a concrete event (not necessary the corresponding concrete event, it could be one of the new events). Notice that the guarding predicate \( P(x, v) \) that is on the left-hand side of the implication now corresponds to an abstract event. This contrasts with all the previous proof statements.

### 5.1.3 B-Action System Refinement

With the Event B refinement we can develop distributed systems. However, we cannot model more advanced features, like procedure refinement, with Event B. By using procedures in the distributed systems, we get a more structured system, as well as a general communication mechanism. B-action systems refinement supports also these features. Hence, the proof obligations generated for Event based B form a subset of the obligations generated for B-action systems. A more detailed description of proving the correctness of B-action systems can be found later in this chapter.
5.2 Iterative Construction of Gluing Invariants for Refinement

In this section we provide some guidance on constructing gluing invariants for proving an Event B-style refinement. We do this through an example refinement involving some timing constraints. The example is a simplification of a system based on the railway case study. We have an abstract model which specifies that within $XX$ time units of being signalled to stop by the trackside system, a train must apply its brakes. To model the timing constraint, the abstract machine has a variable ($clk$) which models the current (discrete) time. It also has variables modelling whether the train has been signalled, the time at which it was signalled and whether the train is currently braking:

```
MACHINE stiming1
CONCRETE_CONSTANTS XX
PROPERTIES  XX ∈ NATURAL ∧ XX > 0
ABSTRACT_VARIABLES clk, braking, signalled, stime
INVARIANT
    clk : NATURAL /* current time */
    ∧
    stime ∈ ( 0..clk ) /* time at which a train is signalled */
    ∧
    signalled ∈ BOOL /* TRUE if the train has been signalled */
    ∧
    braking ∈ BOOL /* TRUE if the train is braking */
```

The timing constraint is modelled by adding the following constraint to the invariant:

$$( \text{signalled}=\text{TRUE} ∧ \text{stime} + XX ≤ \text{clk} ⇒ \text{braking}=\text{TRUE} )$$

This says that if the train has been signalled and the current time has reached the deadline $\text{stime}+XX$, then the train should be braking.

We introduce 3 operations in the machine to model the signalling of the train, the application of the brakes and the passage of time:

```
OPERATIONS
Signal =
    BEGIN
    signalled := TRUE || stime := clk
    END;

Brake =
    BEGIN
    braking := TRUE
    END;

Tick =
    SELECT
    signalled=TRUE ∧ stime + XX ≤ clk + 1 ⇒ braking=TRUE
    THEN
    clk := clk + 1
    END
END
```
The timing constraint is imposed by preventing time from progressing unless the braking property would continue to be satisfied in the next time interval. In reality, we cannot prevent time from progressing, but we can satisfy the specification by ensuring that the timing constraint is met on time, e.g., by applying the brakes of a train using the Brake event.

5.2.1 Refined Machine

Now we present a refinement of this in which the train is signalled by the use of an explicit message. We introduce 2 extra operations in the refinement corresponding to sending and receipt of a signal message. We assume that a message might get delayed or may get lost. To deal with this we also introduce a timeout mechanism.

We assume that the trackside system continually sends messages to a train every $\text{MIT}$ time units. In this way a train knows by what time it should expect to receive a message. If it hasn't received a message within the expected time, it applies the emergency brakes. We also assume that there can be a delay in receiving messages and that the maximum expected delay is $\text{MDT}$ time units. Thus a train should wait no longer than $\text{MIT} + \text{MDT}$ time units in between message receipts before applying the brakes. The refinement has the following constants:

```
CONSTANTS MDT, MIT, TIMEOUT

PROPERTIES

\[ \text{MDT} \in \text{NATURAL} \land \text{MIT} \in \text{NATURAL} \land \text{TIMEOUT} \in \text{NATURAL} \land \text{TIMEOUT} > 0 \land \text{TIMEOUT} \leq \text{MIT} + \text{MDT} \]
```

The refinement has all the variables of the abstract machine as well as some new ones. Messages are delivered to the train via set $\text{msgM}$ which is either empty or a singleton boolean value. Empty means there is no message pending for the train. A message of value $\text{TRUE}$ means that the train has been signalled and should brake. When a message is received, its value is stored in $\text{msgT}$. We also introduce variables modelling the time at which the most recent message was sent and the time at which it was received.

```
ABSTRACT_VARIABLES
  clk, braking, signalled,
  sgM, msgT, mstime, mrtime

INVARIANT

\[ \text{msgM} \in \text{POW(BOOL)} \land \text{card(msgM)} \leq 1 \land \text{msgT} \in \text{BOOL} \land \text{mstime} \in (0 .. \text{clk}) \land \text{mrtime} \in (0 .. \text{clk}) \]
```

As well as the signal operation from the abstract machine, the concrete machine also has operations modelling the sending and receiving of messages:
SendMsg =

\[
\begin{align*}
\text{SELECT} & \\
\text{msgM} = \emptyset & \\
\text{THEN} & \\
\text{msgM} := \{\text{signalled}\} & \\
\text{msgS} := \text{signalled} & \\
\text{mstime} := \text{clk} & \\
\text{END}; & \\
\end{align*}
\]

RecvMsg =

\[
\begin{align*}
\text{ANY} & \text{ mm WHERE} \\
\text{mm} \in \text{BOOL} & \\
\text{msgM} = \{\text{mm}\} & \\
\neg(\text{msgT} = \text{TRUE}) & \\
\text{THEN} & \\
\text{msgT} := \text{mm} & \\
\text{mrttime} := \text{clk} & \\
\text{msgM} := \emptyset & \\
\text{END}; & \\
\end{align*}
\]

The timing constraint on the concrete model says that if more than \text{TIMEOUT} time units have passed since the last message was received, or if \text{msgT} has the value \text{TRUE}, then the brakes should be applied. The timing constraint also says that if there is a pending message for the train and the train is not braking, then that message should be fresh, \text{i.e.}, no more than \text{MDT} time units should have passed since it was sent. The concrete \text{Tick} operation is thus defined as follows:

\[
\begin{align*}
\text{Tick} & = \\
\text{SELECT} & \\
( \text{mrttime} + \text{TIMEOUT} \leq \text{clk} + 1 \Rightarrow \text{braking} = \text{TRUE} ) & \\
( \text{msgT} = \text{TRUE} \Rightarrow \text{braking} = \text{TRUE} ) & \\
\text{THEN} & \\
\text{clk} := \text{clk} + 1 & \\
\text{END}
\end{align*}
\]

\[5.2.2 \text{ Gluing Invariant}\]

The initial gluing invariant we use is simply the typing invariant given above. When we attempt to prove the refinement using the auto-prover and interactive prover of Atelier-B, we find that the following proof obligation (PO) is impossible to prove:

\[
\begin{align*}
\text{PRI} & \text{ > go(Tick.1)} \\
\text{"Local hypotheses"} & \text{ &} \\
\text{mrttime$1$+TIMEOUT<=clk$1+1} & \Rightarrow \text{braking$1$ = TRUE} & \\
\text{msgT$1$ = TRUE} & \Rightarrow \text{braking$1$ = TRUE} & \\
\neg(\text{msgM$1$ = {}}) & \text{ &} \neg(\text{braking$1$ = TRUE}) \Rightarrow \\
\text{clk$1$+1<=mstime$1$+MDT} & \text{ &} \text{stime+XX<=clk$1$+1} & \\
\text{signalled$1$ = TRUE} & \text{ &}
\end{align*}
\]
This PO arises from the need to show refinement between the abstract and concrete Tick operations. We see from one of the antecedents that the goal (braking=TRUE) could easily be shown if \( \text{msgT=TRUE} \). So let us introduce a case split, on \( \text{msgT=TRUE} \). In the interactive prover, we start a deduction and call the case split command:

```plaintext
PRI > dd
Starting Deduction
   braking$1 = TRUE
```

```plaintext
PRI > dc(msgT$1 = TRUE)
Starting Do Cases
   msgT$1 = TRUE => braking$1 = TRUE
```

Since this corresponds to a hypothesis, it is easily discharged using the autoprover, leading to the negative case:

```plaintext
PRI > pr
Starting Prover Call
   not(msgT$1 = TRUE) => braking$1 = TRUE
```

We add this antecedent to the hypothesis:

```plaintext
PRI > dd
Starting Deduction
   braking$1 = TRUE
```

It is also useful to add the negation of the goal to the hypothesis so that it can be used as an assumption in any clause that we add to the invariant. To do this, we perform a case split. The first case is trivially discharged, while the second provides a means to add the negated goal to the hypothesis:

```plaintext
PRI > dc(braking$1 = TRUE)
Starting Do Cases
```
braking$1 = TRUE => braking$1 = TRUE

PRI > pr
Starting Prover Call
not(braking$1 = TRUE) => braking$1 = TRUE

PRI > dd
Starting Deduction
braking$1 = TRUE

The hypothesis list now includes the following:

mrtime$1+TIMEOUT<=clk$1+1 => braking$1 = TRUE
stime+XX<=clk$1+1
signalled$1 = TRUE
not(braking$1 = TRUE)
not(msgT$1 = TRUE)

The goal could be proved by adding the following clause to the invariant

( signalled=TRUE &
  not(braking=TRUE) &
  not(msgT = TRUE)
  =>
  mrtime + TIMEOUT <= clk + 1
)

A general rule of the thumb is that we should make use of as many of the existing hypotheses as possible before strengthening the invariant. This means that later POs will be less difficult to prove. In this case we can make use of the transitivity of inequality and make use of the second hypothesis above to weaken the clause to the following:

( signalled=TRUE &
  not(braking=TRUE) &
  not(msgT = TRUE)
  =>
  mrtime + TIMEOUT <= stime + XX
)
We now save the current proof steps, as we need to exit the prover environment in order to add the clause to the invariant of the concrete machine. After adding the clause, we return to the interactive prover and reply the previous proof to the point at which we left it. We discharge the the additional clause (which now appears as a hypothesis) by applying modus ponens to it. This gives us

\[ \text{mrtime} + \text{TIMEOUT} \leq \text{stime} + \text{XX} \]

as a new hypothesis:

\[ \text{PRI} > \text{mh}(\text{signalled}^1 = \text{TRUE} \& \text{not(} \text{braking}^1 = \text{TRUE} \) \& \text{not(} \text{msgT}^1 = \text{TRUE} \) => } \]

\[ \text{mrtime}^1 + \text{TIMEOUT} \leq \text{stime} + \text{XX} \)

Starting Modus Ponens on Hypothesis

\[ \text{mrtime}^1 + \text{TIMEOUT} \leq \text{stime} + \text{XX} \&
\]

\[ => \]

\[ \text{braking}^1 = \text{TRUE} \]

PRI > dd
Starting Deduction

\[ \text{braking}^1 = \text{TRUE} \]

Our aim now is to prove this goal by discharging the hypothesis:

\[ \text{mrtime}^1 + \text{TIMEOUT} \leq \text{clk}^1 + 1 \Rightarrow \text{braking}^1 = \text{TRUE} \]

Thus we attempt to show that \( \text{mrtime}^1 + \text{TIMEOUT} \leq \text{clk}^1 + 1 \) may be deduced from the current hypothesis. When we try to add this as a hypothesis, it becomes a sub-goal to be proven:

PRI > ah(\text{mrtime}^1 + \text{TIMEOUT} \leq \text{clk}^1 + 1)
Starting Add Hypothesis

\[ \text{mrtime}^1 + \text{TIMEOUT} \leq \text{clk}^1 + 1 \]

To prove this sub-goal, we add 2 appropriate hypotheses to it and call the predicate prover:

PRI > ah(\text{mrtime}^1 + \text{TIMEOUT} \leq \text{stime} + \text{XX})
Starting Add Hypothesis

\[ \text{mrtime}^1 + \text{TIMEOUT} \leq \text{stime} + \text{XX} \Rightarrow \text{mrtime}^1 + \text{TIMEOUT} \leq \text{clk}^1 + 1 \]
PRI > ah(stime+XX<=clk$1+1)
Starting Add Hypothesis
    stime+XX<=clk$1+1 => (mrtim$1+TIMEOUT<=stime+XX => mrtim$1+TIMEOUT<=clk$1+1)

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover

Having proved the sub-goal, we are left with the following main goal:
    mrtim$1+TIMEOUT<=clk$1+1 => braking$1 = TRUE

Since this is in the list of hypotheses, we simply call the auto-prover and we are done:
PRI > pr
Starting Prover Call
    "Local hypotheses" & mrtim$1+TIMEOUT<=clk$1+1 => braking$1 = TRUE & msgT$1 = TRUE => braking$1 = TRUE & not(msgM$1 = {}) & not(braking$1 = TRUE) => clk$1+1<=mstime$1+MDT & stime+XX<=clk$1+1 & signalled$1 = TRUE & "Check operation refinement - ref 4.4, 5.5" & => braking$1 = TRUE

The re-display of the original PO indicates that we have successfully proved it. Recall that to prove the PO, we needed to add the following clause to the invariant:

    /* I1 - required to show refinement between both Tick ops: */

    ( signalled=TRUE &
    not(braking=TRUE) &
    not(msgT = TRUE)
5.2.3 Further Proof Obligations

Strengthening the invariant with this clause has the effect of introducing some new POs which themselves need to be proven. In this case we are left with one remaining PO that cannot be proven automatically or interactively:

```plaintext
PRI > go(RecvMsg.4)

"Local hypotheses" &
mm: BOOL &
not(msgT$1 = TRUE) &
msgM$1 = {mm} &
not(braking$1 = TRUE) &
not(mm = TRUE) &
signalled$1 = TRUE &
=>
clk$1+TIMEOUT<=stime+XX
```

In this case, there are no hypotheses that can be used in an obvious way, we add this PO as a clause to the invariant. We make one simplification, which is to eliminate the variable mm and replace it by the value FALSE since the antecedent includes not(mm = TRUE). Thus we further strengthen the invariant of the concrete machine with the following clause:

```plaintext
/* I2 Required by RecvMsg to preserve I1 */

(signalled=TRUE &
not(braking=TRUE) &
not(msgT = TRUE) &
msgM={FALSE}
=>
clk + TIMEOUT <= stime + XX)
```
Now when proving the PO \textit{RecvMsg.4}, we need to add msgM=\{FALSE\} as a hypothesis (the others are already given in the antecedent of the PO). This is easily achieved by making use of other hypotheses:

\begin{verbatim}
PRI > dd
Starting Deduction
clk$1+TIMEOUT<=stime+XX
\end{verbatim}

\begin{verbatim}
PRI > ah(msgM$1 = \{FALSE\})
Starting Add Hypothesis
msgM$1 = \{FALSE\}
\end{verbatim}

\begin{verbatim}
PRI > ah( msgM$1 = \{mm\} )
Starting Add Hypothesis
msgM$1 = \{mm\} => msgM$1 = \{FALSE\}
\end{verbatim}

\begin{verbatim}
PRI > ah( not(mm = TRUE) )
Starting Add Hypothesis
not(mm = TRUE) => (msgM$1 = \{mm\} => msgM$1 = \{FALSE\})
\end{verbatim}

\begin{verbatim}
PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
msgM$1 = \{FALSE\} => clk$1+TIMEOUT<=stime+XX
\end{verbatim}

\begin{verbatim}
PRI > dd
Starting Deduction
clk$1+TIMEOUT<=stime+XX
\end{verbatim}

We are now in a position to prove this goal by discharging the clause that we just added to the invariant:

\begin{verbatim}
PRI > mh( signalled$1 = TRUE & not(braking$1 = TRUE) & not(msgT$1 = TRUE) &
       msgM$1 = \{FALSE\} => clk$1+TIMEOUT<=stime+XX )
Starting Modus Ponens on Hypothesis
clk$1+TIMEOUT<=stime+XX &
\end{verbatim}
This new invariant clause leads to further POs and there is one of these that is impossible to prove given the current invariant:

This new invariant clause leads to further POs and there is one of these that is impossible to prove given the current invariant:

The third clause of the antecedent for this PO can be discharged so that clk$1+1<=mstime$1+MDT can be used as a hypothesis:
Now the antecedent of the final goal above allows us to deduce easily the following hypothesis:

\[ \text{clk}$1+1+\text{TIMEOUT} \leq \text{stime}+\text{XX} \]

This means that if we had as a hypothesis:

\[ \text{clk}$1+\text{MDT}+\text{TIMEOUT} \leq \text{stime}+\text{XX} \]

then the goal would be easily proven. Thus we construct the following clause to add to the invariant, with the above as a consequent, and many of the current hypotheses as antecedents:

\[ /* \text{I3 Required by Tick to preserve I2 */ \]
(  signalled=TRUE &
    not(braking=TRUE) &
    not(msgT = TRUE) &
    msgM={FALSE}
  =>
    mstime+MDT+TIMEOUT  <=  stime + XX
)

With this as a hypothesis, the proof of the PO can be completed:

PRI > mh(signalled$1 = TRUE & not(braking$1 = TRUE) & not(msgT$1 =
    TRUE) &
    msgM$1 = {FALSE} =>  & mstime$1+MDT+TIMEOUT<=stime+XX )
Starting Modus Ponens on Hypothesis
0<=XX-TIMEOUT-MDT+stime-mstime$1 &
mstime$1+MDT+TIMEOUT<=stime+XX &
=>
    clk$1+1<=mstime$1+MDT  =>  clk$1+1+TIMEOUT<=stime+XX

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
"`Local hypotheses'" &
mstime$1+TIMEOUT<=clk$1+1 => braking$1 = TRUE &
    msgT$1 = TRUE => braking$1 = TRUE &
    not(msgM$1 = {}) & not(braking$1 = TRUE) =>
    clk$1+1<=mstime$1+MDT &
    not(braking$1 = TRUE) &
    not(msgT$1 = TRUE) &
    signalled$1 = TRUE &
    msgM$1 = {FALSE} &
=>
    clk$1+1+TIMEOUT<=stime+XX
Fortunately, we now reach a stage where all the newly introduced POs can be proven with the current invariant and do not require further strengthening of the invariant.

### 5.2.4 Constant Properties

As well as strengthening the invariant, proving the refinement POs also requires that we introduce some constraints between the constant $XX$ used in the abstract and the constants $\text{TIMEOUT}$ and $\text{MDT}$ used in the refinement. Such as constraint is required when proving refinement between the Signal operations:

Example of how property such as $\text{TIMEOUT}+\text{MDT}+1 \leq XX$ required:

```
PRI > go(Signal.3)
    "`Local hypotheses'" &
    not(braking$1 = TRUE) &
    not(msgT$1 = TRUE) &
    msgM$1 = {FALSE} &
    =>
    mstime$1+MDT+TIMEOUT<=clk$1+XX

PRI > dd
Starting Deduction
    mstime$1+MDT+TIMEOUT<=clk$1+XX
```

Now, we have as a hypothesis that $\text{mstime} <= \text{clk} + 1$ (this is easily discovered by performing a search for hypotheses matching $\text{mstime}$) so we add this:

```
PRI > ah(\text{mstime} <= \text{clk} + 1)
Starting Add Hypothesis
    \text{mstime} <= \text{clk} + 1 \Rightarrow \text{mstime}+MDT+TIMEOUT<=clk+XX
```

Now, we see that we require that $\text{TIMEOUT}+\text{MDT}+1 \leq XX$

so we add this constraint to the PROPERTIES clause of the concrete machine. Now we can complete the proof of the PO by adding this property to the goal and calling the predicate prover:

```
PRI > ah(\text{TIMEOUT}+\text{MDT}+1 \leq XX)
Starting Add Hypothesis
```
TIMEOUT+MDT+1<=XX => (mstime$1<=clk$1 => mstime$1+MDT+TIMEOUT<=clk$1+XX)

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
"`Local hypotheses'" &
not(braking$1 = TRUE) &
not(msgT$1 = TRUE) &
msgM$1 = {FALSE} &
=>
mstime$1+MDT+TIMEOUT<=clk$1+XX

The above examples illustrate how the prover can be used to construct additional clauses for the gluing invariant in order to prove refinement POs. Using the prover to guide the construction of the gluing invariant means we are less likely to guess invariants that are too strong or that contain errors.

5.3 Guidance on Proving Loops

5.3.1 Overview

Loops are unavoidable control structures in programming, and it is worthwhile to separate their development from the development of purely sequential operations. To do so, we issue the following recommendations:

- it is necessary to identify in implementation operations subparts which are implemented with loops, to isolate them, and to encapsulate them in dedicated operations (local operations or operations from imported abstract machines) that define them independently from their future implementation.

- the structure of the operation implementing a loop should be as simple as possible, loops should never be nested, and their body should be essentially an operation call, perhaps some conditional substitution. If efficiency considerations allow it, loops bodies should modify only local variables of the operation whose values are assigned to implementation variables or operation results after the loop. For instance, the implementation.

During the modelling phase, it is not mandatory to formulate completely the loop invariant, however, for type checking purposes, it should include at least a predicate, the typing predicate of
the loop index for example. For the same reason a well typed variant should be present. The complete definition of the invariant and variant of the loop will be carried out at detailed design.

5.3.2 Loop Development

The starting point of the development of a loop is the operation which defines abstractly the final effects that it must achieve. If we suppose that this operation have been defined in an abstract machine, then, in the refinement preceding the implementation including the loop, it is recommended to reformulate the abstract definition of the loop with a before/after substitution which uses the variables of the implementation and states the properties of the individual elements that will be treated in the loop. Usually, these properties involve quantified variables ranging over the ordered sets that will be scanned in the loop, or domain and co-domain restrictions over such ordered sets, etc.

It is not always possible to reformulate in a simple manner an abstract definition. In those cases it is preferable to leave the abstract definition as it is.

5.3.3 Loop Invariants

The identification of loop invariants, is one of the most challenging activities of formal development. Below is an approach to obtain loop invariants by systematic transformations of abstract definitions of loops.

Starting from the predicate of the last abstract definition, the loop invariant is constructed as follows:

1. To indicate that the expected result (the abstract definition) is achieved progressively at each loop iteration, one or more constants of the predicate are replaced by loop indices. The constants to be replaced are selected according to the scanning order of the elements in the loop

2. As it is the implementation variables that are constructed progressively in the loop, the “after” variables of the predicate of the abstract definition are replaced by the implementation variables

3. The “before” variables of the predicate of the abstract definition correspond either to variables of the implemented module or to input parameters. Therefore, the “before” variables corresponding to abstract variables are decorated by the $0$.

4. The predicates of the implementation invariant concerned with the abstract and concrete variables involved in the predicate produced by the previous steps are modified in such a manner that they indicate the range of scanned values over which they are still valid; the resulting predicates are added to the loop invariant.

5. It may be necessary to add to the loop invariant predicates assuring that the pre-conditions of the operations invoked within the loop body hold.
6. All the variables updated in the loop that appear in the predicate resulting from steps 1-5 are typed, and the typing definitions are added to the loop invariant.

Introducing an intermediate refinement, which reformulates the abstract definition in terms of indices covering intervals, is sometimes worthwhile, as this formulation is more adequate for loop invariants. Furthermore, this intermediate refinement splits the proof of the loop in two steps. The first step proves that the abstract definition is actually refined by the reformulated abstract definition. The second step is mainly concerned with the specific loop proof obligations (loop initialization establishes the loop invariant, loop body preserves the loop invariant, and loop variant decreases), as the proof that the loop actually refines the reformulated abstract definition has been considerably simplified by the fact that the loop invariant has been derived from the reformulated definition.

5.4 Refining from System Level to Software Level Leading into Decomposition

System decomposition is achieved by decomposing the overall model into a number of sub-models. For instance, in a train system, we could have some control parts replicated and embarked within trains and other control parts replicated and installed along the track, and perhaps, yet another control part at the extremity of some line, etc.

The role of decomposition is clear. Once separated from the main body, a sub-model can be developed further independently from the rest of the system, which becomes its, so-called, environment. Notice that this notion of is essentially relative: each sub-model (or group of sub-models) being the environment of others and vice-versa.

There are two kinds of decompositions:

- The decomposition of a system into several subsystems: for instance, in the railway case study, the railway system is divided into two subsystems: the "on-board" subsystems (that makes the train brake) and the "wayside" subsystems (that makes the switch move).
- The decomposition of a (sub)system into its physical and controller part. For instance, the on-board subsystem is divided into its physical part (the B model of the train, with its speed, brakes, etc.) and its controller part (the B model of the software, that deals with measured speed, brake commands, etc.)

The question is, which kind of decomposition should come first. If the system is decomposed into its physical and software parts, and then each part is decomposed in two subsystems (like the diagram below), there might be incoherence. For example, a system may have three controllers for only two subsystems.
Figure 5.4: Bad example of decomposition.

On the contrary, with the following diagram, the system decomposition will not create such incoherencies.

Figure 5.5: Good example of decomposition.
Decomposition in B

A system $C_{comp}$ can be decomposed into two components $A_{comp}$ and $B_{comp}$. Let us consider the abstract machine $C_{comp}$ below, where $x$ and $y$ are local variables and $u$, $v$, and $z$ are global variables. Furthermore, we assume that we have operation $A_{oper}$ that refers to the variables $x$ and $u$ as well as $B_{oper}$ that refers to $y$ and $v$.

```plaintext
MACHINE C_{comp}
INCLUDES
    Global_u, Global_v, Global_z
VARIABLES
    x, y
INVARIANT
    Inv_A \land Inv_B
INITIALISATION
    x := x_0 \parallel y := y_0
OPERATIONS
    A_oper = SELECT P THEN S END;
    B_oper = SELECT Q THEN T END
END
```

The parallel decomposition of the system $C_{comp}$ into the components $C'_{comp}$, $A_{comp}$ and $B_{comp}$ is then defined by splitting the variables, procedures and operations of $C_{comp}$ as follows.
After the decomposition $A_{\text{comp}}$ contains the variable $x$ and $u$ while $B_{\text{comp}}$ contains the variables $y$ and $v$. The global variable $z$ is a common variable of $A_{\text{comp}}$ and $B_{\text{comp}}$. Due to $B$ restrictions it can only be changed (included) by one of the components. The invariant, the initialisation, the procedures, as well as the operations referring to the variables $x$ and $u$ of $A_{\text{comp}}$ are included in $A_{\text{comp}}$, while the ones referring to the variables of $B_{\text{comp}}$ are included in $B_{\text{comp}}$. This rule can be applied in reverse. In that case it is called parallel composition [Butler96].

### 5.4.1 Healthcare Case Study

A control system, such as the Fillwell workstation in our case study, consists of four components: a controller, sensors, actuators, and a plant. These components are depicted in Figure 5.6.
The purpose of the controller in a control system is to ensure that the plant operates within the predefined requirements. For simplicity, we consider here only controllers with *responsive dynamics*. Such controllers become active only when they observe a discrepancy, or some other important event, in the plant.

As for the plant in a control system, we consider it here to have *autonomous dynamics*. This means that the plant changes its state by itself and, in a way, independently of the controller. Thus, even without the controller, the plant would keep on doing something by itself.

The purpose of the sensors and actuators then is to act as information *converters* between the controller and the plant. The sensors convert *measurements* from the plant into *readings* for the controller. Correspondingly, the actuators convert *commands* from the controller into *control signals* to the plant.

The difficulty in the development of a control system is in finding a suitable combination of control logic, sensors, and actuators that allow the controller to maintain the state of the plant within predefined tolerances. Clearly different choices affect the optimality of the overall performance. However, as a general guideline, one tries to minimise manufacturing costs and maximise reliability. This leads to the development of intelligent control logic to allow the use of simple and reliable sensors and actuators. From this point of view, the use of a co-design method, *i.e.*, a method that allows postponing of hardware details during the development process, supports these general development goals. As an example of postponing hardware details, we first consider a robot arm movement generally and only later introduce motor specific details, like acceleration and deceleration.

When thinking of control systems appearing as part of healthcare systems, another difficulty is that such systems should exhibit provable safety and reliability properties. Thus, during the development of a control system, one has to be able to incorporate safety and reliability measures into the system in such a way that these measures can be demonstrated to the customer when necessary. Typically this means increasing complexity in the control logic, as it has to cope with malfunctioning sensors and actuators, along with duplication of some physical sensors and, sometimes, even actuators.
5.4.1.1 Determining Controller and Plant

When enough implementation details are introduced, we proceed by decomposing the component specifications into controller and plant specification pairs. Thus, for each component we obtain both a controller specification and a plant specification. The obtained plant specification describes autonomous behaviour of the component, whereas the controller specification describes algorithms that guide the plant behaviour. Thus, the role of the controller is to react to changes in the plant. We can note that the parallel decomposition of a B-action system into several B-action systems that have the same effect as the initial B-action system is only a structuring technique.

In B-action systems terms, the plant machine contains the transitions to the new states, but before each such transition takes place, a controller machine procedure is called for handling the required decisions. If the controller machine needs not intervene, the procedure body is modelled with a stuttering operation \texttt{skip}. Thus, the procedures of the controller machine are thought of as interrupt procedures.

To obtain such a decomposition in B-action systems, the operations in the machines to be decomposed have to be split, and the variables are partitioned among the plant and the controller machines. As for the splitting of operations, each operation of the form

\[ \text{Oper} = \text{SELECT} \quad \text{state}=\text{act1} \land A \quad \text{THEN} \quad \text{B}; \quad \text{state}:=\text{act2} \quad \text{END} \]

is replaced with the operation in the plant

\[ \text{Oper}' = \text{SELECT} \quad \text{state}=\text{act1} \land A \quad \text{THEN} \quad \text{Act2} \quad \text{END}, \]

where the procedure \text{Act2} of the controller is

\[ \text{Act2} = \text{PRE} \quad \text{state}=\text{act1} \land A \quad \text{THEN} \quad \text{B}; \quad \text{state}:\text{act2} \quad \text{END}. \]

Obviously, the effect of the new operation \text{Oper}' is the same as the effect of the old operation \text{Oper}. Adding procedures and keeping the old functionality agrees with the superposition refinement step.

As an example, let us consider the operation for delivering a plate. When the operating table with a plate is at the lower end position and the Robot is ready to take the plate \texttt{ReadyFor_blank}, the plate is removed (\texttt{blank := FALSE}) and the liquid level is set to zero. Also the variable modelling the Analyser waiting for the Robot is reset. The operation \texttt{Deliver} in the B-action system to be decomposed,

\[ \text{Deliver} = \text{SELECT} \quad \text{astatel}=\text{aidle1} \land \text{acmd}=\text{deliver} \land \text{plate}=\text{TRUE} \land \text{blank} = \text{TRUE} \land \text{zpos}=\text{zmin} \quad \text{THEN} \quad \text{ReadyFor_blank}; \quad \text{ChangeBlankAndResetPlateLiq(FALSE);} \quad \text{awaited:=FALSE;} \quad \text{astatel:=adeliver1} \quad \text{END}, \]

is replaced with the operation \texttt{Deliver} in the plant,
Deliver = \textbf{SELECT} \quad \text{astatel=aidle1} \land \text{acmd=deliver} \land \text{plate=TRUE} \land \text{blank = TRUE} \land \text{zpos=zmin} \\
\text{THEN} \quad \text{ReadyFor_blank; Ddeliver} \\
\text{END},

where the procedure \textit{Ddeliver} of the controller is:

\begin{align*}
\text{Ddeliver} &= \textbf{PRE} \quad \text{astatel=aidle1} \land \text{acmd=deliver} \land \text{blank=TRUE} \land \text{plate=TRUE} \land \\
&\text{zpos=zmin} \\
\text{THEN} \quad \text{ChangeBlankAndResetPlateLiq(FALSE)} \quad || \quad \text{awaited:=FALSE} \quad || \\
&\text{astatel:=adeliver1} \\
\text{END}.
\end{align*}

We can note that the procedure call on the imported procedure \textit{ReadyFor_blank} in the operation \textit{Deliver} remains in the plant. It can be seen as a procedure call to the controller of the system that exports the procedure. Since the procedures are given in an abstract machine construct, we have multiple assignments instead of sequential composition of assignments in the procedure \textit{Ddeliver}.

\subsection{5.4.1.2 Determining Sensors and Actuators}

In the last development step, we determine the sensors and the actuators for the components from the controller and plant specifications. In particular, the variables of the plant and the controller specification usually model more than just properties of these entities; some of these variables are used for communication. From these variables, we find sensors and actuators that are used to synchronise the plant specification with the controller specification.

In B-action systems, the actuators and sensors are global variables of the plant and the controller, and these variables are put into separate machines. The sensor variables are set by the plant and read by the controller, while the actuator variables are set by the controller and read by the plant. Due to this, the plant \textbf{INCLUDES} the sensor variables and \textbf{SEES} the actuator variables. Dually, the controller \textbf{INCLUDES} the actuator variables and \textbf{SEES} the sensor variables. An invariant relates the state of the whole control system to the plant and the controller state, as well as to the sensor variables and to the actuator variables.

The operations of the plant refine the corresponding operations of the initial specification. The general decomposition schema using B-action systems [Sekerinski98a] is shown in
Figure 5.7. Encapsulating the actuator and sensor variables in separate machines allows abstracting from the details of a particular communication mechanism, which depends on the underlying hardware and operating environment. This allows late decisions regarding the hardware implementation.

```
REFINEMENT MachinePlant
REFINES Component_Ref
INCLUDES MachineCtrl, MachineSensors
VARIABLES
<plant state>
INVARIANT
<variable types> ∧
<refinement invariant for plant attributes>,
<controller attributes, actuators, sensors>
INITIALISATION
<plant state and sensors initialisation>
OPERATIONS
Pl_oper = SELECT
Pl_oper_guard
THEN plant state and
sensor change;

Ctrl_oper(parameters)
END;
...
END;

MACHINE MachineCtrl
INCLUDES MachineActuators
SEES MachineSensors
VARIABLES
<controller state>
INVARIANT
<variable types>
INITIALISATION
<controller state and actuators initialisation>
OPERATIONS
Ctrl_oper(parameters) =
PRE Pl_oper_guard
THEN <controller state and
actuator change>
END;

MACHINE MachineSensors
VARIABLES
sensors
INVARIANT
<sensors types>
INITIALISATION
sensors :∈ <any value>
OPERATIONS
SetSensor(ss) = sensor := ss;
...
END

MACHINE MachineActuators
VARIABLES
actuators
INVARIANT
<actuators types>
INITIALISATION
actuators :∈ <any value>
OPERATIONS
SetActuator(aa)= actuator := aa;
...
END
```
5.4.2 Transportation Case Study

Track decomposition

The line is divided into sectors (from 1 to $n$). Each sector is connected to one or two sector(s), called its adjacent sector(s). A switch is a special sector that has either two incoming or two outgoing sectors. The connectedness relation for the sectors is reflexive.

Each sector is associated with one piece of software, which controls this sector (position of switches, emission of messages to train and to adjacent sector, and so on).

Singularities, such as switches, stopping points, and sections, are located on the line, and therefore are presented on sectors. These singularities are represented in the software by arrays of constant data. In fact, this software is divided into two parts:

- A generic part which is the same for all sectors, deals with the algorithmic nature of the software.
- A specific part that is sector-dependent.

Each sector is divided into sections.

Adjacent sectors communicate: each sector sends and receives messages from its adjacent sector(s). These messages are essentially authorization to enter a section.
A (simplified) message between sectors is composed of:

- The number of the section that emitted the message;
- A boolean (authorization or not, for a train in the adjacent sector, to enter the section that emitted the message)

Sections also transmit messages to trains. Each section emits different messages, but if there are several trains on a single section, they will receive the same message. A (simplified) message to a train is composed of:

- The number of the section that emitted the message;
- A boolean (authorization or not to go forward)

If a train does not receive any message over a 5 second period (14 cycles), then the onboard equipment considers that there is no authorisation to enter an adjacent sector. Similarly, if a sector does not receive any message during 5 second period, then the sector equipment considers that there is no authorisation to enter an adjacent sector.

**Description of the invariants**

Here is a simplified example with absolute and relative identification of stopping points. Below is the line invariant, with absolute identification number of each stopping point:

![Diagram of a line with stopping points and sectors](image)

Now, the line is decomposed into two sectors:

![Diagram of two sectors](image)
Now, we obtain the invariants of the two sectors, with relative identification numbers for each stopping point:

![Diagram of two sectors with stopping points labeled 1 and 2 for Sector 1, and 1, 2, and 3 for Sector 2.]

with the gluing bijection defined according to Table 4.

<table>
<thead>
<tr>
<th>Absolute Stopping Point</th>
<th>Sector</th>
<th>Relative Stopping Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Gluing bijection.

The software is linked with the relative identification, whereas in the system model, both identifications are present, with the gluing relation.

**Software architecture**

The current software architecture is described in . For each sector, an infinite loop calls the generic software, which is linked with the invariant of the sector.
Means that A have access to B through the B clause *contain*.

**The track system**

In this schematic, the invariant term means "constant data describing the line/sector" and is not related to the invariant clause of a B component.
In model 1, the proof of the correct decomposition is performed (basically, $\bigcup \text{sector}$).

In model 2, sector process is introduced. This sector process includes communication (message from a sector to another). the sector process is a refinement of the system process.

In model 3, the sector process is refined as a generic sector process. Invariant' are linked with Invariant by a gluing relation.

In model 4, the generic sector process is refined into a generic software. Invariant" are linked with Invariant' by a gluing relation.

Each model is related to the system development shown in next.

In model 2, "sectors system process" has access to the n "sector invariant".

In model 3, "sectors system process" has access to the n "sector invariant", and to the n "generic sector process". Each "generic sector process" has access to its related invariant.

In model 4, for each sector, an infinite loop calls the generic software, which is linked with the invariant of the sector.
5.5 Distributed Refinement

This section proposes a new refinement operator especially designed for the derivation of concurrent processes in the refinement process of the MATISSE methodology. It will allow splitting monolithic operations of an abstract machine into a set of operations, easier to refine. The resulting abstract machine can be seen as many (sub)processes executed in parallel. All those processes can communicate with each other in order to take coordination and synchronisation into account. Of course, details from the initial formal specification can be integrated all along the development by mean of new processes. As soon as the lower level of implementation is reached, our operator can then be used to reassemble (abstract) some of the operations, to sequentialise some processes and, possibly for mapping purposes, to match a specific topology of the target distributed machine architecture.

5.5.1 Definition

A distributed system $P$ is the opening of the bounded choice of all its processes ($P_i$ for $i = 1..n$).

$$P \approx (P_1 \sqcap P_2 \sqcap ... \sqcap P_n)$$

One can define a specific refinement operator for the design of distributed systems where abstract events are no more refined by their concrete counterpart, but where the opening of the bounded choice of all the concrete events refines the opening of the bounded choice of all the abstract events. In order for our operator to be compatible with both algorithmic and data refinements, we built it on top of data refinement.

<table>
<thead>
<tr>
<th>SYSTEM P OPERATIONS</th>
<th>REFINEMENT Q OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{SELECT } p_1 \text{ THEN } P_1 \text{ END} )</td>
<td>( \text{SELECT } q_1 \text{ THEN } Q_1 \text{ END} )</td>
</tr>
<tr>
<td>... ( \text{SELECT } p_n \text{ THEN } P_n \text{ END} )</td>
<td>... ( \text{SELECT } q_n \text{ THEN } Q_n \text{ END} )</td>
</tr>
</tbody>
</table>

END

END
An abstract system $P$ is distributed refined (noted $\hat{\sqsubseteq}$) by an abstract system $Q$ if and only if the opening of the bounded choice of all the processes of $P$ is data refined by the opening of the bounded choice of all the processes of $Q$.

$$P \hat{\sqsubseteq} Q \Leftrightarrow (P_1 \sqcup P_2 \sqcup \ldots \sqcup P_n) \hat{\leq} (Q_1 \sqcup Q_2 \sqcup \ldots \sqcup Q_m)$$

### 5.5.2 Creating, Splitting and Grouping Processes

The same way event B abstracts logical time through the creation of new events [Diab01], we shall consider that distributed refinement as abstracting communications. This way, an abstract system not only refines its existing abstract events, but may also either introduce new communications that were not perceptible in the abstraction or hide communications that were not compatible with the grain of atomicity. Splitting a process reveals some communication details, grouping several processes hides communication detail.

We define below seven rules of refinement and their condition of use concerning as many cases of refinement, for creating new processes, splitting machines in concurrent components (parallelization) and grouping processes in a unique machine (sequentialisation). Thus, when a developer encounters a matching abstract system, he can directly deduce the corresponding refinement with a minimum of proofs. It is assumed that, as most of our refinement rules do not have prerequisites, it is inadequate to automate the entire refinement process. On the other hand, it would be much more realistic to develop an interactive software which, when recognising a matching abstract system, prompts the user for applying a predefined refinement rule.

#### 5.5.2.1 Process Creation

Following "event B" where one can add new events in a refinement if those events refine `skip`, distributed refinement allows to add new processes progressively all along the design in order to integrate more details of the specification and to refine abstraction (decrease) the grain of atomicity.

<table>
<thead>
<tr>
<th>SYSTEM machine OPERATIONS</th>
<th>REFINE refinement machine OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op_{P_1} =$</td>
<td>$op_{P_1} =$</td>
</tr>
<tr>
<td>$\text{SELECT } p_1 \text{ THEN }$</td>
<td>$\text{SELECT } p_1 \text{ THEN }$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$op_{P_n} =$</td>
<td>$op_{P_n} =$</td>
</tr>
<tr>
<td>$\text{SELECT } p_n \text{ THEN }$</td>
<td>$\text{SELECT } p_n \text{ THEN }$</td>
</tr>
<tr>
<td>$P_n$</td>
<td>$P_n$</td>
</tr>
</tbody>
</table>
5.5.2.2 Grouping by sequence

The use of distributed refinement allows the conception of distributed systems, but, also, simplifies the development of sequential programs. So, we must be able to group some of our processes in order to retrieve standard programming structures. This section concerns the case of the sequence.

```
SYSTEM
machine
OPERATIONS
  op_P =
    SELECT p THEN
    p
  END
  op_Q =
    SELECT q THEN
    Q
  END
  op_R1 =
    SELECT r1 THEN
    R1
  END
  ...
  op_Rn =
    SELECT rn THEN
    Rn
  END
END
```

```
REFINEMENT
refinement
REFINES
machine
OPERATIONS
  op_P,Q =
    SELECT p THEN
    P ; Q
  END
  op_R1 =
    SELECT r1 THEN
    R1
  END
  ...
  op_Rn =
    SELECT rn THEN
    Rn
  END
END
```

provided that \( [op_P]q \leftrightarrow true \)
5.5.2.3 Grouping by loop

Refining an abstract machine to obtain while-loops is quite difficult. Within the framework of distributed refinement, this task is simplified by the intensive use of the opening operator ('). Theorem below allows obtaining while-loops, as far as some simple conditions are satisfied.

\[
\begin{array}{l}
\text{SYSTEM machine}
\text{OPERATIONS}
\text{op}_Q = \\
\quad \text{SELECT } q \text{ THEN } \\
\quad \quad Q \\
\quad \text{END}
\text{op}_R = \\
\quad \text{SELECT } r \land s \text{ THEN } \\
\quad \quad R \\
\quad \text{END}
\text{op}_{P1} = \\
\quad \text{SELECT } p_1 \text{ THEN } \\
\quad \quad P_1 \\
\quad \text{END}
\quad \ldots
\text{op}_{Pn} = \\
\quad \text{SELECT } p_n \text{ THEN } \\
\quad \quad P_n \\
\quad \text{END}
\end{array}
\]

\[
\begin{array}{l}
\text{REFINEMENT refinement}
\text{REFINES machine}
\text{OPERATIONS}
\text{op}_{Q,R} = \\
\quad \text{SELECT } q \text{ THEN } \\
\quad \quad Q ; \\
\quad \quad \text{WHILE } S \\
\quad \quad \text{DO } \\
\quad \quad \quad R \\
\quad \quad \text{END}
\text{END}
\text{op}_{P1} = \\
\quad \text{SELECT } p_1 \text{ THEN } \\
\quad \quad P_1 \\
\quad \text{END}
\quad \ldots
\text{op}_{Pn} = \\
\quad \text{SELECT } p_n \text{ THEN } \\
\quad \quad P_n \\
\quad \text{END}
\end{array}
\]

\[\text{provided that } r \leftrightarrow [\text{op}_R]r \land [\text{op}_Q]r \Rightarrow \text{true}\]

5.5.2.4 Grouping/Splitting by selection

The use of the bounded choice operator in the definition of distributed refinement allows us to combine it very easily with the unbounded choice operator. Hence,

\[
\begin{array}{l}
\text{SYSTEM machine}
\text{OPERATIONS}
\text{op}_s = \\
\quad \text{ANY } x \text{ WHERE } g(x) \land h \text{ THEN } \\
\quad \quad S
\end{array}
\]

\[
\begin{array}{l}
\text{REFINEMENT refinement}
\text{REFINES machine}
\text{OPERATIONS}
\text{op}_{s,T} = \\
\quad \text{ANY } x \text{ WHERE } g(x) \text{ THEN } \\
\quad \quad \text{CHOICE}
\end{array}
\]
The reverse refinement is also correct.

<table>
<thead>
<tr>
<th>SYSTEM machine OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{op}_{S,T} = )</td>
</tr>
<tr>
<td>ANY ( x ) WHERE ( g(x) )</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>CHOICE</td>
</tr>
<tr>
<td>SELECT ( h )</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>OR</td>
</tr>
<tr>
<td>SELECT ( i )</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>( \text{op}_{P1} = )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REFINES machine OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{op}_S = )</td>
</tr>
<tr>
<td>ANY ( x ) WHERE</td>
</tr>
<tr>
<td>( g(x) ) ( \land ) ( h )</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>( \text{op}_T = )</td>
</tr>
<tr>
<td>ANY ( x ) WHERE</td>
</tr>
<tr>
<td>( g(x) ) ( \land ) ( i )</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>( \text{op}_{P1} = )</td>
</tr>
<tr>
<td>SELECT ( p_1 ) THEN</td>
</tr>
<tr>
<td>( p_1 )</td>
</tr>
</tbody>
</table>
5.5.2.5 Grouping/Splitting by Condition

Again, as the definition of distributed refinement integrates the bounded choice operator, we can take advantage of its associativity property. Theorem 5 deals with conditional statements.

\[
\text{SYSTEM machine}
\]
\[
\text{OPERATIONS}
\]
\[
\text{op}_s = \begin{align*}
\text{SELECT } g \text{ THEN} \\
S
\text{ END}
\end{align*}
\]
\[
\text{op}_t = \begin{align*}
\text{SELECT } \neg g \text{ THEN} \\
T
\text{ END}
\end{align*}
\]
\[
\text{op}_{P_1} = \begin{align*}
\text{SELECT } p_1 \text{ THEN} \\
p_1
\text{ END}
\end{align*}
\]
\[
\text{...}
\]
\[
\text{op}_{P_n} = \begin{align*}
\text{SELECT } p_n \text{ THEN} \\
p_n
\text{ END}
\end{align*}
\]
\[
\text{REFINEMENT refinement}
\]
\[
\text{REFINES machine}
\]
\[
\text{OPERATIONS}
\]
\[
\text{op}_{S, T} = \begin{align*}
\text{IF } g \text{ THEN} \\
S
\text{ ELSE} \\
T
\text{ END}
\end{align*}
\]
\[
\text{op}_{P_1} = \begin{align*}
\text{SELECT } p_1 \text{ THEN} \\
p_1
\text{ END}
\end{align*}
\]
\[
\text{...}
\]
\[
\text{op}_{P_n} = \begin{align*}
\text{SELECT } p_n \text{ THEN} \\
p_n
\text{ END}
\end{align*}
\]
5.5.2.6 Splitting by Iteration

All along the development process, events must be transformed (refined) as much as possible to be expressed by guarded commands. Theorem 6 below allows eliminating an unbounded choice.

<table>
<thead>
<tr>
<th>SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine</td>
</tr>
<tr>
<td>VARIABLES</td>
</tr>
<tr>
<td>u</td>
</tr>
<tr>
<td>INVARIANTS</td>
</tr>
<tr>
<td>u ∈ E</td>
</tr>
<tr>
<td>OPERATIONS</td>
</tr>
<tr>
<td>op U =</td>
</tr>
<tr>
<td>ANY v WHERE</td>
</tr>
<tr>
<td>g(v)</td>
</tr>
<tr>
<td>THEN</td>
</tr>
<tr>
<td>u := v</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>op P1 =</td>
</tr>
<tr>
<td>SELECT p1 THEN</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>op Pn =</td>
</tr>
<tr>
<td>SELECT pn THEN</td>
</tr>
<tr>
<td>Pn</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REFINEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>refinement</td>
</tr>
<tr>
<td>REFINES</td>
</tr>
<tr>
<td>machine</td>
</tr>
<tr>
<td>VARIABLES</td>
</tr>
<tr>
<td>u, v</td>
</tr>
<tr>
<td>INVARIANTS</td>
</tr>
<tr>
<td>(u, v) ∈ E^2</td>
</tr>
<tr>
<td>OPERATIONS</td>
</tr>
<tr>
<td>op U1 =</td>
</tr>
<tr>
<td>SELECT g(v) THEN</td>
</tr>
<tr>
<td>u := v</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>op U2 =</td>
</tr>
<tr>
<td>SELECT ¬g(v) THEN</td>
</tr>
<tr>
<td>v ∈ E</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>op P1 =</td>
</tr>
<tr>
<td>SELECT p1 THEN</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>op Pn =</td>
</tr>
<tr>
<td>SELECT pn THEN</td>
</tr>
<tr>
<td>Pn</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

5.5.2.7 Variables suppression

Variables defined in an abstract system can be no longer used in a more concrete one. We have to provide some means to automatically eliminate them. This can be done if the variable respects the following conditions:

- The initialisation of the variable validates just a part or even all the guards of all the operations of the concrete system.
- The value of the variable is invariant within all the operations.
Furthermore, if the variable validates the entire guards of all the operations, we have just attained the implementation level (called $B_0$). All the guards can be suppressed and replaced by scope operators (BEGIN, END).

5.5.3 Methodology

The automation of all the previous mechanisms can be quite hard to implement without prerequisites for all refinement rules. Nevertheless, we propose some guidelines for the refinement of distributed software.

1) Try as much as possible to split the initial abstract system into a large number of guarded operations:
   a. Split unbounded choices in smaller ones (*splitting by selection*).
   b. Transform unbounded choices in guarded operations (*splitting by iteration*).
2) Try as much as possible to apply algorithmic and data refinements on those smaller events. During this stage, one can use the B tool to help for the proofs.

3) From now on, group the processes:
   a. Grouping processes by loop is quite simple to use due to its prerequisites.
   b. Grouping processes by sequence is quite simple to use due to its prerequisites.
   c. Group processes by condition whenever their guards are complementary.
   d. Group processes by selection if their guards are complementary so you can group them by condition.

4) Create events whenever it is suitable:
   a. To add details from the specification.
   b. To adjust the grain of atomicity.

5.5.4 Case study

This case study describes the construction of an algorithm dedicated to the evaluation of the transfer rate of information in a network, namely the Average Bit Rate (ABR) control protocol. Although this is not a real industrial case study in MATISSE, we believe that this class of applications, namely constructing correct communication protocols, has a major impact on constructing correct distributed systems.

Our intention here is not to prove the whole process of development using the B Method (it has already been done elsewhere), but to illustrate how the distributed refinement for the automatic generation of a correct protocol. We won’t prove each intermediate abstract system and algorithmic or data refinement (see [Rolland02]), we will only highlight how distributed refinement simplifies the construction of distributed software.

5.5.4.1 Average Bit Rate Protocol

The ABR algorithm is used to specify every station of a network the transfer rate they must use from now on for all their communications. All the rates received by a station form some sort of a list defined as following:

\[ \{(t_i, v_i) \mid t_i \geq 0 \land t_i \leq T\} \]

The bit rate \( v_i \) was received at the time \( t_i \). Last value was received at a time lesser than or equal to the current time \( T \).

For practical reasons, received values can not be processed immediately, but after a certain delay \( \tau_3 \). The same way, received bit rates are meaningless after a certain delay \( \tau_2 \). That is to say that only values received in the temporal window \( [T-\tau_2, (T-\tau_3)] \) are meaningful.
Finally, the ABR algorithm gives only an approximation of the transfer rate in the form of a value $A$ greater than or equal to the maximum of all the rates received within the temporal window.

### 5.5.4.2 Initial specification

We present first a relatively abstract model of the ABR algorithm. We formalize quite exactly the definition of the previous section without taking time into account. It will be introduce later, in an ulterior step of refinement.

```plaintext
SYSTEM Abr1
VARIABLES
   l, v, j, k, A
INVARIANTS
   l ∈ N
   v ∈ (1..l) → N
   j ∈ 1..l
   k ∈ 1..l
   A ∈ N
   \( \max(v[j..k]) \leq A \)
INITIALISATION
   j = k = l = 1 ||
   A = v(1)
OPERATIONS
   ext =
      ANY X WHERE X ∈ N THEN
      l, v(l + 1) := l + 1, X
   END
   in =
      SELECT k < l THEN
      A
      : (\( \max(v[j..k+1]) \leq A \)) ||
      k := k + 1
   END
   out =
      SELECT j < k THEN
      A
      : (\( \max(v[j+1..k]) \leq A \)) ||
      k := j + 1
   END
```
5.5.4.3 First refinement

The first distribution-oriented refinement steps consist in simplifying or eliminating unbounded choices. For instance, we can refine the three operations `in`, `out` and `inout`, where $A : P(A)$ is syntactic sugar for an any construction on which we can apply refinement rule 6: splitting by iteration.

```
REFINEMENT Arb2 (iterating)
REFINES Abr1
VARIABLES
   l, v, j, k, A,
   A'
INVARIANTS
   l ∈ N
   v ∈ (1..l) → N
   j ∈ 1..l
   k ∈ 1..l
   A ∈ N
   A' ∈ N
   max(v[j..k]) ≤ A
OPERATIONS
   ext =
      ANY X WHERE X ∈ N THEN
      l, v(l + 1) := l + 1, X
      END
   in1 =
      SELECT k < l ∧ max(v[j..k+1]) ≤ A' THEN
      A := A' ||
      k := k + 1
      END
   in2 =
      SELECT k < l ∧ A' < max(v[j..k+1]) THEN
```
\[
A' : \in N
\]

\textbf{END}

\textbf{out1} =
\begin{itemize}
  \item \textbf{SELECT} \ j < k \land \max(v[j+1..k]) \leq A' \ \textbf{THEN}
  \item A := A' ||
  \item j := j + 1
\end{itemize}
\textbf{END}

\textbf{out2} =
\begin{itemize}
  \item \textbf{SELECT} \ j < k \land A' < \max(v[j+1..k]) \ \textbf{THEN}
  \item A' : \in N
\end{itemize}
\textbf{END}

\textbf{inout1} =
\begin{itemize}
  \item \textbf{SELECT} \ j < k \land k < l \land \max(v[j+1..k+1]) \leq A' \ \textbf{THEN}
  \item A := A' ||
  \item j, k := j + 1, k + 1
\end{itemize}
\textbf{END}

\textbf{inout2} =
\begin{itemize}
  \item \textbf{SELECT} \ j < k \land k < l \land A' < \max(v[j+1..k+1]) \ \textbf{THEN}
  \item A' : \in N
\end{itemize}
\textbf{END}

\textbf{5.5.4.4 Second refinement}

The six events \textit{in1}, \textit{in2}, \textit{out1}, \textit{out2}, \textit{inout1} and \textit{inout2} can now be simplified as \(A'\) can be eliminated.

\textit{in1} reflects the arrival of a new value in the sensitive zone greater than the previous ones.

\textit{out1} denotes an empty sensitive zone.

\textit{inout1} is a mix of the previous cases.

We can, then, deduce the refinement of those events and obtain the following abstract machine.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{REFINEMENT} & Abr3 (data refinement) \\
\textbf{REFINES} & Abr2 \\
\textbf{VARIABLES} & l, v, j, k, A \\
\textbf{INvariants} & \\
\hline
\end{tabular}
\end{center}
\[ l \in N \]
\[ v \in (1..1) \rightarrow N \]
\[ j \in 1..1 \]
\[ k \in 1..l \]
\[ A \in N \]
\[ \max(v[j..k]) \leq A \]

**OPERATIONS**

\[
\text{ext} =
\]
\[ \text{ANY } X \text{ WHERE } X \in N \text{ THEN} \]
\[ l, v(l + 1) := l + 1, X \]
\[ \text{END} \]

\[
\text{in1} =
\]
\[ \text{SELECT } k < l \land A < v(k+1) \text{ THEN} \]
\[ A := v(k+1) \mid | \]
\[ k := k + 1 \]
\[ \text{END} \]

\[
\text{in2} =
\]
\[ \text{SELECT } k < l \land v(k+1) \leq A \text{ THEN} \]
\[ k := k + 1 \]
\[ \text{END} \]

\[
\text{out1} =
\]
\[ \text{SELECT } j < k \land j + 1 = l \text{ THEN} \]
\[ A := v(l) \mid | \]
\[ j := j + 1 \]
\[ \text{END} \]

\[
\text{out2} =
\]
\[ \text{SELECT } j < k \land j + 1 \neq l \text{ THEN} \]
\[ j := j + 1 \]
\[ \text{END} \]

\[
\text{inout1} =
\]
\[ \text{SELECT } j < k \land k < l \land A < v(k + 1) \text{ THEN} \]
\[ A := v(k + 1) \mid | \]
\[ j, k := j + 1, k + 1 \]
\[ \text{END} \]

\[
\text{inout2} =
\]
\[ \text{SELECT } j < k \land k < l \land v(k + 1) \leq A \text{ THEN} \]
\[ j, k := j + 1, k + 1 \]
\[ \text{END} \]
5.5.4.5 Third refinement
This is a data refinement.

```
REFINEMENT Abr4 (data refinement)
REFINES Abr3
VARIABLES
    F, L, f, j, k, l, A
INITIALISATION
    F, L ∈ N x N
    f ∈ 1..l
    k < f ⇔ (k < l ∧ A < max(v[k+1..l]))
    k < f ⇒ f = min({n | n ∈ k + 1..l ∧ A < v(n) }) ∧ F = max(v[k+1..l])
    L = v(l)
OPERATIONS
    ext =
    ANY X WHERE X ∈ N THEN
        l, L := l + 1, X ||
        CHOICE
            SELECT k < F ∧ F < X THEN
                F := X
            END
        OR
            SELECT f ≤ k ∧ A < X THEN
                f, F := l + 1, X
            END
        END
    END
    in1 =
    SELECT k < l ∧ k + 1 = f THEN
        k, A := k + 1, F
    END
    in2 =
    SELECT k < l ∧ k + 1 ≠ f THEN
        k := k + 1
    END
    out1 =
```
SELECT j < k ∧ j + 1 = l THEN
    j, A := j + 1, L
END

out2 =
SELECT j < k ∧ j + 1 ≠ l THEN
    j := j + 1
END

inout1 =
SELECT j < k < l ∧ k + 1 = f THEN
    j, k, A := j+1, k+1, F
END

inout2 =
SELECT j < k < l ∧ k + 1 ≠ f THEN
    j, k := j+1, k+1
END

END

5.5.4.6 Fourth refinement
Here, we will try to simplify then event ext. To do so, we are going to apply rule of refinement 4: splitting by selection. We must, previously, transform the inner operations of the unbounded choice in the bounded choice of guarded operations. If is trivially done by integrating the parallel affectation in each guarded operations and to create a third one whose guard is the complement of the two others. In fact, we create two events to get four disjunctive guards.

REFINEMENT Abr5 (selecting)
REFINES Abr4
VARIABLES F, L, f, j, k, l, A
INITIALISATION
F, L ∈ N x N
f ∈ 1..l
k < f ⇔ (k < l ∧ A < max(v[k+1..l]))
k < f ⇒ f = min({n | n ∈ k + 1..l ∧ A < v(n)}) ∧ F = max(v[k+1..l])
L = v(l)
OPERATIONS
ext1 =
    ANY X WHERE X ∈ N ∧ k < f ∧ F < X THEN
        l, F, L := l+1, X, X
    END

ext2 =
\[
\begin{align*}
\text{ANY } X \text{ WHERE } X \in \mathbb{N} \land k < f \land X \leq F \text{ THEN} \\
&1, L := 1 + 1, X \\
\text{END}
\end{align*}
\]

\[
\text{ext3 =}
\begin{align*}
\text{ANY } X \text{ WHERE } X \in \mathbb{N} \land f \leq k \land A < X \text{ THEN} \\
&1, f, F, L := 1 + 1, 1 + 1, X, X \\
\text{END}
\end{align*}
\]

\[
\text{ext4 =}
\begin{align*}
\text{ANY } X \text{ WHERE } X \in \mathbb{N} \land f \leq k \land X \leq A \text{ THEN} \\
&1, L := 1 + 1, X \\
\text{END}
\end{align*}
\]

\[
\text{in1 =}
\begin{align*}
\text{SELECT } &k < l \land k + 1 = f \text{ THEN} \\
&k, A := k + 1, F \\
\text{END}
\end{align*}
\]

\[
\text{in2 =}
\begin{align*}
\text{SELECT } &k < l \land k + 1 \neq f \text{ THEN} \\
&k := k + 1 \\
\text{END}
\end{align*}
\]

\[
\text{out1 =}
\begin{align*}
\text{SELECT } &j < k \land j + 1 = l \text{ THEN} \\
&j, A := j + 1, L \\
\text{END}
\end{align*}
\]

\[
\text{out2 =}
\begin{align*}
\text{SELECT } &j < k \land j + 1 \neq l \text{ THEN} \\
&j := j + 1 \\
\text{END}
\end{align*}
\]

\[
\text{inout1 =}
\begin{align*}
\text{SELECT } &j < k < l \land k + 1 = f \text{ THEN} \\
&j, k, A := j + 1, k + 1, F \\
\text{END}
\end{align*}
\]

\[
\text{inout2 =}
\begin{align*}
\text{SELECT } &j < k < l \land k + 1 \neq f \text{ THEN} \\
&j, k := j + 1, k + 1 \\
\text{END}
\end{align*}
\]

\[
\text{END}
\]
5.5.4.7 Fifth refinement

Once again, a data refinement.

<table>
<thead>
<tr>
<th>REFINEMENT</th>
<th>Abr6 (data refinement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFINES</td>
<td>Abr5</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>T, t, l, j, k</td>
</tr>
</tbody>
</table>
| INVARIANTS | T ∈ N \
| | t ∈ (1..l) → N \
| | ∀ n · (n ∈ 1..l - 1 ⇒ t(n) < t(n+1)) \
| | t(l) ≤ T \
| | k = max({i | i ∈ 1..l ∧ t(i) + τ3 ≤ T}) \
| | j = max({i | i ∈ 1..l ∧ t(i) + τ2 ≤ T}) \
| | τ3 < τ2 |
| OPERATIONS | ext1 = \
| | ANY X WHERE X ∈ N ∧ T < t(f) + τ3 ∧ F < X ∧ t(l) \
| | # T THEN \
| | l, F, L := l+1, X, X || \
| | t(l+1) := T \
| | END \
| | ext2 = \
| | ANY X WHERE X ∈ N ∧ T < t(f) + τ3 ∧ X ≤ F ∧ t(l) \
| | # T THEN \
| | l, L := l+1, X || \
| | t(l+1) := T \
| | END \
| | ext3 = \
| | ANY X WHERE X ∈ N ∧ t(f) + τ3 ≤ T ∧ A < X ∧ t(l) \
| | # T THEN \
| | l, f, F, L := l+1, 1+l, X, X || \
| | t(l+1) := T \
| | END \
| | ext4 = \
| | ANY X WHERE X ∈ N ∧ t(f) + τ3 ≤ T ∧ X ≤ A ∧ t(l) \
| | # T THEN \
| | l, L := l+1, X || \
| | t(l+1) := T \
| | END |
in1 =

    SELECT
        k < l ∧
        j < l ∧
        T + 1 = t(k+1) + \tau_3 ∧
        T + 1 ≠ t(j+1) + \tau_2 ∧
        T + 1 = t(f) + \tau_3 ∧
        T + 1 ≠ t(l) + \tau_2
    THEN
        k, A, T := k+1, F, T+1
    END

in2 =

    SELECT
        k < l ∧
        j < l ∧
        T + 1 = t(k+1) + \tau_3 ∧
        T + 1 ≠ t(j+1) + \tau_2 ∧
        T + 1 ≠ t(f) + \tau_3 ∧
        T + 1 ≠ t(l) + \tau_2
    THEN
        k, T := k+1, T+1
    END

out1 =

    SELECT
        T + 1 = t(j+1) + \tau_2 ∧ T + 1 = t(l) + \tau_2
    THEN
        j, A, T := j+1, L, T+1
    END

out2 =

    SELECT
        j < l ∧
        k < l ⇒ T + 1 ≠ t(k+1) + \tau_3 ∧
        T + 1 = t(j+1) + \tau_2 ∧
        T + 1 ≠ t(f) + \tau_3 ∧
        T + 1 ≠ t(l) + \tau_2
    THEN
        j, T := j+1, T+1
    END
inout1 =
  SELECT
    k < l ∧
    j < l ∧
    T + 1 = t(k+1) + τ₃ ∧
    T + 1 = t(j+1) + τ₂ ∧
    T + 1 = t(f) + τ₃ ∧
    T + 1 ≠ t(l) + τ₂
  THEN
    j, k, A, T := j+1, k+1, F, T+1
  END
inout2 =
  SELECT
    k < l ∧
    j < l ∧
    T + 1 = t(k+1) + τ₃ ∧
    T + 1 = t(j+1) + τ₂ ∧
    T + 1 ≠ t(f) + τ₃ ∧
    T + 1 ≠ t(l) + τ₂
  THEN
    j, k, T := j+1, k+1, T+1
  END
tick =
  SELECT
    k < l ⇒ T + 1 ≠ t(k+1) + τ₃ ∧
    j < l ⇒ T + 1 ≠ t(j+1) + τ₂ ∧
    T + 1 ≠ t(f) + τ₃ ∧
    T + 1 ≠ t(l) + τ₂
  THEN
    T := T + 1
  END

5.5.4.8 Sixth refinement
We will now group events together:

  The guards of inout2, ou2, inout2 and tick can be factorized.

  The guards of inout1 and inout1 can also be factorized.
**REFINEMENT**  
Abr7 (conditioning)

**REFINES**  
Abr6

**OPERATIONS**

\[
\text{ext}1 = \\
\begin{align*}
\text{ANY} \, X \, \text{WHERE} \, X \in N \land T < t(f) + \tau_3 \land F < X \land t(l) \neq T \\
\text{THEN} \\
\, l, F, L := l+1, X, X || \\
\, t(l+1) := T \\
\text{END} \\
\text{ext}2 = \\
\begin{align*}
\text{ANY} \, X \, \text{WHERE} \, X \in N \land T < t(f) + \tau_3 \land X \leq F \land t(l) \neq T \\
\text{THEN} \\
\, l, L := l+1, X || \\
\, t(l+1) := T \\
\text{END} \\
\text{ext}3 = \\
\begin{align*}
\text{ANY} \, X \, \text{WHERE} \, X \in N \land t(f) + \tau_3 \leq T \land A < X \land t(l) \neq T \\
\text{THEN} \\
\, l, f, F, L := l+1, l+1, X, X || \\
\, t(l+1) = T \\
\text{END} \\
\text{ext}4 = \\
\begin{align*}
\text{ANY} \, X \, \text{WHERE} \, X \in N \land t(f) + \tau_3 \leq T \land X \leq A \land t(l) \neq T \\
\text{THEN} \\
\, l, L := l+1, X || \\
\, t(l+1) := T \\
\text{END} \\
\end{align*}
\]

**in2_out2_inout2_tick =**

\[
\begin{align*}
\text{SELECT} \, T + 1 \neq t(f) + \tau_3 \land T + 1 \neq t(l) + \tau_2 \, \text{THEN} \\
\, T := T + 1 || \\
\, \text{CHOICE} \\
\, \, k := k + 1 \\
\, \text{OR} \\
\, \, j := j + 1 \\
\, \text{OR} \\
\, \, j, k := j+1, k+1 \\
\, \text{OR} \\
\, \, \text{skip} \\
\text{END} \\
\end{align*}
\]
5.5.4.9 Seventh refinement
The last data refinement.

REFINEMENT Abr8 (data refinement)
REFINES Abr7
VARIABLES
    TF, TL, T, F, L, τ₂, τ₃, A
OPERATIONS
    ext1 =
        ANY X WHERE X \in N \land T < TF \land F < X \land TL \neq T
        + τ₂ THEN
            F, L, TL := X, X, T + τ₂
        END
    ext2 =
        ANY X WHERE X \in N \land T < TF \land X \leq F \land TL \neq T
        + τ₂ THEN
            L, TL := X, T + τ₂
        END
    ext3 =
        ANY X WHERE X \in N \land TF \leq T \land A < X \land TL \neq T
        + τ₂ THEN
            F, L,
            TL, TF := X, X, T + τ₂, T + τ₃
        END
\[ \text{ext4} = \]
\[
\text{ANY } X \text{ WHERE } X \in \mathbb{N} \land TF \leq T \land X \leq A \land TL \neq T \]
\[ + \tau_2 \text{ THEN} \]
\[ L, TL := X, T + \tau_2 \]
\[ \text{END} \]

\[ \text{in2_out2_inout2_tick} = \]
\[
\text{SELECT } T + 1 \neq TF \land T + 1 \neq TL \text{ THEN} \\
T := T + 1 \\
\text{END} \]

\[ \text{in1_inout1} = \]
\[
\text{SELECT } T + 1 = TF \text{ THEN} \\
A, T := F, T + 1 \\
\text{END} \]

\[ \text{out1} = \]
\[
\text{SELECT } T + 1 = TL \text{ THEN} \\
A, T := L, T + 1 \\
\text{END} \]

\[ \text{END} \]

5.5.4.10 Last refinement: implementation

We group once more. \text{in2_out2_inout2_tick}, \text{in1_inout1} and \text{out1} become the scheduler. The four \text{ext} events group into the simplified \text{abr} algorithm.

\begin{tabular}{l}
\textbf{REFINEMENT} & \textbf{Abr9} \\
\textbf{REFINES} & \textbf{Abr8} \\
\textbf{VARIABLES} & A, T, F, X, TF, TL, \tau_2, \tau_3 \\
\textbf{INVARIANTS} & TF = t(f) + \tau_3 \\
& TL = t(l) + \tau_2 \\
\textbf{OPERATIONS} & \text{scheduler} = \\
& \text{BEGIN} \\
& \quad T := T + 1 || \\
& \quad \text{IF } T + 1 = TF \text{ THEN} \\
& \quad \quad A := F \\
& \quad \text{ELSE} \\
& \quad \quad \text{IF } T + 1 = TL \text{ THEN} \\
& \quad \quad \quad A := L \\
& \quad \text{END} \\
& \text{END} \\
\end{tabular}
5.5.5 Conclusion

We have presented a new refinement operator dedicated for the derivation of distributed / multi-process / multi-threaded software from their specifications. The methodological approach we are adopting follows some essential choices.

First, we do not build one abstract machine for each component of a distributed system and prove afterwards the correction of their interactions. The components will be generated progressively in the successive refinements. That way, "communications" between processes are correct by construction.

Second, we intensively use the "divide and conquer" paradigm; we split as much as possible specifications into very simple events. That way, algorithmic and data refinements are easier to prove; preservation of invariants is easier to ensure. It is only after that we group events to obtain standard programming structures: loops, conditional statements and sequences.

Third, communications are taken into account in a very simple manner, through global variables. Of course, one can deal with synchronization and coordination, but data exchanges are still quite limited. This can be addressed furthermore when targeting a specific standard implementation.

This chapter foreshadows an overall method to specify and construct critical distributed systems. A non trivial example has been presented in full for pedagogical purpose, as a case study (e.g., ABR for ATM networks) for constructing correct control protocols.
5.6 Safety Requirements in Refinement

In the healthcare case study, after translating the basic statechart diagram (Figure 3.8) into a B-action system, we gradually refine the system towards a B-model. For the refinement process we identify the attributes suggested in the use case descriptions. These attributes/variables are then added gradually to the specification making it more concrete. At each refinement step we add one or more variables. For each variable we state the safety conditions and the properties of the variable that can be found in the requirements specification. We add the computation concerning the new variable(s) to the existing operations. New operations that only assign the new variable(s) may also be added. A particular refinement method consists in adding new functionality to a specification in this way preserving the old behaviour. This type of refinement is referred to as superposition refinement. When dealing with complex control systems it is especially convenient to stepwise introduce details about the system to the specification and not to have to handle all the implementation issues at once.

The refinement Component_Ref (Figure 5.8) of the machine Component (see Figure 3.10 in Section 3.2.2.5) can be derived from the state diagram in Figure 5.10. Some of the variables in variable_list in the VARIABLES clause are additional to those in the abstract machine Component. The types of the variables, their properties and internal relations as well as their safety requirements are given in the INVARIANT clause. The variables are updated in the operations. Compared to the abstract machine specification Component the refinement machine has more detailed operations. The guards have more precise conditions under which the operation can be performed and also the newly introduced variables can be changed in the operation. In the operation Service1 the new variable(s) are referenced to in the predicate Service1_prec as well as in the substitutions in Service1_comp.

Furthermore, new operations concerning each service/subroutine may be introduced that take care of new features introduced by the new variables. The exported procedures are incorporated in the system as before.

5.6.1 Safety Issues During the Refinement

As the system development proceeds we obtain more elaborated information about faults and conditions of failure occurrence. The refinement step introduces distinction between faults. The operation Service1_fail models a fault resulting from an attempt to provide a service from an incorrect initial state:

Service1_fail = SELECT ¬Service1_prec ∧ state = Idle THEN state := Suspension1 END

This situation might be caused by faults occurred previously or by a logical error in the calling command. The operation Service1_notok models fault occurrence during the execution of the action:

Service1_notok = SELECT ¬Service1_postc ∧ state=Service1 THEN state := Suspension1 END
These kinds of faults are caused by the physical failures of the system components involved in the execution.

```plaintext
REFINEMENT Component_Ref
REFINES Component
VARIABLES variable_list
INVARIANT
    variable_types ∧
    variable_relations ∧
    variable_requirements
INITIALISATION variable_initialisation
OPERATIONS
    New_command(ss) =
        PRE ss ∈ {serv1,serv2,serv3}
        THEN
            SELECT state = Idle THEN cmd := ss END
        END;
    Service1 =
        SELECT Service1_prec ∧ state = Idle
        THEN
            state := Service1 || Service1_comp
        END;
    Service1_fail =
        SELECT ¬Service1_prec ∧ state = Idle
        THEN
            state := Suspension1
        END;
    Service1_ok =
        SELECT Service1_postc ∧ state = Service1
        THEN
            state := Idle || Service1_endcomp
        END;
    Service1_notok =
        SELECT ¬Service1_postc ∧ state = Service1
        THEN
            state := Suspension1
        END;
    Remedy1 =
        SELECT state = Suspension1
        THEN
            fix_the_error1 || state := Service1
        END;
    Services_notready =
        SELECT state = Idle
        THEN
            state := Suspension
        END;
    Remedy =
        SELECT state = Suspension
        THEN
```
We also introduce a distinction between different repair procedures by adjusting the Remedy operation for each fault accordingly.

The action Failure models system shut down:

```
Failure = SELECT state ∈ {Suspension, Suspension1, Suspension2, Suspension3} THEN state := Abort END
```

The action becomes enabled when the attempts to fix occurred faults fail.

Safety analysis proceeds by an identification of different failure modes of the system. It is conducted by considering the consequences of multiple faults. In the system specification we proceed by specifying the status of each component (failed or functioning), specifying system failure modes and finally by introducing error messages.

### 5.6.2 Proving the Correctness of the Refinement

Within Atelier-B we can actually prove formally that the refinement is sound. For this a number of proof obligations first need to be generated. Let us assume that we have two systems $A_{\text{comp}}$ and $C_{\text{comp}}$ as below. The variable $x$ is a variable of both $A_{\text{comp}}$ and $C_{\text{comp}}$. In the refinement step the variable $y$ is a new variable of $C_{\text{comp}}$. The operations $A_1$, $A_2$ and $A_3$ are changed to also take $y$ into account in $C_{\text{comp}}$. The operations $B_1$ and $B_2$ are created during the refinement step.

```plaintext
MACHINE A_{\text{comp}}
VARIABLES x
INVARIANT Inv_A
INITIALISATION x := x0
OPERATIONS
A1 = SELECT P1 THEN S1 END;

REFINEMENT C_{\text{comp}}
REFINES A_{\text{comp}}
VARIABLES x, y
INVARIANT Inv_C
INITIALISATION x := x0 || y:= y0
OPERATIONS
A1 = SELECT P1' THEN S1' END;
A2 = SELECT P2' THEN S2' END;
```
The proof obligations needed for proving that a concrete system \( C\_comp \) is a correct refinement of a more abstract system \( A\_comp \) are as follows [Back96, Walden98a]. For each proof obligation the corresponding proof obligation in Event B is given in the parentheses.

1. The initialization in \( C\_comp \) should be a refinement of the initialization in \( A\_comp \), and the initialization should establish the invariant \( \text{Inv}_C \). (Proofs of Correct Refinement)

2. Each operation \( A_i \) (\( i=1,2,3 \)) in \( C\_comp \) should refine the corresponding operation (the operations with the same name) in \( A\_comp \), and they should preserve the invariant \( \text{Inv}_C \). (Proofs of Correct Refinement)

3. Each new operation \( B_j \) (\( j=1,2 \)) in \( C\_comp \) (that do not have a corresponding operation in \( A\_comp \)) should only concern the new variable \( y \), and preserve the invariant \( \text{Inv}_C \). (Proofs of Correct Refinement)

4. The new operations \( B_j \) (\( j=1,2 \)) in \( C\_comp \) should terminate, if they are executed in isolation. (Proofs of the Impossibility of Monopoly of New Events)

5. Whenever an operation \( A_i \) (\( i=1,2,3 \)) in \( A\_comp \) is enabled, either the corresponding operation \( A_i \) in \( C\_comp \) is enabled or then one of the new operations \( B_j \) (\( j=1,2 \)) in \( C\_comp \) is enabled. (Proofs of the Limitation of Deadlocks)

6. Whenever a fault occurs in \( A\_comp \) (a fault-operation in \( A\_comp \) is enabled), a fault could also occur in \( C\_comp \) (a fault-operation in \( C\_comp \) could be enabled). (Proofs of the Limitation of Deadlocks)

With the fault-operations in (6) we mean the operations leading to a suspension state. Hence, according to (6) a general fault is partitioned into distinct faults during the refinement process.

The proof obligations (1)–(3) above are automatically generated by Atelier-B. Proof obligations (5) and (6) can be generated automatically if an exit- as well as an exitFail-operation is added to the system. The exit-operation should be enabled when all the other non-failure operations in the system are not enabled. In \( A\_comp \) the exit-operation would be

\[
\text{SELECT } \neg P1 \land \neg P2 \land \neg P3 \text{ THEN skip END.}
\]

The exitFail-operation can be created similarly with the guards of the failure-operations. The proof obligation (4) requires a variant that is decreased by the new operations. Using Atelier-B we need to introduce some additional machine constructs discussed in [Walden98b,]. With the help of the Evt2b translator [ClearSy01] also proof obligation (4) can be generated automatically as discussed in [Butler96].

During the refinement process global procedures, procedures that are declared in one system and called from other systems as \( P \) above, are refined in the same way as the operations of the system. Hence, let us assume that we have an abstract system \( A\_comp \) with a global procedure \( P \) and a refined system \( C\_comp \) with the refined procedure \( P' \). Then for \( C\_comp \) to be a refinement of \( A\_comp \), the six proof obligations concerning refinement of distributed systems given earlier should hold, as well as:
7. Procedure $P$ in $A_{\text{comp}}$ should be refined by the corresponding procedure $P'$ in $C_{\text{comp}}$, and $P'$ should also preserve the invariant.

8. If procedure $P$ is enabled so should $P'$ be, or then the operations in $C_{\text{comp}}$ should enable $P'$.

Proof obligation (7) is automatically generated by Atelier-B and corresponds to the proof obligations concerned with Correct Refinement. Proof obligation (8) requires some extra constructs to be created. It has to be proven that when executing the operations they will eventually enable the refined global procedure $P'$ when the abstract procedure $P$ is enabled. Hence, we should provide a variant that is decreased by each operation. For more details on procedures and their proof obligations, see [Walden98b].

The proof obligations can be discharged with the help of the auto-prover and interactive prover in Atelier-B. The auto-prover uses a database of proof rules in order to perform the proofs. In case the needed proof rules are not found in this database, some new rules can be added via the interactive prover of Atelier-B. The proof obligations that were not proved automatically can be proven interactively with the interactive prover.

5.7 Refinement with UML+B

In order to get a graphical interface to the formal method to facilitate the reading and analysing of the development, UML artefacts are integrated in the formal development process. The stepwise introduction of implementation details (features) can be applied by adding these features into class and statechart diagrams. Part of the refined diagrams is automatically generated from the more abstract ones. New variables and operations can be added to the class diagrams.

The more concrete behaviour of the system can then be modelled with new states and more complex transitions in the statechart diagrams taking into consideration the new variables and operations of the corresponding class diagram. While adding the new features to the statecharts, the refinement rules for B-action systems should be applied. For each refinement step a new set of class and statechart diagrams are generated. The refined diagrams are then automatically translated to B-action systems with the U2B-tool.

Currently the U2B tool provides the following support for refinement.

a) The B component is created as a refinement instead of a machine (i.e., it has a `REFINEMENT` header and a `REFINES` clause).

b) Refinement of events can be indicated on statecharts by merging and splitting transitions on statecharts.
5.7.1 Refinement in Class Diagrams

During the development more implementation details are added to the system. The new features are modelled with new attributes. Sometimes even new operations have to be added in the class diagrams. In Figure 5.9 the attributes awaited, move_counter and analyse_counter have been added to the Analyser class to keep track of the services performed by the Analyser. In the class A_PROC the new features on the operating table of the Analyser are given. The attribute zpos gives the height of the operating table in the Analyser and blank states whether there is a plate on the operating table in the Analyser or not. Furthermore, new failure-operations are added to the methods of the Analyser.

We state the services of the basic statechart with more details in a refined statechart diagram. This is shown in Figure 5.10 where the component can evolve from a state Idle to a state which performs each of the required services, provided certain conditions, ServiceN_prec, are satisfied. If these services are successfully performed, when ServiceN_postc holds, the component returns to its state Idle. An exception of a service suspends the autonomous behaviour of a component, that evolves to the state Suspension. If a remedy for an occurred exception is found, the component can resume its service-providing state. If no remedy exists, then the exception is a failure and the component aborts its execution.
Figure 5.9: A refined class diagram of the Analyser.
Figure 5.10: A refined statechart diagram for a component.

The refined statechart diagram is derived from the use case description (subroutine description) and the refined class diagram. All the conditions shown in the diagram are expressions on the class attributes and methods. At this level, the initial informal specification regarding the respective component is completely captured by this refined statechart diagram. This diagram can still be refined to a more detailed statechart diagram involving more attributes and methods. The class and statechart diagrams of each refinement step are translated to B. Figure 5.11 and Figure 5.11 show part of the refined statechart diagram of the Analyser. Note that similar diagrams for the move and deliver commands and the service_not_ready condition exist, but are not shown.

Figure 5.11: Transitions associated with a receive command.
5.7.2 Refinement using Statecharts

The rest of the section describes in detail how both transitions and states can be involved in the refinement.

5.7.2.1 Transition Refinement

One technique that was employed was to break down a single transition, representing one event, into several different transitions representing alternative events that are either refinements of the original event or additional events (refining the non-event, skip).

For example, in the analyser example the state transitions associated with a command to analyse were

![Diagram of state transitions]

Figure 5.12: Transitions associated with an analyse command.
These were refined by adding new transitions representing additional failure conditions. In doing so the transition path from *aidle* to *aanalyse* may be refined by additional guards and actions.

The corresponding fragments of B (produced by U2B) are:

For the abstract actions of *ANALYSER*:

```plaintext
Analyse =
BEGIN
  SELECT 0=0
  THEN astate:=aanalyse ||
      PlateInAnalyser ||
      SELECT astate=aidle ∧ acmd=analyse THEN skip END
END ;
End
AnalysePlateFail =
BEGIN
  SELECT 0=0
  THEN astate:=asuspended ||
      NoPlateInAnalyser ||
      SELECT astate=aidle ∧ acmd=analyse THEN skip END
END
End
```

For the refined actions of *ANALYSER1*:

```plaintext
Analyse =
BEGIN
  SELECT 0=0
  THEN astate:=aanalyse1 ||
      CheckPosOk(zmax);analyse_counter:=analyse_counter+1 ||
```
SELECT 0=0 THEN BlankInAnalyser ||
SELECT 0=0 THEN  PlateInAnalyser ||
SELECT analyse_counter < aMaxMoves then
SELECT astate=aidle1 \ acmd=analyse
THEN skip END
END
END
END

AnalyseMaxFail =
BEGIN
SELECT  analyse_counter=aMaxAnalyse
THEN astate:=asuspended1 ||
SELECT astate=aidle1 \ acmd=analyse THEN skip END
END

AnalysePlateFail =
BEGIN
SELECT 0=0 THEN  astate:=asuspended1 ||
NoPlateInAnalyser ||
SELECT analyse_counter<aMaxMoves
THEN SELECT astate=aidle1 \ acmd=analyse THEN skip END
END

AnalyseBlankFail =
BEGIN
SELECT 0=0 THEN  astate:=asuspended1 ||
NoBlankInAnalyser ||
SELECT 0=0 THEN  PlateInAnalyser ||
SELECT analyse_counter<aMaxMoves
THEN SELECT astate=aidle1 \ acmd=analyse THEN skip END
END

AnalysePosFailed =
BEGIN
SELECT 0=0 THEN  astate:=asuspended1 ||
CheckPosNotOk(zmax) ||
SELECT 0=0 THEN  BlankInAnalyser ||
SELECT 0=0 THEN  PlateInAnalyser ||
SELECT analyse_counter<aMaxMoves
5.7.2.2 State Refinement

Another refinement that was used on the statechart models was to break down a single state into several sub-states. Currently, the U2B translation does not support hierarchical states. Therefore, we refined the original set of states with a complete new set of states replacing the refined state with its sub-states. This involves renaming all the states, even those that are not being decomposed. For example, the `asuspended` state of `ANALYSER` was broken down into different sub-states in order to facilitate enabling of the correct remedy action for the refined failures. This meant that, in the translated B, the state variable was refined and glued as shown below:

```plaintext
5.7.2.2 State Refinement

Another refinement that was used on the statechart models was to break down a single state into several sub-states. Currently, the U2B translation does not support hierarchical states. Therefore, we refined the original set of states with a complete new set of states replacing the refined state with its sub-states. This involves renaming all the states, even those that are not being decomposed. For example, the `asuspended` state of `ANALYSER` was broken down into different sub-states in order to facilitate enabling of the correct remedy action for the refined failures. This meant that, in the translated B, the state variable was refined and glued as shown below:

```

```plaintext
MACHINE ANALYSER
SETS
ASTATE=(aidle, amove, aanalyse, areceive, adeliver, aabort, asuspended)
VARIABLES
astate, ...
INVARIANT
astate : ASTATE & ...
INITIALISATION
astate := idle || ...

is refined by:

```

```plaintext
REFINEMENT ANALYSER1
REFINES
ANALYSER
... 
SETS
ASTATE=(aidle1, amove1, aanalyse1, areceive1, adeliver1, aabort1, asuspended1, 
amsuspended1, aasuspended1, adsuspended1, arsuspended1)
VARIABLES
astate, ...
INVARIANT
astate ∈ ASTATE ∧ ...

```

```plaintext
... (astatel=aidle1 ⇒ astate=aidle) ∧ 
(astatel=amove1 ⇒ astate=amove) ∧ 
(astatel=aanalyse1 ⇒ astate=aanalyse) ∧ 
(astatel=adeliver1 ⇒ astate=adeliver) ∧ 
(astatel=areceive1 ⇒ astate=areceive) ∧ 
(astatel=asuspended1 ⇒ astate=asuspended) ∧ 
(astatel=amsuspended1 ⇒ astate=asuspended) ∧ 
```

```plaintext
...
5.7.2.3 Hierarchical State Refinement

As observed above, the correspondence of all the states has to be explicitly stated in the invariant even when there is no real change to the state (e.g., \texttt{aidle1} is equivalent to \texttt{aidle}). An improvement to the U2B translator, so that it handles hierarchical states would enable the implicit refinement correspondence of B to be utilised and the hierarchical correspondence would become more obvious. The resulting superposition refinement would also make the proofs easier to discharge. The proposed new method of handling state refinement is shown below.

This diagram shows the states with no transitions. It illustrates the hierarchical relationship between the states in the statechart model of \textit{ANALYSER1\_H} (a version of \textit{ANALYSER1} utilising this proposed method):

Note that, apart from the addition of sub-states of \texttt{asuspended}, the set of states are exactly the same as in \textit{ANALYSER} (the class refined by \textit{ANALYSER1\_H}) in contrast to previous refinements where all the states were renamed by post fixing the number 1.

\[
\begin{align*}
(astate1\&astate1 = aasuspended1 \Rightarrow astate = asuspended) \land \\
(astate1\&astate1 = adsuspended1 \Rightarrow astate = asuspended) \land \\
(astate1\&astate1 = arsuspended1 \Rightarrow astate = asuspended) \\
\end{align*}
\]

\texttt{INITIALISATION}
\[
astate := \texttt{aidle1} || 
\]

\[
\begin{align*}
\text{(astate1} = \text{aasuspended1} \Rightarrow \text{astate} = \text{asuspended}) \land \\
\text{(astate1} = \text{adsuspended1} \Rightarrow \text{astate} = \text{asuspended}) \land \\
\text{(astate1} = \text{arsuspended1} \Rightarrow \text{astate} = \text{asuspended}) \\
\end{align*}
\]
sub-state wherever it is. The parent state, `asuspended`, is not used on any statechart diagram, it is implied by the transition entering one of its sub-states. For example, the (partial) analyser statechart shown in Figure 3.9 becomes:
Note that showing the parent state, *asuspended*, on the diagram is superfluous. This is because the hierarchy is captured in the model, not on the diagram. The B refinement that we would like the U2B translator to produced is shown below.

Note:

1. Only operations relevant to the analyser command are shown.
2. The gluing invariant is generated automatically from the state hierarchy whereas before it had to be manually entered as information attached to the class.

```plaintext
REFINEMENT ANALYSER1
REFINES ANALYSER
SEES DEF0, DEF1
INCLUDES R_PROC,
GLOBALVAR_PLATE,
A_PROC1
SETS
ASTATE={aidle,amove,aanalyse,arecieve,adeliver,aabort,asuspended}
ASUSPENDED_SUBSTATE={asuspended1,amsuspended1,aasuspended1,adsuspended1,
arsuspended1}
VARIABLES
astate,
asuspended_substate,
acmd,
awaited,
move_counter,
analyse_counter
INVARIANT
astate ∈ ASTATE ∧
asuspended_substate ∈ ASUSPENDED_SUBSTATE ∧
acmd ∈ ACOMMAND ∧
awaited ∈ BOOL ∧
move_counter ∈ 0..aMaxMoves ∧
analyse_counter ∈ 0..aMaxAnalyse ∧
(aasuspended_substate=asuspended1 ⇒ astate=asuspended) ∧
(aasuspended_substate=amsuspended1 ⇒ astate=asuspended) ∧
(aasuspended_substate=aasuspended1 ⇒ astate=asuspended) ∧
(aasuspended_substate=adsuspended1 ⇒ astate=asuspended) ∧
(aasuspended_substate=arsuspended1 ⇒ astate=asuspended)
INITIALISATION
astate := aidle ||
asuspended_substate ∈ ASUSPENDED_SUBSTATE ||
acmd := receivee ||
awaited := FALSE ||
move_counter := 0 ||
```
analysed_counter := 0

OPERATIONS
AnalyseFail=
BEGIN
  SELECT astate=aidle ∧ acmd=analyse
  THEN astate:=asuspended || asuspended_substate:=asuspended1
END
END;
AnalyseReady=
BEGIN
  SELECT astate=aanalyse ∧ acmd=analyse
  THEN astate:=aidle
END
END;
AnalyseNotReady=
BEGIN
  SELECT astate=aanalyse ∧ acmd=analyse
  THEN astate:=asuspended || asuspended_substate:=asuspended1
END
END;
AnalysePosRemedy=
BEGIN
  SELECT astate=asuspended ∧ asuspended_substate=asuspended1 ∧ acmd=analyse ∧ analyse_counter=aMaxAnalyse
  THEN astate:=aidle || Safe; MoveZ(zmax)
END
END;
AnalyseMaxRemedy=
BEGIN
  SELECT astate=asuspended ∧ asuspended_substate=asuspended1 ∧ acmd=analyse ∧ analyse_counter=aMaxAnalyse
  THEN astate:=aidle || Safe; MoveZ(zmax)
END
END;
AnalyseBlankRemedy=
BEGIN
  SELECT astate=asuspended ∧ asuspended_substate=asuspended1 ∧ acmd=analyse ∧ blank=FALSE
  THEN astate:=aidle || ChangeBlank(TRUE)
END
END;
AnalyseRemedy=
BEGIN
  SELECT astate=asuspended ∧ asuspended_substate=asuspended1 ∧ acmd=analyse
  THEN astate:=aanalyse || PlateInAnalyser; BlankInAnalyser; CheckPosOk(zmax)
END
END;
AnalyseMaxFail=
BEGIN
  SELECT analyse_counter=aMaxAnalyse
  THEN astate := asuspended || asuspended_substate:=asuspended1 ||
SELECT astate = aidele \land \text{acmd}=analyse \text{ THEN skip END}

END

AnalyzePlateFail=
BEGIN
  SELECT 0=0
  \text{THEN} astate := asuspended \mid \mid \text{asuspended\_substate:=asuspended1} \mid \mid \text{NoPlateInAnalyser} \mid \mid
  SELECT \text{analyse\_counter}<\text{aMaxMoves}
  \text{THEN}
  SELECT astate = aidele \land \text{acmd}=analyse \text{ THEN skip END}

END

END;

AnalyzeBlankFail=
BEGIN
  SELECT 0=0
  \text{THEN} astate := asuspended \mid \mid \text{asuspended\_substate:=asuspended1} \mid \mid \text{NoBlankInAnalyser} \mid \mid
  SELECT 0=0
  \text{THEN} \text{PlateInAnalyser} \mid \mid
  SELECT \text{analyse\_counter}<\text{aMaxMoves}
  \text{THEN}
  SELECT astate = aidele \land \text{acmd}=analyse \text{ THEN skip END}

END

END;

Analyze=
BEGIN
  SELECT 0=0
  \text{THEN} astate := aanalyse \mid \mid \text{CheckPosOk}(zmax) ; \text{analyse\_counter:=analyse\_counter+1} \mid \mid
  SELECT 0=0
  \text{THEN} \text{BlankInAnalyser} \mid \mid
  SELECT 0=0
  \text{THEN} \text{PlateInAnalyser} \mid \mid
  SELECT \text{analyse\_counter}<\text{aMaxMoves}
  \text{THEN}
    SELECT astate = aidele \land \text{acmd}=analyse \text{ THEN skip END}

END

END;

END;

AnalyzePosFailed=
BEGIN
  SELECT 0=0
  \text{THEN} astate := asuspended \mid \mid \text{asuspended\_substate:=asuspended1} \mid \mid \text{CheckPosNotOk}(zmax) \mid \mid
  SELECT 0=0
  \text{THEN} \text{BlankInAnalyser} \mid \mid
  SELECT 0=0
  \text{THEN} \text{PlateInAnalyser} \mid \mid
    SELECT \text{analyse\_counter}<\text{aMaxMoves}
THEN
SELECT status = 'idle' \∧ acmd=analyse
THEN
skip
END
6 Implementing Models

This chapter provides guidelines on automatically generating code from B implementations and integrating this with other code.

6.1 Producing Efficient Code Automatically

One of the main advantages of using the B method is automatic code generation. In fact, at the end of the refinement process, the final component is an implementation in a subset of the B language, B0. One of the main questions when starting this formal development was about efficient code generation. We have chosen to develop a simple code translator. The idea is to use it as a prototype to figure out what kind of improvements can be implemented and what kind of improvements are necessary. Note that improvement can be dedicated to a single smart card chip target but we do not consider this point here.

Our translation principles are similar to the conversion scheme of the Atelier-B Tool, with some minor modifications to reduce memory consumption. For example, the constant values are converted to pre-processor directives, instead of variables. Each implemented machine is converted into a C source file and a corresponding C header file. Although only the code contained within the implementation machine generates corresponding C source code, the conversion process needs to take into account the abstract machine and all its refinement steps. For example, concrete variables can be introduced at every refinement step. However, the signature of the operations is fully defined in the abstract machine: this is a B requirement, since an implementation that imports a machine can call operations defined in that machine, but has no knowledge of the implementation details.

It should be noted, however that a user of the converter only needs to provide the implementation to the converter: the implementation machine contains enough information for the converter to retrieve the refinements and the abstract machine.

The header file defines the enumerated sets definitions, the constants definitions and the C functions prototypes. The C source file contains the actual C conversion of the machine’s operations. Although those files are generated for most machines, not every machine is converted to a C source file. For example, context machines that only define constants or enumerated sets are translated into a C header file defining the corresponding constants or enumeration.

6.1.1 Machine Conversion

Although the B0 language is similar to classical imperative languages, an important point when converting machines is that a system may contain multiple instances of a given machine. This makes converting B0 to C similar to converting an object oriented language to C. To allow the use of multiple machine instances, the state of the converted machine is defined within a C structure. The state consists of all the concrete variables, and of the imported machines’ state.
To allow operations to modify a specific machine instance, a parameter corresponding to a pointer to the instance is added to each operation. This pointer is similar to the `this` pointer used by C++ or Java.

Enumerated sets are converted to C `enums`, and the pre-processor is used to define constant values.

Architecture clauses such as `IMPORTS` and `SEES` are all translated using the include directive of the C pre-processor: although this directive does not enforce the access rules of the B clauses, as the B specification complies with those rules, so will the generated code.

Finally, the initialisation clause is treated as a special operation that performs the initialisation of the variables and calls the initialisation function of the imported machines. This function has to be called before any other functions are invoked.

### 6.1.2 Operation Conversion

Converting operations is quite straightforward, since the constructs allowed by B0 match the constructs allowed in C. The conversion involves replacing the B substitutions by the equivalent C instructions and the variables by their C equivalents.

Whereas B operations may have several return parameters, C functions have only one. When an operation has more than one return parameter, the first parameter is returned as the C function return value, and the next return values are passed by reference as pointers to the destination variables. This increases the converter complexity, since the converter has to know how a given variable will be returned. Moreover, for return values passed by reference, the converter has to differentiate occurrences of those variables appearing on left or right sides of substitutions.

Finally, in order to prevent name clashes, all operations names are prefixed with the name of the machine.

A converter prototype has been developed using Java and the JavaCC B parser distributed on [Tatibouet99]. The aim of this converter is mainly to test the feasibility of the approaches considered and it currently only handles a subset of the B language.

Another limitation of this converter is that the verification of the correctness of the optimisations performed has not been studied at this prototype stage. A translator meant to be used in industrial development should be developed so that it could be certified, for example by a Common Criteria evaluation.

### 6.2 Timing Constraints in Code Generation

The objective is a cycle time of the executable code, running on target, less or equal than 330 ms.

First of all, the time performance can depend on the controller, but not on the physical part. And the controller is only a specification, the implementation is made at software level, so the time optimization (if required) is also made at a low software level.
A conclusion of the railway case study is that the controller has very little influence on timing performances of the software, which depends quite entirely on the choices made during the implementation of the software.

### 6.3 Resource Constraints in Code Generation

This section describes some optimisations that can be added to the previous translation scheme in order to reduce the memory consumption of the generated code. The objective of those optimisations is not to replace the optimisations performed by the C compiler, but to address optimisations that the compiler cannot perform on its own, since the converted code would not contain enough information.

Most of these optimisations will require the translation to be performed on a project basis. That is, a machine cannot be converted without knowing how the other machines are converted. The new conversion process is shown in Figure 6.1.

![Figure 6.1: Optimised conversion of a B project.](image)

#### 6.3.1 Smart Card Constraints

Smart cards are devices with severe constraints both for memory size and computing power (The chips used are usually 8-bit chips, and smart card memory rarely exceeds 64 kilobytes of ROM, 64 kilobytes of EEPROM, and 4 kilobytes of RAM). So, although formal methods have shown their efficiency for proof and validation of specification, the code generated from those formal specifications can usually not be embedded within a smart card. This requires either limiting the use of formal methods to specification, or manually translating the specification.

The B method has already been used to specify some smart card operating system components. Although the code automatically generated from B specifications cannot be embedded yet, it is very close to meet smart cards constraints, and could meet them if an optimising code generator is used.
6.3.2 Machine Instances

When a machine has no state on its own, it is possible to suppress the use of the additional instance parameter. We consider that a machine has no state when it does not define any concrete variable, and imports only machines that have no state.

An example of machine without state would be a machine encapsulating an algorithm. Figure 6.2 shows a simple example of such a machine, and its corresponding implementation.

```
MACHINE Average
OPERATIONS
res ← average(val1, val2) =
PRE
  val1 ∈ NAT ∧
  val2 ∈ NAT ∧
  val1 + val2 ∈ NAT
THEN
  res := (val1 + val2) / 2
END
END

IMPLEMENTATION Average_i
REFINES Average
OPERATIONS
res ← average(val1, val2) =
BEGIN
  res := (val1 + val2) / 2
END
END
```

Figure 6.2: A machine encapsulating an algorithm.

A conversion of this machine removing the instance parameter is shown in Figure 6.3. The same strategy can be used to translate machines that are instantiated only once within the project to convert: in this case the pointer used to access multiple instances can be suppressed, since its value will always be the same, and known statically.

```
int average(int val1, int val2) {
  int res;
  res = (val1+val2)/2;
  return res;
}
```

Figure 6.3: C Translation of the Average machine.

It should be noted that, as the operations calling convention changes depending on the way the machines are translated, such an optimisation could not be performed to translate machines without knowledge on how the other machines are converted.

6.3.3 Variables Storage

A simple conversion from B0 to C translates all the integer values into C int. However, it is possible to use smaller integers depending on the range used by the B variables. This allows using the short and char C types to store the variables, and can greatly decrease the memory consumption. For smart cards applications, which mainly use byte and short values, the memory required can be divided by more than two if the adequate type is used.
For example, assuming that the variables of the Range machine are specified as byte intervals (Figure 6.4), the converter could use this information to assign a corresponding C type.

A possible C translation of the getRange operation is provided in Figure 6.5. It assumes that the variables begin and end have been typed as char.

```c
unsigned char getRange(unsigned char *r_max)
{
    unsigned char r_min;
    r_min = begin;
    *r_max = end;
    return r_min;
}
```

Using different C types for storing variables can however introduce memory corruption if not handled carefully: as some results are passed by reference using C pointers, the types of the variables have to be the same.
The Range_User machine presented in Figure 6.6 is a valid B machine that needs a special conversion: as the variables var1 and var2 are natural numbers, the results of the getRange operation can safely be stored inside var1 and var2 without overflow. However, assuming that the variables var1 and var2 are allocated as integer variables, and that the getRange operation is translated as shown in Figure 6.5, passing the address of the variable var2 to the getRange operation would only update one byte of this variable, leading to erroneous results.

As shown in Figure 6.7, a workaround is to use a temporary variable to pass the parameter when calling the operation. The overhead of using temporary variables can however balance the savings obtained by using smaller types if the specification is not written with this concern in mind. So, always allocating the smallest type for every variable may not be a good choice. A better solution would be to perform an analysis of the complete machine set to be translated, and to allocate variable types depending on the number of conversions that will be needed.

```c
int var1, var2;
void useRange()
{
    unsigned char tmp_var2;
    var1 = getRange(&tmp_var2);
    var2 = tmp_var2;
}
```

Figure 6.7: Safe conversion of the useRange operation.

### 6.3.4 Initialisation Translation

Initialisation can often be performed statically instead of using a dedicated initialisation function. This solution allows suppressing all the initialisation code, but is more difficult to implement, since this initialisation operation may have to be interpreted by the converter.

### 6.3.5 Operation Inlining

Inlining an operation call corresponds to replace the corresponding function call by the body of the function.

Operation inlining can be used basically for two objectives. The first one corresponds to the classical use of in-lining. That is inlining simple functions where the inlined code is cheaper than a function call. For example, for simple function that only stores a value in a variable, the cost of the function call can be avoided. Moreover, in such cases, the number of assembly instructions generated is likely to be less than the number of instructions that a function call would generate.

The second objective of inlining is to remove operation calls that have not been introduced to factorise code, but for specification purpose: we have found that, in order to simplify the
specification, or to ease the proof process, specifications are structured using multiple machines and operations.

Although this strong decomposition is usually a good specification practice, it also involves a large number of operation calls. Those calls are costly both in terms of computing resources, and of memory consumption.

Inlining those operations allows the use of a clear and modular specification, without the overhead of the generated function calls. To achieve this, a simple inlining criterion would be to inline all the operations that are called from only one location within the specification.

### 6.4 Integrating Formal and Non-Formal Code

In the smart case study the chip target is an ATMEL platform. First, we provide an implementation on a ATMEL AT90 SC 3232. This chip contains 32 kb for the software, 32 kb for the data and 1.5 kb of RAM. This implementation allows us to demonstrate the feasibility of embedding a formally developed verifier into a smart card. This first implementation embeds a complete type verifier except the *integer* management, and a partial structural verifier where external tests are excluded. This first implementation required 29 kb of compiled code and fits in the ATMEL chip. The code of the verifier is stored in the first 32 kb for the software. Applets are loaded into the 32 kb dedicated to data.

The second implementation we provide is on an ATMEL AT90 6464 C with 64 kb for software, 64 kb for data and 3 kb of RAM. On this platform, we aim to embed the complete verifier, with the *integer* management for the type verifier and all the structural tests. With no specific optimisation, the size of this complete verifier is 45 kb.

The compilation chain provided by ATMEL, which has some very efficient tools such as the compiler, which enable significant space (in terms of code size) savings to be made. Among the different options available for compiling the code for ATMEL, we chose to focus on code size and not yet for code execution efficiency. Our priority is to restrict code size over runtime efficiency, in order to embed the byte code verifier within a smart card.

### 6.5 Integrating Testing and Formal Development

Software development requires a description of the requirements. These requirements generally do not describe the solution, but the needs by using a functional approach. This description mainly concerns the inputs and outputs of the system, and the expected results in some specific circumstances. One of the project leader responsibilities is to define a specification of the solution, and to validate that this solution corresponds to the needs informally expressed. A formal development can then be used, in order to ensure that the product is compatible with its specification.
Considering the Java Card verifier, the verifier requirements are expressed through the Java Card specification, but no verifier specification is given. The first step when developing a Java Card verifier is to write specifications that are supposed to satisfy explicit and implicit needs. Then, a verifier can be developed using a formal development process. The overall process is illustrated in Figure 6.8.

Two major problems can still arise, that are very time and money consuming:

- The requirements can often be ambiguous or incomplete, and an informal validation may not identify this. Thus, although the resulting software conforms to its specification, it will not meet its requirements. One solution is to formalise the software environment, in order to limit the risk of ambiguity or incompleteness. It then becomes possible to generate test suites based on the environment, and to validate software integration within this environment.

- The execution platform does not conform to its specification, generating failures during execution. The software developed does not provide the expected results because of them. In order to detect these errors, the test process has to include not only functional tests but also robustness tests.

The risk of these problems occurring can be reduced using testing in the manner illustrated in Figure 6.9.
6.5.1 Test Generation Automation

Several techniques exist to automate test generation. In a formal development, where a trusted translator automatically generates the code, and when the code is executed on a trusted platform, no conformance testing is required. The last problem is then to check that the application fulfils its environment’s needs. In order to automate these tests, a possible solution is to formalise a model of the environment and to generate tests that check for conformity to the environment behaviour. When the trusted formal chain is incomplete, conformity tests are relevant; as they limit the risk of errors which are generated by the translator or are present in the execution platforms. The process consists in using the formal specifications of the application to generate tests.

Different types of tests can be generated, depending on the kind of application. Applications that are not critical can be tested using only a few tests. One strategy can be to cover each function, each state or each transition of the model only one time. For more critical applications, tests can cover all, or particular, paths in the model.

In any case, the test strategy is strongly dependent on the client needs in term of quality and security. The major difficulty consists in defining the end of the test process, considering that exhaustive test is almost impossible.

6.5.1.1 Example of Flaws

We considered that most of errors have been identified and eliminated after the proof process. If the refinement process ensures that the implementation conforms its specification, it does not ensure that the corresponding formal specification is correct. In fact, when translating informal specification into formal ones, errors may have been introduced. This is due to the misunderstanding of the informal specification by the developer. For this reason, tests are still necessary to make sure that these kinds of errors do not exist. Moreover, as the B code is automatically translated into C code by a tool and that this tool is not “qualified”, it is also necessary to test the C code in order to check that it is correctly generated. These are the two reasons why testing is still necessary. 14 errors have been discovered during the testing step on the type verifier. It is important to note that these errors cannot be discovered by the proof as it concerns the higher specification. Half of errors concern the model itself. That is, a wrong writing of the formal model. 8 errors concern directly the model of the byte code instructions. These errors are due to the misunderstanding of the informal specification provided to the formal developer. It concerns for example the `invokeinterface`, the `invokespecial` or the `athrow` instructions that are not really easy to understand and thus to model. One may not that for example some errors have already been found on instruction such as the `invokespecial` and the `checkcast` during the code review and the proof activity. That enforces the fact that these instructions are really difficult to model and to implement. The proof activity has removed some of implementation errors and the testing phase helps identifying the specification errors.
6.5.1.2  Testing a Verifier: A Manual Test Strategy

An automated test generation process, requires inputs of the application under test to be automatically generated. For the Java Card verifier, the system under test is shown in Figure 6.10.

![Diagram showing applet, Verifier under test, and boolean]

Figure 6.10: Black box testing of a verifier.

In order to automatically generate test suites, the test generation process has to automatically generate different type of test applets. These applets have to be expressed in a specific format in order to be checked by the verifier. In its actual version, this format seems to be too complex to enable any kind of test applet generation (including invalid applets). Another solution could be to generate abstract applets corresponding to concrete applets. Eliciting the different type of applets, valid and invalid, is a process similar to partition analysis but on a very much more complex data type. This could be pertinent if the test process was dedicated to verifier behaviour, with only a few kinds of applets. However, it is exactly the contrary. The verifier is supposed to have no state, focusing on the data only (which in this case is the applet). For all these reasons, the test generation process used in this situation has been manual.

Observe that when the first applet elicitation is performed, it becomes possible to generate automated tests for a verifier, using all the elicited applets as an equivalent of partition analysis.
7 Emerging Complementary Developments

7.1 Event Based Model Decomposition

7.1.1 Introduction

The process of developing an event model by successive refinement steps usually starts with a very few events (sometimes even a single event) dealing with a very few state variables. It usually ends up with many events and many variables. This is because one of the most important mechanisms of this approach consists of introducing new events during refinement steps. The refinement mechanism is also used at the same time to significantly enlarge the number of state variables.

The new events, let us recall, are manifestation of the refinement of the time grain within which we might, more and more accurately, observe and analyse our dynamic system. At some point, we might have so many events and so many state variables that the refinement process might become quite heavy. And we may also figure out that the refinement steps we are trying to undertake are not involving any more the totality of our system (as was the case at the beginning of the development): only a few variables and events are concerned, the others only playing a passive, but noisy, rôle. The idea of model decomposition is thus clearly very attractive: it consists of cutting a heavy event system into smaller pieces which can be handled more comfortably than the whole. More precisely, each piece should be refinable independently of the others. But, of course, the constraint that must be satisfied by this decomposition is that such independently refined pieces could always (in principle) be easily recomposed. This process should then result in a system which could have been obtained directly without the decomposition, which thus appears to be just a kind of “divide-and-conquer” artefact.

This chapter contains the feasibility study of such a mechanism. After proposing an informal definition of decomposition in the next section, we outline the outcome and constraints of this approach (Section 7.1.3). We then present its main difficulty (Section 7.1.4) and propose a solution to it (Section 7.1.5).

7.1.2 Informal Definition

Decomposing an event model \( M \) is a complex process which can be defined in the following way:

1. \( M \) is first “splitted ” (in a certain way to be defined) into several models, say \( N, \ldots, P \).
2. These new models are then be refined independently of each other yielding \( NR, \ldots, PR \).
3. These refined models are eventually “put together” (also in a certain way to be defined) to form yet another model \( MR \).
4. The recomposed model \( \text{MR} \) is guaranteed to be a refinement of \( M \).

It is important to notice here that point 3 above (the recomposition) will never be performed in practice. One has only to figure out that it can be done under certain circumstances and that the refinement condition stated in point 4 (\( \text{MR} \) is a refinement of \( M \)) is then satisfied. The events of the component models \( N, \ldots, P \) of the decomposition are supposed to form a partition of the events of the initial model \( M \). Concerning the way the variables of \( M \) are splitted among \( N, \ldots, P \), the problem is not so simple as we shall see below in section 7.1.4.

7.1.3 Outcome and Constraints of Decomposition

This decomposition process may play three important practical methodological rôles:

1. It is certainly easier (less proofs) to refine (possibly several times) \( N, \ldots, P \) independently of each other rather than together.

2. This process of decomposition is “monotonic” in that refinements of \( N, \ldots, P \) can be further decomposed in the same way, and so on.

3. The models \( N, \ldots, P \) could already possess (off the shelf) some refinements that can then be reused in several projects.

In building this mechanism, we shall observe several “constraints”:

1. We must define a process that is totally robust mathematically speaking.

2. We do not want to modify in any way the mathematical definition and concept of refinement.

3. We figure out that the process of decomposition of an event model is distinct (although close) from that of importation as described in “classical B”.

7.1.4 The Main Difficulty: Variable Splitting

Suppose that we have a certain model, \( M \), with four events \( m_1, m_2, m_3, \) and \( m_4 \). We would like to decompose \( M \) into two separate models:

(1) \( N \) dealing with events \( m_1 \) and \( m_2 \), and

(2) \( P \) dealing with events \( m_3 \) and \( m_4 \).

We are interested in doing this decomposition because we “know” that there are some nice refinements that can be performed on \( m_1 \) and \( m_2 \) (possibly adding some new events) and independently on \( m_3 \) and \( m_4 \) in the same way.

But in doing this event splitting we must also perform a certain variable splitting. Suppose that we have three variables \( v_1, v_2 \) and \( v_3 \) in \( M \). Like the events, the variables must be splitted too. For instance, we might put \( v_1 \) and \( v_2 \) with \( N \) (because \( m_1 \) and \( m_2 \) are supposedly working with them and not with \( v_3 \)), and thus \( v_3 \) goes, quite naturally, with \( P \). But the difficulty here is that \( m_3 \)
and m4, which certainly work with v3, might also work with v2, so that, besides v3, P certainly also requires v2 to deal correctly with m3 and m4.

The problem seems unsolvable since apparently it will always exist some shared variables. As a matter of fact, we have the very strong impression that the splitting of the events will always conflict with that of the variables. Suppose it is not the case. In other words, suppose that, in our example, m1 and m2 only work with v1 and v2, while m4 only does with v3. Clearly then, M is made of two completely separated groups of events (m1 and m2 in one hand, and m3 and m4 in the other) which do not communicate in any way with each other. In this case, M is obviously made of two distinct models, which could have been handled separately. So, in all interesting cases, the question of shared variables, v2 in our example, is unavoidable. How are we going to solve this difficulty?

7.1.5 The Solution: Variable Sharing

7.1.5.1 Shared Variables Replication
We have no choice: the shared variables must clearly be replicated in the various components of our decomposition. Notice that the shared variables in question can be modified by any of the components: we do not want to make any “specialisation” of the components, some of them being only allowed to “read”, and some other to “write” these variables. We know that it is not possible in general.

The new difficulty that arises immediately at this point concerns the problem of refinement. In principle, each component can freely data-refine its state space. So that the same replicated variable could, in principle, be refined in one way in one component and differently in another: this is obviously not acceptable.

7.1.5.2 A Notion of External Variable
The price to pay in order to solve this difficulty is to give the replicated variables a special status in the components where they stay. Let us call this status: external. An external variable has a simple limitation: it must always be present in the state space of any refinement of the component. In other words, an external variable cannot be data-refined.

7.1.5.3 A Notion of External Event
But this is not sufficient. Suppose that in a certain component an external variable is only read, not written. The trouble with that external variable is that it has suddenly become a constant in that component, which is certainly not what we want. What we need thus in each component, is a number of extra events simulating the way our external variables are handled in the initial model. Such events are called external events. Each of them “mimics”, using the external variables only, an event of the initial model that modifies the external variables in question. The reader has
understood: “mimic” simply means “is an abstraction of”. Of course such external events cannot be refined in their component. Notice that there is a distinction to be made between an external variable and an external event. An external variable is external in all sub-models where it can be found, whereas an external event always has a non-external counterpart elsewhere. An event, however, can be external in several sub-models.

7.1.5.4 Final Recomposition
The recomposition of the initial model by means of refinements of the various components is now extremely simple. We put together all the variables of the individual components (“dereplicating” the various shared variables) and we simply throw away all the external events of each component. It remains now for us to prove that the recomposed model is indeed a refinement of the initial one. Notice again that this recomposition is usually not done explicitly. It is just something that could be done, and which must then yield a refinement of the initial model.

7.2 Refining Iterative Programs in the B Method

7.2.1 Introduction
As stated before in the Handbook, there are mainly two ways of exploiting formal methods: the verification \textit{a posteriori} of the correctness of already implemented programs, or the development of “correct by construction” software, starting from very abstract specification and applying correctness preserving transformations until the final implementation is obtained. MATISSE project is related to the later approach, more precisely in the framework of the \textit{B method} [Abrial96]. Starting from an initial \textit{abstract machine} that gives a very high level expression of the user requirements (state variables, invariants, high level specification of the operations), stepwise refinements can be built in \textit{B} to get eventually a concrete machine called the \textit{implementation}. All along this process, the \textit{validity} of the program under construction is guaranteed by formally verifying \textit{proof obligations}. In the commercially available tool Atelier-B, a theorem prover has the role of verifying all these proof obligations.

In this framework, we focus on the \textit{design of iterative programs}. Currently, B supports neither recursion nor iteration; more precisely, recursive functions are forbidden, and iterative forms (e.g., while loops) are only allowed late in the refinement process mainly at \textit{implementation} level. It means that a loop can only be introduced in its definitive form, thus preventing from performing any refinement. The resulting proof obligations are often complicated and often more difficult to verify. In a Matisse working paper and research report [Klenov02], we have proposed a methodology that allows to introduce “while” loops at any refinement step, and to refine further the components of this loop up to its final form. We have given the templates of the corresponding proof obligations, hence giving the possibility of automatically generating them. We will here briefly recall the main results and illustrate them on a case study taken from [Butler00] which will help practitioners to appreciate the use of loops refinement lemmas in the process of constructing iterative programs.
The standard way of reasoning with the B method consists in performing data refinements on the original abstract specification, and then algorithmic refinements. In [Butler00], the authors propose a method that gives the possibility of starting with algorithmic refinements (introduction of complex algorithmic structures), and then refining data types. They define specific “data refinement laws”, one of them is related to “while” loops. This approach only allows data refinement, and it does not take into account loop variants and invariants. Here we consider a more general framework; the main restriction that has to be respected with our method is to treat all the variables involved in the refinement process as state (i.e., global) variables. This approach can be used to extend further the B method with such control structures earlier in the refinement process.

7.2.2 Problem under Consideration in the Framework of the B Method

As usual in the B method [Abrial96], each machine has an identifier; in the example of Figure 7.2, the is name is $M_1$-ballot. The state variables of the machine are listed in a section introduced by the keyword variables (apaper and aballot in this example), their values must satisfy the invariant, stated as a predicate. It is also possible to declare constants, after the keyword constants, and to specify properties that characterize them, using the keyword properties (for instance see the constant make_ballot and its definition in the example). The keyword operations introduces the section in which the operations that can be executed by the machine are given (operation Make_Ballot in the example of Figure 7.2). Each operation can have a precondition. Both the initialisation and the operations are defined using generalised substitutions, which are similar to assignments statements, but have a semantic based on the weakest precondition calculus of Dijkstra: $[GS]E$ represents the weakest precondition under which the generalized substitution $GS$ is guaranteed to terminate and establish $E$. A while loop can be introduced in any operation at the last level of refinement, called implementation. Such a loop has the form:

$$T; \text{WHILE } P \text{ DO } GS \text{ INVARIANT } I \text{ VARIANT } V \text{ END}$$

$T$ is the initialisation, $P$ is the condition, $GS$ is the generalized substitution that corresponds to the loop body, $I$ is the loop invariant, and $V$ is the variant. The “loop proof rule” below is associated with the correctness of such a loop:

$$\begin{align*}
[T] & \quad I \\
\forall x : (I \land P \Rightarrow [GS] I) & \quad \text{(initialisation)} \\
\forall x : (I \Rightarrow V \in N) & \quad \text{(typing)} \\
\forall x : (I \land P \Rightarrow [w := V] [GS] (V < w)) & \quad \text{(termination)}
\end{align*}$$
∀ x : ( I ∧ ¬P ⇒ R )
⇔ [ T ; WHILE P DO GS INVARIANT I VARIANT V END] R

The problem that we consider, referred to as \((Pb_0)\), is illustrated in its simplified form in Figure 7.1. We start from a machine \(M_1\), where the operation \(\text{op}\) can have a precondition \(Q_1\) and its body is referred as \(W_1\). At any refinement step \(M_{n-1}\) \((n – 1 \geq 1)\), an iterative computation can implicitly be specified in the body \(W_{n-1}\) of the operation \(\text{op}\), and at the next refinement \(M_n\), the loop is explicitly introduced in the body \(W_n\) of this operation (this constitute an algorithmic refinement). Then the subsequent refinements can transform this while loop, by modifying any of its components \(T_n\), \(P_n\), \(GS_n\), \(I_n\) or \(V_n\). The corresponding transformations can be either data or algorithmic refinements.

### Machine \(M_1\)
- **Properties**: \(PROP_1\)
- **Invariant**: \(INV_1\)
- **Operations**:
  - \(\text{op} = \) PRE \(Q\) THEN \(W_1\) END

### Refinement \(M_n\)
- **Refines**: \(M_{n-1}\)
- **Properties**: \(PROP_n\)
- **Invariant**: \(INV_n\)
- **Operations**:
  - \(\text{op} = \) PRE \(Q_n\) THEN \(T_n\) WHILE \(P_n\) DO \(GS_n\) INVARIANT \(I_n\) VARIANT \(V_n\) END END

### Refinement \(M_{n+1}\)
- **Refines**: \(M_n\)
- **Properties**: \(PROP_{n+1}\)
- **Invariant**: \(INV_{n+1}\)
- **Operations**:
  - \(\text{op} = \) PRE \(Q_{n+1}\) THEN \(T_{n+1}\) WHILE \(P_{n+1}\) DO \(GS_{n+1}\) INVARIANT \(I_{n+1}\) VARIANT \(V_{n+1}\) END END

**Figure 7.1: Refinement with while loops.**

Problem \((Pb_0)\) is the simplest case of loop refinement. A more elaborate case, called \((Pb_1)\) in the following, is the one on which the “while” loop is followed by the assignment of a global variable:

MATISSE/WP1-D10/April03
\[ W_{n+1} = T_{n+1}; \text{ WHILE } P_{n+1} \text{ DO } GSn+1 \text{ INARIANT } I_{n+1} \text{ VARIANT } V_{n+1} \text{ END; } x_{n+1} = \text{EXP}_{n+1} \]

\[ W_n = T_n; \text{ WHILE } P_n \text{ DO } GS_n \text{ INARIANT } I_n \text{ VARIANT } V_n \text{ END; } x = \text{EXP}_n \]

In the original problem, the operation \( \text{op} \) returns no result, it is a procedure. Another variant of \( (Pb_0) \) corresponds to the case where \( \text{op} \) returns a result, it is a function. We will call this problem \( (Pb_2) \):

In refinement \( M_{n+1} \):

\[ \text{res} \leftarrow \text{op} = \text{PRE } Q_{n+1} \text{ THEN } W_{n+1} \text{ END} \]

In refinement \( M_n \):

\[ \text{res} \leftarrow \text{op} = \text{PRE } Q_n \text{ THEN } W_n \text{ END} \]

We can remark that it is often the case in \( (Pb_2) \) that \( W_{n+1} \) and \( W_n \) are instances of the ones of \( (Pb_1) \), where \( x_{n+1} = x_n = \text{res} \) (in that case, it does not have the status of global variable).

### 7.2.3 Proof Obligations for the Loop Refinement

In [Klenov02] we have given the minimal sets of proof obligations that are needed to verify that \( M_{n+1} \) correctly refines \( M_n \) (denoted \( M_n \preceq M_{n+1} \) below) for the machines of Figure 7.1, in the case of independent refinements, and in the case where the five components of the loop are simultaneously transformed. In both cases, the minimal set of proof obligations has been produced thanks to an inductive reasoning based on the following rule:

\[
\text{PO}_1, \forall n, n-1 \geq 1, \ (M_{n-1} \preceq M_n \Rightarrow M_n \preceq M_{n+1})
\]

\[
\forall n \geq 1, \ M_n \preceq M_{n+1}
\]

where \( \text{PO}_1 \) denotes the conjunction of the proof obligations for the validity of \( M_1 \) and for the correctness of the first refinement. In the case where the five components of the loop are refined simultaneously, the following lemma (proofs are omitted here) gives seven sufficient proof obligations.

**Lemma 1.** To prove that \( W_{n+1} \) correctly refines \( W_n \), the proof obligations are:

\[
\bigwedge_{i=1}^{n+1} \text{HYP}_i \land Q_i \land P_{i+1} \land I_{n+1} \land [w:=V_n][GS_n](V_n < w) \Rightarrow [w:=V_{n+1}][GS_{n+1}](V_{n+1} < w)
\]  

(1)

\[
\bigwedge_{i=1}^n \text{HYP}_i \land Q_i \land \text{HYP}_{n+1} \land P_{n+1} \land I_{n+1} \land [GS_n]I_n \Rightarrow [GS_{n+1}]I_{n+1}
\]  

(2)

\[
\bigwedge_{i=1}^n \text{HYP}_i \land Q_i \land \text{HYP}_{n+1} \land P_{n+1} \land I_{n+1} \Rightarrow P_n
\]  

(3)

\[
\bigwedge_{i=1}^n \text{HYP}_i \land Q_i \land \text{HYP}_{n+1} \land \neg P_{n+1} \land I_{n+1} \Rightarrow \neg P_n
\]  

(4)
\[ \bigwedge_{i=1}^{n} HYP_i \land Q_1 \land HYP_{n+1} \land I_{n+1} \land V_n \in N \Rightarrow V_{n+1} \in N \]  
(5)

\[ \bigwedge_{i=1}^{n} HYP_i \land Q_1 \land HYP_{n+1} \land [T_n]I_n \Rightarrow [T_{n+1}]I_{n+1} \]  
(6)

\[ \bigwedge_{i=1}^{n} HYP_i \land Q_1 \land HYP_{n+1} \land I_{n+1} \Rightarrow I_n \]  
(7)

For the problem \( (Pb_1) \), the proof obligations above have to be verified, where (4) is replaced by:

\[ \bigwedge_{i=1}^{n} HYP_i \land Q_1 \land HYP_{n+1} \land \neg P_{n+1} \land I_{n+1} \Rightarrow \neg P_n \land [x_{n+1}:=\text{EXP}_{n+1}] [x_n := \text{EXP}_n] \text{INV}_{n+1} \]  
(8)

For the problem \( (Pb_2) \), (4) becomes:

\[ \bigwedge_{i=1}^{n} HYP_i \land Q_1 \land HYP_{n+1} \land \neg P_{n+1} \land I_{n+1} \Rightarrow \neg P_n \land \text{EXP}_n = \text{EXP}_{n+1} \]  
(9)

### 7.2.4 Case Study

This example, taken from [Butler00], is slightly different from the original one, in particular from the point of view of variants and invariants (see Figure 7.2). We also introduce a new variable \( \text{aballot2} \) in the first refinement \( M_2\text{-ballot} \) in order to use the (Finalisation) part of theorem 1 to validate the behaviour of the first while loop with respect to the specified \( \text{make_ballot} \) function. The input to the system is a collection of votes, that contains errors. The example specifies a preprocessing of the raw input that retains only valid votes (called ballots); this operation is called \( \text{Make_Ballot} \), see Figure 7.2. A paper is a partial function from Candidate to \( \mathbb{N} \) (type \( \text{Paper} \)). Valid preferences on a paper are that set of preferences that are unique, contiguous and start at 1. Duplicate preferences are disregarded as are all higher preferences on the paper. A skip in preferences invalidates all preferences after the skipped one. It may happen that a paper has no valid preferences, in which case the resultant ballot is empty.

Figure 7.3 and Figure 7.4 show the introduction of the “while” loop to refine the body of the operation \( \text{Make_Ballot} \), and the refinement of this loop. In the second part of the case study, another “while” loop is introduced and then refined to implement the computation variable \( \text{BB} \). Our set of proof obligations allows to generate the minimal set of theorems needed to validate the operation \( \text{Make_Ballot} \) of \( M_3\text{-ballot} \) is a correct refinement of its counterpart in \( M_2\text{-ballot} \), and similarly for the algorithm related to \( \text{BB} \). All these proof obligations have easily
been discharged using Atelier B. A prototype tool is being implemented in Perl to mechanise the generation of these lemmas, given the B sources for the machines under construction (for instance the ones depicted in Figure 7.3 and Figure 7.4).

**MACHINE** M1-ballot

**CONSTANTS** make_ballot, no_cands

**PROPERTIES**

\[
\begin{align*}
\text{no_cands} & \in \mathbb{N} \land \\
\text{make_ballot} & \in \text{Paper} \rightarrow \text{Ballot} \land \\
\forall \text{paper} \cdot (\text{paper} \in \text{Paper} \Rightarrow \text{make_ballot} (\text{paper}) = \\
1 & \ldots \min \{\{\text{nn} \mid \text{nn} \in 1..\text{no_cands} + 1 \land \\
\text{card} (\text{paper}^{-1}[\{\text{nn}\}] \neq 1\}) - 1\} < \text{paper}^{-1})
\end{align*}
\]

**VARIABLES** apaper, aballot

**INVARIANT**

\[
\begin{align*}
apaper & \in \text{Paper} \land \\
aballot & \in \text{Ballot}
\end{align*}
\]

**OPERATIONS**

\[
\text{Make_Ballot} = \\
\text{BEGIN} \\
\text{aballot} := \text{make_ballot} (\text{apaper}) \\
\text{END}
\]

**END**

*Figure 7.2: Specification of Make_Ballot.*

**REFINEMENT** M2-ballot

**REFINES** M1-ballot

**VARIABLES** K, so_far, aballot2

**INVARIANT**

\[
\begin{align*}
\text{aballot2} & = \text{aballot} \land \\
K & \in \mathbb{N} \land \\
\text{so_far} & \in 1..K \land
\end{align*}
\]
\[
K = \min \{ nn | nn \in 1..no\_cands + 1 \land \text{card (apaper}\^{-1}([{nn}]) \neq 1\}
\]

**OPERATIONS**

\[
\text{Make\_ballot} =
\]

\[
\begin{align*}
\text{BEGIN} \\
\text{so\_far} := 1; \\
\text{aballot2} := [ ]; \\
\text{WHILE} \text{so\_far} < K \text{ DO} \\
\text{aballot2 (so\_far)} := \text{apaper}\^{-1}(\text{so\_far}); \\
\text{so\_far} := \text{so\_far} + 1;
\end{align*}
\]

**INVARIANT**

\[
\text{aballot2} = \{nn \mapsto cc | nn \in 1..\text{so\_far} - 1 \land cc \mapsto nn \in \text{apaper}\}
\]

**VARIANT**

\[
K - \text{so\_far}
\]

**END**

**END**

*Figure 7.3: First refinement.*

---

**REFINEMENT** \( M_3\)-ballot

**REFINES** \( M_2\)-ballot

**VARIABLES**

\[
\text{BB, cballot}
\]

**INVARIANT**

\[
\text{cballot} = \text{aballot2} \land \\
\text{BB} = \{1..\text{no\_cands} \times ((\bot, 0)) \} \leftarrow (\text{ii} \mapsto (\text{cand, no}) | \text{ii} \in \text{ran(\text{apaper})} \land \\
\text{cand} \in \text{apaper}\^{-1}([\text{ii}]) \land \text{no} = \text{card (apaper}\^{-1}([\text{ii}]))
\]

**OPERATIONS**

\[
\text{Make\_Ballot} =
\]

\[
\begin{align*}
\text{BEGIN} \\
\text{so\_far} := 1; \\
\text{cballot} := [ ];
\end{align*}
\]
7.3 Vulnerability Analysis

7.3.1 Smart Card Certification Process

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 focuses mainly on the first part of the lifecycle: requirements, specifications, design, development and test. The CC takes 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 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, \textit{i.e.}, the assessment whether they are correctly implemented,
- The confidence in the effectiveness of the security functions, \textit{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 \textbf{formal} model of the TOE security policy and a \textbf{semiformal} presentation of the functional specification and high-level design and a \textbf{semiformal} demonstration of \textit{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.

**Covert Channel Analysis**

The covert channel analysis allows the existence of exploitable channels to be discovered. These covert channels can be classified into two types:

- storage channels, and

- timing channels.

Although fundamentally the same, storage channels differ in the way that information is encoded. In a storage channel, there is a shared global variable in the system that acts as the medium for information transfer. A timing channel requires the ability to reference a real time clock. If the receiver detects a timing difference in code execution, it can infer information. Therefore, information may be transferred. A covert channel can exist in the smart card if between two (or more) applets the two subject have the potential to communicate (through the shared interface) and if this communication is not allowed under the non-discretionary security policy.

For this purpose we have developed a tool and a methodology for automatically analysing this potential illegal information flow as described in [Bieber99] and [Bieber00]. The PACAP project\(^{15}\) aimed to check Java Card applet interaction. More precisely, it checks the data flows between objects on the card by static analysis prior to applets downloading, for a given configuration. Information flows between applets that share data through shareable interfaces are verified. The shareable interface is the means to transfer information from one applet to another in Java Card. A shareable interface can be called by an applicative command or an external call. In all the interactions, we check if the level associated with all the variables (system and application) does not exceed the allowed sharing level that defines the security policy. The tool verifies automatically if a set of applications correctly implements a given security policy, as proposed in [Girard99].

An application is composed of a finite number of interacting applets. Each applet contains several methods. For efficiency reasons, the number of applets and methods analysed when a new applet is downloaded or when the security policy is modified are limited. The method to verify the security property on the application byte code is based on three elements:

- **Abstraction**: all values of variables are abstracted by computed levels,
- **Sufficiency**: an invariant that is a sufficient condition of the security property is verified;
- **Model checking**: this invariant is verified by model checking.

The abstraction mechanism and the invariant definition have been described in [Bieber00]. The tool needs as inputs, a representation of the lattice and the configuration (i.e., the set of applets).

---

\(^{15}\) The PACAP project was partially founded by MENRT contract number 98B0252.
With this information the tool transforms automatically the byte code into a formal semantics, adds the relevant invariant and performs the verification of the invariant.

In the example described in the previous reference, a quarter of the properties are verified within 5 seconds and 90% are verified with less than 10 minutes. One property required 23 minutes to be checked on a Sun Ultra 80. Within this tool it is possible to provide evidence to the evaluator that the applet respects the security policy. If a property cannot be validated, a counter example is given, which need to be analysed. In fact not all the counter examples express a real covert channel due to the abstraction that have been made.

This tool provides an efficient means that covers the EAL 7 requirement: “The analysis documentation shall provide evidence that the method used to identify covert channel is systematic”.

**Vulnerability Analysis**

At the highest level (EAL7) it is not required to use formal methods. But as demonstrated in the covert channel analysis, any form of automation saves time. One of the requirements is to provide a justification that the analysis completely addresses the TOE deliverables. The evaluator will use the analysis that the vulnerabilities have been addressed but will also perform an independent vulnerability analysis. Vulnerability analysis deals with the threats that a user will be able to discover flaws that allow unauthorised access to resources (e.g., data). Such an attack allows the authorised capabilities of other users to be interfered with. The vulnerability analysis enables checks to be made on the coherence of the security policy and the enforcement of this security policy in the product implementation. It requires first an informal model of the interactions that take place between applets and between the selected applet and the terminal. The vulnerability analysis consists in examining all the application responses. The set of this responses are called traces. The analysis of these traces determines whether data is accessible or not. To perform this analysis, we have to model inference rules that represent dependence and the security policy. The FDR model checker [FDR97] is used to check properties over the model by examining the set of traces, looking for illegal transactions or private data that are publicly accessible. If data is accessible, we determine if it is in accordance with the security policy.

**7.3.2 Motivation**

The B refinement method produces powerful results; we have a guarantee, through mathematical proof, 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 identify that mistake.
Consider the Needham-Schroeder Public Key Authentication Protocol [Needham78]. The protocol aims to establish mutual authentication between two parties, \textit{initiator} and \textit{responder}, and uses \textit{public key cryptography} in order to do this. The protocol was verified as secure using a variety of techniques, including the \textit{BAN} logic – a logic for reasoning about “belief” of an entity about the system (or context) it is in. But there are multiple attacks upon this protocol [Lowe96], the most well known involving message replay. In the case of the \textit{BAN} logic the rules for reasoning about belief contained some subtle assumptions, which led to a \textit{BAN} model of the protocol system (\textit{i.e.}, the protocol and its context) that was subtly flawed. Hence the proof did not correspond to the protocol as it would operate within its real world context.

If the Needham-Schroeder protocol was written as a B specification (easily achieved since protocols are a form of specification), the process of refining it to code could not have identified the security flaw. It is this limitation on the B method that we address here.

### 7.3.3 Requirements of the Specification

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 should not do, rather than what it should. For example:

\begin{quote}
\textit{“The system should not reveal the contents of the transaction table to anyone other than the administrator”},
\end{quote}

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 [O’Halloran99]. The technique described here facilitates the identification of critical properties of systems, and is designed to be used in conjunction with, and to compliment, the B method.

### 7.3.4 Vulnerability Analysis

The aim is to identify the critical properties of complex systems (\textit{i.e.}, the properties that the corresponding specifications must ensure). Such systems can be viewed as a collection of communicating components. The range of behaviours such systems exhibit is a direct result of the variety of communications that components can enter into, the possible interleaving of such events, and the resulting states of the system. There can be so many different sequences of communications 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 are 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 vulnerability analysis.

The process algebra CSP [Hoare85, Roscoe98] 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 [FDR97]. 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 [O'Halloran99] to identify if, when, and how a hazard could arise. In the next section a CSP/FDR capability modelling analysis technique is described.

### 7.3.5 CSP/FDR Capability Analysis

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 is now illustrated by the example in the next section, where the formal CSP statements are written in a typewriter font, and the keywords are in bold. For a brief overview of the CSP syntax refer to Appendix B; and for further details refer to [Roscoe98,FDR97].

7.3.6 Simplified Smart Card CSP/FDR Capability Analysis Example

7.3.6.1 Interface / Syntax Specification

The simplified example presented here was inspired by the GemPlus smartcard case study. Here we consider the context to be an abstract Java smart card, which contains one purse applet and three loyalty applets, which are uniquely identified by their names. Note that it is useful to have additional no applet identifier (no_aid) for the modelling of the applet interfaces.

\[
\text{datatype AID = purse | loyalty1 | loyalty2 | spy | no_aid}
\]

\[
\text{LoyaltyAID = \{ loyalty1, loyalty2, spy \}}
\]

In order for the applets to interact they send messages to each other over specified interfaces, namely the purse-to-loyalty interface, the loyalty-to-purse interface, and the loyalty-to-loyalty interface. Given that PurLoyMsg, LoyPurMsg, and LoyLoyMsg represent the messages they are allowed to send on these interfaces respectively, then the following CSP models the interface description, where the first argument represents the source of the message, the second the destination, and the third the content.
channel pur_loy_int : AID . AID . PurLoyMsg
channel loy_pur_int : AID . AID . LoyPurMsg
channel loy_loy_int : AID . AID . LoyLoyMsg

The following diagram illustrates the model to date.

![Diagram](image)

Figure 7.5: Simple CSP case study – structural overview.

The following table summarises the messages, in this case commands and their responses, which can be sent over these interfaces.

<table>
<thead>
<tr>
<th>Cmd. Name</th>
<th>var:Type</th>
<th>Command Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_full</td>
<td></td>
<td>The purse sending a command to indicate to the registered loyalties that its transaction database is full.</td>
</tr>
<tr>
<td>get_next_trn</td>
<td>1:LoyaltyAID</td>
<td>A loyalty sending the get next transactions command for loyalty 1. Note that for the purposes of modelling a transaction can be represented by the loyalty it is associated with.</td>
</tr>
<tr>
<td>get_next_trn_r</td>
<td>r:AID</td>
<td>The get next transaction command returns a result r, which is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none if either there is no next transaction record for loyalty 1, or this action is refused for some reason</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 otherwise</td>
</tr>
<tr>
<td>avail_funds</td>
<td>1:LoyaltyAID</td>
<td>A loyalty 1 sending an “available funds” command to another loyalty.</td>
</tr>
<tr>
<td>avail_funds_r</td>
<td>r:Set(AID)</td>
<td>The available funds command returns a result r, which is a loyalty-indexed set of funds, where the indexes are used to identify how much each loyalties has contributed. For the purposes of modelling this can be abstracted to the set of applet identifiers that contribute to the funds.</td>
</tr>
</tbody>
</table>

Table 5: Simple CSP case study – message overview.

The information in the above table is modelled formally by the following CSP, where each command is represented by a pair \((c:CmdName, v:Value)\), where Value is either an AID (or LoyaltyAID), Set(AID), or None (representing a void, null, or missing value). For example, the pair \((\text{get}\_\text{next}\_\text{trn}, \text{aid\.loyalty1})\) represents issuing the get_next_trn command with
argument \( \text{loyalty1} \); its response is represented by the pair \((\text{get\_next\_trn\_r}, \text{aids.r})\), where \( r \) is either \( \text{none} \) or \( \text{loyalty1} \).

**Datatype**

\[
\text{CmdName} = \begin{cases} 
\text{log\_full} \\
\text{get\_next\_trn} | \text{get\_next\_trn\_r} \\
\text{avail\_funds} | \text{avail\_funds\_r} 
\end{cases}
\]

\[
\text{Value} = \text{aid.AID} | \text{aids.Set(AID)} | \text{none}
\]

**Nametype**

\[
\text{None} = \{ \text{none} \}
\]

**Nametype**

\[
\text{Command} = (\text{CmdName}, \text{Value})
\]

Continuing the previous example, the \text{spy} loyalty applet might want to pretend to be the \text{loyalty1} applet, in which case it would issue the spoofed message \text{loy\_pur\_int.spy.purse.(get\_next\_trn, loyalty1)}, with an expected response of \text{loy\_pur\_int.spy.purse.(get\_next\_trans\_r, aid.r)}, where \( r \) is either \( \text{none} \) or \( \text{loyalty1} \).

Now that the main interfaces to the system have been defined, it is necessary to model their semantics by a set of inference rules. An inference rule is represented by a triple \((n, X, Y)\), where \( n \) represents the rule’s unique name\(^{16}\), \( X \) represents the set of capabilities required for the

\[16\] Uniqueness of rule names simplifies the techniques required to model some forms of fault injection.
rule to fire, \( Y \) represents the set of capabilities gained from firing the rule. The inference rules, which represent the abstract semantics of the case study now follows, where each operation – a command with its response (if it has one) – is dealt with in turn.

First, the “log full” operation is considered. When the log_full command is issued, it enables the loyalty interfaces of each of the loyalties that have registered for the log full event; the idea being that the loyalties have a chance to check the transaction table, before entries that have not already been processed are overwritten. However, once a loyalty has been activated it may perform any of its operations. This is formally modelled by enabling the get_next_trn and avail_funds commands of that loyalty. Further, as the set of rules associated with the “log full” operation is dependent on the set of registered loyalties, the formal model of the “log full” rules (LogFullRules) is parameterised by this set of loyalties (RegLoys). Hence, the set of inference rules that are associated with the log full operation is:

\[
\text{LogFullRules}(\text{RegLoys}) = \\
\{ \text{log_full_op.1}, \\
\{ \text{pur_loy_int.purse.1.(log_full, none)}, \\
\{ \text{loy_pur_int.1.purse.(get_next_trans, aid.13),} \\
\text{loy_loy_int.1.12.(avail_funds, aid.13)} \mid 12 \leftarrow \text{LoyaltyAID} \\
\text{13} \leftarrow \text{LoyaltyAID} \} \\
\} \mid 1 \leftarrow \text{RegLoys}
\]

where the rule name log_full_op.1, is informally speaking\(^\text{17}\) the name that is constructed by replacing \( i \) by a member of RegLoys.

Next, the rules associated with the “get next transaction” operation are discussed. Here the intention is that a loyalty can get the next unprocessed transaction from the purse applet’s transaction list, by supplying the purse with its applet identifier. However, before scanning the transaction list, the authenticity of the request is checked by ensuring that the applet identifier belongs to the requesting applet. This is formally modelled by a special capability known as self.11.12, which enables an applet 12 to “authentically” claim that it is applet 11.

\[
\text{channel self: AID . AID}
\]

Now the “get next transaction” rules can be formally defined by the GetNextTrnRules set, where each “get next transaction” command returns either the found transaction (12) or the empty transaction (aid.no_aid). Further, the empty transaction is returned in two distinct circumstances: (1) for requests that fail authentication; and (2) for requests that have no unprocessed transaction within the purse’s transaction list.

\(^{17}\) It is better to delay the formal definition of the set of rule names until all the rules have been discussed.
GetNextTrnRules =
let
calc(l, l2) = ( if l == l2 then { l2, no_aid } else { no_aid } )
within
{ ( get_next_trn_op.l1.l2.l,  
  { loy_pur_int.l1.purse.(get_next_trans, aid.l2), self.l1.l1 },  
  { loy_pur_int.l1.purse.(get_next_trans_r, aid.r) | r <- calc(l, l2) }  
) |  
11 <- LoyaltyAID,  
12 <- LoyaltyAID,  
1 <- LoyaltyAID }

Next, the rules for the “available funds” operation are discussed. Here the intention is to enable a 
loyalty to calculate the amount of funds that are available to it; that is its own funds, and the 
funds available to its co-operating loyalties. Thus, the rules associated with the “available funds” 
operation are dependent on the co-operation loyalty relation (CoOpLoy). These rules are formally 
modelled by the following CSP where: the operation is only enabled if the loyalty can 
authentically claim to be who it says it is; and the result is calculated by the avail_funds_calc 
function (which performs the appropriate recursive co-operating loyalty available funds 
lookup).18

AvailFundsRules(CoOpLoy) =
{ ( avail_funds_op.l1.l2.l,  
  { loy_loy_int.l1.l2.(avail_funds, aid.l), self.l1.l1 },  
  { loy_loy_int.l1.l2.(avail_funds_r, aids.avial_funds_calc(l,l2,CoOpLoy)) }  
) |  
11 <- LoyaltyAID,  
12 <- LoyaltyAID,  
1 <- LoyaltyAID }

Having defined all the rules associated with each operation, it is now possible to provide a formal 
description of the set of rule names, where:

• the rules associated with the “log full” operation are parameterised by the registered loyalty 
  identifier;
• the rules associated with the “get next” transaction operation are parameterised by the actual 
  and claimed source loyalty identifiers, as well as another loyalty identifier used for 
  authentication;
• the rules associated with the “available funds” operation are parameterised by the actual 
  source and destination loyalty identifiers, as well as the claimed source loyalty identifier.

18 The details of the “available funds calculation” operation are fairly straightforward but lengthy, due to the lack of 
built-in support for relations within the CSP/FDR language.
The second type of critical property is that a loyalty should never be able to get funds from a loyalty that it does not co-operate with. In other words, the returned available funds to a loyalty $l$, as represented by a set of loyalty identifiers $L$, should not contain a loyalty identifier that does not belong to $l$’s set of co-operating loyalties $co_op_loy(l, CoOpLoy)$, where $CoOpLoy$ represents the co-operating loyalty relationship. This is formally represented by the following CSP, where it is formally necessary to insist that a loyalty can co-operate with itself.

$AvailFundsCritInfo(CoOpLoy) = \{ \text{loy_loy_int.} l. l1.(avail_funds_r, aids.L) \mid l \leftarrow \text{LoyaltyAID},
\quad l1 \leftarrow \text{LoyaltyAID},
\quad \text{diff}(L, \text{union}({l}, co_op_loy(l, CoOpLoy))) \neq \{\} \}$

The information required for the vulnerability analysis is almost complete. All that remains is to:

1. choose an appropriate set of initial capabilities;
2. specify the set of registered loyalties (default all loyalties);
3. specify the co-operating loyalty relation;

4. link the resulting specification in with one of the general purpose CSP inference engines.

First, point 4 in the above list is addressed. The case study has been specified in a manner that enables it to be used with an existing general purpose CSP inference engine, which takes a collection of rules, some starting capabilities, and a set of misuse (hazardous) capabilities, and determines whether it is possible to reach a misuse capability. If it is possible to reach a misuse capability then the minimal sequence of inferences that led to that misuse can be displayed. Note that more advanced versions of the inference engine are able to cope with mode switching and interaction with external processes, such as for interacting with a state based model of the environment.

Next, points 1 and 2 are addressed. Recall that the “log full” operation enables all the operations of the registered loyalties; thus, if all loyalties are registered, then enabling the log full operation is sufficient to enable all of the operations within the case study. In addition to this, each loyalty applet should be able to authentically claim to be itself, so self.l.1 is also enabled for all loyalty applets with applet identifier l.

Last, point 3 is addressed. This is slightly more complex as there is no obviously correct choice. For the purposes of discussion, two relations will be examined, namely the empty relation, where no loyalty co-operates with any other loyalty, and the relation CoOpLoy_rel where both the spy and loyalty2 loyalties co-operate with loyalty1, but not each other.

CoOpLoy_rel = { (spy, loyalty1), (loyalty1, spy),
    (loyalty1, loyalty2), (loyalty2, loyalty1) }

When the inference engine analysis is run with the above CoOpLoy_rel co-operating loyalty relationship, the inference engine identifies a misuse, which states that the spy can discover loyalty2’s available funds, even though it is not in its set of co-operating loyalties (i.e., loyalty1). This results from the transitive nature of the calculation of the available funds operation, where the funds available to loyalty1 are also available to the spy (and loyalty2). Having identified the problem in the design, there are a number of solutions. For example, a simple restriction that the co-operating loyalties must be transitively closed is sufficient. Alternatively, the available funds calculation could be updated so that it only returns the funds of the loyalty that was explicitly asked. For the purposes, of the case study the first option is taken, as this requires no rewriting of the rules.

The second configuration of the inference engine, namely with the empty co-operating relation, is trivially transitively closed, and thus should not suffer from the transitive misuse problem identified in the previous paragraph. Indeed, when this configuration of the case study is analysed (model checked by FDR) no misuse can be found, and thus the design is not inherently “hazardous”. Having determined that the design is not inherently “hazardous”, how robust is it to faults in its implementation and use. The existing inference engines already have some ability to answer these questions, via two fault injection techniques: one which essentially adds n additional
capabilities chosen from a faulty-capability set; and the other, which removes individual “firing”
conditions from the rule. Following some experimentation with the fault sets, it quickly became
established that providing the spy with the capability to authentically pretend to be another
loyalty was critical; that is the spy could claim to be a different loyalty and pass the
authentication check. For example if \texttt{self.spy.loyalty1}, is added to the initial set of
capabilities, then it can quickly (by 2 rules) be inferred that the \texttt{spy} can get hold of a \texttt{loyalty1}
transaction as represent by the \texttt{loy_pur_int.spy.purse(get_next_trn_r, aid.loyalty1)}
capability. Further, it was established that the rules that check authentication were also vulnerable
to the same type of failure, but purely for that rule.

Before leaving the case study, it should be noted that care must be taken in the construction of the
rules, and fault models, in order to make the CSP/FDR vulnerability analysis tractable. Current,
research activities are examining design patterns that aid in this task.

7.3.7 Summary

Once the correct specification has been constructed, using CSP, it can be incorporated into the B
model. Using the techniques described elsewhere in this 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 code development cycle.

It isn’t obvious how one would use 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. Our effort is
focused on constructing the models, as it should be, and not on finding a way to make the B
formalism meet our needs.
8 Conclusions

8.1 Summary
This handbook is intended to provide work practices and guidelines to software practitioners in using a formal development methodology. This methodology is centred around the B method, and it provides guidelines on how to use B (or a combination of B with other notations) in the various stages in the development lifecycle:

- **Modelling notations**: Besides presenting the B method, this section describes: event B that extends B to take into account abstract systems, and; the UML+B tool that converts UML class diagrams and state charts into B.

- **Constructing models**: In this section we present two approaches on how to model a system: in the system-level model one constructs a formal model of the overall system; while in a computational model one constructs only a formal model of the part of the system we want study, and then verify that it interacts correctly with the overall system.

- **Analysing models**: This section shows different ways in which formal models may be analysed for validity and soundness. We start by presenting a procedure for the review of B models by domain experts. Next, we show how B can be used to verify safety and identify vulnerabilities. Last, we describe strategies to prove (within Atelier-B) *proof obligations*, and in this way validate a model.

- **Refining models**: This section describes how an abstract model can be successively refined until a correct system implementation is obtained. For complex models, in addition to refinement is necessary to use decomposition - where the overall system is partitioned into several sub-systems. Furthermore, this section gives guidance on how to prove refinement and how to refine UML+B diagrams.

- **Implementing models**: This section provides guidelines on automatically generating code from B implementations. Moreover, it discusses the generation of code that must comply with timing and resources constraints. Another important issue discussed is the integration of formal code with existing non-formal code.

The MATISSE project aims to exploit and enhance existing generic methodologies and associated technologies that support the correct construction of software-based systems. In particular, a strong emphasis was placed on the use of the B Method, and on integrating B with other methods, namely UML and CSP. The techniques developed by MATISSE were driven by three major industrial case studies: transportation, healthcare, and smart card.
8.2 Related Work

MATISSE provides a methodology on how to use formal methods, in special the B method. The methodology proposed in this handbook covers all the steps of a typical development cycle. Furthermore, it provides examples and guidelines on how to formally specify complex systems, as within this project three industrial case studies were developed.

In [Ciapessoni99] the authors describe their experiences with introducing formal methods within an industrial setting. Its argued in the paper that instead of trying persuade the industry to use formal methodologies it is more acceptable to extend informal methods with formal notations. They illustrate this approach by briefly describing the informal and formal notations involved, and giving a number of examples from real-life applications. The approach presented in [Ciapessoni99] is very limited as they only consider the formalisation of the informal requirements, the authors do not consider the remaining steps of the development cycle.

The DES (Dependable Embedded Systems) roadmap [DES03] intends to define future challenges within the application areas of embedded systems. DES is part of the AMSD (Accompanying Measure System Dependability) project [AMSD03] that will provide an overall roadmap in system dependability. Two of the MATISSE case studies are within the application areas of DES: the railway system is in the transport area, and the healthcare is in the medical electronics area. From the several challenges identified by DES two of them were address within the MATISSE project. The first one, formal verification and validation was used in all MATISSE case studies. The second one, integration of informal and formal methods was addressed in the MATISSE project with the UML+B notation and associated U2B tool. Furthermore, the [DES03] document identifies the use of formal methods (at several phases of development life cycle) as a challenge, as for example, requirements and realisation phase.

8.2.1 Railway

SIEMENS had a previous experience of using the B method within the METEOR project, where B was systematically used for developing safety critical functions. The good results obtained with the METEOR project made SIEMENS willing to bring the formalisation development to system level in industrial projects.

The formal model developed in [Haxthausen99] of a railway system using the Raise method is also a system-level model similarly to the MATISSE model of the railway example of Section 3.2.1. The refinement approach described in this handbook in Section 5.1.1.1 has a more comprehensive way of refining the model to derive models for the individual controllers than found in [Haxthausen99].

8.2.2 Smart Cards
The goal of the smart card case study was to formally develop a byte code verifier, and more specifically to verify if formal methods can be used in developing smart cards in such a way that gains in quality come at predictable and acceptable cost. [Cassel02] presents a comparison between formal and traditional development for the smart case study. The author concludes that the development overhead introduced by formal methods is offset by the increase confidence gain in the obtained code, which is particularly important for critical environments where security or safety must be assured.

Also in [Cassel02], the author compares the MATISSE smart card formal development with other formal work on Java byte code verification. Most of those studies focus on the type verification part of the algorithms. One of the most complete formal models of the Java virtual machine is given by in [Qian99]. He considers a large subset of the byte code and aims at proving the runtime correctness from its static typing. Then, he proposes the proof of a verifier that can be deducted from the specifications of the virtual machine. In a more recent work [Coglio00] the authors also propose a correct implementation of almost all aspects of the Java byte code verifier. They view the verification problem as a data flow analysis, and aims to formally describe the specification to extract the corresponding code.

Within the Bali project, [Pusch98] proves a part of the JVM using the Isabelle/HOL prover. Building from [Qian99] work, she gives the verifier specification and then proves its correctness. She also defines a subset of Java, µjava [Pusch00] and aims to prove properties. More precisely, they formalise the type system and the semantics of this language using the Isabelle theorem prover. More recent work [Nipkow00] introduces the formal specification of the Java byte code verifier in Isabelle. The idea is to come up with the generic proof of the algorithm and then to instantiate it with a particular JVM.

The verification scheme presented in [Rose98] has been proven safe using the Isabelle theorem prover described in [Klein00], and a similar scheme for a Smart Card specific language has been proved correct using B in [Requet00].

Work prior to the one described in this article has also been performed using the B method on the formalisation of a simple verifier [Casset99], and its implementation [Casset01]. Similar work is presented in [Bertot01] using Coq’s theorem prover. He proves the correctness of the verification algorithm and generates an implementation using Coq’s extraction mechanism.

The RESET project [RESET03] that has as its aim to assess and introduce technology and marketing priorities to the smart card industry. In [RESET03] several challenges were identified regarding smart cards and the global information and communication society, and two of them:

- improved smart card integration in IT infrastructures, more specifically in the context of IPV6 for Internet, for enhanced access to trusted e-services;
- Security handling in product design and manufacturing, through co-design and extended cooperation in security certification procedures.
where already partially addressed in the MATISSE smart case study. Furthermore, [RESET03]
mentions the need to comply to legal issues and regulations, making increasingly important to
assure secure access to network and online services, and for trusted users consent in transactions.

8.2.3 UML+B

An important aspect of MATISSE was the use of UML+B in the formal development of the
healthcare case study. As UML is an accepted notation in industry, the notation UML+B can be
used not only to introduce the B method to non-formal methods users, but also as a common
notation between formal and non-formal methods users. Furthermore, as the tool U2B can
generate B specifications from UML+B diagrams, those specifications can be validated and
refined within a B tool as Atelier-B.

Next we compare several approaches involving formal languages and UML with UML+B. Most
of these approaches focus either on formalising an UML notation or producing a formal
specification from a UML model. The UML+B notation is related to the latter, as its aim is to use
UML as a graphical representation for B specifications, and to generate B specifications from
those graphical models.

Other work with a similar perspective is that of [Meyer99] and [Sekerinski02].

As said before, the UML+B notation (and associated U2B tool) builds from the translation of
OMT class diagrams and state charts into B specifications presented in [Meyer99]. The main
difference between these two translations, is that UML+B has a greater concern in the
consistency of the generated B specifications.

In [Sekerinski02] the authors present a translation from the UML statecharts to B AMN. This
translation supports hierarchy, concurrency, and communication. The aim of their work differs
from UML+B, as the translation is intended to provide a formalisation of statecharts, and to
provide a formal development method for statecharts.

8.2.4 Safety Analysis

An application of formal methods to the development of dependable systems and the qualitative
reasoning about their dependability has a long history. Obviously, an overview of all the research
conducted in this area is rather unfeasible. An extensive bibliography of research in this field can be
found elsewhere [Bowen93]. Here we narrow the scope of our overview by only considering
research attempting to incorporate safety analysis techniques in software development process.

The influential book by Leveson [Leveson95] focuses on the development of software embedded in
safety-critical systems. In general, Leveson is rather sceptical about practicality of formal
mathematical frameworks for the software development. She, however, claims that a good discipline
of software development is an intrinsic part of achieving system safety. The rich industrial
experience allows the researcher to argue that only if the software development is carried out hand-in-hand with safety analysis an overall system safety can be provided. The same idea underlies our research on the incorporation of safety analysis in the stepwise process of system development. We, however, use a formal framework to ensure a good discipline of software development.

The problem of embodying safety consideration in the software development process is addressed in the work of Hansen et al. [Hansen98]. They argue that the semantic gap between the terms of reasoning used in safety analysis and software development might cause serious flaws in a system design. The employed safety technique – fault tree analysis [Leveson95,Storey96] – is linked to program development by using a common system model. The program development framework of the approach is the duration calculus [Zhou91]. Hansen et al.[ Hansen98] give a formal semantics to fault trees in terms of the real-time interval logic, the duration calculus. They show how to derive a safety invariant, which embedded software should preserve. The idea of linking safety analysis and software development underlies our work, too. The approach proposed by Hansen et al. [Hansen98] has a greater expressiveness in treating dynamic dependencies. An advantage of our approach is better support of abstraction and hence a better management of the complexity of the requirements.

In her thesis Troubitsyna [Troubitsyna00] proposes an approach to the development of dependable systems within the formal framework of the refinement calculus. She argues that substantial improvement in dependability of computer-based systems can be achieved by an incorporation of safety analysis in the formal development of dependable systems. In the MATISSE project we put forward this approach by using UML to facilitate multidisciplinary communication.
A The specification of the Analyser

\[ \textbf{Machine ANALYSER.mch} \]
We give the abstract specification of the Analyser in ANALYSER.mch. This machine is derived from the basic UML statechart-diagram.

The variables of the abstract system are \( a\text{state} \) and \( acmd \), modelling the state of the Analyser and the command given to the Analyser, respectively.

\begin{verbatim}
MACHINE ANALYSER
SEES
DEF0
INCLUDES
GLOBALVAR_PLATE,
R_PROC,
A_PROC
SETS
ASTATE={aidle,amove,aanalyse,arecieve,adeliver,aabort,asuspended}
VARIABLES
a\text{state},
acmd
INVARIANT
a\text{state} \in ASTATE \land
acmd \in ACOMMAND
INITIALISATION
a\text{state} := idle \mid
acmd := receive
OPERATIONS
NewCommand (cc) =

\begin{verbatim}
PRE
 cc \in ACOMMAND
\end{verbatim}

\end{verbatim}
\begin{verbatim}
SELECT astate=aidle THEN acmd:=cc END

MoveA (pos) =
PRE
  pos ∈ NAT
THEN
  SELECT astate=aidle ∧
       acmd=move
  THEN astate:=amove ||
       Safe
  END
END;

MoveAFail =
BEGIN
  SELECT astate=aidle ∧
       acmd=move
  THEN astate:=asuspended ||
       NotSafe
  END
END;

MoveAMinRemedy =
BEGIN
  SELECT astate=asuspended ∧
       acmd=move
  THEN astate:=aidle ||
       Safe
  END
END;

MoveAMidRemedy =
BEGIN
\end{verbatim}
MoveAReady (pos) =
    PRE
    pos ∈ NAT
    THEN
    SELECT astate=amove ∧
            acmd=move
    THEN astate:=aidle
    END
END;

MoveANotReady (pos) =
    PRE
    pos ∈ NAT
    THEN
    SELECT astate=amove ∧
            acmd=move
    THEN astate:=asuspended
    END
END;

MoveRemedy (pos) =
    PRE
    pos ∈ NAT
    THEN
    SELECT astate=asuspended ∧
            acmd=move
    THEN astate:=aidle
    END
END;
acmd=move
THEN  astate:=amove ||
Safe
END

END;

Analyse  =
BEGIN
SELECT  0=0
THEN  astate:=aanalyse ||
PlateInAnalyser ||
SELECT  astate=aidle ∧ acmd=analyse THEN  skip END
END

END;

AnalyseFail =
BEGIN
SELECT  astate=aidle ∧
acmd=analyse
THEN  astate:=asuspended
END

END;

AnalysePlateFail =
BEGIN
SELECT  0=0
THEN  astate:=asuspended ||
NoPlateInAnalyser ||
SELECT  astate=aidle ∧ acmd=analyse THEN  skip END
END

END;

AnalysePosRemedy =
BEGIN
SELECT astate=asuspended ∧ acmd=analyse
THEN astate:=idle || Safe
END

AnalyseMaxRemedy =
BEGIN
SELECT astate=asuspended ∧ acmd=analyse
THEN astate:=idle
END
END;

AnalyseReady =
BEGIN
SELECT astate=aanalyse ∧ acmd=analyse
THEN astate:=idle
END
END;

AnalyseNotReady =
BEGIN
SELECT astate=aanalyse ∧ acmd=analyse
THEN astate:=asuspended
END
END;

AnalyseRemedy =
BEGIN

SELECT astate=asuspended ∧
    acmd=analyse
THEN astate:=aanalyse ||
    PlateInAnalyser
END

END;

Deliver =
BEGIN

SELECT 0=0
THEN astate:=adeliver ||
    PlateInAnalyser || ReadyFor_blank ||
    skip END

END

END;

DeliverFail =
BEGIN

SELECT astate=aidle ∧
    acmd=deliver
THEN astate:=asuspended
END

END;

DeliverPlateFail =
BEGIN

SELECT 0=0
THEN astate:=asuspended ||
    NoPlateInAnalyser ||
    SELECT astate=aidle ∧ acmd=deliver THEN skip END

END

END;
DeliverFailAtRobot = 
BEGIN
   SELECT astate=aidle ∧
      acmd=deliver
   THEN astate:=asuspended ||
      NotReadyFor_blank
   END
END;

DeliverPosRemedy = 
BEGIN
   SELECT astate=asuspended ∧
      acmd=deliver
   THEN astate:=aidle ||
      Safe
   END
END;

DeliverReady = 
BEGIN
   SELECT astate=adeliver ∧
      acmd=deliver
   THEN astate:=aidle ||
      NoPlateInAnalyser
   END
END;

DeliverNotReady = 
BEGIN
   SELECT astate=adeliver ∧
      acmd=deliver

THEN  astate:=asuspended
END;

DeliverNotReadyPlate =
BEGIN
SELECT  astate=adeliver ∧
         acmd=deliver
THEN  astate:=asuspended ||
       PlateInAnalyser
END

END;

DeliverRemedy =
BEGIN
SELECT  astate=asuspended ∧
        acmd=deliver
THEN  astate:=adeliver ||
       Safe || NoPlateInAnalyser
END

END;

Receive =
BEGIN
SELECT  0=0
THEN  astate:=areceive ||
       NoPlateInAnalyser || Available_blank ||
SELECT  astate=aidle ∧ acmd=receive THEN skip END
END

END;

ReceiveFail =
BEGIN
SELECT  astate=aidle ∧
acmd=receive
THEN astate:=asuspended
END;

ReceivePlateFail =
BEGIN
SELECT 0=0
THEN astate:=asuspended || PlateInAnalyser ||
SELECT astate=aidle ∧ acmd=receive THEN skip END
END;

ReceiveFailAtRobot =
BEGIN
SELECT astate=aidle ∧
acmd=receive
THEN astate:=asuspended ||
NoAvailable_blank
END;

ReceivePosRemedy =
BEGIN
SELECT astate=asuspended ∧
acmd=receive
THEN astate:=aidle ||
Safe
END;
ReceiveReady =
BEGIN
  SELECT astate = areceive ∧
  acmd = receive
  THEN astate := idle ||
  PlateInAnalyser
END;

ReceiveNotReady =
BEGIN
  SELECT astate = areceive ∧
  acmd = receive
  THEN astate := asuspended
END;

ReceiveNotReadyPlate =
BEGIN
  SELECT astate = areceive ∧
  acmd = receive
  THEN astate := asuspended ||
  NoPlateInAnalyser
END;

ReceiveRemedy =
BEGIN
  SELECT astate = asuspended ∧
  acmd = receive
  THEN astate := areceive ||
  Safe || PlateInAnalyser
END;
AServiceNotReady  =
BEGIN
  SELECT astate=aidle
  THEN  astate:=asuspended
  END;
END

ARemedy  =
BEGIN
  SELECT astate=asuspended
  THEN  astate:=aidle
  END;
END

AFailure  =
BEGIN
  SELECT astate=asuspended
  THEN  astate:=aabort
  END;
END

∧ Machine DEF0.mch
This machine defines the commands used in the Analyser.
MACHINE  DEF0
SETS
  ACOMMAND=\{move,analyse,receive,deliver\}
END

∧ Machine GLOBALVAR_PLATE.mch
The global variable plate is given in a separate machine GLOBALVAR_PLATE.mch.
MACHINE  GLOBALVAR_PLATE
VARIABLES

    plate

INARIANT

    plate ∈ BOOL

INITIALISATION

    plate := FALSE

OPERATIONS

ChangePlate (bb) =

    PRE

        bb ∈ BOOL

    THEN

        plate := bb

    END;

PlateInAnalyser =

    BEGIN

        SELECT plate = TRUE THEN skip END

    END;

NoPlateInAnalyser =

    BEGIN

        SELECT plate = FALSE THEN skip END

    END

END

∧

Machine A_PROC.mch

The exported global of the Analyser are given in A_PROC.mch.

MACHINE A_PROC

OPERATIONS

ReadyToDeliver =

    BEGIN skip END;

MATISSE/WP1-D10/April03
NotReadyToDeliver =
    BEGIN skip END;

ReadyToReceive =
    BEGIN skip END;

NotReadyToReceive =
    BEGIN skip END;

BlankInAnalyser =
    BEGIN skip END;

NoBlankInAnalyser =
    BEGIN skip END
END

∧

Machine R_PROC.mch
The imported procedures from the Robot are given in R_PROC.mch. These procedures are only given as skip, since the Analyser need not be aware of their implementation.

MACHINE R_PROC

OPERATIONS
Safe =
    BEGIN skip END;

NotSafe =
    BEGIN skip END;

Available_blank =
    BEGIN skip END;

NotAvailable_blank =
BEGIN skip END;

ReadyFor_blank =
  BEGIN skip END;

NotReadyFor_blank =
  BEGIN skip END
END
B The Language CSP\text{M}

CSP\text{M} is the machine readable dialect of CSP [Roscoe98,FDR97], used to define both the specification and implementation of models verified by FDR. It is very close to pure CSP and it is embedded in a Gofer/Haskell like functional programming language, which augments its utility substantially. A brief description of the key features of the CSP\text{M} syntax follows, where the formal syntax is in a typewriter font and keywords and symbols are highlighted in bold.

- **Channel**
  channel channel_id : ChannelType is a CSP\text{M} declaration of a channel, which is identified by its channel_id and communicates values of type ChannelType.

- **Datatype**
  datatype name = string1 . X11 . . . X1n | . . . | stringN . XN1 . . . XNm, where name and string1 to stringN represent globally unique identifiers, and XIJ represents either explicit or named sets of data. Note that CSP\text{M} has two inbuilt datatypes, namely \text{Bool} and \text{Int}, representing booleans (true, false) and integers between ±(2^{31}-1).

- **NameType**
  nametype name = X, where a set expression (X) is given a name which is intended to be used as a type.

- **Stop**
  STOP is the simplest CSP process; it never engages in any action, and never (successfully) terminates; that is it cannot be used as a successful termination for sequential composition.

- **Skip**
  SKIP is the process that terminates successfully, and is the unit of sequential composition (i.e. \text{P} ; \text{SKIP} = \text{P} = \text{SKIP} ; \text{P}). Note that the precise formal semantics of \text{SKIP} is surprisingly complex.

- **Action**
  a \rightarrow \text{P} is the most basic program constructor. It waits to perform the event a and after this has occurred subsequently behaves as process \text{P}. The same notation is used for outputs (c!v \rightarrow \text{P}), inputs (c?v \rightarrow \text{P}(v)) and simple communication (c.x \rightarrow \text{P}(x)) of values along named channels.

- **Boolean Expression**
  bool_exp & a \rightarrow \text{P} is the process which behaves as \text{a} \rightarrow \text{P} if the bool_exp is true.

- **Choice**
  \text{P} |\sim| \text{Q} represents the non-deterministic or internal choice between \text{P} and \text{Q}.

  \text{P} [\sim] \text{Q} represents the external choice between \text{P} and \text{Q}, which is deterministic so long as the set of initial events of \text{P} and \text{Q} are disjoint. The process \text{P} [\sim] \text{Q} offers the initial actions of both \text{P} and \text{Q} to its environment; its subsequent behaviour is like \text{P} if the initial action chosen was possible only for \text{P} and like \text{Q} if the action selected \text{Q}. If both \text{P} and \text{Q} have common initial actions, its subsequent behaviour is non-deterministic (like |\sim|). Note that \text{STOP} is the unit of external choice (i.e. \text{P} [\sim] \text{STOP} = \text{P} = \text{STOP} [\sim] \text{P}).
[] x : X @ x -> P represents replicated external choice.

P \ A is the CSP hiding operator. This process behaves as P except that events in set A are hidden from the environment and are solely (and internally) determined by P; the environment can neither observe nor influence them. P \ {<|channel_id|>} hides all communications along the channel channel_id.

P [\ A ] Q represents shared parallel (concurrent) composition. P and Q evolve separately, except that events from A occur only when P and Q agree (i.e. synchronise) to perform them.

P [\ Alphabet1 || Alphabet2 ] Q represents alphabetised parallel composition. Any events in P which are not in Alphabet1 are blocked from occurring and similarly for Q. P and Q evolve separately except or events in the intersection of Alphabet1 and Alphabet2 occur only when P and Q agree (i.e. synchronise) to perform them.

\ i : Indexing_set \ A(i) P(i) represents alphabetised parallel composition. For example \ i : \ {1,2,3} \ A(i) P(i) is equivalent to (P(1) [\ A(1) || A(2)] P(2)) [Union(A(1),A(2)) || A(3)] P(3).

P [[ channel_id1 <-> channel_id2 ]] Q represents linked parallel composition. The two processes communicate via a common new hidden channel resulting from a renaming of channel_id1 and channel_id2.

\ name \ represnets the set of all expansions of name; that is the set containing { name.x } for all values x that can be prefixed by “name.”.

diff(X, Y) represnets the set difference operation.

union(X, Y) represnets the set union operation.

Union({X1, ..., Xn}) represnets the multi argument (distruibuted) set union operation; that is the union of sets X1, ..., Xn.

Set(X) represnets the power set operation; that is the set of all subsets of X.

if bool_exp then exp1 else exp2 represents the conditional evaluation of an expresion, where exp1 is returned if the bool_exp is true, and exp2 if bool_exp is false.

let name=val within body represents a technique for local aliasing, where each occurrence of name within the body is replaced with val. Note that the name can optionally include have a bracketted set of comma separated arguments, in which case a local function has been declared.

chase(P) The process P with the ‘chase’ operator applied to it. As a consequence, internal events, called tau-events, in P, are give priority over all other events. Where there is an option to perform several tau-events, one of them is chosen arbitrarily. Note the use of chase changes the semantics of the process P, if the order in which the hidden tau-events take place matters.
References


