

Formal Methods

Formal Methods

Industrial Use from Model to the Code

Edited by
Jean-Louis Boulanger



First published 2012 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms and licenses issued by the CLA. Enquiries concerning reproduction outside these terms should be sent to the publishers at the undermentioned address:

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

© ISTE Ltd 2012

The rights of Jean-Louis Boulanger to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Library of Congress Cataloging-in-Publication Data

Formal methods : industrial use from model to the code / edited by Jean-Louis Boulanger.
p. cm. -- (Industrial implementation of formal methods series)
Includes bibliographical references and index.
ISBN 978-1-84821-362-3
1. Railroads--Management--Data processing. 2. Formal methods (Computer science) 3. Application software--Development. I. Boulanger, Jean-Louis.
TF507.F66 2012
385.0285'53--dc23

2012011496

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN: 978-1-84821-362-3

Printed and bound in Great Britain by CPI Group (UK) Ltd., Croydon, Surrey CR0 4YY



Table of Contents

Introduction	xi
Jean-Louis BOULANGER	
Chapter 1. From Classic Languages to Formal Methods	1
Jean-Louis BOULANGER	
1.1. Introduction	1
1.2. Classic development	2
1.2.1. Development process	2
1.2.2. Coding	6
1.2.3. Specification and architecture	18
1.2.4. Verification and validation (V&V)	27
1.2.5. Summary	33
1.3. Structured, semi-formal and/or formal methods	33
1.3.1. E/E/PE system	33
1.3.2. Rail sector	35
1.3.3. Taking into account techniques and formal methods	36
1.4. Formal methods	39
1.4.1. Principles	39
1.4.2. Examples of formal methods	39
1.5. Conclusion	45
1.6. Bibliography	49
Chapter 2. Formal Method in the Railway Sector the First Complex Application: SAET-METEOR	55
Jean-Louis BOULANGER	
2.1. Introduction	55
2.2. About SAET-METEOR	56
2.2.1. Decomposition of the SAET-METEOR	57

2.2.2. Anticollision functions	61
2.2.3. Restrictions	62
2.3. The supplier realization process	62
2.3.1. Historical context	62
2.3.2. The hardware aspect	64
2.3.3. The software aspect	67
2.3.4. Assessment of the processes	78
2.4. Process of verification and validation set up by RATP	78
2.4.1. Context	78
2.4.2. RATP methodology	79
2.4.3. Verification carried out by RATP	79
2.4.4. Validation	103
2.4.5. Assessment of RATP activity	112
2.5. Assessment of the global approach	114
2.6. Conclusion	115
2.7. Appendix	116
2.7.1. Object of the track	116
2.7.2. Block logic.	120
2.8. Bibliography	122
Chapter 3. The B Method and B Tools	127
<i>Jean-Louis BOULANGER</i>	
3.1. Introduction	127
3.2. The B method	128
3.2.1. The concept of abstract machines	128
3.2.2. Machines with implementations	133
3.3. Verification and validation (V&V)	137
3.3.1. Internal verification	137
3.4. B tools	141
3.4.1. General principles	141
3.4.2. Code generation	142
3.4.3. Prover	142
3.4.4. Atelier B	144
3.5. Methodology	146
3.5.1. Layered development	146
3.5.2. Link between the project structure and the POs	148
3.5.3. Cycle of development of a B project	148
3.6. Feedback	150
3.6.1. Some figures	150
3.6.2. Some users	151
3.7. Conclusion	155
3.8. Bibliography	155

Chapter 4. Model-Based Design Using Simulink – Modeling, Code Generation, Verification, and Validation	159
Mirko CONRAD and Pieter J. MOSTERMAN	
4.1. Introduction	159
4.2. Embedded software development using Model-Based Design	162
4.3. Case study – an electronic throttle control system	164
4.3.1. System overview	164
4.3.2. Simulink® model	164
4.3.3. Automatic code generation	169
4.3.4. Code optimization	170
4.3.5. Fixed-point code	170
4.3.6. Including legacy code	172
4.3.7. Importing interface definitions	172
4.3.8. Importing algorithms	173
4.4. Verification and validation of models and generated code	173
4.4.1. Integrating verification and validation with Model-Based Design	173
4.4.2. Design verification	175
4.4.3. Reviews and static analyses at the model level	175
4.4.4. Module and integration testing at the model level	175
4.4.5. Code verification	176
4.4.6. Back-to-back comparison testing between model and code	176
4.4.7. Measures to prevent unintended functionality	176
4.5. Compliance with safety standards	177
4.6. Conclusion	178
4.7. Bibliography	178
Chapter 5. Proving Global Properties with the Aid of the SIMULINK DESIGN VERIFIER Proof Tool	183
Véronique DELEBARRE and Jean-Frédéric ETIENNE	
5.1. Introduction	183
5.2. Formal proof or verification method	184
5.2.1. Model verification	186
5.2.2. Formal methods and proof of correction	189
5.2.3. Combining models and proof tools	192
5.3. Implementation of the SIMULINK DESIGN VERIFIER tool	193
5.3.1. Reminders of the MATLAB modeling and verification environment	194
5.3.2. Case study	201
5.3.3. Modeling	204
5.3.4. Modeling	211
5.4. Experience feedback and methodological aspects	211

5.4.1. Modeling rules and convergence control	211
5.4.2. Modular proof phase	213
5.4.3. Proof of global properties	214
5.4.4. Detection of counterexamples	217
5.5. Study case feedback and conclusions	218
5.5.1. SIMULINK model	218
5.5.2. Proofs achieved	218
5.5.3. Incremental verification approach	220
5.6. Contributions of the methodology compared with the EN50128 normative referential	220
5.7. Bibliography	222
Chapter 6. SCADE: Implementation and Applications	225
Jean-Louis CAMUS	
6.1. Introduction	225
6.2. Issues of embedded safety-critical software	225
6.2.1. Characteristics of embedded safety-critical software	225
6.2.2. Architecture of an embedded safety-critical application	226
6.2.3. Criticality and normative requirements for embedded safety-critical applications	226
6.2.4. Complexity, cost and delays	227
6.3. Origins of SCADE	228
6.3.1. Introduction	228
6.3.2. Initial industrial approaches	228
6.3.3. “Real-time” extensions of current languages	230
6.3.4. Synchronous formal languages dedicated to “real-time” created in laboratories	230
6.3.5. Birth of SCADE	231
6.4. The SCADE data-flow language	231
6.4.1. Introduction	231
6.4.2. Synchronous language	232
6.4.3. “Data-flow” functional language	233
6.4.4. Scalar data types	234
6.4.5. Structured data types	235
6.4.6. Clocks, temporal operators, and causality	235
6.4.7. Selectors	237
6.4.8. Imperative structures	238
6.4.9. Rigor and functional safety	239
6.5. Conclusion: extensions of languages for controllers and iterative processing	240
6.5.1. Objectives	240
6.5.2. Control flow	241
6.5.3. Iterative processing	243

6.6. The SCADE system	246
6.6.1. Outline of the SCADE workbench	246
6.6.2. Model verification	247
6.6.3. Performance prediction	252
6.6.4. The qualified code generator	253
6.7. Application of SCADE in the aeronautical industry	256
6.7.1. History: Aérospatiale and Thales Avionique	256
6.7.2. Control/command type applications	257
6.7.3. Monitoring/alarm type applications	260
6.7.4. Navigation systems	261
6.8 Application of SCADE in the rail industry	261
6.8.1. First applications	262
6.8.2. Applications developed for the RATP and other French metros	262
6.8.3. Generic PAI-NG applications	263
6.8.4. Example of automated door control	264
6.9. Application of SCADE in the nuclear and other industries	265
6.9.1. Applications in the nuclear industry	265
6.9.2. Deployment of SCADE in the civil nuclear industry	268
6.10. Conclusion	269
6.11. Bibliography	270
Chapter 7. GATEL: A V&V Platform for SCADE Models	273
Bruno MARRE, Benjamin BIANC, Patricia MOUY and Christophe JUNKE	
7.1. Introduction	273
7.2. SCADE language	275
7.3. GATEL prerequisites	276
7.3.1. GATEL kernel	277
7.3.2. Example	278
7.4. Assistance in the design of test selection strategies	279
7.4.1. Unfolding of SCADE operators	279
7.4.2. Functional scenarios	281
7.5. Performances	283
7.6. Conclusion	284
7.7. Bibliography	285
Chapter 8. ControlBuild, a Development Framework for Control Engineering	287
Franck CORBIER	
8.1. Introduction	287
8.2. Development of the control system	289
8.2.1. ERTMS	290
8.2.2. Development process equipment	291

8.2.3. A component-based approach	293
8.2.4. Development methodology	294
8.3. Formalisms used	300
8.3.1. Assembly editor	301
8.3.2. IEC1131-3 languages for embedded control	302
8.3.3. Electrical schematics for conventional control	309
8.3.4. Electromechanical and physical environment	311
8.4. Safety arrangements	311
8.4.1. Metrics	313
8.4.2. Assertions	314
8.4.3. Automatic test procedure execution	314
8.4.4. Functional tests	315
8.4.5. Code coverage	315
8.4.6. SSIL2 code generation	315
8.4.7. Management of the project documentation	316
8.4.8. Traceability of requirements	317
8.5. Examples of railway use cases	318
8.5.1. Specification validation	318
8.5.2. TCMS development	319
8.5.3. Progressive integration bench – HiL	320
8.6. Conclusion	323
8.7. Bibliography	323
Chapter 9. Conclusion	325
Jean-Louis BOULANGER	
9.1. Introduction	325
9.2. Problems	326
9.3. Summary	327
9.3.1. Model verification	327
9.3.2. Properties and requirements	328
9.3.3. Implementation of formal methods	330
9.4. Implementing formal methods	332
9.4.1. Conventional process	332
9.4.2. Process accounting for formal methods	333
9.4.3. Problems	335
9.5. Realization of a software application	337
9.6. Conclusion	339
9.7. Bibliography	340
Glossary	345
List of Authors	351
Index	353

Introduction

Context

Although formal analysis programming techniques (see works by Hoare [HOA 69] and Dijkstra [DIJ 75]) are relatively old, the introduction of formal methods only dates from the 1980s. These techniques enable us to analyze the behavior of a software application, described in a programming language. Program correction (good behavior, program stop, etc.) is thus demonstrated through a program proof based on the weakest precondition calculation [DIJ 76].

It took until the end of the 1990s before formal methods (Z [SPI 89], VDM [JON 90] or the B-method [ABR 96, ARA 97]) could be used in industrial applications and settings.

One of the stumbling blocks was implementing them in the framework of an industrial application (large application, cost constraints or delays, etc.). This implementation is only possible using “sufficiently” mature and high-performance tools.

Where safety requirements are critical, at least two formal methods are used: the B-method [ABR 96] and the LUSTRE language [HAL 91, ARA 97] and its graphic version, named SCADE [DOR 08]. These cover one part of the specification production process according to the code and integrate one or more verification processes.

The B-method and the SCADE environment are associated with industrial tools.

Introduction written by Jean-Louis BOULANGER.

For example, Atelier B and the B-Toolkit, marketed by CLEARSY¹ and B-Core² respectively, are tools that completely cover the development cycle proposed by the B-method comprising specification, refinement, code, and proof generation. It should be noted that Atelier B³ can be accessed for free from version 4.0 onward.

Formal methods rely on different formal verification techniques such as proofs, *model checking* [BAI 08] and/or simulation.

The use of formal methods while in full development remains marginal, given the number of lines of code. In effect, there are currently many more Ada [ANS 83], C [ISO 99] or C++ lines of code, which have been produced manually rather than through a formal process.

That is why other formal techniques have been implemented to verify the behavior of a software application written in a programming language such as C or Ada. The technical principle known as *abstract interpretation* [COU 00] of programs makes it possible to evaluate all the behaviors of a software application through a static analysis. This type of technique has, in these last few years, given rise to several tools such as POLYSPACE⁴, Caveat⁵, Absint⁶, Frama-C⁷, and/or ASTREE⁸.

The efficacy of these static program analysis techniques has progressed greatly with the increase in power of business machines. It should be noted that these techniques generally necessitate the integration of complementary information such as pre-conditions, invariants, and/or post-conditions in the manual code.

SPARK Ada⁹ [BAR 03] is one approach where the Ada language [ANS 83] has been expanded to introduce these complementary elements (pre-, post-, invariant), and an adapted suite of tools has been defined.

Objective of this book

In [BOW 95, ARA 97], the first industrial feedback involving formal techniques can be found, and notably, a report on the B-method [ABR 96], the LUSTRE

1 To find out more about the CLEARSY company and Atelier B, visit www.clearsy.com.

2 The B-Toolkit was distributed by B-Core (UK) Ltd.

3 Atelier B and associated information can be obtained from www.atelierb.eu/.

4 For more information on Polyspace, visit www.mathworks.com/polyspace.

5 To find out more about Caveat, visit www-list.cea.fr/labos/fr/LSL/caveat/index.html.

6 To find out more about Absint, visit www.absint.com.

7 To find out more, visit <http://frama-c.com/>.

8 To find out more about ASTREE, visit www.astree.ens.fr.

9 The site www.altran-praxis.com/spark.aspx offers further information about SPARK Ada technology.

language [HAL 91, ARA 97] and SAO+, the predecessor to SCADE¹⁰ [DOR 08]. Other works such as [MON 00, MON 02, HAD 06] provide a panorama of formal methods with a more scientific point of view.

Given the presentation of the context and of the state of the literature, our objective is to present concrete examples of industrial use of formal techniques.

By formal techniques, we mean the different mathematical approaches, which make it possible to demonstrate that a software application obeys some properties.

While the standard use of formal techniques consists of making specification and/or design models, they are seen by a verification subject to static analysis of code, demonstration of abiding by properties, good management of floating-point calculations, etc.

This work is related to two other books by the same authors published by ISTE and John Wiley & Sons in 2012 [BOU 12a] and [BOU 12b].

The current book is dedicated to the presentation of different formal methods, such as the B-method (Chapters 2 and 3), SCADE (Chapters 6 and 7), MATLAB/SIMULINK (Chapters 4 and 5) and ControlBuild¹¹ (Chapter 8).

[BOU 12a] involves industrial examples of implementation of formal techniques based on static analysis such as abstract interpretation; examples of the use of ASTREE (Chapters 2 and 3), CAVEAT (Chapter 3), CODEPEER (Chapter 6), Frama-C (Chapters 3 and 7), and POLYSPACE (Chapters 4 and 5) tools.

[BOU 12b] is dedicated to the presentation of different formal techniques, such as the SPARK Ada (Chapter 1), MaTeLo¹² (Chapter 2), AltaRica (Chapter 3), Polyspace (Chapter 4), Escher verification Studio Perfect Developer (Chapter 5) and the B method (Chapters 6 and 7).

In conclusion to this introduction, I have to thank all the manufacturers who have taken the time to redraft and improve upon these chapters.

¹⁰ It should be noted that SCADE was initially a development environment basing itself on the LUSTRE language and that since version 6, SCADE has become an entirely separate language (the code generator for version 6 takes most of its input from a SCADE model, and not a LUSTRE code).

¹¹ To find out more about the ControlBuild tool, visit www.geensoft.com/en/article/controlbuild.

¹² To find out more about MaTeLo, visit www.all4tec.net/index.php/All4tec/matelo-product.html.

Bibliography

- [ABR 96] ABRIAL J.R., *The B-Book - Assigning Programs to Meanings*, Cambridge University Press, Cambridge, August 1996.
- [ANS 83] ANSI, Norme ANSI/MIL-STD-1815A-1983, Langage de programmation Ada, 1983.
- [ARA 97] ARAGO, “Applications des méthodes formelles au logiciel”, *Observatoire français des techniques avancées* (OFTA), vol. 20, Masson, Paris, June 1997.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, The MIT Press, Cambridge, MA, 2008.
- [BAR 03] BARNES J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison Wesley, Boston, 2003.
- [BOU 12a] BOULANGER J.-L. (ed.), *Static Analysis of Software – The Abstract Interpretation*, ISTE Ltd, London and John Wiley and Sons, New York, 2012.
- [BOU 12b] BOULANGER J.-L. (ed.), *Industrial Use of Formal Methods*, ISTE Ltd, London, John Wiley and Sons, New York, 2012.
- [BOW 95] BOWEN J.P., HINCHEY H.G., *Applications of Formal Methods*, Prentice Hall, Upper Saddle River, 1995.
- [COU 00] COUSOT P., “Interprétation abstraite”. *Technique et science informatiques*, vol. 19, no. 1-3, p. 155-164, Hermès, Paris, 2000.
- [DIJ 75] DIJKSTRA E.W., “Guarded commands, non-determinacy and formal derivation of programs”, *Communications of the ACM*, vol.18, no. 8, p.453-457, August 1975.
- [DIJ 76] DIJKSTRA E.W., *A Discipline of Programming*, Prentice Hall, Upper Saddle River, 1976.
- [DOR 08] DORMOY F.-X., “Scade 6 a model based solution for safety critical software development”, *Embedded Real-Time Systems Conference*, 2008.
- [HAD 06] HADDAD S., KORDON F., PETRUCCI L. (ed.), *Méthodes formelles pour les systèmes répartis et coopératifs*, Hermès, Paris, 2006.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous dataflow programming language lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305-1320, September 1991.
- [HOA 69] HOARE C.A.R., “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, p. 576-580-583, 1969.
- [ISO 99] ISO/IEC 9899:1999, Programming languages – C, 1999.

- [JON 90] JONES C.B., *Systematic Software Development using VDM*, Prentice Hall International, Upper Saddle River, 1990.
- [MON 00] MONIN J.-F., *Introduction aux méthodes formelles*, préface by HUET G., Hermès, Paris, 2000.
- [MON 02] MONIN J.-F., *Understanding Formal Methods*, preface by HUET, G., TRAD. M., Hinckley, Springer Verlag, New York, 2002.
- [SPI 89] SPIVEY J.-M., *The Z notation – a reference Manual*, Prentice Hall International, Upper Saddle River, 1989.

Chapter 1

From Classic Languages to Formal Methods

1.1. Introduction

The introduction to this book has provided the opportunity to set formal analysis techniques in a general context. In this chapter, we are going to focus on formal methods and their implementation.

The classic development process of a software application is based on the use of a programming language (for example, Ada [ANS 83], C [ISO 99] and/or C++ [ISO 03]). These languages have a certain abstraction level in comparison to the code finally executed on the computer, a program is a set of line of code write manually.

The size of applications has gone from several thousands of lines to several hundreds of thousands of lines of code (possibly several millions for new applications). Considering the number of faults introduced by developers, it is then important to use techniques to limit the number of faults introduced and to more easily identify potential faults.

As we will show later, formal methods enable us to fulfill this double objective.

1.2. Classic development

The objective of this section is to analyze the weaknesses of the *classic* (meaning non-formal) process, which is implemented to make a software application.

Chapter written by Jean-Louis BOULANGER.

2 Formal Methods

1.2.1. Development process

1.2.1.1. Presentation

The creation of a software application is broken down into stages (specification, design, coding, tests, etc.). We refer to the lifecycle. The lifecycle is necessary to describe the dependencies and sequencing between activities.

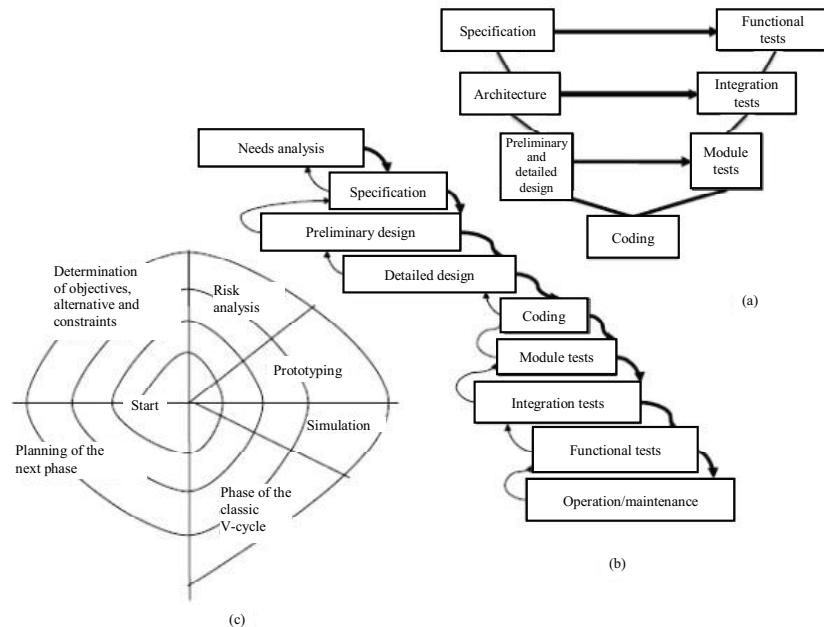


Figure 1.1. Three possible lifecycles

The lifecycle must take into account the progressive refinement aspect of the development as well as possible iterations. In this section, we present the lifecycle, which is used to make a software application.

As Figure 1.1 shows, there are several cycles: a) V-cycle, b) waterfall cycle, c) spiral cycle, etc. for making a software application, but the cycle recommended by different standards (CENELEC EN 50128 [CEN 01], DO 178 [ARI 92], IEC 61508 [IEC 98], ISO 26262 [ISO 09]) remains the V-cycle.

Figure 1.2 presents the V-cycle as it is generally presented. The objective of needs analysis is to verify adequacy to the expectations of the client and technological feasibility. The objective of the specification phase is to describe what

the software must do (and not how it will do it). In the context of architecture definition, the aim is to create a hierarchical breakdown of the software application into modules/components and to identify interfaces between these elements.

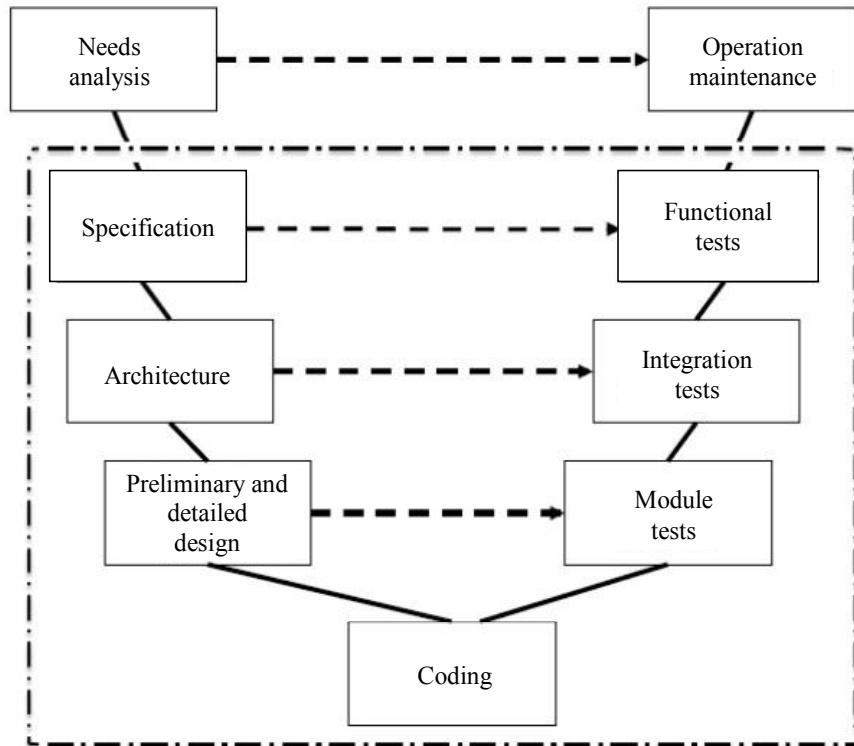


Figure 1.2. V-cycle

Description of each module/component (data, algorithms, etc.) is achieved within the framework of the design. The design phase is often separated into two stages. The first stage, named preliminary design, aims to identify manipulated data and necessary services; the second stage, named detailed design, aims to describe all services through their algorithms. The design phase then gives rise to the coding phase.

Figure 1.2 shows that there are different test phases: module tests (focused on the lowest-level components), integration tests (focused on software and/or hardware interfaces), and functional tests (sometimes known as validation tests), which show that a product conforms to its specification. As for the operation/maintenance phase, it involves operational life and control of potential evolutions.

4 Formal Methods

There is a horizontal correspondence (dotted arrow) between activity specification and design and activity testing. The V-cycle is thus broken down into two phases: bottom-up phase and top-down phase. Top-down phase activity (execution of the MT/IT and FT) must be processed during the bottom-up phase. Figure 1.3 is thus closer to the V-cycle recommended.

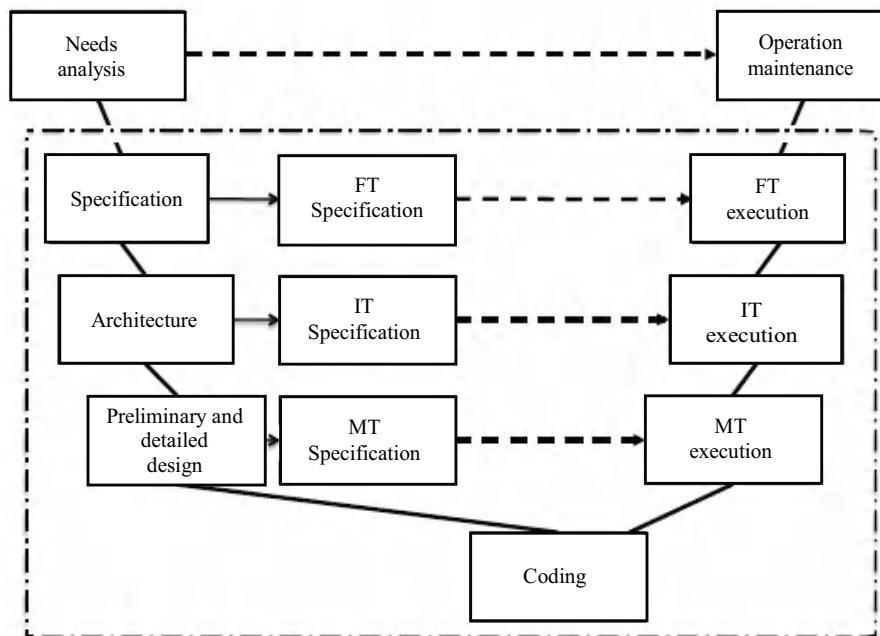


Figure 1.3. *V-cycle including test specifications*

1.2.1.2. Advantages/disadvantages

The V-cycle of Figure 1.3 reveals that faults introduced in the making of the software application will be detected during the top-down phase, which has a direct impact on the cost and delays of making the software.

Experience in safety-critical applications shows that activity testing accounts for 50% to 75% of the cost of production and that the presence of faults can multiply delays in production two or three times over.

Increased delays are caused by the discovery of anomalies, their identification, analysis of their effects (impact on the safety and/or reliability of the software application), selection of anomalies to correct, analysis of anomalies, implementation

of corrections and verification of corrections (in general, verifying correct implementation of modifications is achieved through test runs, but it will be necessary to verify that no additional modification has been implemented).

Analysis of anomalies is achieved through an impact analysis (definition 1.1) and a non-regression analysis (definition 1.2). In certain cases, non-regression is said to be total, and for this, it is necessary to re-execute the series of tests on one phase or all phases.

The objective of non-regression analysis is to minimize the cost of a new version.

DEFINITION 1.1.– IMPACT ANALYSIS. Impact analysis of an anomaly consists of identifying modifications to make in the bottom-up phase (impact on the documents, impact on the code, impact on the description and test implementations) of production.

DEFINITION 1.2.– NON-REGRESSION ANALYSIS. Non-regression analysis consists of determining a series of tests, which make it possible to demonstrate that the modification made has not had an effect on the rest of the software application¹.

In addition, it should be noted that the cost of correcting a fault is directly linked to the phase during which it is identified. In effect, detecting a fault during the functional testing phase is 10 to 100 times more expensive (not to mention higher in certain cases) than a fault identified in the module testing phase. This cost is linked to resources that have been used right up to discovery of the fault and to the difficulty of carrying out functional testing (usage of targeting equipment, necessity of taking real time into account, difficulty of observation, technical expertise of people involved, etc.).

Our experience feedback (in a railway system² evaluator/certifier capacity) leads us to conclude that the unitary and integration testing phases are not effective, given that manufacturers consider that:

- module testing is useless (as a general rule, module tests are defined from the code);
- software/software integration is reduced to a big-bang integration (integration of all the code in place of a module-by-module integration); at worst, all the code is compiled suddenly and integration is reduced to an interface verification by the compiler;

¹ It should be noted that a non-regression analysis can be carried out on a software application or on a more important element such as equipment, a subsystem, and/or a system.

² The author of this chapter is an evaluator/certifier within the CERTIFER association see: www.certifer.asso.fr.

6 Formal Methods

- software/hardware integration is supported by functional testing on a target. If all of the software is being executed correctly on the target machine, the integration is correct.

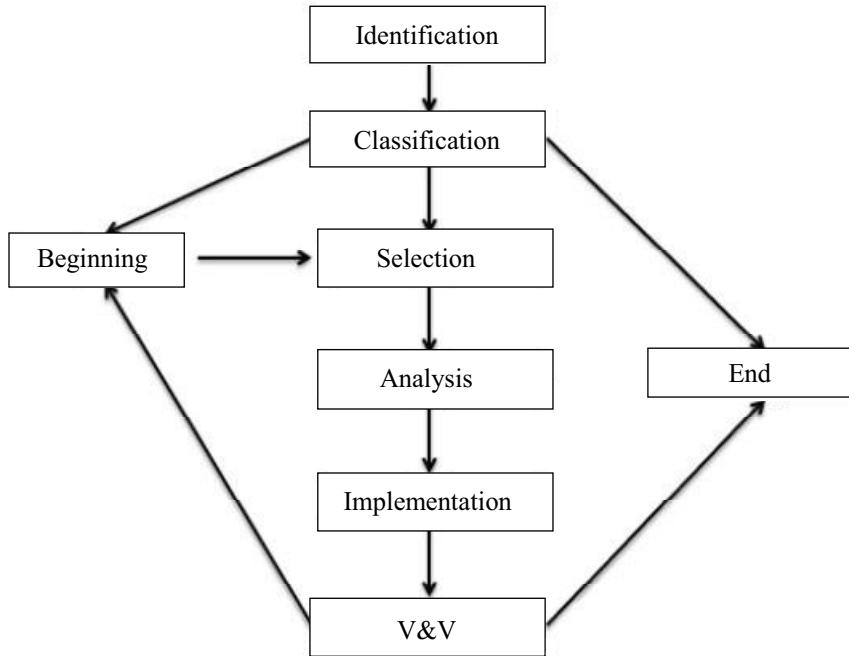


Figure 1.4. Cycle management of anomalies

Cost and delay management imply two necessities:

- N.1: reducing the number of faults introduced into the application during the bottom-up phase of the V-cycle;
- N.2: identifying faults introduced within the software application as early as possible.

1.2.2. Coding

1.2.2.1. Presentation

The classic development process of a software application is based on the use of programming language, for example Ada [ANS 83, ISO 95] C [ISO 99] and/or C++ [ISO 03].

Even if these languages have a certain abstraction level with respect to the code ultimately executed on the computer, they necessitate the writing of a line of code.

It is not possible to analyze all the current programming languages for all industries. We shall analyze the advancements that have taken place in the rail sector.

1.2.2.2. Assessment

1.2.2.2.1. The Ada language

The first rail applications in France were programmed in the middle of the 1980s with the Modula 2 language. Since then however, the Ada 83 language [ANS 83] has become the reference language for the development of safety-critical applications [RIC 94].

As Table 1.1 shows, in the context of applications that have a high level of criticality (SSIL3/SSIL4), the Ada language is only R (recommended); it is necessary to establish a sub-assembly of the language so that use of Ada be HR (highly recommended).

	SSILO ³	SSIL1	SSIL2	SSIL3	SSIL4
Ada	R	HR	HR	R	R
MODULA-2	R	HR	HR	R	R
PASCAL	R	HR	HR	R	R
C or C++ without restriction	R	-	-	NR	NR
Sub-assembly of C or C++	R	R	R	R	R

Table 1.1. CENELEC EN 50128 [CEN 01] – Table A.15

Ada was designed on the initiative of the DoD (the US Department of Defense) to federate more than 400 languages or dialects used by this body since the 1970s.

Ada is very widely used in the framework of embedded software applications in avionics (airbus), the space (the Ariane rocket), and the rail sector. The principal characteristic of these systems is that they require a correction of the execution.

³ The CENELEC [CEN 01], [CEN 00], [CEN 03] referential, like the CEI/IEC 61508 [IEC 98] standard, introduces the notion of SIL for *Safety Integrity Level*, which can take four values, from 1 to 4. To find out more on SIL management, we advise you to read Chapter 7 of [BOU 11]. The notion of SSIL (*Software SIL*) concerns assignment of a safety level to the software aspect. The SSIL level relates to systematic failures. SSILs can go from 0 up to 4. SSIL0 software is software that has no impact on safety and must, at the very least, implement the requirements of the ISO 9001:2008 [ISO 08].

8 Formal Methods

The Ada 83 language [ANS 83] has advanced toward a second major standard, Ada 95 [ISO 95], which is the first object-normalized language. It provides the possibility of constructing object-oriented models. The latest version to date is called Ada 2005.

```
with Ada.Text_IO;
procedure Hello is
begin Ada.Text_IO.Put_Line("Hello, world!");
end Hello;
```

Figure 1.5. Example of Ada code

DEFINITION 1.3.– CERTIFICATION. Certification consists of achieving a certificate, which is a commitment that the product complies with a normative referential. Certification is based on the results of an evaluation and on production of a certificate.

Regarding certification (see definition 1.3) of Ada compilers, the existence of a standard and of a semantic of Ada has made it possible to define a compiler certification process.

This process has been implemented on different compilers and is based on a suite of tests named ACATS (*Ada Conformity Assessment Test Suite*); see the [ISO 99a] standard. To find out more, consult [ADA 01].

At present, these new versions of Ada have not been adopted by the embedded systems sector, because of their object-oriented aspect.

Given their efficacy, however, Ada 95 compilers are used for compilation provided that a sub-assembly of the language, which does not use the “object-oriented” features is relied upon.

John Barnes, in the context of the AdaCore article makes a presentation on the strength of the Ada 2005 language (syntax and semantics, strong typing, sound management of the pointer and the memory, etc.) and the impact of its implementation to demonstrate that the software is reliable.

The “object-oriented” aspect is not taken into account by the CENELEC EN 50128 ([CEN 01], DO 178 [ARI 92], CEI/IEC 61508 [IEC 98], ISO 26262 [ISO 09]) standards applicable to safety-critical applications.

To get around this stumbling block, the ISO 15942 [ISO 00] standard delineates a restriction on Ada 95 language constructions and defines some rules of use

(programming style), which enable creation of a supposedly-certifiable application (see definition 1.4).

DEFINITION 1.4.– CERTIFIABLE APPLICATION. A certifiable software application is a software application that has been created to be certified.

The SPARK Ada language [BAR 03] is a programming language, which is a sub-assembly of Ada. All the complex structures of Ada regarded as risky or not allowing for an easy safety demonstration are not to be found in SPARK Ada. A mechanism enabling the addition of annotations in the code has been introduced.

SPARK Ada tools contain a compiler, but also a verifier for annotations. There is a free version of the SPARK Ada tools⁴. Chapter 1 of a future book by the same authors (to appear in 2012) will provide the opportunity to present the SPARK Ada tool as well as some industrial examples of its implementation.

1.2.2.2. The C language

The C⁵ language [KER 88] was one of the first languages to be made available to developers to create complex applications. The principal difficulty of C resides in the partial definition of the language, which means that different compilers generate one executable with different behaviors. It has since been subject to a standardization process by the ANSI [ISO 99].

Regarding the use of the C language [ISO 99], contingent upon the safety level required, the CENELEC EN 50128 [CEN 01] standard recommends defining a sub-assembly of the language (see Table 1.1), the execution of which is controllable.

Table 1.2 (an extract from the new version of the CENELEC EN 50128 [CEN 11] standard) shows that there was sufficient experience feedback for the Ada, C and C++ languages, enabling no further explicit mention of the notion of sub-assembly of the language as it is taken as established.

Figure 1.6 presents a piece of C code, which can give two different codes contingent upon anomaly (a) or (b) that is implemented.

This example emphasizes the weaknesses of the C language, small programming errors that are not detected in the compilation. It should be noted that this type of error is detected if the programming language used is Ada.

4 To find out more about AdaCore and free tools such as GNAT and SPARK Ada, visit: www.libre.adacore.com.

5 Although “Kerdigan and Ritchie” [KER 88] does not follow ANSI on the C language, it remains one of the most interesting books on the subject.

10 Formal Methods

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
ADA	R	HR	HR	HR	HR
MODULA-2	R	HR	HR	HR	HR
PASCAL	R	HR	HR	HR	HR
C or C++	R	R	R	R	R
C #	R	R	R	R	R
JAVA	R	R	R	R	R

Table 1.2. New CENELEC EN 50128 [CEN 11] – Table A.15 (partial)

Certain weaknesses of C may be overcome by implementing some programming rules; by way of an example, to avoid an anomaly of the type if (a = cond) instead of if (a == cond), a rule of the following form can be implemented: “in the framework of comparison with a variable, this must be in the left part of the expression”.

From 1994, some experience feedback regarding the implementation of C (see for example [HAT 94]) has revealed that it was possible to define a sub-assembly of C usable to create software applications needing a high level of safety (SSIL3, SSIL4).

In fact, since the end of the 1990s, the MISRA-C [MIS 98, MIS 04] standard, which was developed by the *Motor Industry Software Reliability Association* (MISRA⁶), has become a standard for the C language.

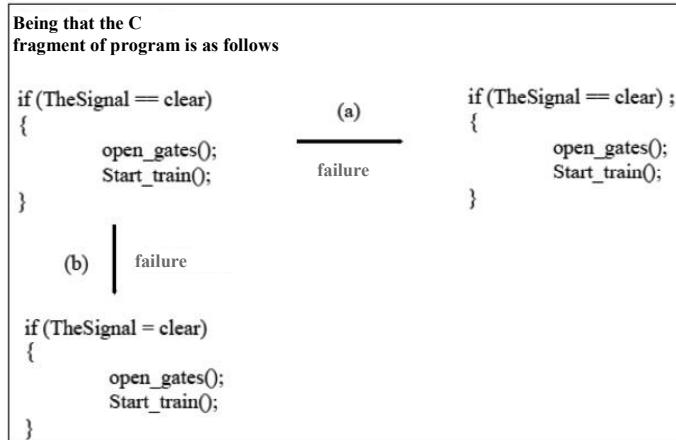


Figure 1.6. Example⁷ of a fault in C

⁶ To find out more, visit: www.misra.org.uk.

⁷ This example is retrieved from [ADA 05].

MISRA-C [MIS 04] specifies some programming rules (see the examples of Table 1.3) making it possible to avoid execution errors provoked by poorly defined constructions, unforeseen behaviors (certain C language structures are not completely defined) and misunderstandings between those people in charge of production (legible code, code with implicit cast, etc.). Several tools enable the MISRA-C rules to be automatically verified.

The MISRA-C [MIS 04] standard repeats some rules, which are explicit in several standards (see for example 14.4 and 14.7):

- rule 14.4: in the EN 50128 standard – Table A.12 or the IEC 61508 standard – Table B.1;
- rule 14.7: in the EN 50128 standard – Table A.20 or the IEC 61508 standard – Table B.9;
- etc.

MISRA-C [MIS 98] was created in 1998 and updated in 2004 [MIS 04], which shows that some experience feedback has been made use of.

Id	Status ⁸	Description
Rule 1.1	Required	All of the code must conform to the ISO 9899:1990 norm “Programming languages – C”, amended and corrected by ISO/IEC9899/COR1: 1995, ISO/IEC/9899/AMD1: 1995 and ISO/IEC9899/COR2: 1996.
Rule 5.4	Required	Each tag is a single identifier.
Rule 14.1	Required	There must not be any dead code.
Rule 14.4	Required	No unconditional jumps (goto) in programs.
Rule 14.7	Required	A function must have a single output point at the end of the function.
Rule 17.1	Required	The pointer arithmetic can only be used for pointers which address a table or tabular element.
Rule 17.5	Advisory	An object declaration must not contain more than two levels of pointer indirection

Table 1.3. Some MISRA-C: 2004 [MIS 04] rules

⁸ A MISRA rule can be “Required” or Advisory. A rule tagged “required” should be taken into account by the developer, a rule tagged “advisory” can be ignored.

12 Formal Methods

The principal difficulty of C remains the choice of a compiler having sufficient experience feedback at its disposal for the chosen target and safety level to be achieved.

In the absence of a precise and complete standard, there is currently no certification process for C compilers, even though there are initiatives such as [LER 09]⁹.

1.2.2.2.3. Object-oriented language

As already stated, the “object-oriented” aspect is not taken into account by the CENELEC EN 50128 ([CEN 01], DO 178 [ARI 92], CEI/IEC 61508 [IEC 98], ISO 26262 [ISO 09]) standards applicable to safety-critical applications.

The “object-oriented” aspect is cited in the CENELEC EN 50128 [CEN 01] standard, but the constraints applying to languages do not allow for development of a safety-critical application (SSIL3 and SSIL4) with this type of language (see Table 1.4).

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
No dynamic objects	-	R	R	HR	HR
No dynamic variable	-	R	R	HR	HR
Limited usage of pointers	-	R	R	R	R

Table 1.4. CENELEC EN 50128 [CEN 01] – Table A.12

As shown by Tables 1.1 and 1.2, the C++ language [ISO 03] is cited as applicable but certain recommendations are not compatible with the use of an object-oriented language, as shown in Table 1.4.

C++ was developed during the 1980s; it proceeded from an improvement of the C language. C++ introduces the notion of class, inheritance, virtual functions, and overload. It was standardized by the ISO in 1998 and in 2003 [ISO 03].

Since the beginning of the 2000s, many attempts have been made to define a framework enabling the use of the C++ language for the development of high-safety level applications (SSIL3, SSIL4). The following works can be cited:

⁹ It is to be noted that [LER 09] features a verification job by a limited C compiler, but that if the term “certification” is to be introduced, it does not concern certification by an external body. Certification does not only involve demonstration that the compiler is correct but must also apply to the process implemented, to the elements produced (documents, sources, etc.) and to the tools used (Ocaml, Coq, and other tools must be demonstrated, like safety).

- JSF++ (*Join Strike Fighter C++*), which has published a guide [LOC 05] regarding the current work, notably the MISRA-C: 1998 [MIS 98] standard;
- MISRA, which has developed the MISRA-C++: 2008 [MIS 08] standard; Table 1.5 shows some MISRA-C++: 2008 rule examples;
- OOTIA¹⁰ (*Object Oriented Technology in Aviation*), which has published several guides [OOT 04a, OOT 04b, OOT 04c, OOT 04d].

Id	Status ¹¹	Description
Rule 0-1-1	Required	A software must not contain non-achievable code.
Rule 0-1-2	Required	A software must not contain non-executable paths.
Rule 1-0-1	Required	All the code must conform to ISO/IEC 14882: 2003 “ <i>The C++ standard incorporating Technicql Corrigendum 1</i> ”.
Rule 2-10-4	Required	A “class”, “union” or “enum” denomination must be a single identifier.
Rule 5-2-3	Advisory	Operation of “cast” of a base class towards a derived class must not be made using polymorphous types.
Rule 15-5-1	Required	A class destructor must not quit with an exceptions.
Rule 17-0-4	Document	All libraries must conform to MISRA-C++.
Rule 18-0-1	Required	The C library must not be used.

Table 1.5. Some MISRA-C++: 2008 [MIS 08] rules¹²

C++ [ISO 03] is therefore quite an old language. Some approaches identifying the weak points of C++ and proposing some rules appeared quite early on [MEY 98, SUT 05], but the definition of a framework for using C++ for high safety-level applications is quite recent [OOT 04, LM 05, MIS 08], which explains how applications in C++ are to be found even at SSIL2.

As Table 1.5 shows, the MISRA C++: 2008 [MIS 08] standard introduces some rules inspired by those current for the C language, but which do not allow for all the difficulties of C++ to be taken into account.

10 To find out more, visit: www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot.

11 A MISRA rule may be “required”, “advisory”, or “document”. A “required” rule must mandatorily be implemented by the developer, an “advisory” rule cannot be disregarded, even if it is not mandatory to implement it and a “document” rule is mandatory.

12 [MIS 08] introduces 198 rules.

14 Formal Methods

Under various pressures (a reduction in the number of Ada and C programmers, for example), the updating of standards such as CENELEC EN 50128 or D0 178 has given rise to initiatives seeking to introduce object-oriented languages.

	SSILO	SSIL1	SSIL2	SSIL3	SSIL4
The classes should only have a single objective.	R	R	R	HR	HR
Inheritance used only if the derived class is a refinement of its base class.	R	HR	HR	HR	HR
Depth of inheritance limited by coding norms.	R	R	R	HR	HR
Neutralization of operations (methods) under strict control.	R	R	R	HR	HR
Multiple inheritance used only for interface classes.	R	HR	HR	HR	HR
Inheritance from unknown classes.	-	-	-	NR	NR

Table 1.6. New CENELEC EN 50128 [CEN 11] – Table A.23

Thus, the new version of the CENELEC EN 50128 [CEN 11] standard has expanded the list of usable object-oriented languages to JAVA and to C#, as Table 1.2 shows. Yet this new version of the standard introduces some restrictions (limitations on inheritance) as Table 1.6 shows.

The C version of the D0 178 standard should use a specific annex aiming to define the constraints of implementing an object-oriented language for production of a safety-critical application.

As for C, the difficult point with C++ remains the demonstration that the compiler for the chosen target and associated libraries complies with the safety objectives, which have been defined by safety studies. As there is currently no certification for C++ compilers, it will be necessary to establish a justification based on experience feedback and/or qualification.

1.2.2.3. Programmable controller

The “continuous processes” CEI/IEC 61511 [IEC 05] industry standard brings precision in restricting the initial field of application of the CEI/IEC 61508¹³ [IEC 98] standard to the traditional context of continuous processes.

¹³ In Chapter 10 of [BOU 09b], there is a presentation on the use of SISs (Safety Instrumented Systems) in industry and Chapter 6 of [BOU 11c] presents a complete example



Figure 1.7. Example¹⁴ of an actual programmable controller

The CEI/IEC 61511 [IEC 05] standard applies in the framework of E/E/PE systems in the context of safety instrumented functions for controlling a process (chemical products, petrol refining, petrol and gas production, non-nuclear electricity production). It does not apply to manufacturers or system providers, who must use the CEI/IEC 61508 standard.

For synthesizing, the CEI/IEC 61511 standard is used when already-certified components (of programmable controllers) are used and when they are programmed with limited variability languages.

of implementation of a programmable safety controller for managing ventilation of the Soumagne tunnel and the Walhorn trench of the high-speed L3 line.

14 Photograph taken by Jean-Louis BOULANGER.

16 Formal Methods

With respect to programming languages, the CEI/IEC 61511 [IEC 05] standard only concerns programming languages usual to the industry, which are specified through the CEI/IEC 61131-3 [IEC 03] standard.

This in no way prevents anyone from programming their E/E/PE system with the Ada or C++ languages, but in these cases, it is necessary to refer to the CEI/IEC 61508 [IEC 98] standard, which defines how these languages can be used.

Part 3 of the CEI/IEC 61131-3 [IEC 03] standard describes the principles for producing an application for a programmable controller and introduces several technical terms: FBDs for functional block diagrams, SFCs for *sequential function charts*, suites of instructions (written IL), structured text (ST), and contact diagrams (LD for *ladder diagrams*).

The languages introduced by the CEI/IEC 61131-3 [IEC 03] standard are used more and more for making software applications, as these graphic languages are easily assimilated and standardized. There are thus some tools which cover all the formalisms described in the CEI/IEC 61131-3 [IEC 03] standard.

One of the important points regarding these tools is that they are supplied for making programmable controller applications and may thus come with a certificate for a safety level, which might be SSIL2 or SSIL3.

The ControlBuild tool is presented in Chapter 4 of this book. ControlBuild is used by several actors in the rail sector to make SSIL2-level safety-critical software applications.

1.2.2.4. Qualification

The different standards (DO 178, ISO 26262, and CENELEC EN 50128) reveal/or will reveal the notion of the “qualification file”.

Qualification of a tool depends on its impact on the final product:

- an impact on the executable (compiler, data generator, etc.);
- an impact on the verification and/or validation (test tools, etc.);
- no impact.

The qualification file can rely on different activity types:

- analysis of the use of a certificate (commonly termed *cross-acceptance*): if a certificate exists, it is necessary to verify that the tool is suited to what it is intended to be used for (same normative referential, similar safety level, etc.);

– experience feedback construction: an inventory of all uses (those that are known and those of the business). It must then be capable of justifying the version used, levels of safety, size, sectors, etc.;

– establishment of a qualification; test runs of the tool can be undertaken so as to show that the former is usable for a given safety level.

The notion of qualification is an important notion, which is linked to implementation of tools in the production process of a software application.

All tools used can be subjected to a qualification phase.

Regarding formal methods, the question of qualification of tools is of fundamental importance, since the complexity of technologies implemented (prover, *model-checking*, etc.), the confidentiality aspect (licensed algorithm, etc.), the innovation aspect (new technology, few users, etc.), and the maturity aspect (product resulting from research, product made using a “free” license, etc.) do not permit trust to be built easily.

1.2.2.5. Advantages/disadvantages

The programming language used must facilitate anomaly detection and for this, it is necessary to implement strong typing and modular and structured programming.

	SSIL1	SSIL2	SSIL3	SSIL4
Adequate programming language	HR	HR	HR	HR
Strongly typed programming language	HR	HR	HR	HR
Sub-assembly of the language	-	-	HR	HR
Certified tools	R	HR	HR	HR
Tools: confidence in which has increased as a result of use	HR	HR	HR	HR
Certified translator	R	HR	HR	HR
Translator: confidence has been increased as a result of use	HR	HR	HR	HR
Library of modules software and proven/verified components	R	HR	HR	HR

Table 1.7. IEC 61508 [IEC 98, Part 3] – Table A.3

As Table 1.7 shows, one of the principal difficulties while using a language such as C or C++ resides with demonstrating the “proven” character of the translator. Use of a programming language:

- N.3: necessitates definition of a sub-assembly of the language (limiting the difficulties) and using a set of design and coding rules;
- N.4: necessitates using sufficient experience feedback to show that the tools and the compiler especially, are usable for the expected safety level.

1.2.3. Specification and architecture

1.2.3.1. Requirements management

The standards applicable in different industries render identification of the software application requirements mandatory. It should be noted that the literature presents several terms: prescriptions, recommendation, requirement, constraint and property, which seem equivalent. The generalist CEI/IEC 61508 [IEC 98] standard uses the notion of *prescription*, but the most-used term remains the *notion of requirement*.

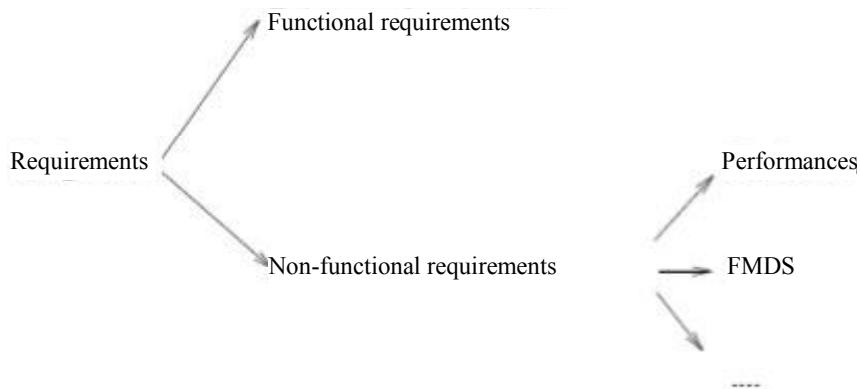


Figure 1.8. Different types of requirements

As Figure 1.8 shows, the requirements are of at least two types: functional and non-functional requirements (safety, reliability, performance, etc.).

In Chapter 2 of [RAM 09] and Chapter 3 of [RAM 11], we have presented some examples of requirements management in the automobile and rail industries.

Table 1.8 is an extract from [STA 94], which shows that over 43% of the causes of failure in production originate from incomplete requirements, something lack in the needs description and unrealistic expectations.

In [STA 01], it is mentioned that implementation of a requirements management environment is the best way to have a strong impact on the success of a project.

Definition of a minimal set of requirements makes it possible to have a basis that can be managed, the tool then being a communication vector between teams.

Description	%
Incomplete Requirement	13.1%
Lack of user involvement	12.4%
Lack of Resources	10.6%
Unrealistic expectations	9.9%
Lack of executive support	9.3%
Changing Requirement/specification	8.7%
Lack of planning	8.1%
Didn't need it any longer	7.5%

Table 1.8. Distribution of the causes of failure

The difficulty then resides in defining the notion of requirement. There are currently several studies, which attempt to identify what a requirement is and how to take it into account. [HUL 05] features one of the most complete summaries. We will recall the following definition, which has resulted from studies carried out by manufacturers.

DEFINITION 1.5.– REQUIREMENT. A requirement is a formulation that translates a need and/or some constraints (technical, costs, delays, etc.). This formulation is written in a language that may be natural, mathematical, etc¹⁵.

Figure 1.9 shows how the recommendations of the client can be declined on the system and how processing can continue right down to the software and hardware type elements.

In the context of the verification phase of level n_i , it is necessary to show that the requirements of this level are in relation to the higher n_{i-1} level. The links between requirements of different levels are made during a phase of traceability.

Implementation of traceability involves defining at least one link between two objects. In the context of the requirements, traceability links must make it possible to show that an n_i -level requirement is linked to a need of the preceding n_{i-1} level.

The inverse link makes it possible to show that no requirement has been forgotten during the production process.

¹⁵ The AFIS is *l'Association française d'ingénierie système* (French Association of Systems Engineering). One of its work groups is specifically dedicated to requirements management. For more information: www.afis.fr.

20 Formal Methods

As Figure 1.10 shows, there are currently several basis transformations of requirements. Among these, two cases are particularly interesting; the addition and the abandonment of a requirement; in both cases, it is absolutely necessary to attach a justification to the requirement.

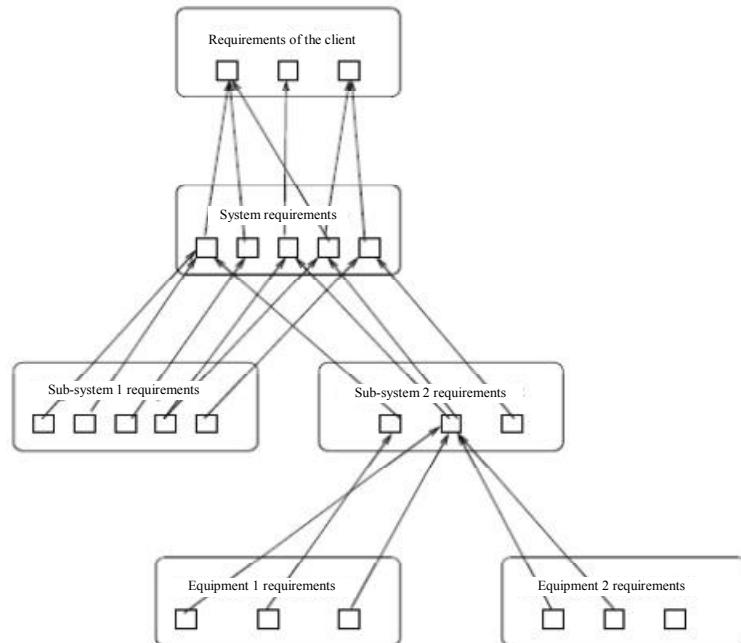


Figure 1.9. Partial traceability between the requirements of the client and requirements linked to the equipment

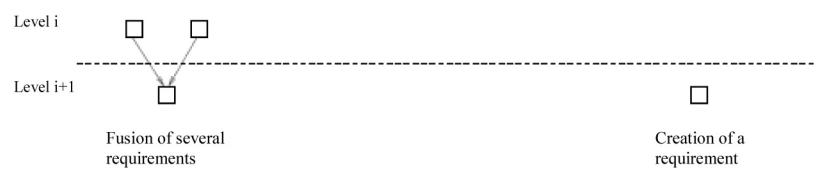
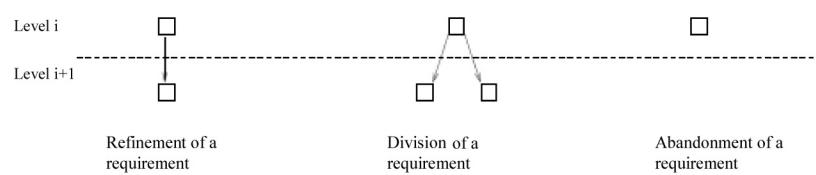


Figure 1.10. Basic transformation of requirements

1.2.3.2. Specification

The specification of a software application is thus at the very least, a set of requirements. A first analysis of the specifications provided by the client must make it possible to identify functional needs. Figure 1.11 presents the proposed process.

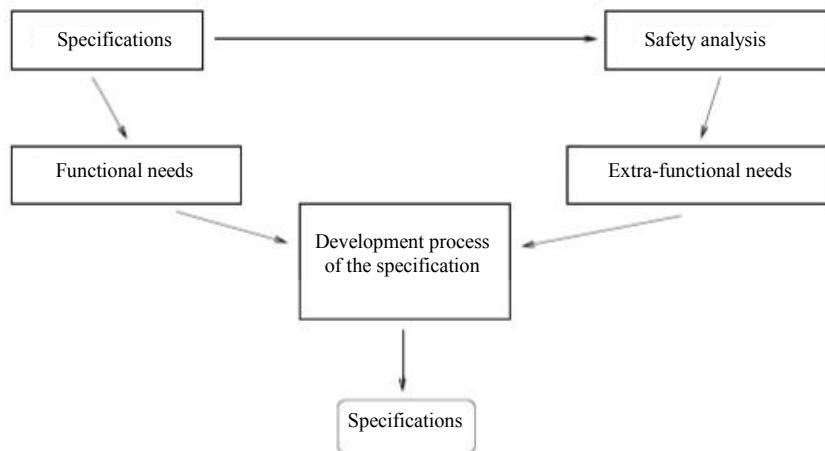


Figure 1.11. Development of the specification

In parallel with this work, it is possible to begin analyses linked to dependability, the objective of which is to define the non-functional requirements: safety requirements but also availability, reliability or other requirements.

It is, however, necessary to know a bit more about the needs of the software application. Within the application, it is thus necessary to introduce:

- interfaces with the environment;
- the notion of state in establishing a partition between safe functioning, decline and dangerous states;
- the notion of correct behavior and of dangerous behavior;
- the notion of requirement linked to safety; this type of requirement must allow for characterization of dangerous behaviors;
- the integrity level of the software (written SSIL/DAL).

The environment of the software application is composed of interfaces with the hardware resources (memory, specific address, input/output, *watchdog*, etc.), with other software applications (base software, related application, etc.) and/or with the operating system.

Figure 1.12 reveals a software application environment, which is composed of three entries (E_i), two exits (S_j), and three interfaces (I_k) with the hardware resources (for example, access to a specific memory address).

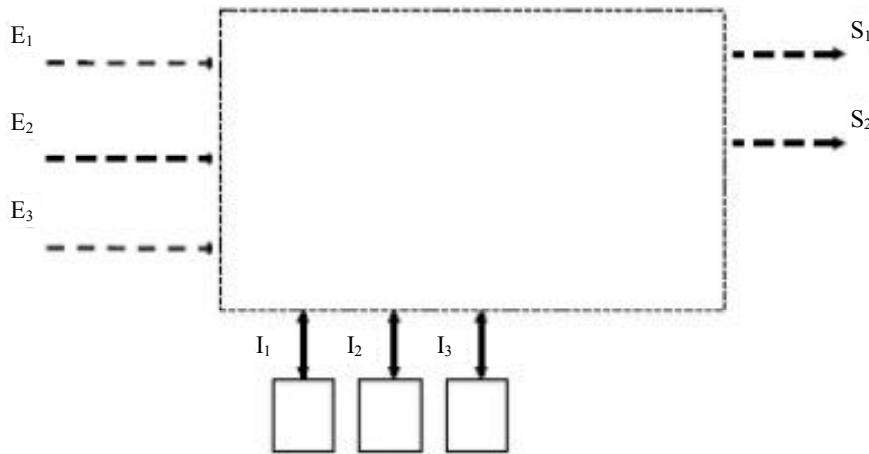


Figure 1.12. Software application environment

Standards recommend that the specification of a software application be composed from a textual description of the need (the requirements) and all the notations necessary for facilitating understanding of the need. So, for the EN 50128 [CEN 01] standard, there is Table A.2 (Table 1.9).

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
Formal methods comprising for example, CCS, CSP, HOL, LOTOS, OBJ, VDM, Z, B	-	R	R	HR	HR
Semi-formal methods	R	R	R	HR	HR
Structured methodology comprising for example, JSD, MASCOT, SADT, SDL SSADM and YOURDON	R	HR	HR	HR	HR

Table 1.9. CENELEC EN 50128 [CEN 01] – Table A.2

Classically, designers of a software application go directly from the textual requirements to the code without verifying the the requirements or mastering their unity.

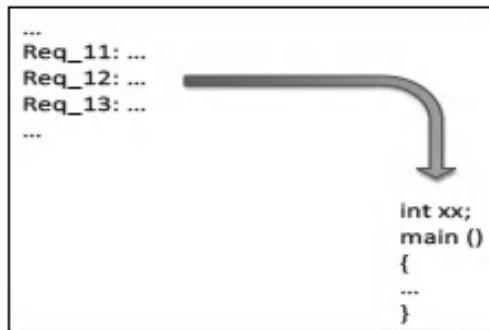


Figure 1.13. From the requirements to the code

Successful development of a maintainable software application consists of going through at least two stages:

- a formalization of the requirements phase (see Figure 1.15). This formalization phase can rely on structured methods (for example, Figure 1.14 in SADT¹⁶ makes it possible to establish a functional analysis, etc.), on semi-formal or formal methods (controller, Petri net, Grafset, B-method¹⁷, SCADE¹⁸, etc.);
- an architecture phase (see section 1.2.3.3).

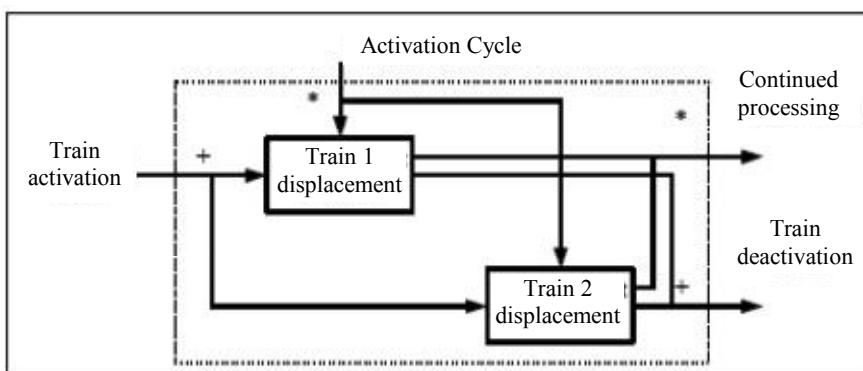


Figure 1.14. Example of static introducing different types of communication

16 The acronym SADT stands for *Structured Analysis and Design Technic*. This method was invented by the Softech company in the US. The SADT method is a method of analysis by successive levels of descriptive approach of a set, whatever it may be.

17 In Chapter 2, we present an example of the use of SADT [LIS 90] and of the B-method [ABR 96, BEH 96] in the context of the SAET-METEOR project [BOU 06].

18 Chapters 6 and 7 presents some examples of SCADE [DOR 08] use.

The formalization phase is important because it enables translation of an abstract requirement into modeled elements, like the following P1 property:

P1: “there must not be any risk of collision”

which can be translated into set-theoretic logic of the first order in $\forall \{t_1, t_2\} \in [T]$, hence $D_{t_1} \cap D_{t_2} = \emptyset$, if $t_1 \neq t_2$, with D_{t_i} which is the domain of train speed i and $[T]$, which is the whole set of trains.

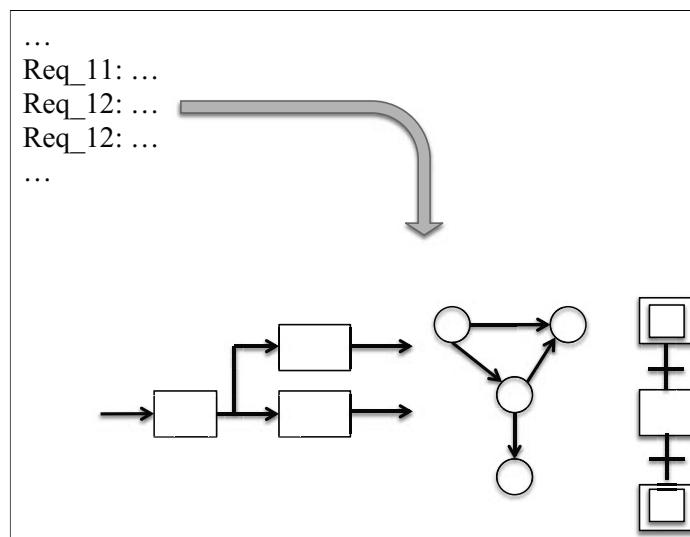


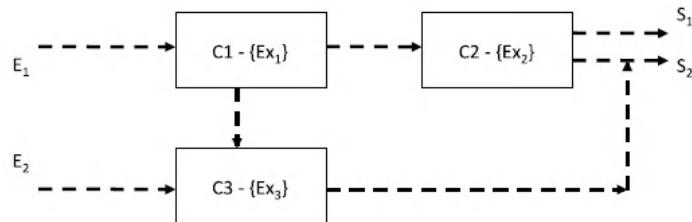
Figure 1.15. Formalization of requirements

1.2.3.3. Architecture

Once the software application specification is created, it is then possible to establish an architecture. This architecture aims to break down the software application into components¹⁹ (or modules, following the vocabulary used).

Figure 1.16 presents an example of architecture. The software application has been broken down into three components (C_1 , C_2 , and C_3); there are interfaces between the environment (E_1 , E_2 , S_1 , S_2) and components and some interfaces between components; for each component, the list of requirements to take into account is given.

¹⁹ In the new versions of the standards, the notion of components is used whereas in preceding versions, “module” (file, functions, class, etc.) was used.

**Figure 1.16.** Example of software application architecture

1.2.3.4. Model

The new version of the EN 50128 [CEN 11] standard (Table A.3) recommends that software application architecture is based a structured method (SADT, etc.).

Yet it is possible to establish a model that is based on the techniques of Table A.17 (see Table 1.10).

	SSILO	SSIL1	SSIL2	SSIL3	SSIL4
Data modeling	R	R	R	HR	HR
Data organization	-	R	R	HR	HR
Command data flow diagram	R	R	T	HR	HR
Finite state controllers/transition schematics	-	HR	HR	HR	HR

Formal methods	R	R	R	HR	HR
Sequence diagrams	R	HR	HR	HR	HR

Table 1.10. New CENELEC EN 50128 [CEN 01] – Table A.17

The creation of an M model is a means of understanding and/or stopping a problem/situation. In general, the specification phase, which allows for adaptation of the specifications, involves creation of an M model.

A model may be more or less close to the system studied, which would then be termed abstraction. The closer the model is, the closer the results obtained will be to those observed on the final system.

Another characteristic of models stems from availability (or lack of) a semantic for the language support.

The presence of a semantic enables implementation of reasoning techniques, which guarantee correction of obtained results.

A model (Figure 1.17) is, in general, composed of two complementary parts:

- a static model (Figures 1.14 and 1.16) describing the entities constituting the system and the states that can be associated with them;

- a dynamic model describing authorized changes of state.

Figure 1.17 presents an example of a UML model²⁰ [OMG 06a, OMG 06b], and [OMG 07] of a rail system implementing different models: use case, sequence diagram, state/transition diagram, and class diagram.

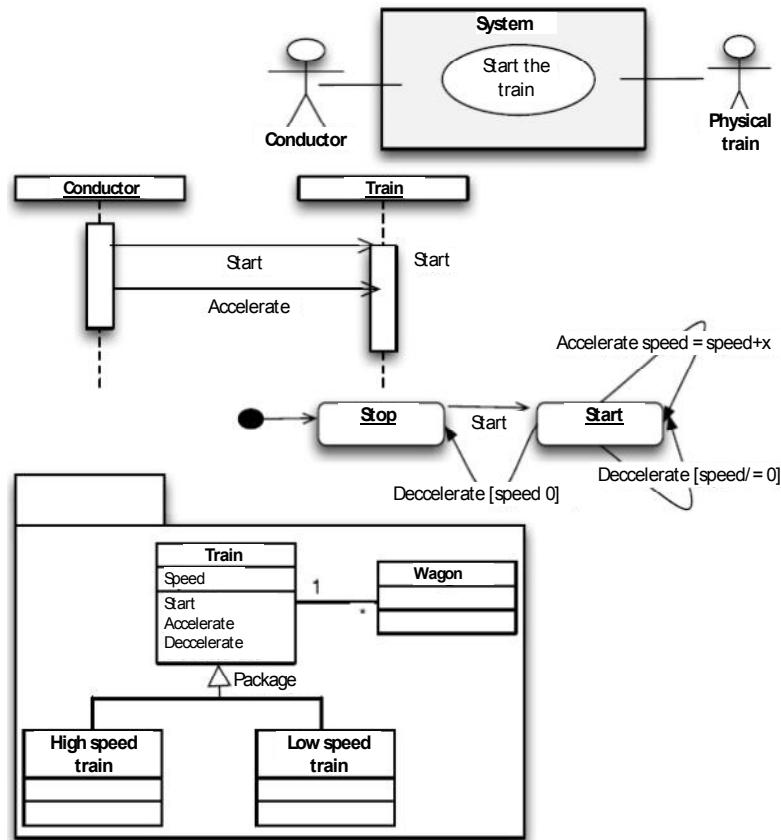


Figure 1.17. Example of a UML model

20 To find out more, visit: the OMG site: www.omg.org.

Using UML notation is thus not without certain questions [BOU 07, OKA 07]. How can we use a notation without a semantic? How can we evaluate an application based on UM notation? etc. Several studies aim to offer answers to these questions, for example [OOT 04, MOT 05].

At present, UML notation [OMG 06a, OMG 06b, OMG 07] is not recognized as a structured and/or semi-formal method in the standards, although many would like to implement it completely or partially.

In [BOU 07, BOU 09a] and Chapter 19 of [RAM 09], we have shown how UML notation can be used to create some models of safety-critical systems.

In the context of the RT3-TUCS, ANR-SAFECODE, and ANR-RNTL-MEMVATEX projects, we have studied different ways of introducing UML as a means of modeling a safety-critical system; see for example [RAS 08, BOU 08, IDA 07a, IDA 07b, OKA 07, BOU 05].

In recent years, several studies have enabled a formalization of UML notation to be proposed through returning to formal languages, for example [IDA 09, IDA 06, MAR 04, MAR 01, MAM 01, LED 01]. Works such as [MOT 05] must also be noted, which have made it possible to suggest some additional rules for verifying UML models.

1.2.3.5. *Advantages/disadvantages*

A specification and/or architecture cannot be reduced to a list of requirements; it is necessary to use a supporting model of requirements. This supporting model (see Table 1.10) will make it possible to verify the coherence and completion of requirements. Creation of a software application specification and/or architecture (though the same could be said of the design phase) entails:

- N.5: creating a model associated with the requirements;
- N.6: having a method of verifying the model so as to verify the coherence and/or completion of the requirements;
- N.7: being able to continue processing (Figure 1.18) of the specification phase up to the design phase; not to mention the coding phase.

1.2.4. *Verification and validation (V&V)*

1.2.4.1. *Presentation*

Creation of a software application must take into account the design of the software application, but also enable the software application to reach a certain quality standard.

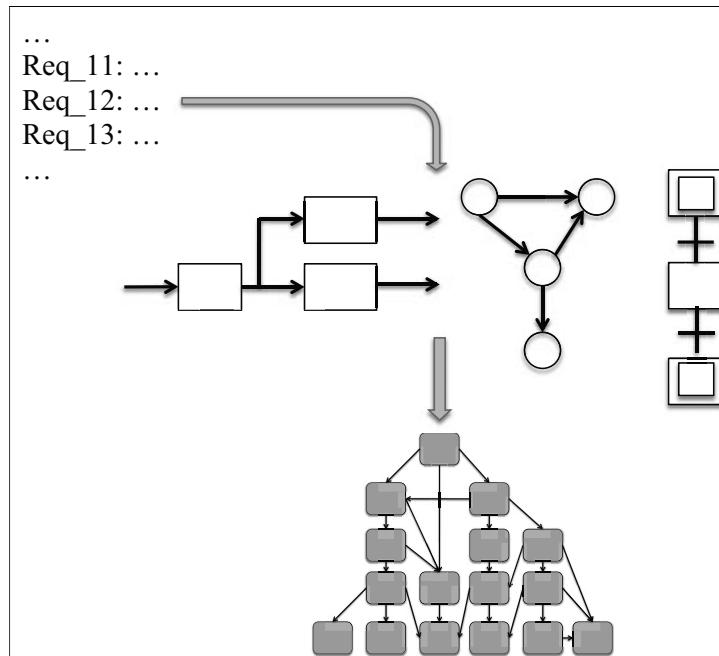


Figure 1.18. From the requirements to the architecture

Achievement of a quality standard requires demonstration that:

- no fault has been introduced during the design;
- the product corresponds to the needs that have been identified.

Above all else, it is necessary for us to recall what verification and validation are.

DEFINITION 1.6.– VERIFICATION. Confirmation by tangible proof that the specified requirements have been met at each stage of the production process.

DEFINITION 1.7.– VALIDATION. Confirmation by tangible proof that the anticipated requirements for a specific use or application have been met.

The two preceding definitions are extracts from the ISO 9000:2008 standard (see [ISO 08]). They introduce the notions of requirements and of proof.

To be more precise, we can reprise the presentation made by I. Sommerville [SOM 07]. He cites Boehm who, in 1979, stated that:

- verification ensures that the product conforms to its specification;
- validation ensures that the system implemented corresponds to the expectations of future users.

Through this definition, validation aims to show accordance of the product with the initial need.

Figure 1.19 depicts the main problem in production of a software application. In effect, there is a need to fulfill and there is a fulfillment; verification is to show that the set of needs is accounted (specified functions) for by the fulfillment (coded functions) and that there are no unforeseen elements.

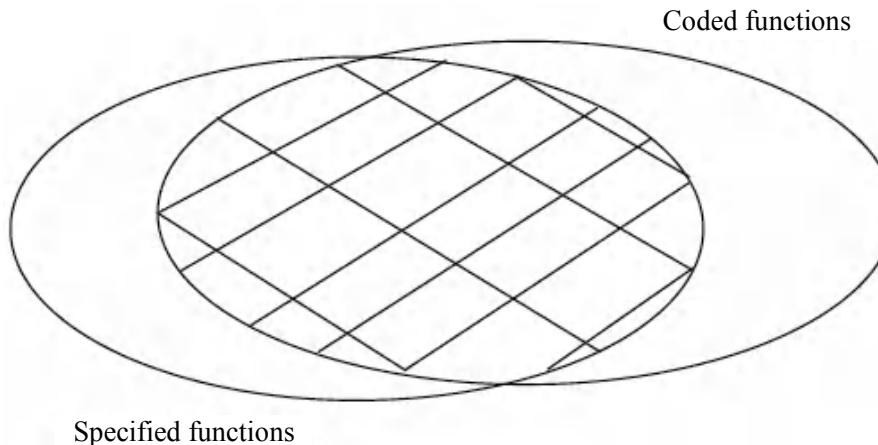
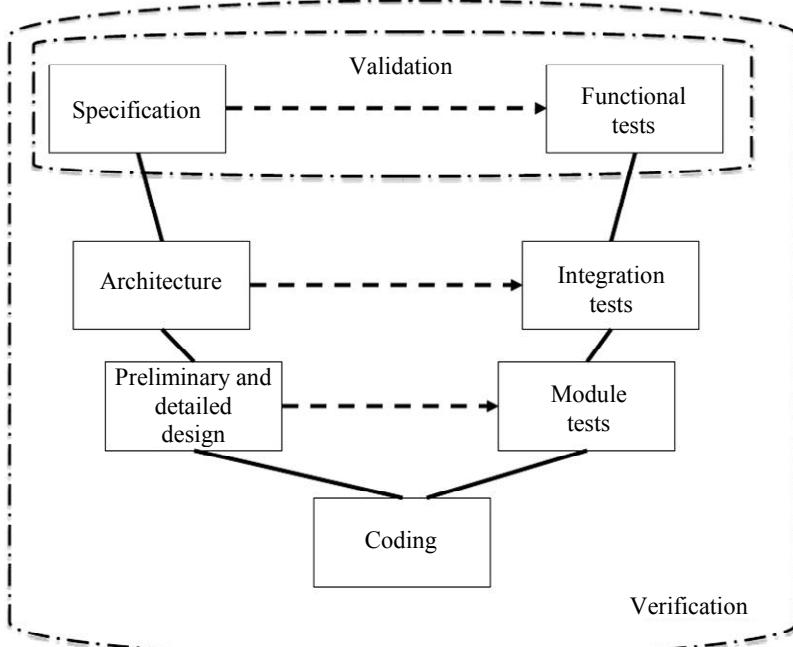


Figure 1.19. Problem in software development

The development team will always have good reason to introduce bits of undesired code (re-usable functions, service addition, etc.) and for not taking into account the whole set of needs (technical difficulties, omissions, etc.).

The definitions allow us to conclude (see Figure 1.20) that verification applies to every stage of the V-cycle and that validation is an activity, which aims to show that the final product abides by the specification; this activity involves functional tests, also referred to as validation tests.

As Figure 1.20 shows, verification involves seeking faults within the V-cycle; validation involves demonstration that the product corresponds to its need or to its localization at the level of the high part of the V-cycle. Verification overlaps with validation.

**Figure 1.20.** *V&V on the V-cycle*

1.2.4.2. Verification

In the context of the 61508 [IEC 98], for each phase, “verification” is an activity, which consists of demonstrating by an analysis and or by testing that the deliverable fulfill the methodological requirements and the functional and no-functional requirements.

Here is a list of verification activities:

- reviews relative to the phase exit (documents concerning every phase of the safety lifecycle) intended to ensure conformity with the phase objectives and prescriptions and taking into account the specific input at this phase;
- design reviews;
- tests carried out on the products made available to ensure that their functioning conforms to their specification;
- integration tests carried out during the element-by-element assembly of different parts of a system, from environmental testing, to ensure that each and every part functions in accordance with specifications.

Verification is a process that deals with the production phases and which concerns:

- the structure of the system, how it is made, with reference to standards and to the properties to be met (to verify the product);
- the resources implemented; the production processes; with reference to some rules on the working method, how one should proceed (to verify the process).

The objective of verification is to show that the product is well-made. The notion of a “well-made product” signifies that no fault has been introduced during the phases associated with production.

There are two types of verification:

- static verification, which does not require execution of all or part of the system analyzed;
- dynamic verification, which is based on all or part of the system analyzed.

Besides verifying the product, it is necessary to *verify the quality of the product*, which will be carried out by the quality team through a quality audit (on the product, on the project, or on the application of a process), review of the elements produced (documentation etc.), and control activities (monitoring of anomalies, non-conformities, client feedback, etc.).

1.2.4.3. Validation

In the context of lifecycle of a system, validation groups together activities, which guarantee and build faith in the system, its ability to satisfy anticipated uses, and achieve assigned aims and objectives.

In the CEI/IEC 61508 [IEC 98] standard, validation is the activity, which consists of demonstrating that the system, before or after installation, corresponds entirely to the prescriptions contained in the specification of this system.

So, for example, validation of the software consists of confirmation, by examination and provision of tangible proof, that the software responds to the safety prescriptions of the software specification.

Validation thus consists of showing that we have created a good product. Validation may be seen as an external verification.

1.2.4.4. Advantages/disadvantages

Development of a certifiable software application (see definition 1.4) is constrained by the requirements of standards associated with each industry

32 Formal Methods

(aeronautical [ARI 92], automobile [ISO 09], rail [CEN 01, CEN 11], nuclear [IEC 06], generic [IEC 98]).

These normative requirements recommend implementation of a “V-cycle” type development process based on the activities of verification and validation, based on the carrying out of tests (MT, IT, FT).

Activity testing implementation suffers from several problems, which are:

- the cost and complexity of activity testing;
- late detection of faults;
- the difficulty of carrying out all the tests.

This is why it is necessary to implement other practices, which must enable early detection of faults in the software application.

One of the possible orientations consists of implementing formal methods (for example, the B-method [ABR 96], SCADE²¹ [DOR 08], VDM [JON 90], Z [SPI 89], etc.) which, on the basis of a model and set of properties, enable demonstration that the software product verifies the stated properties.

It may, however, be interesting, on the basis of classic development languages (like C) to explore the program behaviors and to demonstrate that it verifies some properties.

This analysis of behaviors can be made through implementation of static analysis techniques of code based on abstract interpretation [COU 00] and proof of programs [HOA 69]. The first volume of this series [BOU 11a] is dedicated to abstract interpretation.

It should be noted that one of the difficulties in implementing these techniques lies in the absence of recognition of these techniques within the current standards. In effect, certain standards (see Table 1.16, extract from the standard CENELEC EN 50128 [CEN 01a] for example) advocate the implementation of formal methods but they do not mention the notion of abstract interpretation (or derived methods).

An improvement of V&V (verification and validation) entails:

- N.8: carrying out abstract interpretation type formal verification on the code;
- N.9: using models enabling test production (generation of test cases, model exploration, etc.);
- N.10: etc.

²¹ SCADE is distributed by Esterel Technologies; see: www.estrel-technologies.com.

1.2.5. Summary

In the preceding sections, we have identified some needs, which are still to be covered:

- N.1: reducing the number of faults introduced into the software application during the bottom-up phase of the V-cycle;
- N.2: identifying faults introduced within the software application as early as possible;
- N.3: defining a sub-assembly of the language (limiting difficulties) and using a set of design and coding rules;
- N.4: using sufficient experience feedback to show that the tools and the compiler especially, are usable at the expected safety level;
- N.5: creating a model associated with the requirements;
- N.6: using a method of verifying the model so as to verify the coherence and/or completion of the requirements;
- N.7: being able to continue processing (Figure 1.17) from the specification phase right up to the design phase, not to mention the coding phase;
- N.8: carrying out abstract interpretation-type formal verifications on the code;
- N.9: using a model that facilitates test production (generation of test cases, model exploration, etc.).

1.3. Structured, semi-formal and/or formal methods

1.3.1. E/E/PE system

The CEI/IEC 61508 [IEC 98] standard is a generic standard, which may be applied to all electronically-based and/or programmable electronic complex systems termed E/E/PE (electric/electronic/programmable electronic). This standard is edited by the *International Electrotechnical Commission*²² (IEC), which is the international standardization organization responsible for electricity, electronics, and related techniques.

The CEI/IEC 61508 [IEC 98] standard proposes a globalized approach to safety, in the sense of safety/verifiable safety²³, comparable to the ISO 9001:2008 [ISO 08]

²² To find out more about the IEC, visit: www.iec.ch.

²³ The CEI/IEC 61508 [IEC 98] standard does not cover the confidentiality and/or integrity aspects that are linked to establishment of precautions aiming to prevent non-authorized

system for quality. The CEI/IEC 61508 [IEC 98] standard is totally consistent with the convergence observable between different industrial sectors (aeronautical, nuclear, rail, automobile, machinery, etc.), the content of the CEI/IEC 61508 standard is sufficiently complex, not to mention unusual, so that the reader needs “to be guided through”. See [ISA 05] or [SMI 07].

The CEI/IEC 61508 [IEC 98] standard defines the notion of SIL (*Safety Integrity Level*). The SIL allows the safety level of a system to be quantified. Safety integrity level 1 (SIL1) is the lowest and safety integrity level 4 (SIL4) is the highest.

The CEI/IEC 61508 standard outlines the requirements necessary to create and implement a system and/or software, which achieves its safety objectives (SIL). The higher the safety level to be obtained, the more severe are the requirements to be implemented (financial impact). In effect, to obtain a low probability for a dangerous failure, greater financial investment is necessary.

The SIL (*Safety Integrity Level*) enables prescriptions concerning safety functions integrity to allocate to E/E/PE systems to be specified.

For several years, the generic CEI/IEC 61508 [IEC 98] standard has devolved through sister-standards that cover different sectors, as Figure 1.21 shows.

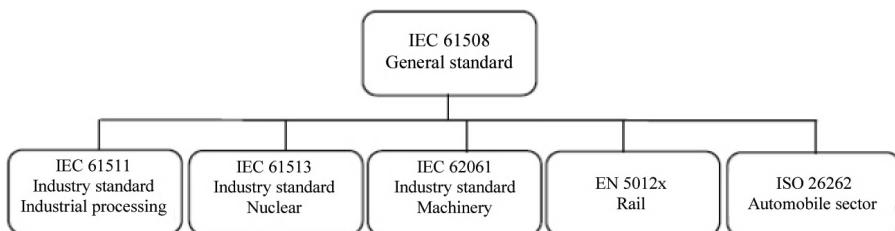


Figure 1.21. The CEI/IEC 61508 [IEC 98] standard and these decensions²⁴

The CEI/IEC 61508 [IEC 98] standard is used for the development of “safety-critical” software in the automation and automobile sectors, as well as for the installation of industrial process controls. Part 3 of the CEI/IEC 61508 standard is dedicated to production of software applications.

persons from damaging and/or affecting safety achieved by the E/E/PE system or systems. Notably, management of the network aspect so as to avoid intrusions comes to mind here.

24 Figure 1.21 reveals a link between the CEI/IEC 61508 standard and the CEI/IEC 61513 [IEC 01] standard, but this link is incorrect. In effect, nuclear standards existed well before the CEI/IEC 61508 standard and the link is only a link of naming.

1.3.2. Rail sector

Today, rail projects are governed by some documents (decrees, orders, etc.) and a normative referential (CENELEC²⁵ EN 50126 [CEN 00], EN 50129 [CEN 03], and EN 50128 [CEN 01a]), which aims to define and achieve certain RAMS (*Reliability, Maintainability, Availability, and Safety*) objectives.

Three standards (see Figure 1.22) make it possible to cover aspects concerning safety/verifiable *safety* of the system, right down to the hardware and/or software elements.

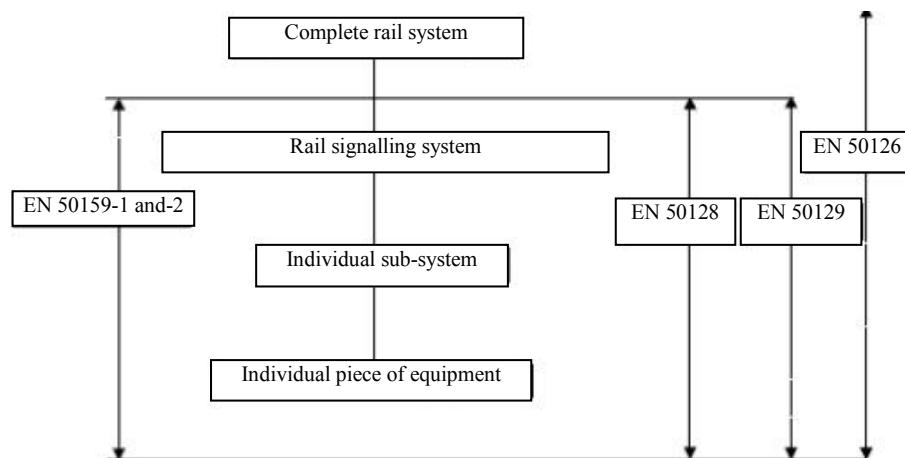


Figure 1.22. CENELEC referential

For the notion of safety-confidentiality, the CENELEC standard is completed by the EN 50159-1 [CEN 01b] and EN 50159-2 [CEN 01c] standards, which are appropriate if the application uses some open or closed networks.

These standards now have a European statute, applying to all the member states of the community, and have been endorsed by the International Electrotechnical Commission; application of CEI/IEC 61508 [IEC 98] to the rail sector also conferring an international statute on them.

Safety, as an element of *safe functioning*, is obtained by establishment of designs, methods, tools, all throughout the lifecycle. A safety study necessitates

²⁵ CENELEC for *Comité européen de normalisation électrotechnique* (European Electrotechnical Standardization Committee), see: www.cenelec.eu.

36 Formal Methods

analysis of the failures of the system. It must set out to identify and quantify the severity of the potential consequences and as far as possible, the predictable frequency of these failures.

Among the risk reduction actions enabling achievement of an acceptable level of risk, the CENELEC EN 50129 [CEN 03] standard describes allocation of safety objectives to the functions of the system and its sub-systems. The safety integrity level (SIL) is defined as one of the discrete levels for specifying safety integrity requirements of safety functions allocated to safety systems. The SIL goes from the value of 1 up to the value of 4.

The notion of SIL in the sense of the standard CENELEC EN 50129 [CEN 03] is not the same as in the standard CEI/IEC 61508 [IEC 98]. The differences concern the identification method and the impact. Usage of SIL is identical in all standards.

The CENELEC EN 50128 [CEN 01a] standard is specifically dedicated to the development aspects of software for the rail sector. Regarding software, the SSIL (Software SIL) makes it possible to define different levels of criticality: from 0 (no danger) to 4 (critical).

The CENELEC EN 50128 [CEN 01a] standard specifies *the procedures, the technical prescriptions* applicable to the development of programmable electronic systems used in applications for rail command and protection. The CENELEC EN 50128 standard is thus not normally applicable to all software applications of the rail sector. The EN 50155 [AFN 01] standard is normally applicable for all the embedded applications in a train. However, the EN 50155 standard calls directly the CENELEC EN 50128 standard.

1.3.3. Taking into account techniques and formal methods

It is interesting to note that the definitions of the B and C annexes (Part 7 of the CEI/IEC 61508 standard) use some articles and/or books, which are dated between 1970 and 1998, as references. It seems necessary to bring this annex up to date.

1.3.3.1. Definitions

1.3.3.1.1. Semi-formal method

According to the CEI/IEC 61508 (B.2.3) standard, a *semi-formal method* offers a means of developing a description of a system at a stage of development (specification, architecture, and/or design).

The description can be analyzed or animated so as to verify that the system modeled satisfies the requirements (real and/or specified). It is indicated that state diagrams (finite controllers) and temporal Petri nets are two semi-formal methods.

The CENELEC EN 50128 standard identifies semi-formal methods (see Table 1.11).

In the outline of the new CENELEC EN 50128 [CEN 11] standard, semi-formal methods as techniques have disappeared. This is due to the ambiguity that existed.

In effect, how do we define the limit between formal and semi-formal and between semi-formal and structured? Certain manufacturers consider the existence of a modeling tool as enough to demand establishment of a semi-formal method.

The existence of a tool has only a slight link with the existence of a semantic; see for example, the Labview environment, which, although graphic, does not use a semantic. The question is all the more relevant for environments based on UML notation.

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
Logic/function block diagrams	R	R	R	HR	HR
Sequence diagrams	R	R	R	HR	HR
Data flow diagrams	R	R	R	R	R
Finite state controllers/state transition schematics	-	R	R	HR	HR
Temporal Petri nets	-	R	R	HR	HR
Decision/truth table	R	R	R	HR	HR

Table 1.11. CENELEC EN 50128 [CEN 01] – Table A.18

1.3.3.1.2. Formal method

Section B.30 of the CEI/IEC 61508 standard indicates that a formal method (HOL, CSP, LOTOS, CCS, linear temporal logic (LTL), VDM²⁶ [JON 90] and Z²⁷ [SPI 89]) enables an unambiguous and coherent description of a system at a stage of

²⁶ In the IEC 61508 standard, the references to VDM include VDM++ [DUR 92], which is a real-time and object-oriented extension of VDM. To find out more, visit: www.vdmportal.org.

²⁷ It is to be noted that in the IEC 61508 standard, the B-method [ABR 96] is seen as a method associated with Z.

development (specification, architecture, and/or design) to be produced. The description takes a mathematical form and can be subjected to a mathematical analysis, which may be toolled.

A formal method generally offers a notation, a technique for processing a description in this notation and a verification process for controlling correction of the requirements.

It is to be noted that the CEI/IEC 61508 standard indicates that it is possible to make transformations right down to “a logic circuit design”²⁸.

Petri nets and state machines (mentioned in the outline of semi-formal methods) can be considered as formal methods, according to their degree of conformity to a rigorous mathematical basis of their uses.

1.3.3.1.3. Structured method

The aim of *structured methods* (EN 50128, B.60) is to promote quality of software development by looking at the first phase of the lifecycle.

This type of method calls for some precise and intuitive notations (generally computer-aided) so as to produce and document the requirements and characteristics of installation of the product.

Structured methods are tools for analysis. They are based on an order of logical reflection, an analysis of the system taking into account environment, production, and documentation of the system, breakdown of data and of the functions and the elements to be verified; they impose a weak intellectual service (simple, intuitive and pragmatic method). Among the structured methods are SADT [LIS 90], JSD, CORE, Yourdon real-time, and MASCOT.

Structured specification is a technique, which aims to give prominence to the simplest visible relationships between the partial requirements and the functional specification.

Analysis is refined by stages. A hierarchical structure of partial requirements is then obtained, which will allow the total requirements to be specified. The method highlights interfaces between requirements and enables interface failures to be avoided.

28 These are the terms of the CEI/IEC 61508 standard.

1.3.3.2. Computer-aided specification tools

In section B.2.4 of the outline of the CEI/IEC 61508 standard, it is indicated that implementation of computer-aided specification tools allows a specification to be obtained in the form of a database, which can be automatically inspected to evaluate coherence and completion. The tool can enable animation.

The application can be made at any of the different phases: specification, architecture, and/or design. For the design phase, use is recommended once there are tools, but it will be necessary to show correction of the tool (experience feedback and/or independent verification of the results).

1.4. Formal methods

1.4.1. Principles

The aim of formal methods is to eliminate ambiguities, misunderstandings and bad interpretations, which may occur in natural language descriptions.

The term *formal method* groups together:

- languages;
- with a formally-defined vocabulary and syntax,
- the semantic of which is mathematically defined;
- transformation and verification techniques based on mathematical proof.

The formal specification always gives a description of what the software must do and not how it must do it.

1.4.2. Examples of formal methods

1.4.2.1. Introduction

It should be noted that in the context of safety-critical applications, at least two formal methods have a currently-used and recognized environmental design, which covers one part of the specification production process according to the code, while integrating one or more verification processes; namely the B-method [ABR 96] and the LUSTRE language [HAL 91, ARA 97] and its graphic version, named SCADE [DOR 08].

The B-method [ABR 96] and the SCADE environment [DOR 08] are associated with industrial tools.

The appeal of formal methods resides in the fact that some proof of property-type verifications (through proof activities, exhaustive simulation [BAI 08], etc.) can be implemented as early as possible in the development cycle. The underlying idea is to have software, which is reliable by construction.

1.4.2.2. From Z to the B-method

In the context of the SACEM²⁹ [GEO 90], the RATP performed Hoare's proof [HOA 69] to show acknowledgement of the requirements – for more information, consult [GUI 90]. Hoare's proof makes it possible, through a P program and a set of C pre-conditions, to bring the set of post-conditions to light.

Hoare's proof, run in the framework of the SACEM, enabled some of the properties of the code to be exposed, but it was not possible to link it to the safety-related requirements (the requirement for non-collision, for example).

Faced with this situation, it was decided to make a formal model with Z [SPI 89]. This formal model has enabled the properties to be broken down and a link to be made between the requirements and the code. A score of important anomalies was then detected by the team of experts responsible for Z re-specification.

As a result of the difficulties encountered during SACEM's execution (manual processing, introduction of error, complexity of obtained properties, difficulty of carrying out proof, traceability problems, etc.), the industrial engineer in charge of SAET-METEOR [MAT 98, LEC 96] development was asked to use a formal process, which integrated formal proof and which could be equipped. It was in this context that the B-method was chosen.

This is why, in the French rail system, use of formal methods and notably of the B-method, is increasingly common in the context of development of safety-critical systems. The software of these safety systems (rail signaling and automatic driving) must respond to some very strict criteria of quality, reliability, and robustness.

The B-method [ABR 96] was developed by Jean-Raymond Abrial. It is a model-oriented formal method like Z and VDM, but enables an incremental development of the specification right down to the code, through the notion of refinement [MOR 90], and that too in a single formalism, the language of abstract machines. At each stage of B development, some proof obligations are generated so as to guarantee the validity of the refinement and the consistency of the abstract machine. The B-method is thus based on the notion of proof.

²⁹ *Système d'aide à la conduite, à l'exploitation et à la maintenance* (driving, operation, and maintenance assistance system).

Some more recent projects, such as CTDC, KVS, SAET-METEOR [BEH 93, BEH 96, BOU 06], CDGVAL, and line 1 of the Parisian metro, use the B-method all through the process (see Figure 1.23) of development (from the specifications right down to the code).

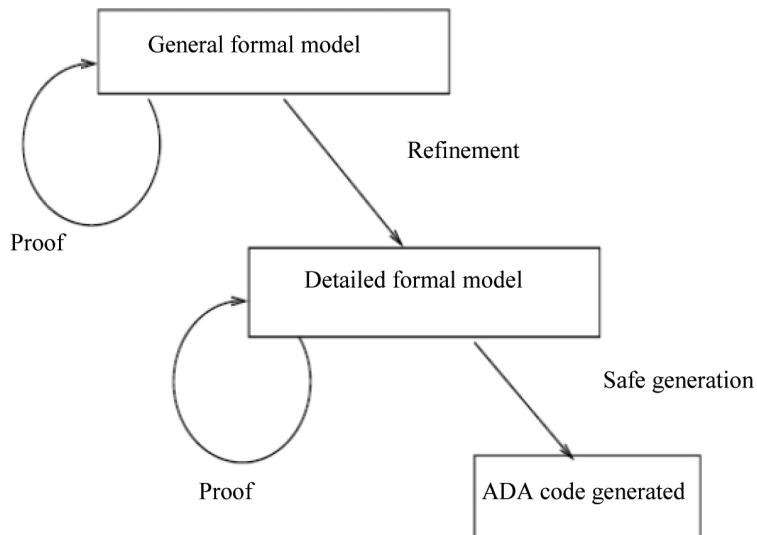


Figure 1.23. *B development cycle*

It should be noted that the B-method uses an industrial environment. Atelier B and the B-toolkit (marketed respectively by CLEARSY and B-Core) are tools which completely cover the development cycle proposed by the B-method (specification, refinement, code generation and proof).

In the context of the example of Figure 1.24, we introduce the specification of the Hello program and its implementation. This example shows that a B model is an assembly of components (known as an *abstract machine*) and that there is a separation between the “what” (the MACHINE) and the “how” (the IMPLEMENTATION).

Figure 1.25 is an example of formalization of the P1 property (introduced in section 1.2.3.2, “there must not be any risk of collision”), which was formalized in set-theoretic logic of the first order in $\forall t1, t2 \in T, \text{ hence } Dt \cap Dt = \emptyset, \text{ if } t1 \neq t2. Dt$ is the domain of train movement limit i and $[T]$ is the whole set of trains.

Chapter 3 of [BOU 11b] presents use of the B-method in the context of the SAET-METEOR.

<pre> MACHINE HelloWorld OPERATIONS Hello = skip END </pre>	<pre> IMPLEMENTATION HelloWorld_n REFINES HelloWorld IMPORTS BASIC_IO OPERATIONS Hello = STRING_WRITE("Hello World") END </pre>
---	---

Figure 1.24. The “Hello” program in B

<pre> MACHINE Example SETS TRAIN ; POSITION VARIABLE Industry INVARIANT Industry : TRAIN -> POW(POSITION) & ! (t1,t2). ((t1,t2) : TRAIN * TRAIN => ((Industry(t1) ((Industry(t2)= {}) & (t1!=t2))) INITIALIZATION Industry : (Industry : TRAIN -> POW(POSITION) & !(t1,t2). ((t1,t2) : TRAIN * TRAIN => ((Industry(t1) ((Industry(t2)= {}) & (t1!=t2)))) END </pre>
--

Figure 1.25. Second example of B

1.4.2.3. From LUSTRE to SCADE

After, acknowledging that classic computing languages were inadequate for producing automatic and real-time applications. The LUSTRE language was developed in the VERIMAG³⁰ laboratory in the 1980s [HAL 91, BEN 03, HAL 05].

From LUSTRE, we moved on to SCADE³¹ through different phases and connections (SAGA with Merlin Gérin, SAO³² with AIRBUS). SAO had two strengths:

³⁰ To find out more, visit: www.verimag.fr.

³¹ SCADE is distributed by Esterel Technologies; see: www.estrel-technologies.com.

³² SAO (*spécification assistée par ordinateur*) (computer-aided specification) is an “in-house” formalism developed by AIRBUS, which reprises the concept of analog block diagrams, that provide a specification tool with code generation, then of code generation.

– code generation from automatic models allows the number of errors to be reduced;

– the graphic language relates closely to the knowledge of aeronautical engineers, facilitating communication within AIRBUS; there was then an acceleration in updating and capitalization on know-how. SAO is one of the factors behind the success of the A320.

[HAL 05] presents the history of LUSTRE and of SCADE in more detail along with the differences that can exist.

The SCADE environment implements an extension of the (textual) LUSTRE language. This extension preserves the properties of the LUSTRE language and enables modeling based on the notions of functional block diagrams and data-streams.

LUSTRE is part of the “formal methods” in the sense that it is based on a precise syntax and semantics. The LUSTRE language is a textual language.

The SCADE environment enables an application to be modeled in graphic form through components (boxes, state/transition graphs, etc.), which are grouped together in “rectangles” (see Figure 1.26).

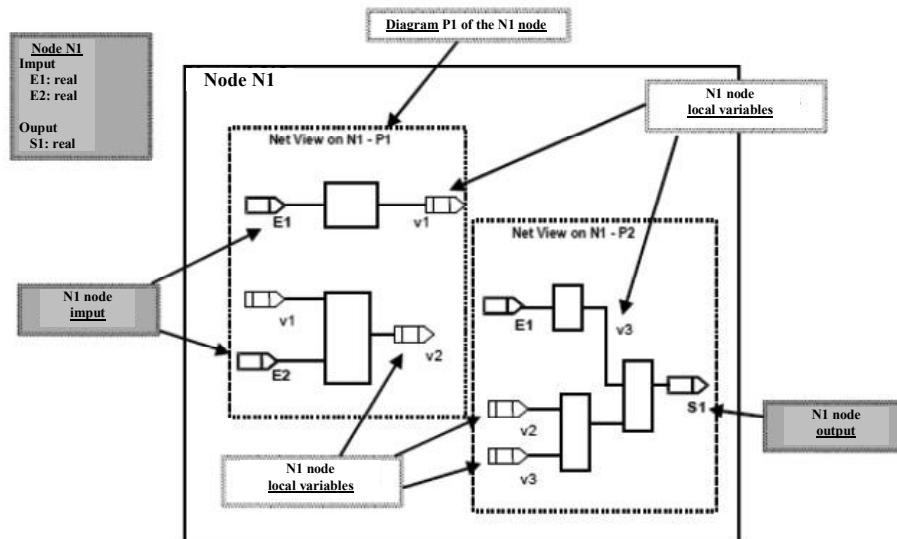


Figure 1.26. Example of a “rectangle”

SCADE (*Safety critical application development environment*) is a language and development environment used in sectors such as the avionic (AIRBUS, Eurocopter), nuclear (Schneider Electrics, CEA), rail (Ansaldo STS, AREVA, THALES, RATP) and since only recently, the automobile. Chapter 6 will present examples in more detail. SCADE is meant to be compatible with the recommendations of different industry standards (DO-178, CEI/IEC 61508, EN 50128, and CEI/IEC 60880).

Figure 1.27 puts the tools of the SCADE environment back in the V-cycle. If the SCADE environment is used in the specification or design phases, an initial document titled “specification”, is always used along with it. This document can be a textual specification or a formal specification (SART); it may be system level or software level, etc.

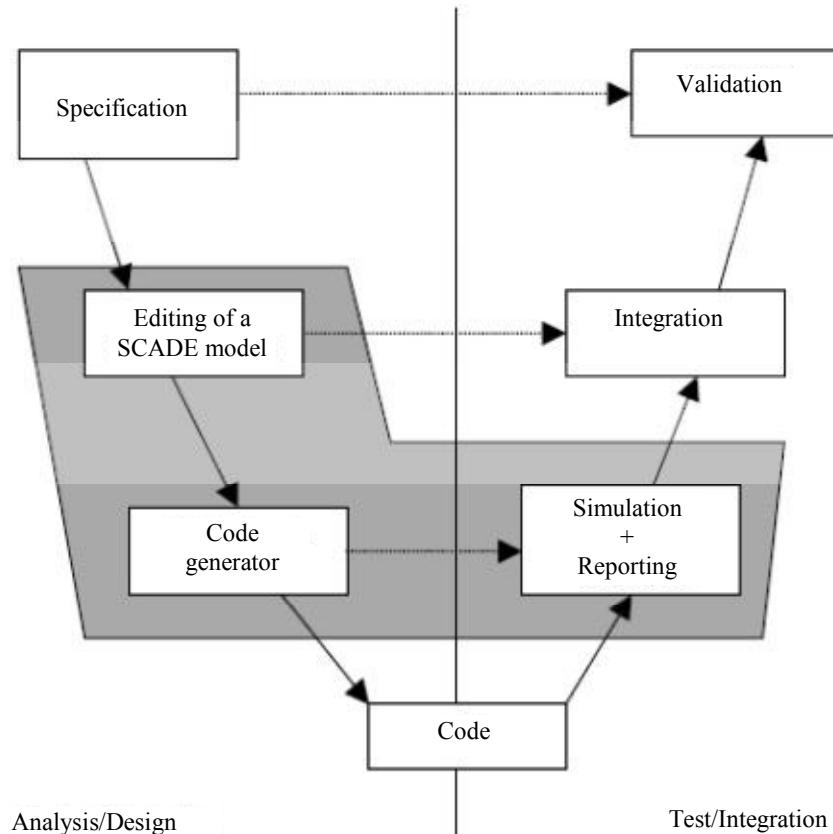


Figure 1.27. SCADE as a design environment

At the global level, it is interesting to establish a model that describes the functional architecture of the software. The system to model is thus broken down depending upon the needs that will correspond to the SCADE models. This breakdown enables:

- the global specification to be divided up into several SCADE models (one model per function of need);
- each SCADE model to develop independently;
- the corresponding document of interfaces to be produced.

Figure 1.28 details the models used for managing SCADE nodes as well as the relations that govern them. Chapter 2 is devoted to presentation of SCADE and some examples of industrial use are presented there.

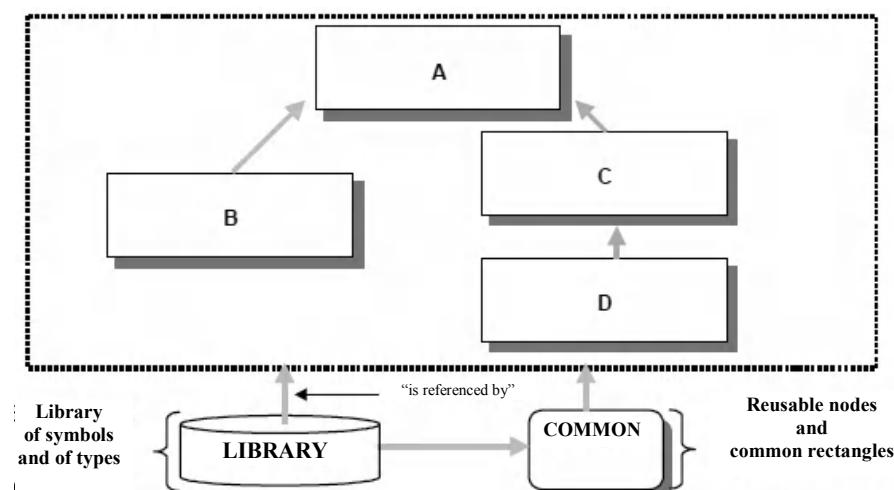


Figure 1.28. Architecture of SCADE models

1.5. Conclusion

This chapter presents the needs that appear when a software application with a high safety objective (SSIL3/SSIL4) is to be created. In the outlines of the automobile [ISO 09], rail [CEN 01] and generic [IEC 98] industry standards, formal methods are advocated (see Table 1.5) as a means of satisfying needs and achieving a controlled code production.

46 Formal Methods

ID	Description	Impact of formal methods	Method volume (V) chapter (C) ³³
N.1	N.1 reducing the number of faults introduced into the application during the bottom-up phase of the V-cycle;	1) establishment of a formal model on the whole of the cycle enables verifications to be carried out as early as possible and to show that there are no faults which have been introduced; 2) the weak point is the coding phase, implementation of a formal method makes it possible to use a semantic and thus to propose a process of automatic translation of the specification in code.	ASA+ (T2-C2) SADT (T2-C2) B-method (T2 – C2, T2- C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7)
N.2	Identifying faults introduced within the software application as early as possible;	Implementation of a formal model enables verifications to be carried out as possible.	ASA+ (T2-C2) SADT (T2-C2) B-method (T2-C2, T2-C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7)
N.3	Having to define a subassembly of the language (limiting difficulties) and of using a set of design and coding rules;	Implementation of a formal method makes it possible to use a semantic and thus to propose a process of automatic translation of the specification in code, enabling management of the sub-assembly and design to be facilitated.	B-method (T2-C2, T2-C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7)
N.4	Having to use sufficient experience feedback to show that the tools, and the compiler especially, are usable for the expected safety level;	This is a recurring problem with the new standards. Code generation enables part of the problem to be fixed, but it must be capable of qualifying the tools.	SPARK Ada (T3-C1)

33 We define Tx-Cy where Tx is the number of the book and Cy the chapter. T1 = [BOU 12], T2 = this book and T3 the third book to appear.

N.5	Having to create a model associated with the requirements;	This is the objective of formal methods.	ASA+ (T2-C2) SADT (T2-C2) B-method (T2-C2, T2-C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7) CB (T2-C8) MATELO (T3-C2)
N.6	Having to use a method of verifying the model so as to verify the coherence and/or completion of requirements;	Creation of a formal model enables verification of the coherence and completion of the requirements through techniques such as proof or model-checking.	SADT (T2-C2) B-method (T2-C2, T2-C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7) CB (T2-C8) MATELO (T3-C2) SPARK Ada (T3-C1)
N.7	Having to be able to continue processing (Figure 1.17) from the specification phase right up to design phase, not to mention the coding phase;	Formal methods such as SCADE or the B-method enable models to be made which cover the different bottom-up phases of the V-cycle.	B-method (T2-C2, T2-C3, T3-C6, T3-C7) SCADE (T2-C6, T2-C7) SPARK Ada (T3-C1)
N.8	Having to carry out abstract interpretation type formal verifications on the code;	Abstract interpretation and program proof can now be executed on the C, C++ or Ada code, with a reasonable temporal cost ³⁴ .	See [BOU 12].
N.9	Having to use a model which enables facilitation of test production (generation of test cases, model exploration, etc.).	A formal model can be made specifically so as to be used for the production of test cases (be it by exploration, or by pseudoautomatic generation).	ASA+ (T2-C2) MATELO (T3-C2)

Table 1.12. Impact of formal methods

³⁴ Before 1998, the RATP had carried out an analysis by abstract interpretation of the code of the SAET-METEOR application, and approximately a week of processing was needed to carry out one “run”. Currently, it is possible to carry out one “run” (on an actual size code) in a few hours.

Some formal methods such as the B-method³⁵ [ABR 96], SCADE [DOR 08], VDM [JON 90], Z [SPI 89], etc. enable us to acquire the customer need formalized as a model. From this model, it is possible to demonstrate some properties (formalization of requirement) and to generate some code automatically, this code is the image of the model.

If the tools implemented (simulator, prover, code generator, etc.) are shown to abide by safety level-linked objectives to be achieved by the application, this type of approach enables the activities of V&V (verification and validation) to be replaced by other activities (for example, module testing can be replaced by a proof activity and a qualification/certification of the code generator, see for example [BOU 06]).

The different industry standards (aeronautical [ARI 92], automobile [ISO 09], rail [CEN 01, CEN 11], nuclear [IEC 06] and generic [IEC 98]) necessitating a high safety level take into account the use of formal techniques when they are associated with models, connected with design documents (specification, architecture, design).

Currently, the coding phase is based on the use of Ada [ANS 83], C [ISO 99] and/or C++ type programming language. Originally, these languages were not associated with formal verification techniques. As we have stated, it can be quite difficult to carry out the activities of V&V (verification and validation) like review and tests on the final (Ada, C or C++ type) code. It may then be interesting to implement some formal techniques such as abstract interpretation [COU 00] and/or proof (such as Hoare's proof [HOA 69]), which will enable identification of properties that the code abides by and/or demonstration that the code verifies certain properties. For more information on this subject, see the first volume of this series, [BOU 11a]. This is why it is necessary to implement other practices, which must enable faults in the software application to be detected as early and as broadly as possible.

It should be noted that one of the difficulties in implementing these techniques resides in the absence of recognition of these techniques within the current standards. In effect, certain standards (see Table 1.9, extract from the CENELEC EN 50128 [CEN 01] standard for example) advocate implementation of formal methods but they do not mention the notion of abstract interpretation (or derived methods).

³⁵ The list is not meant to be complete; for more information, see: for example [MON 00].

1.6. Bibliography

- [ABR 96] ABRIAL J.R., *The B-Book – Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.
- [ADA 01] ADA RESOURCE ASSOCIATION, Operating Procedures for Ada Conformity Assessments, version 3.0, www.ada-auth.org/procs/3.0/ACAP30.pdf, April 2001.
- [ADA 05] ADACORE, *Safe and Secure Software – An Invitation to Ada 2005*, www.adacore.com/home/ada_answers/ada_2005/safe_secure, 2005.
- [ANS 83] ANSI, Norme ANSI/MIL-STD-1815A-1983, Langage de programmation ADA, 1983.
- [ARA 97] ARAGO, “Applications des méthodes formelles au logiciel”, *Observatoire français des techniques avancées* (OFTA), vol. 20, Masson, Paris, June 1997.
- [ARI 92] ARINC, *Software Considerations in Airborne Systems and Equipment Certification, DO 178B* and EUROCAE/no. ED12B, 1992.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, The MIT Press, Cambridge, MA, 2008.
- [BAR 03] BARNES J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, Boston, 2003.
- [BEH 93] BEHM P., “Application d’une méthode formelle aux logiciels sécuritaires ferroviaires”, *Atelier Logiciel Temps Réel, 6e Journées internationales du Génie Logiciel*, 1993.
- [BEH 96] BEHM P., “Formal development of safety critical software of METEOR”, *First B Conference*, Nantes, 24-26 November 1996.
- [BEN 03] BENVENISTE A., CASPI P., EDWARDS S.A., HALBWACHS N., LE GUERNIC P., DE SIMONE R., “The synchronous languages 12 years later”, *Proceedings of the IEEE*, vol. 91, no. 1, January 2003.
- [BOU 99] BOULANGER J.-L., DELEBARRE V., NATKIN S., “METEOR: validation de spécification par modèle formel”, *Revue RTS*, no. 63, p. 47-62, April-June 1999.
- [BOU 05] BOULANGER J.-L., BERKANI K., “UML et la certification d’application”, *ICSSEA CNAM*, Paris, 1-2 December 2005.
- [BOU 06] BOULANGER J.-L., Expression et validation des propriétés de sécurité logique et physique pour les systèmes informatiques critiques, thesis, University of Technology of Compiègne, 2006.
- [BOU 07a] BOULANGER J.-L., “UML et les applications critiques”, *Proceedings of Qualita ’07*, p. 739-745, Tanger, Maroc, 2007.
- [BOU 07b] BOULANGER J.-L., BON P., “BRAIL: d’UML à la méthode B pour modéliser un passage à niveau”, *Revue RTS*, no. 95, p. 147-172, April-June 2007.

50 Formal Methods

- [BOU 08] BOULANGER J.-L., “RT3-TUCS: How to build a certifiable and safety critical railway application”, *17th International Conference on Software Engineering and Data Engineering, SEDE-2008*, p. 182-187, Los Angeles, 30 June-2 July 2008.
- [BOU 09a] BOULANGER J.-L., IDANI A., PHILIPPE L., “Linking paradigms in safety critical systems”, *Review ICSA*, September 2009.
- [BOU 09b] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques – exemples concrets*, Hermès Lavoisier, Paris, 2009.
- [BOU 11a] BOULANGER J.-L. (ed.), *Utilisation industrielles des techniques formelles – interprétation abstraite*, Hermès Lavoisier, Paris, 2011.
- [BOU 11b] BOULANGER J.-L. (ed.), *Techniques industrielles de modélisation formelle pour le transport*, Hermès Lavoisier, Paris, 2011.
- [BOU 11c] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques industrielles*, Hermès Lavoisier, Paris, 2011.
- [BOU 12] BOULANGER J.-L. (ed.), *Mise en oeuvre de la méthode B*, Hermès Lavoisier, Paris, forthcoming.
- [CEN 00] CENELEC – NF EN 50126, Applications Ferroviaires, Spécification et démonstration de la fiabilité, de la disponibilité, de la maintenabilité et de la sécurité (FMDS), January 2000.
- [CEN 01] CENELEC – EN 50128, Railway Applications, Communications, Signalling and Processing Systems, Software for Railway Control and Protection Systems, May 2001.
- [CEN 03] CENELEC – NF EN 50129, Norme européenne, Applications ferroviaires: systèmes de signalisation, de télécommunications et de traitement systèmes électroniques de sécurité pour la signalisation, 2003.
- [CEN 11] CENELEC – EN 50128, Railway Applications, Communications, Signalling and Processing Systems, Software for Railway Control and Protection Systems, January 2011.
- [COU 00] COUSOT P., “Interprétation abstraite”, *TSI*, vol. 19, no. 1-3, www.di.ens.fr/~cousov/COUSOTpapers/TSI00.shtml, 2000.
- [DOR 08] DORMOY F.-X., “Scade 6, a model based solution for safety critical software development”, *Embedded Real-Time Systems Conference*, 2008.
- [GEO 90] GEORGES J.P., “Principes et fonctionnement du système d'aide à la conduite, à l'exploitation et à la maintenance (SACEM), Application à la ligne A du RER”, *Revue générale des Chemins de fer*, no. 6, June 1990.
- [GUI 90] GUIHOT G., HENNEBERT C., “Sacem software validation”, *ICSE*, p.186-191, 26-30 March 1990.
- [HAD 06] HADDAD S., KORDON F., PETRUCCI L. (ed.), *Méthodes formelles pour les systèmes répartis et coopératifs*, Hermès Lavoisier, Paris, 2006.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous dataflow programming language Lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p.1305-1320, September 1991.

- [HAL 05] HALBWACHS N., “A synchronous language at work: the story of lustre, MEMOCODE ’05”, *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, 2005.
- [HAT 94] HATTON L., SAFER C., *Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1994.
- [HOA 69] HOARE C.A.R, “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, p. 576-580-583, 1969.
- [HUL 05] HULL E., JACKSON K., DICK J., *Requirements Engineering*, Springer, New York, 2005.
- [IDA 06] IDANI A., B/UML: mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B, PhD thesis, University Joseph Fourier, Grenoble, November 2006.
- [IDA 07a] IDANI A., BOULANGER J.-L. PHILIPPE L., “A generic process and its tool support towards combining UML and B for safety critical systems”, *CAINE 2007*, San Francisco, 7-9 November 2007.
- [IDA 07b] IDANI A., OKASA OSSAMI D-D., BOULANGER J.-L., “Commandments of UML for safety”, *2nd International Conference on Software Engineering Advances IEEE CS Press*, August 2007.
- [IDA 09] IDANI A., BOULANGER J.-L., PHILIPPE L., “Linking paradigms in safety critical systems”, *revue ICSA*, September 2009.
- [IEC 03] IEC 61131, Programmable controllers. International standard, May 2003.
- [IEC 05] IEC 61511, Sécurité fonctionnelle – Système instrumenté de sécurité pour le secteur des industries de transformation, Norme européenne, March 2005.
- [IEC 06] IEC 60880, Centrales nucléaires de puissance – instrumentation et contrôles commande importants pour la sécurité, Aspects logiciels des systèmes programmés réalisant des fonctions de catégories A. Norme internationale, 2006.
- [IEC 98] IEC 61508, Sécurité fonctionnelle des systèmes électriques électroniques programmables relatifs à la sécurité. Norme internationale, 1998.
- [ISA 05] ISA, Guide d’interprétation et d’application de la norme IEC 61508 et des normes dérivées IEC 61511 (ISA S84.01) et IEC 62061, April 2005.
- [ISO 00] ISO/IEC 15942, Technologies de l’information – Langages de programmation – Guide pour l’emploi du langage de programmation Ada dans les systèmes de haute intégrité, 2000.
- [ISO 03] ISO/IEC 14882:2003(E), Programming Languages – C++, American National Standards Institute, New York 10036, 2003.
- [ISO 08] ISO 9000:2008, Systèmes de management de la qualité – Principes essentiels et vocabulaire, 2008.

52 Formal Methods

- [ISO 09] ISO/CD-26262, Road vehicles – Functional safety – non publié, 2009.
- [ISO 95] ISO/IEC 8652:1995, Information technology, Programming languages – Ada, 1995.
- [ISO 99] ISO C standard 1999, Technical report, www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, 1999.
- [ISO 99a] ISO/IEC 18009:1999, Information Technology – Programming Languages – Ada: Conformity Assessment of a Language Processor, 1999.
- [JON 90] JONES C.B., *Systematic Software Development Using VDM*, Prentice Hall, Upper Saddle River, 1990 (2nd edition).
- [KER 88] KERNIGHAN B.W., RITCHIE D.M., *The C programming Language*, Prentice Hall, Upper Saddle River, 1988 (2nd edition).
- [LEC 96] LECOMPTÉ P., BEAURENT P.-J., “Le système d’automatisation de l’exploitation des trains (SAET) de METEOR”, *Revue Générale des Chemins de fer*, vol. 6, p. 31-34, June 1996.
- [LED 01] LEDANG H., “Des cas d’utilisation à une spécification B”, *AFADL 2001: Approches formelles dans l’assistance au développement de logiciels*, 2001.
- [LER 09] LEROY X., “Formal verification of a realistic compiler”, *Communication of ACM*, vol. 52, no. 7, p.107-115, July 2009.
- [LIS 90] LISSANDRE M., *Maîtriser SADT*, Armand Collin, Paris, 1990.
- [LOC 05] LOCKHEED M., Joint strike fighter air vehicle C++ Coding standards for the system development and demonstration program, document no. 2RDU00001, rev.C, December 2005.
- [MAM 01] MAMMAR A., LALEAU R., “An automatic generation of B-specification from well defined uml notations for database applications”, *CNAM*, Paris, 2001.
- [MAR 01] MARCANO R., LEVY N., “Transformation d’annotations OCL en expressions B”, *Journées AFADL 2001, Approches formelles dans l’assistance au développement de logiciels*, 2001.
- [MAR 04] MARCANO R., COLIN S., MARIANO G., “A formal framework for uml modelling with timed constraint: application to railway control system”, *SVERTS ’04, Specification, Validation of UML Models for Real Time and Embedded Systems*, 2004.
- [MAT 98] MATRA/RATP, “Naissance d’un Métro. Sur la nouvelle ligne 14, les rames METEOR entrent en scène. PARIS découvre son premier métro automatique”, *La vie du Rail et des transports*, no. 1076, hors-série, October 1998.
- [MEY 98] MEYERS S., *Effective C++: 50 Specific Ways to Improve Your Programs and Design*, Addison-Wesley, 1998 (2nd edition).
- [MIS 98] MISRA, “MISRA-C: 1998 guidelines for the use of the C language in vehicle based software”, *Motor Industry Research Association*, April 1998.

- [MIS 04] MISRA, “MISRA-C: 2004 guidelines for the use of the C language in critical systems”, *Motor Industry Research Association*, October 2004.
- [MIS 08] MISRA, “MISRA-C++: 2008 guidelines for the use of the C++ language in critical systems”, *Motor Industry Research Association*, June 2008.
- [MON 00] MONIN J.-F., *Introduction aux méthodes formelles*, Hermès, Paris, 2000.
- [MOR 90] MORGAN C., *Deriving Programs from Specifications*, Prentice Hall International, 1990.
- [MOT 05] MOTET G., “Vérification de cohérence des modèles UML 2.0”, *Première journée thématique Modélisation de Systèmes avec UML, SysML et B-Système*, Association française d’ingénierie système, Toulouse, 2005.
- [OKA 07] OKALAS O., MOTA J.-M., THIRY L., PERRONNE J.-M., BOULANGER J.-L., “A method to model guidelines for developing railway safety-critical systems with UML”, *ICSOFT ’07 - International Conference on Software and Data Technologies*, Barcelona, 2007.
- [OMG 06a] OMG, Unified Modeling Language: Superstructure, v. 2.1, OMG document ptc/06-01-02, January 2006.
- [OMG 06b] OMG, Unified Modeling Language: Infrastructure, v. 2.0, OMG document formal/05-07-05, March 2006.
- [OMG 07] OMG, Unified Modeling Language (uml), v. 2.1.1, 2007.
- [OOT 04a] OOTIA, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, vol. 1: Handbook Overview, Revision 0, 26 October 2004.
- [OOT 04b] OOTIA, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, vol. 2: Considerations and Issues, Revision 0, 26 October 2004.
- [OOT 04c] OOTIA, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, vol. 3: Best Practices, Revision 0, 26 October 2004.
- [OOT 04d] OOTIA, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, vol. 4: Certification Practices, Revision 0, 26 October 2004.
- [RAM 09] RAMACHANDRAN M., ATEM DE CARVALHO A. (ed.), *Software Engineering and Productivity Technologies: Implications of Globalization*, Handbook of Research, August 2009.
- [RAM 11] RAMACHANDRAN M. (ed.), *Knowledge Engineering for Software Development Life Cycles: Support Technologies and Applications*, Leeds Metropolitan University, Leeds, April 2011.
- [RAS 08] RASSE A., BOULANGER J.-L., MARIANO G., THIRY L., PERRONNE J.-M., “Approche orientée modèles pour une conception UML certifiée des systèmes logiciels critiques”, CIFA, *conférence internationale francophone d’Automatique*, Bucarest, November 2008.

54 Formal Methods

- [RIC 94] RICHARD-FOY M., LEGOFF G., “On-board with safety critical software: implementing safety critical software for high-speed railway transportation”, *Alsys World Dialogue*, vol. 8, no. 2, 1994.
- [SOM 07] SOMMERVILLE I., *Software Engineering 8*, Addison Wesley, Boston, 2007.
- [SPI 89] SPIVEY J.M., *The Z Notation: a Reference Manual*, Prentice Hall International, 1989.
- [STA 01] THE STANDISH GROUP, Extreme chaos, Technical Report, 2001.
- [STA 94] THE STANDISH GROUP, The chaos report, Technical Report, 1994.
- [SUT 05] SUTTER H., ALEXANDRESCU A., *C++ Coding Standards, 101 Rules, Guideline, and Best Practices*, Addison Wesley, Boston, 2005.

Chapter 2

Formal Method in the Railway Sector the First Complex Application: SAET-METEOR

2.1. Introduction¹

The railway sector has always been a strong force in innovation. This is all the more true in the field of systems based on software. RATP² understood early on the difficulties inherent in software development and integration in demonstrating system safety.

This second chapter shows how formal methods are implemented to develop software with “zero defect”.

This chapter is separated into four sections: the purpose of the first section (section 2.2) is to provide a quick description of the SAET-METEOR line principles [MAT 98, LEC 96]. Section 2.3 describes the development process followed by Siemens³ transportation systems (Matra Transport International, at the time).

The validation process that was set up within RATP is described in section 2.4. The last section sets out an assessment of the work completed when commissioning the SAET-METEOR.

Chapter written by Jean-Louis BOULANGER.

1 This chapter is very largely inspired by my thesis [BOU 06] and by the articles [DEL 99a, GAL 99, BOU 00, BOU 02, BOU 03, BOU 07].

2 For more information on the RATP, see the Website www.ratp.fr.

3 To learn more, see www.siemens.com.

2.2. About SAET-METEOR

At the beginning of the 1990s, RATP decided to implement a new system intended to reduce the use of the RER A⁴ and to open up certain districts (southern Paris and the 13th arrondissement).

This new system [MAT 98] that has equipped Line 14 of the Paris metro since October 1998 is called SAET-METEOR (French: *Système d'automatisation de l'exploitation des trains – METro Est Ouest Rapide*; English: Automation system of train operations).

Line 14 is a complex *real-time distributed* system whose principal function is to ensure passenger transport while guaranteeing a very high level of safety for passengers. It therefore should ensure that this system respects the operational constraints called *safety* [CHA 96]. In addition, the real-time characteristic indicates that the railway system interacts with its physical environment whose behavior is by nature *uninterruptible* and *irreversible*.

Concerning the technical aspects, the service currently runs with nineteen trains composed of six cars (to be increased to eight) with a commercial speed of 40 km/h and an interval between the trains of 85 seconds in integral automatic operation (French: CAI, *conduite automatique intégrale*).

A characteristic of this metro line is that it is *driverless*. This line will allow the circulation of trains with a driver (a conventional non-equipped train) and automatically equipped trains. The equipped trains can be used according to two modes of control, manual or automatic. In the case of an equipped train or non-equipped train in manual control, we speak of CM, and in the case of an equipped train in automatic mode, we talk about CAI. The set of train movements is managed by computer processors, the supervision remaining under human control through a control center.

The SAET-METEOR is not the first driverless metro, but it has two specialties: the safety functions are fulfilled by computers (contrary to the VAL system⁵ where they are carried out by electronics) and it allows mixing the trains. Indeed, it is possible to have trains without equipment (with drivers) circulate on the line managed by the SAET-METEOR.

⁴ Initially, the prevision was for 12,000 passengers and the current number is more than 25,000 passengers per hour.

⁵ The VAL (French: *véhicule automatique léger*, a light automatic transport system) is a light metro that is established in as many areas of France (for example: Lille, Toulouse and the ORLY-VAL) as overseas (for example: Chicago, Taipei and Turin). It should be noted that the Charles de Gaulle VAL is mixed since there is an electronic VAL and software processing.

2.2.1. Decomposition of the SAET-METEOR

We give a brief description of the operating system (SAET-METEOR) managing Line 14 of the Paris metro [MAT 98].

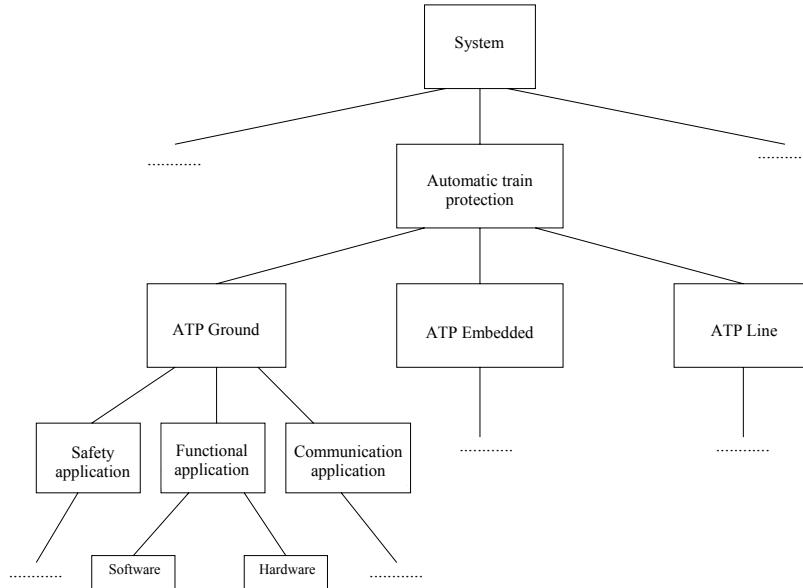


Figure 2.1. From the system to the software and hardware

SAET-METEOR:

- orders the routes on the line and in the terminal;
- ensures the regulation of train voyage;
- manages the station stops, the opening and closing of train doors and station platform doors;
- obeys signaling;
- controls the speed of trains while allowing the circulation of non-automated trains during passenger service;
- manages the electric traction power supply safely;
- manages various alarms related to the movements of the trains;
- allows phonic and visual supervision of the platforms and interior of the trains by an operator.

At this level of complexity, it is necessary to add reliability objectives such as a failure rate contrary to equipment safety from 10^{-7} to 10^{-9} per hour and which results in:

- all the occurrences of a material failure lead to the release of the emergency brake and, if necessary, to a cut in the traction energy;
- the software must have no fault with respect to safety remaining in the software.

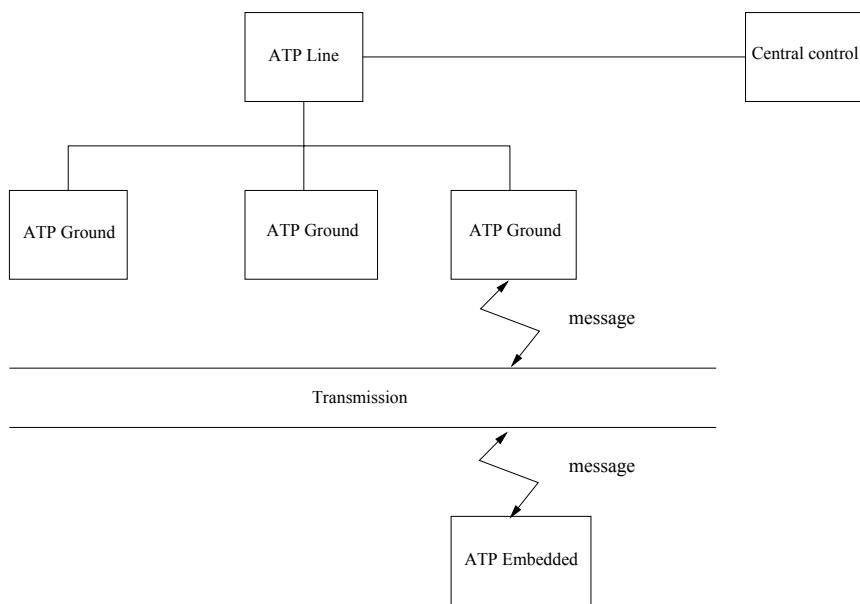


Figure 2.2. Decomposition of the SAET-METEOR automatic train protection

The line management is performed by the Automatic Train Protection subsystem (Figure 2.2), which is used throughout the line and in equipped trains. The control system has complete control over the automatic operation trains (acceleration, emergency stop, etc.). However, the manual operation trains are autonomous and can override a transmitted stop order through signaling. The line is managed by an autopilot line (ATP-Line) and is divided into automation sections [LEC 96].

Each section is managed by a computer, called *ground* Automatic Train Protection (ATP-Ground). The equipped trains have an embedded computer called *embedded autopilot* (ATP-Etched) which, among other things, regularly sends a localization message to the ground autopilot to deal with it. The processing of computers is made cyclically and is uninterrupted.

The Automatic Train Protection subsystem has complete control of the trains in automatic control (acceleration, emergency stop, etc.).

On the other hand, the trains in manual control are autonomous and can ignore a stop order transmitted through lateral signaling. For a train in manual control (an equipped or unequipped train) there are principles of safety and equipment associated with signaling that apply.

As indicated previously, the piloting subsystem is thus broken up throughout the set of equipment. The ATP-Ground is responsible for the prevention of a collision between trains and thus for transmitting to ATP-Embedded targets that must be respected. ATP-Embedded systems are also responsible for transmitting a precise location, as often and as regularly as possible.

We will present the basic principles of the SAET-METEOR and build on the concepts, which are introduced in section 2.7.

The treatment of the anticollision function is primarily based on the concept of fixed block sections [CON 96], common in the railway sector. A block section is a portion of the railway, which can be occupied only by one train at a given moment. The presence of the trains is detected by sensors called *track circuits*. The portion of the controlled railway may also be incorrectly called CdV.

The section of track thus controlled has a minimal length determined by technical and economic constraints, which, consequently, limit railway flow.

To mitigate this drawback, we use the concept of a *virtual section* and a *negative detector* (optical barriers referred to as ND for the rest of this chapter), which authorize a more precise train location. The negative detector allows a specific detection and is used to refine the state of occupation provided by the track circuits. The virtual sections (CV) are a subdivision of CdV.

A virtual section is a portion of the track where the presence of the trains is not solely determined by the state of the sensors (CdV, ND) but also by a distributed dynamic calculation, carried out by the control system according to the speed of the trains and the characteristics of the track.

The state of the virtual block sections is calculated at each cycle of the application's execution and makes it possible to obtain a cartography of the track through the real or potential occupation of the virtual section blocks by equipped or unequipped trains according to the state of CdV, ND, received messages of location, and the state of the virtual block section in the preceding cycle.

The line is divided into sections of automatism. Each section of automatism is controlled by ATP-Ground equipment. The example in Figure 2.3 defines a (very simple) section of automatism that is physically made up of three CdVs and comprises three NDs represented by the double vertical arrows, the NDs nd1 and nd3, respectively, represent the input and the output of the section managed by this ATP-Ground.

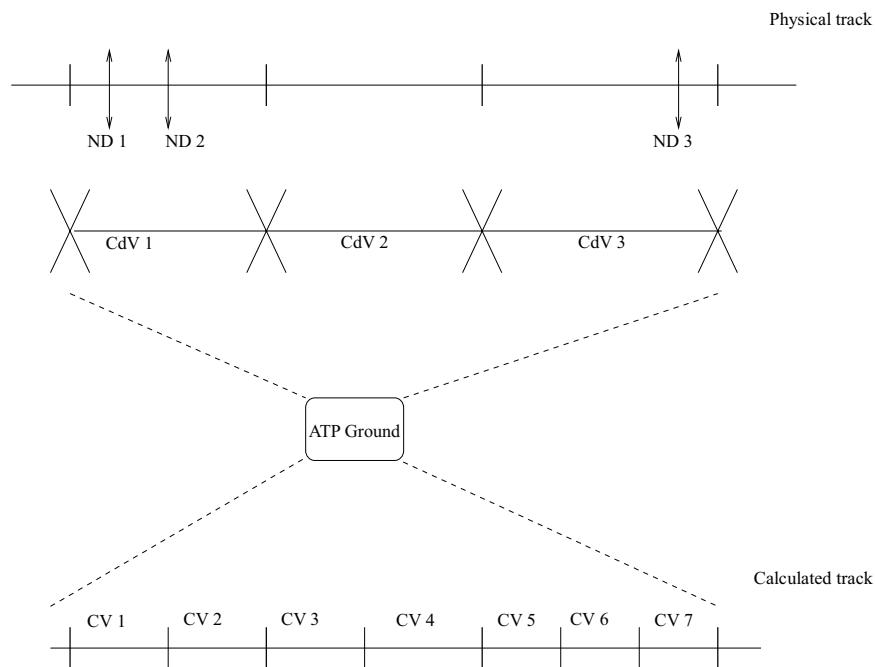


Figure 2.3. Description of a track

It becomes complex in the links between objects of the track, the NDs are always connected to two CVs (at the end of the track, we add a CV at the most restrictive state) but supported by a CdV; the CVs are supported by at least a CdV and the CdVs can support several NDs and CVs. Note that a section of track is made up of two tracks and that the section can comprise one or more railroad switches, making it possible to pass from one track to the other.

The complexity of the problem is introduced by the number of objects handled (a real section of automatism can consist of forty CVs, twenty NDs and of twenty CdVs), by the complexity of the links between objects of the section, and the number of states of each object (one CV has five states, the ND and the CdV have two states).

To allow a continuous assessment of the trains, the sections of automatism are recovered. The virtual block sections represent the interpretation made by the ATP-Ground of the occupation of the track by the trains.

2.2.2. Anticollision functions

The autopilot subsystem's principal function is to provide operation orders for all train types. For the trains in manual control, the orders are transmitted through signaling. For those in automatic control, orders are transmitted as messages containing the target to be reached. A target is a physical point on the line that should not be crossed by the train. Generating targets is a complex safety calculation that is carried out by the anticollision function, which is responsible for avoiding collisions between the trains in both automatic and manual control.

For each section of automatism, the ground autopilot cyclically provides the target to the equipped trains present in the section. These targets are elaborate starting from the state of CVs, the space constraints related to topology and the authorizations of the operation in progress. This safety action is carried out by the anticollision function.

Figure 2.4 presents a functional division of the *anticollision* function to be analyzed. The decomposition takes the functional division introduced in the functional specification of equipment and preserved in the technical specification of software needs.

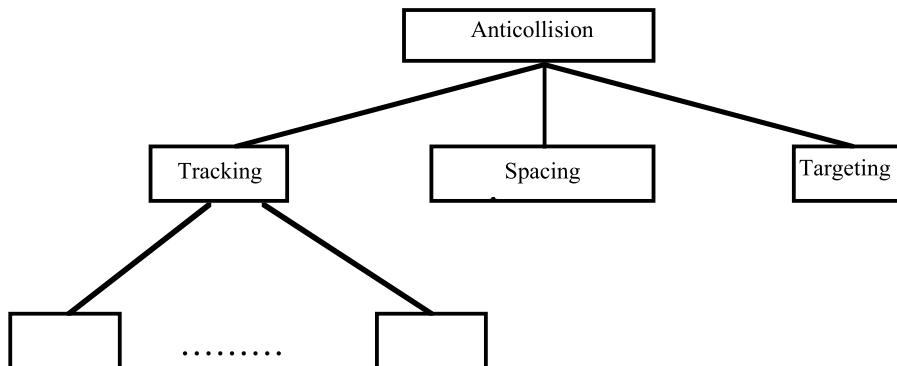


Figure 2.4. Dividing the anticollision function

The function of *tracking* elaborates the states of occupation of CVs. As the movement of the trains in manual control cannot be predicted, an overlap is

developed by the sub-function *spacing* around them to protect the trains in automatic control from possible operation errors. A buffer zone is oriented and delimits a zone that no train can enter in the direction indicated. The calculation of the targets for the equipped trains in automatic control that are dealt with is carried out by the function *target* according to the occupation of the section, overlaps in spacing, and the orders of operation received.

CVs have a complex structure that makes it possible to record a certain amount of information relating to the trains detected. It is necessary to have a state distinguishing the free state from the occupied state by an equipped train or the state occupied by a non-equipped train.

In the case of an equipped train, it is necessary for us to store the number of the train, the operation mode, and the direction it runs. Calculating spacing will generate the update of an attribute buffer (none, growing, decreasing, or bi_sens). For just the virtual block (CV) the data structure is composed of seven fields that can take 2 or 3 values. A section of track consists of at least twenty virtual block sections. This very complex environment can in no case be exhaustively simulated.

The function *anticollision* is one to store in the sense that the functions *spacing* and *target* use the output of the calculation of the *tracking* and this calculation needs the data of the preceding cycle.

2.2.3. Restrictions

The work on specification, design, verification and validation concerns the entirety of the system, but the work that we focus on is implemented for the aspects of critical embedded applications (software + hardware architecture). In the following section of the chapter, we focus on the *software* aspect.

2.3. The supplier realization process

2.3.1. Historical context

Within the context of SACEM⁶ [GEO 90, CHA 90], RATP carried out a Hoare proof [HOA 69] to show that the requirements were taken into account; for more information, consult [GUI 90].

⁶ The SACEM, which has been onboard the RER A since the end of the 1980s, is the first implementation of a software and a processor on a train.

The Hoare proof makes it possible to highlight the set of postconditions O starting from a program P and a set of preconditions I .

This is called a “Hoare triple”, which is noted as:

$$I \{P\} O$$

As shown in Figure 2.5, the Hoare proof, which was carried out within the SACEM, made it possible to show a certain number of properties of the code, but it was unable to link the requirements related to safety (a non-collision requirement, for example).

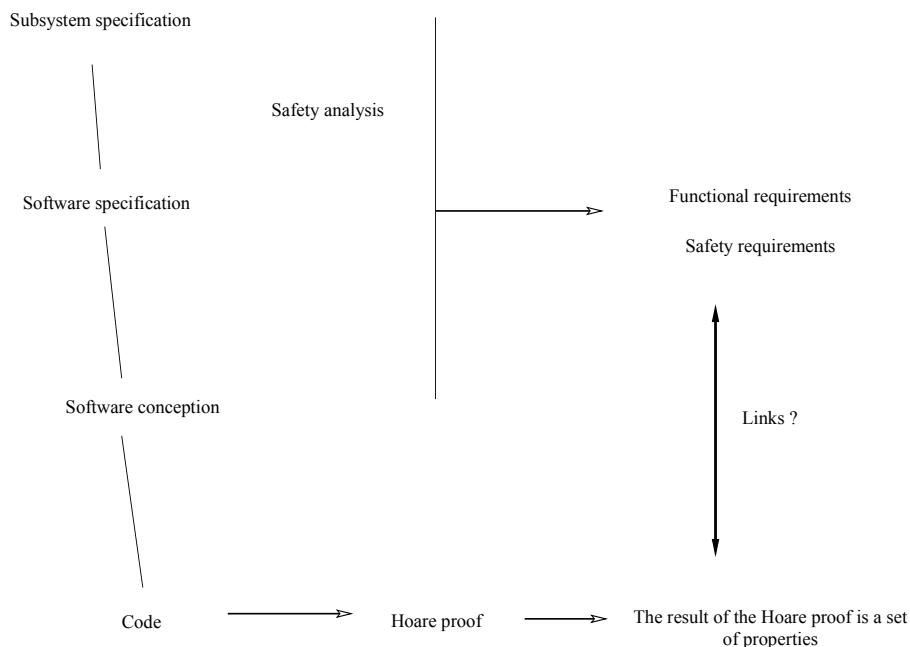


Figure 2.5. Implementation of the Hoare proof

Confronted with this situation, we decided to carry out a formal model in **Z** [SPI 89]. This formal model made it possible to break up the properties and establish a link between the requirements and the code. A score of important anomalies were then detected by the team of experts in charge of the re-specification in **Z**.

Following the difficulties that were encountered at the time of creating the SACEM (manual treatment, introduction of an error, complexity of the properties

obtained, difficulty in carrying out the proofs, traceability problems, etc.), the supplier in charge of developing the SAET-METEOR requested to use a formal process that integrates the equipped formal proof. It is in this context that *B method* [ABR 96] was selected.

2.3.2. The hardware aspect

At the beginning of the 1980s, the installation of a driver assist system for the RER A meant that microprocessors were introduced into trains. This assisted driving, control and maintenance system was given the acronym SACEM [GEO 90, CHA 90, MAR 90, HEN 94] from the French “*système d'aide à la conduite, à l'exploitation et la maintenance*”.

At the beginning of the 1980s, safety rested on redundant systems (2 out of 2 or 2 out of 3). The duration of use (40 to 50 years) is a large constraint, which led to choosing another type of safety implementation. Indeed, making a redundant structure safe (against a common mode's failures) means installing controls that are dependent on the components implemented. Given the duration of rail systems (40 to 50 years), it is preferable to have independent safety principles of hardware components.

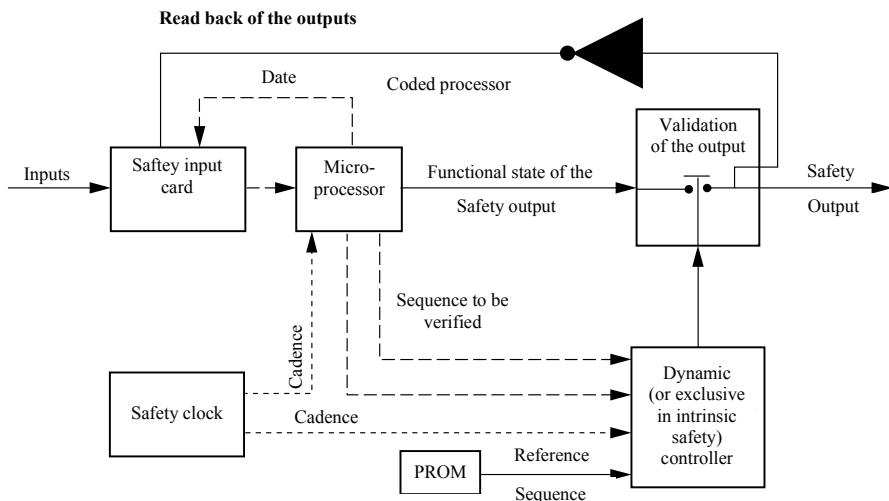


Figure 2.6. Hardware structure of the PSC

With hardware redundancy, it is thus preferable to install information coding [FOR 89] making it possible to ensure safety. Therefore, only one processor is used.

“Coded Mono-processor” technique⁷ (noted PSC) is based on a probabilistic approach developed in the 1980s. It should be noted that at the time of the first stages of creating the SACEM, two models had been produced and compared: a prototype based on a double redundancy and a mono-processor structure based (see Figure 2.6) on coding (arithmetic). As all the internal data is coded, manipulating the data can be carried out only through elementary operations called OPEL, which preserve the characteristics of coding.

Arithmetic coding is implemented through three coding techniques:

- detecting the errors related to the manipulation of data (memory failure, failure in communications, failures during the manipulation of data, etc.) with an arithmetic code;
- detecting the errors related to the execution of the program (a poor instruction, calculation, connection, etc.) through a signature;
- dating the data to prevent the use of out-of-date data.

The effectiveness of the PSC was demonstrated when creating several automatic and semi-automatic railway systems (SACEM [FOR 89], MAGGALY [MAI 93, STU 94], etc.). As [FOR 96] indicates, the SAET-METEOR’s equipment is structured around the PSC. Since the PSC’s performance suffered, the SAET-METEOR brought about the opportunity to add a *co-processor*.

Within the context of PSC structure, two types of safety are implemented⁸:

- *probabilistic safety*: for a system, objectives of safety, expressed in the number of victims (or the number of accidents) per year, are allotted and shown by probability calculations that the system respects its objectives;
- *intrinsic safety*: a system is deemed safe if there is the assurance that the failure of one or more components can make it run only in a less permissible situation (with respect to safety) than the situation in which it is found at the time of the failure, so that it results in a safe configuration.

The acquisition chart of safety input, the safety clock, the dynamic controller, and the safety output manager depend on intrinsic safety. The processing creation (processing unit) depends on probabilistic safety.

⁷ To learn more about the coded mono-processor and its implementation in the SACEM and the SAET-METEOR, consult Chapter 2 of [BOU 09].

⁸ The goal of this chapter is not to lecture about the concept of safety, but since the principal characteristic of the PSC is related to the duality between probabilistic safety/intrinsic safety, we could not avoid mentioning it. If interested, we propose reading the summary report [BIE 98], which effectively clarifies these two concepts.

Figure 2.7 presents the generation mechanism for the target with the predetermination phase of the signatures. The offline phase is predominant here for the safety of the application.

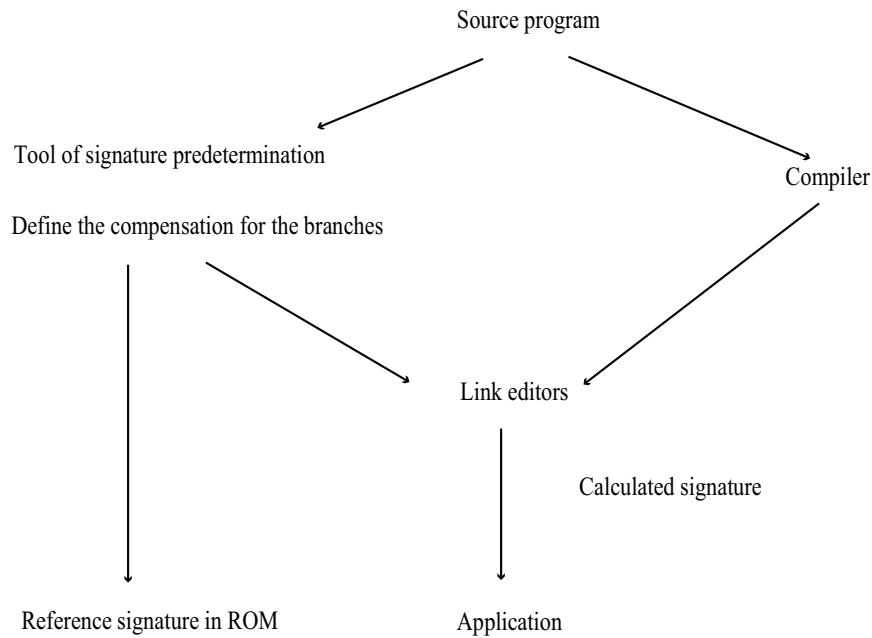


Figure 2.7. Example of signature determination

Starting from the source code of the application, the installation of coding makes it possible to detect the failures of the set of the tools (see Figure 2.7) of the chain of generation (compiler, assembler, etc.), processes of downloading (handling, programming the memories, etc.), and material elements (processor, memory, etc.).

The safety thus implemented is independent of the technology used but it remains related to the effectiveness of the coding⁹ (number of bits, Hamming distance, etc.).

The manipulation of code increases the processing time rather significantly. This is why, for the following systems, coprocessors and FPGA (*Field-Programmable Gate Array*) were set up to relieve the processor. Within the context of the new processors' effectiveness, the execution time should no longer be a problem.

⁹ Chapter 2 of [BOU 09] presents the various safety techniques of hardware structure and the guiding principles of coding.

The first use of the software was based on the language MODULA 2; for the following uses, the language of reference is Ada 83 [ANS 83] and its evolution Ada 95 [ISO 95].

2.3.3. The software aspect

2.3.3.1. B method

In the French railway industry, using formal methods, especially using the B method [ABR 96], is becoming more and more common in developing critical systems. The software of these safety systems (railway signals, automatic operation) must meet very strict criteria of quality, reliability and robustness.

One of the first applications of formal methods was made *a posteriori* in SACEM [GUI 90].

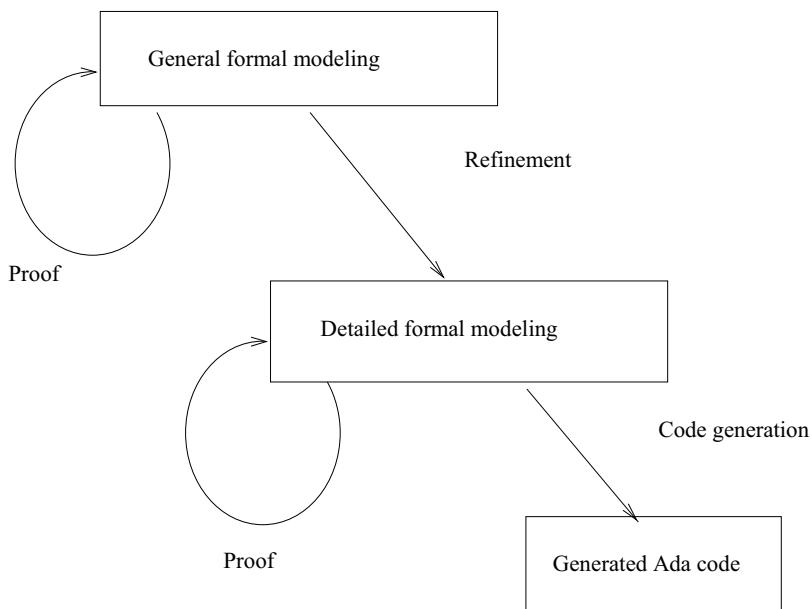


Figure 2.8. Cycle of the B development

More recent projects such as CTDC, KVS or the SAET-METEOR [BEH 93, BEH 96, BEH 97] use the B method throughout the development process (see Figure 2.8) (from specifications to code). Table 2.1 provides information about the complexity of B developments carried out in the railway industry.

The B method, developed by Jean-Raymond Abrial [ABR 96], is a targeted formal method model like Z [SPI 89] and VDM [JON 90], but it allows an incremental development from specification to code using refinement [MOR 90] in a single formalism, the language of abstract machines.

System Name	Line of B code	Line of generated code	Number of proof obligations
CDTC	5,000	3,000 (Ada)	700
KVB	60,000	22,000 (Ada)	10,000
KVB-SN	9,000	6,000 (Ada)	2,750
KVS	22,000	16,000 (Ada)	6,000
SACEM-simplifié	3,500	2,500 (Modula 2)	550
SAET-METEOR	115,000	90,000 (Ada)	27,800
CdG-VAL	PADS: 186440	30632 (Ada)	62,056
	UCA: 50085	11662 (Ada)	12,811

Table 2.1. Example of the complexity of a B model [BOU 06]

As shown in Figure 2.9, the process of refinement [MOR 90] is normally represented as a succession of independent stages with associated verifications.

A component $i+1$ (refinement or implementation) refines a component i (machine or refinement).

The external verification (consistency of the specification) and the internal verifications (validity of the refinements) are carried out through proof obligations.

The proof obligations form an integral part of the B method.

As we presented in [WAË 95], the use of refinement within the B method returns to carrying out models described as layered and shown in Figure 2.10. The layered model used two types of link: the decomposition and the refinement. The principal characteristic of this type of model is that it limits the impact of evolutions (repeating proofs) on the module being questioned and the modules from lesser layers.

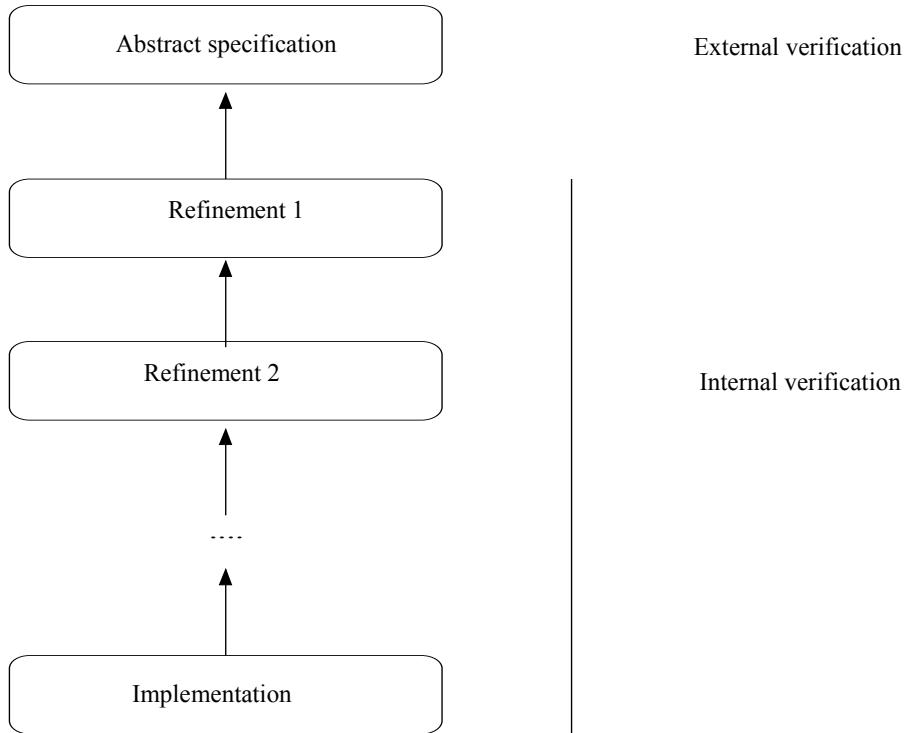
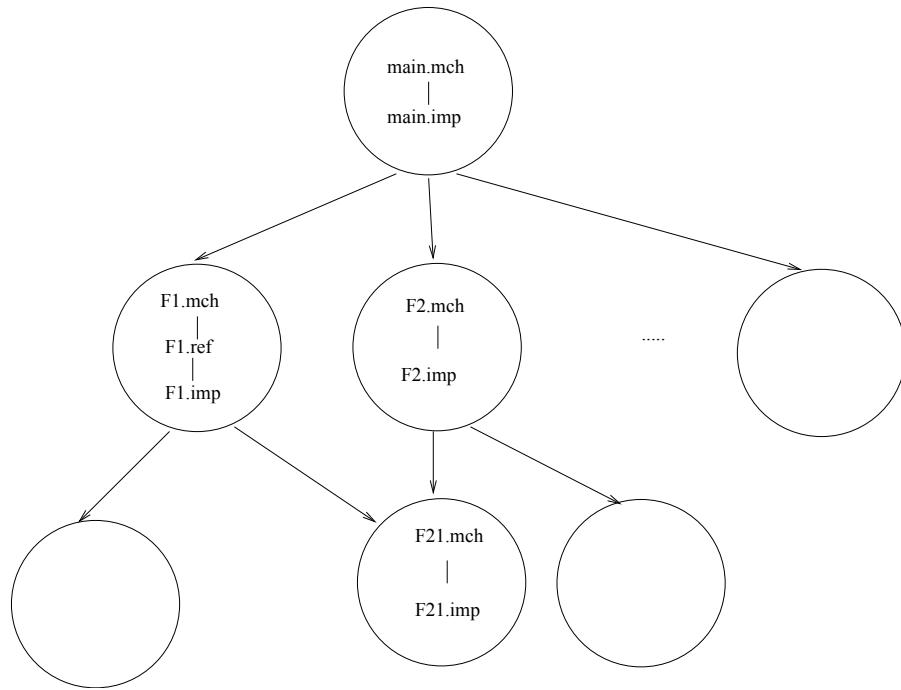


Figure 2.9. Refining an application

In addition to its relative simplicity on the theoretical level, the strength of the B method also resides in the availability of software workshops that take into account each phase of the B development. The commercial workshops are AtelierB, developed by the company DIGILOG (then taken over by the company Clearsy). Industrials (SIEMENES, ALSTOM, AREVA) and RATP used AtelierB.

Note that the B method forms a part of the formal methods, which seem to be most accepted in the industry for developing critical software used in the railway sector [DEH 95]. The B method makes it possible to effectively build a bridge between mathematical modeling and its computing realization.

The software is carried out in the form of a layered model [BEH 96]. The validation phase then consists of generating the set of proof obligations for coherence and refining and testing the proof.

**Figure 2.10.** Refinement within a B model

2.3.3.2. Development process in the B method

In this section, we present the fundamental points of the development process for safety software (to learn more, refer to [CHA 96]).

The development process of the software (see Figure 2.11) can begin after setting up the functional specification of the equipment needs. There are as many specifications as there is equipment.

As shown in Figure 2.1, the *autopilot* of the SAET-METEOR is broken up into three pieces of equipment: ATP-Embedded; ATP-Ground; ATP-Line.

Starting from the functional specification of the equipment needs, the design of the software begins with the definition of a *technical specification of software needs*, which focuses on the software aspects. The functional specification of equipment needs for each piece of equipment is then modeled using the B method [ABR 96].

The B model obtained is a layered model (see Figure 2.10) that can be broken up into two parts: the formal general modeling and the formal detailed modeling, both

of which are made up of abstract machines of various levels (machine, refinement and implementation).

Each implementation is described in a subset of the B language, called B0, which is directly translatable into a conventional language like Ada [ANS 83] or C. This generation is carried out automatically.

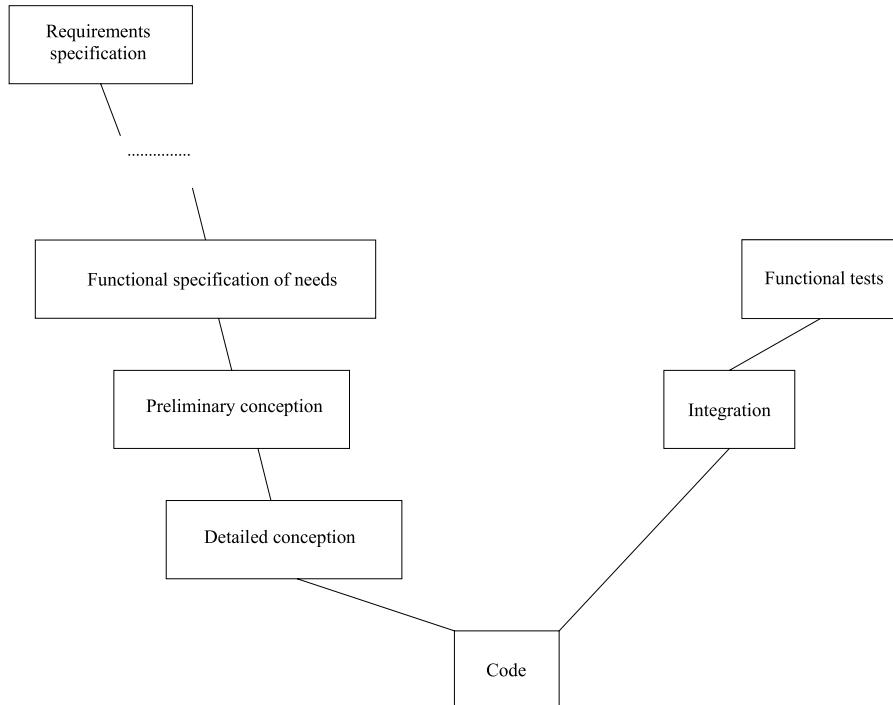


Figure 2.11. The lifecycle implemented by industrials

2.3.3.3. Generic and specific application

As the example in Figure 2.12 shows, a metro line is made up of several tracks (two main tracks allowing displacement in a direction, additional tracks allowing the insertion and the withdrawal of trains and track sidings) and a means making it possible to pass from a track to the other, called a switch or turnout. In section 2.8, we present more precisely the objects constituting a metro line.

The data manipulated by a system can be of different types:

- data of service activation/inhibition;

- fixed data not changing with time, which describe the characteristics of the system (topology, speed curve, stops, etc.);
- data changing with time, this time being either the cycle or a more important duration.

DEFINITION 2.1.– PARAMETER DATA. Parameter data is data that does not change during the execution of the software application.

As definition 2.1 indicates, “parameter data”¹⁰ is the first type of data. The parameter data, in general, gathers two groups of data defining the technical features (speed, weight, size, number, etc.) and data defining the context of use (topology, speed curves, stops, etc.). The parameter data characterizes the implementation of a software application described as *generic* in a specific case.

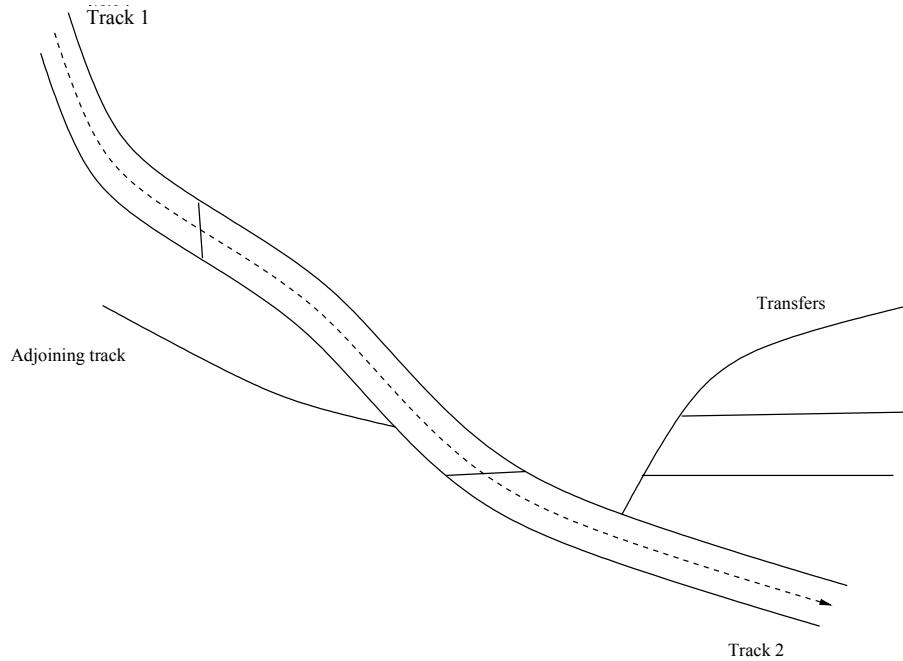


Figure 2.12. Line example

DEFINITION 2.2.– GENERIC APPLICATION. A generic software application is defined according to a set of parameter data that has to be instantiated according to the final use (depending on the site, the services to be activated, the technical features, etc.).

¹⁰ In the railway sector, the parameter data is called “invariant”.

DEFINITION 2.3.– SPECIFIC APPLICATION. A specific software application is a generic software application with which a parameterization is associated. This specific application is usable only for one installation.

A specific application is dedicated to one use and, within the context of the railway industry, to a particular site and/or line. Overall, the data handled by a railway system can be of two types:

- fixed data called *invariant*, which characterize the environment;
- data changing with the state of the system (known as *variant*).

The invariant data is known during the generation of the final version of the system and it defines a railway system configuration. It is thus possible to define a system starting from generic software and parameterization.

From the description of a line, it is possible to deduce a set of data that is common to the set of system equipment, within the context of *autopilot*. This data depends on the topology of the line, the characteristics of the trains, and the technological choices (response time, computing time of the processor, etc.).

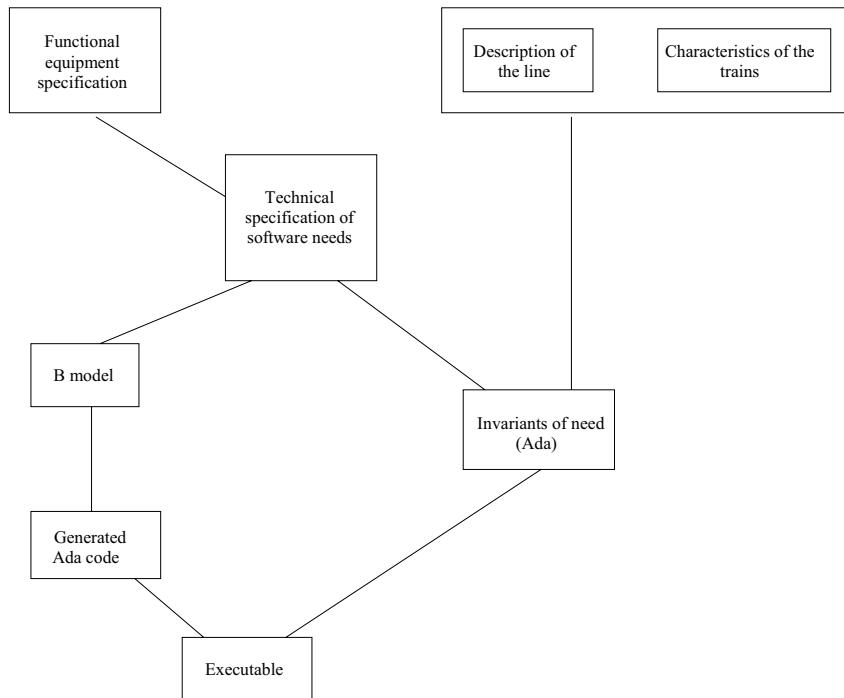


Figure 2.13. Cycle of development

It is called a topological invariant [GEO 90]. From the topological invariants and description of the software needs, it is possible to define necessary invariants. Necessary invariants constitute the set of parameter data that makes it possible to instantiate a version of the generic software.

The separation between the application's data and code allows the compatibility errors between the equipment to be corrected, but more importantly, it makes it possible to obtain a generic description of the equipment.

This generic description simplifies the re-use and installation of the extensions (in 2002, the SAET-METEOR was extended toward Saint-Lazard).

Figure 2.13 thus presents the separation that exists between the software aspect and the “datum” aspect, but it makes it possible to see the links and, in particular, the design process of the final software.

The supplier development process that was introduced [DES 96] in Figure 2.11 takes into account the use of the formal B method. Indeed, we see that the set of validation phases, in particular the test phases (UT, IT and FT¹¹), are only reduced with the functional tests.

However, ignoring the debate about the use of formal methods within the framework of software safety, [BEH 96] says that the unitary and integration tests are redundant with complete proof and the safe code generation. [WAË 95] recommends preserving all the test phases, in the expectation of returning to B tool methodology and validation; this discussion is always valid if the development is executed in a context outside of a PSC.

It is necessary to point out that creating and validating a B model does not guarantee that the code generator, the production tools of executable programs (compiler, linker, etc.), the installation tools, the management tools for configuration, and the hardware structure target transform the execution and therefore render the proofs carried out null and void.

2.3.3.4. Data generation

2.3.3.4.1. Data development

As we have already stated, a line (see Figure 2.12) is made up of a set of objects that corresponds to data that the control/command control system must handle.

¹¹ The functional tests correspond to the validation tests, which result from the functional specification of needs. We could discuss validation tests.

This data [CHA 96, BOU 03, BOU 07] describes the topology of the track (track circuit, negative detector, switches, etc.), the characteristics of the trains (identification, length, kinematic information, etc.), and the constants of the system (cycle time, etc.). In this presentation, we are not interested in either the process of acquisition or the process of validation for this ground data.

Despite the data (description of the line and characteristics of the trains) thus gathered, we still need to set up a process of generation, which makes it possible to obtain data corresponding to the specifications introduced into the development phase of the software.

Within the context of the SAET-METEOR, there are three types of equipment: equipment embedded in each train (ATP-Embedded), one piece of equipment per section of the line (ATP-Ground)¹², and line equipment (ATP-Line). To avoid any ambiguity between the equipment, it was decided to start from a single database.

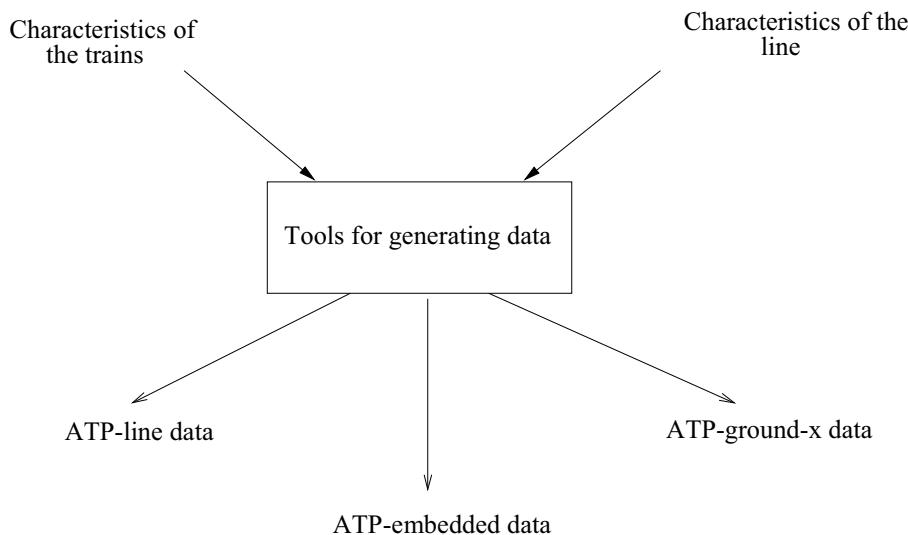


Figure 2.14. Tools for data generation

2.3.3.4.2. Verification of the data

The data verification step consists of showing that the realization of the data produces usable data for the final application. Therefore, it is necessary to show that

¹² In 1998, there were five sections, but there have been six since 2002.

all predicted data is produced, that the format of all the data conforms to what is provided with the required precision, and that unpredicted data is not produced.

Data verification is done through review activities, which consist of verifying:

- the existence of the data handled by the development process;
- the existence of the data handled by the software;
- the existence and correction of the generation process;
- the traceability of the requirements applied to the data.

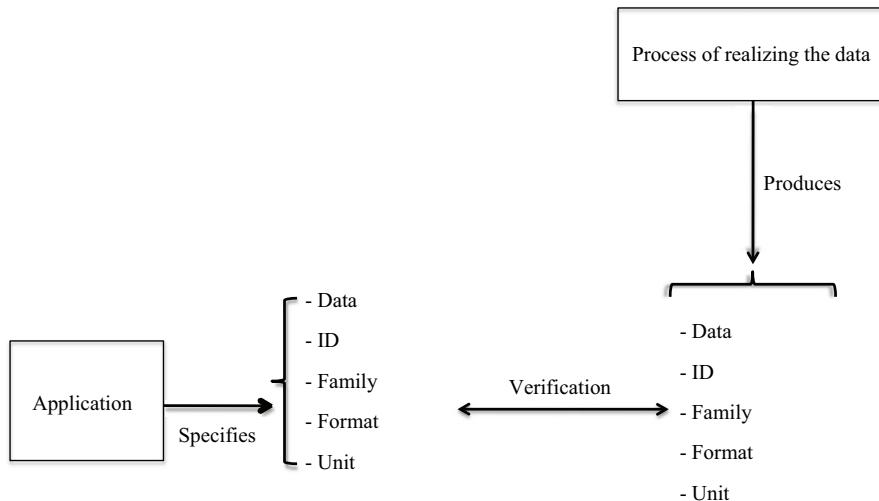


Figure 2.15. Comparability of the data

2.3.3.4.3. Validation of the data

The safety equipment software, starting with the B model, uses the topological invariants of the equipment [GAL 99, BOU 03, BOU 07]. The B model was proven by the introduction of assertions that allow these topological invariants to be characterized. These assertions are accounted for like a specification of the tool for the data generation builder (see Figure 2.14). This process makes it possible to have a separation between the code and the data; we therefore introduce a purely generic code.

As shown in Figure 2.16, the industry sets up a validation, which is based on the existence of a second tool that carries out the same processing. The data generated by

each tool is subjected to a comparator. Given the problems of precision (passing the topological invariants to the necessary invariants is carried out through a transformation based on mark changes) and the difference of the objectives (the second development is focused on the safety aspect, it does not seek to reproduce all the transformations), the comparator carries out an analysis of similarity. This analysis of similarity makes it possible to verify the correction of the data.

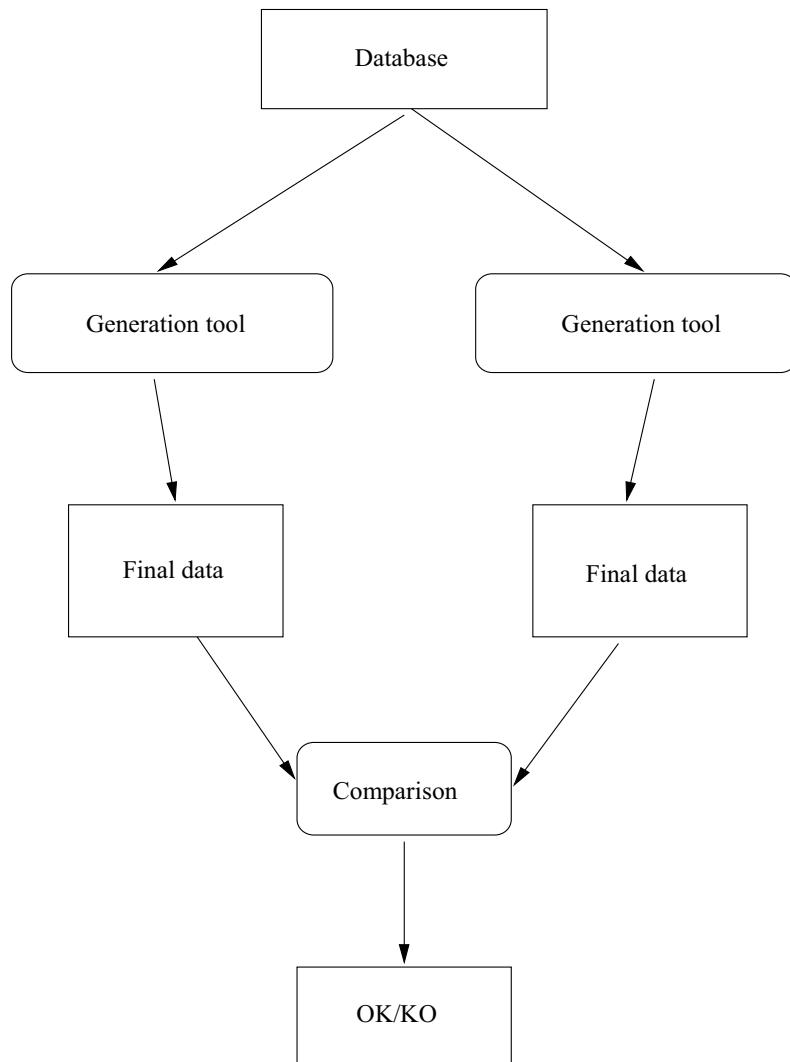


Figure 2.16. Process of descending double generation

2.3.4. Assessment of the processes

As shown in [BEH 96], coupling the use of the B method for the software aspect and the coded processor for the hardware structure makes it possible to increase confidence in the system. The placement of a separation between the code of the application and the data makes it possible to obtain generic software (definition 2.2), which can be verified and validated independently of the final implementation.

As the addition of a new portion of tracks acts only on the given aspect (there is no impact on the specification of the data and code of the generic application), this process makes it possible to replace the test of the set of the equipment by a data validation.

SAET-METEOR's autopilot, distributed on three pieces of equipment (ATP-xx), consists of 1,150 B components, that is to say approximately 115,000 lines of B code that generated 27,800 proof obligations. All these proof obligations were proven. The set of applicative codes (generated starting from the B code) and data makes a total of 150,000 lines of Ada code.

2.4. Process of verification and validation set up by RATP

2.4.1. Context

Since the computerization of railway systems, RATP has always been very involved in ensuring confidence in its systems. This confidence building results in establishing, within RATP, a verification and validation stage independent of the system [GAL 99]. To complete its work, RATP uses methods and tools different from those used by the manufacturer.

The purpose of the verification phase is to analyze the industrial's productions (codes, tests, measurements and documents) to show the correction of work that was realized and risk control.

The double validation is independent of that carried out by the industrial; its purpose is to show that the system meets the needs described in the requirements specifications. RATP validation is based on the functional test carried out on a target.

Within the framework of this chapter, we will present work completed by RATP teams designing the SAET-METEOR, and we will focus in particular on the software aspects. Part of the application of this work to the *anticollision* function was described in [DEL 99a].

2.4.2. RATP methodology

The method employed consists of working out a set of formal properties (founded on mathematical logic) to establish them in a system model or in a system to be validated. This model must be established in a computing tool allowing simulations of test scenarios to be reasoned and carried out.

The set of formal properties makes it possible to precisely describe the behavior of the software to be validated, making it unambiguous and demonstrable.

The validation process implemented by RATP is presented as follows:

- analyze documents of the specification (functional specification of the equipment and the technical specification of software needs);
- carry out a critical analysis of each function;
- remove the list of safety function properties;
- model the safety functions (in relation to the technical specification of software needs);
- verify the properties by introducing them into the model or by verification of the model;
- develop the test sets;
- execute the test sets in test bays¹³;
- use ASA+ models such as Test Oracle¹⁴.

The set of properties is formalized in the form of a document of validation (see section 2.4.3.2).

In [DEL 99a], we presented the set of validation processes and its results for the anticollision function, which is one of the safety functions of the *ground autopilot* equipment.

2.4.3. Verification carried out by RATP

2.4.3.1. Context

As we mentioned briefly, the verification phase implemented by the RATP teams aims to show that the work completed by the industrial is correct. This phase

¹³ A “test bay” is a hardware and software set that includes (real) equipment of the line (AP-xxx) and a simulator of the environment allowing a real time simulated execution.

¹⁴ A TEST ORACLE is a software component that makes it possible to decide if the result of a test execution is correct or incorrect.

therefore consists of verifying the existence of the products (documents, sources, results of analysis, etc.) that the industrial predicted would be delivered.

Verification applies to the group of processes of the studied system. As shown in Figure 2.1, any system can be broken up into subsystems and equipment, and each piece of equipment can then be separated into a group of software and hardware.

Within the framework of verification, we pay particular attention to the *specification*. Indeed, any problem introduced on this level, whether it is concerning specifications, subsystem, or equipment will have financial or other consequences on the set of activities.

Verification of the specification can be carried out through the realization of a model. According to the degree of formalization of the model, it is possible to implement more or less formal verification techniques: animation/simulation, exhaustive simulation, or proof.

2.4.3.1.1. ASA and ASA+ modeling+

Within the context of the SACEM [GEO 90], the RATP teams used the ASA environment¹⁵ (analysis structured by a state machine) for modeling and verifying the specification. The environment of ASA verification implements an IDEF0 view (SADT hierarchical functional decomposition) and in addition proposes syntactic and semantic verifications, a tool of interactive and/or exhaustive simulation (*model-checking*).

As shown in Figure 2.17, the ASA environment makes it possible to execute a hierarchical model that is composed of a set of boxes and wire (static description). The functional analysis represents the statics of the model; it is structured through modules, interface points, and channels of communication between the interface points.

Each module represents a behavior that can be broken up into other behaviors in a lower level; we therefore repeat the process until obtaining basic behaviors represented by module leaves.

The modules communicate with each other by exchanging messages transmitted or received through the channels of communication, which are connected *via* interface points.

¹⁵ RATP and EDF developed the ASA environment and its evolution ASA+ [VER 95]. These modeling environments are based on a structure described with SADT [LIS 90] and the establishments in the form of extended state machines (using algorithms) that communicate (exchange messages).

The modules communicate with each other by exchanging messages transmitted or received by the respective interface points and are propagated along the channels connecting these modules with each other. The communication can be synchronous (rendezvous) or asynchronous (file).

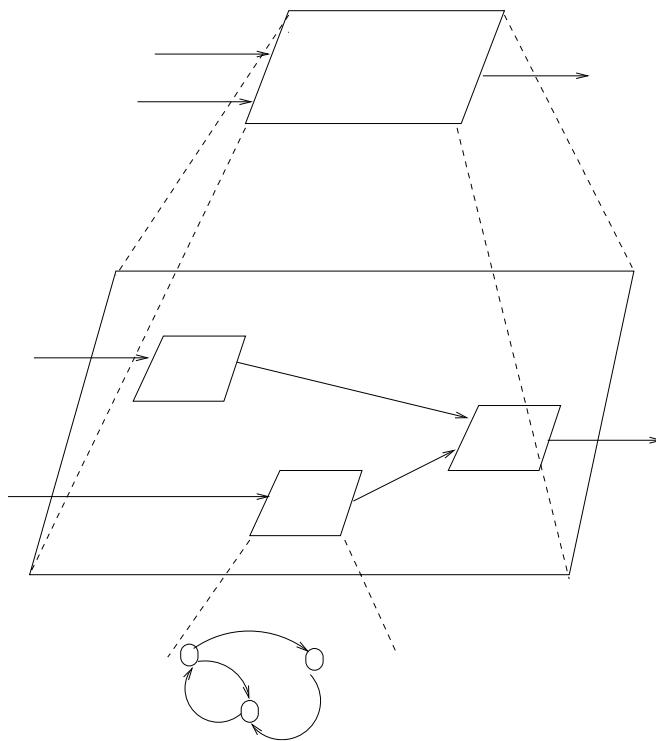


Figure 2.17. Model arranged hierarchically

The ASA product [VER 95] was used to verify and validate the design of the SACEM system (1986), which makes it possible to control the speed of the RER A.

The ASA+ environment has been used to validate the safety software of the autopilot system of the SAET-METEOR (1994–1998) and the KCVP (1999), which takes part in controlling the RER B.

2.4.3.1.2. ASA and ASA+ simulation

There are three modes of simulation within the ASA+ environment:

- interactive simulation, based on a manual construction of the scenarios, which generates textual traces and graphs;

- automatic simulation, based on a random construction of scenario(s), which generates statistics and scenarios;
- exhaustive simulation, based on a targeted research, which allows formal verification by exploring the behavior.

As in the case of the validation of MAGGALY [STU 93, STU 94, STU 95], the verification process set up by RATP is based on the concept of property [STU 93, STU 95]. In [NAT 97], we proposed an approach that showed that we could check the properties on the software resulting from the safety studies (FMECA, fault-trees, AEEL, etc.).

2.4.3.1.3. Finally

Within the context of the SAET-METEOR, the RATP teams used two approaches:

- an approach based on the concept of the Petri network, which allowed the general behavior to be checked. This is based on a very abstract modeling of functions and allows only one validation per animation. It will not be dealt with here;
- an approach based on a model of an extended communicating state machine, which makes it possible to model and verify the behavior of each safety function. We will describe this model here.

2.4.3.2. Requirement specification

Before being able to carry out a model, it is necessary to define its contour, to understand the behavior of the model's object, and to identify the verification objectives. To do this, the first stage consists of carrying out a *requirement specification*.

This *requirement specification* is a document equivalent to the specification that identifies the model's object and its interface with the environment, which describes the predicted behavior and lists its properties that it is necessary to verify. This document is traceable with the documents to be verified for both the functional aspects and the properties.

DEFINITION 2.4.– PROPERTY. A property is a characteristic that a (hardware or software) component must verify.

The properties must be analyzed and classified. We propose a classification into two parts: safety and liveness.

DEFINITION 2.5.– SAFETY PROPERTY. A safety property asserts that something bad never occurs during execution.

As an example of the safety property, we can cite the absence of blocking, mutual exclusion, etc. and in our case, the absence of a collision between trains.

DEFINITION 2.6.– PROPERTY OF LIVELINESS. The property of *liveliness* expresses that something good inevitably occurs during execution.

The termination of an application, the absence of famine (constant progression of applications), and the guarantee of service are many examples of properties of liveliness. An example of the property of liveliness: if the elevator is called, it will arrive in the immediate future.

2.4.3.2.1. Functional requirements

Starting with the documents of the specification (equipment and software), the behavior of each elementary function is described by equations linking the input and output of the function. In our case, these equations make it possible to indicate how the output is generated locally on each train present on the track. These functional requirements can be translated into behavioral properties.

Each equation counts for only one particular situation. No factorization was introduced to decrease the number of equations. This makes it possible to have a list of the anticipated situations and behaviors. These situations are described through traversed working areas, the state of the elements of the track encountered (high or low CdV, ND obscured or not, switch controlled or not, etc.) and information concerning the train (type, control mode, speed, etc.).

As an example, we can state that the property: only the trains equipped, located, and in automatic control mode can have a target. We introduced tree temporal properties [AUD 89], which make it possible to connect the histories of each sub-function of the anticollision function. These properties make it possible to define ways through the various equations.

These properties could be verified only by exhaustive simulation because they are of the “A each time that...then...” type.

2.4.3.2.2. Extra-functional requirements

In the set of extra-functional requirements, there is a subset related to the safety aspect. These requirements will be expressed through safety properties. In the case of the anticollision function, the safety operation is globally guaranteed by restricting its output. This restrictivity can be done in relation to the anticipated behavior but especially in relation to what it is possible to do on the physical track.

The properties related to safety could be formulated by good properties (*liveliness*) and bad behavior (*safety*).

The properties of correct operation are global properties for the set of input/output in the system. There are properties of comparison of the physical world (state of the environment) and the calculated world (virtual state of sections, overlaps, and targets). There are properties of persistence, or lack thereof, from one cycle to the other and transverse tree temporal properties with the various anticollision functions.

The properties of faulty operation are enumerated in the form of unattained situations (continuation of states of incoherent CV, target exceeding another train, etc.).

2.4.3.3. Modeling

For the realization of the behavior model (Figure 2.18 presents an example of a closed model), the ASA+ environment was used. ASA and ASA+ were used within the context of railway applications but also in the field of electricity; see [SAB 97] for a report of experiments in this field. These tools were marketed by the company VERILOG and then by the company TELELOGIC; they stopped production in 2000.

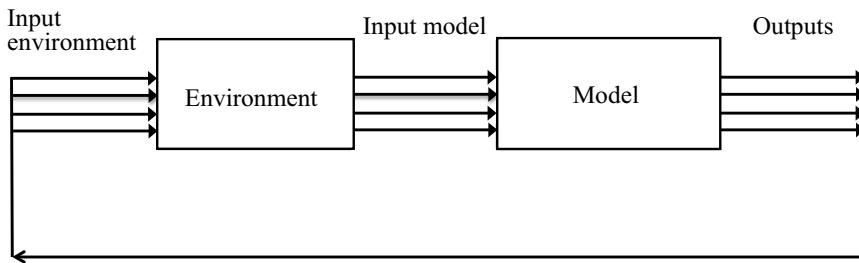


Figure 2.18. Closed model

ASA+ [VER 95, VER 94] is an extension of the ASA environment that makes it possible to capture the structure of the system to be modeled through an IDEF0 view (see Figure 2.17) and the behavior through a concretization of the boxes leaves in the form of an extended communicating state machine.

The ISO language ESTELLE [COU 87, ISO 89] is used to describe the state machines. Figure 2.20 presents an example of a state machine that characterizes the behavior of a train.

The *communicating* characteristic implies that there is a process of sending and receiving messages and the *extended* characteristic is brought about by the fact that it is possible to put PASCAL or C code in transition. The use of a programming language is brought about by the need to set up realistic dynamics of the environment.

As Figure 2.19 shows, the functional analysis represents the static part of the model; it is structured through modules, interface points and channels of communication between the interface points. Each module represents a behavior, which can be broken up into other behaviors on a lower level; we therefore repeat the process until we obtain the basic behaviors represented by module leaves.

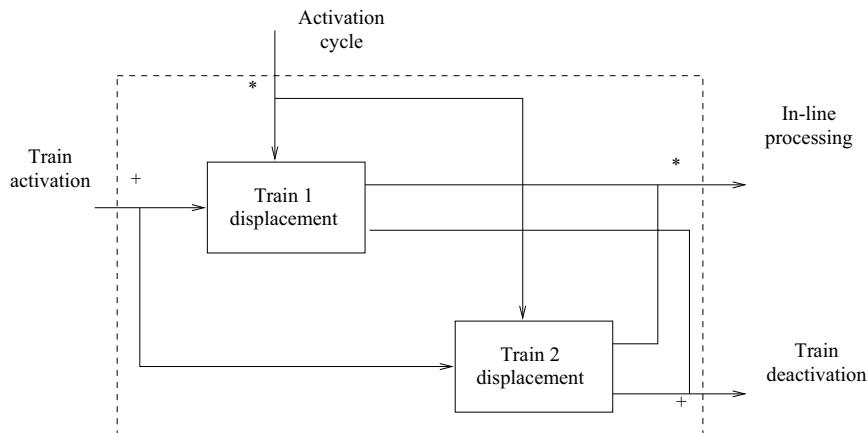


Figure 2.19. Example of statics introducing various types of communication

The static view has two points of view: the hierarchical point of view that makes it possible to visualize the tree of decomposition; the composition point of view, which makes it possible to visualize the communications between modules of the same level. This supports the black box vision of the modules.

In synchronous communication, we can choose two modes of rendezvous: rendezvous with everything (all the recipients are on standby), noted as *; or rendezvous with at least one (at least one recipient is on standby), noted as +.

In managing the displacement of the trains, we had to use the two types of rendezvous (see Figure 2.19). With each cycle, all the trains must carry out an

action, which is why the signal *Activation Cycle* must be sent to all the modules of displacement management. In the same way, *In-line Processing* information makes it possible to know by counting that all the trains have finished their displacements. On the other hand, the message *Activation_Train* corresponds to the input of a train onto the track, if it is *train 1* that carries out its track input, this message must arrive only at the module managing the movements of *train 1*: it is the same for the deactivation message for a train.

The behavior of an unsheeted module results from the cooperation of its wire modules. The installation of the behavior of the sheets is done by communicating extended state machines (see Figure 2.20); for this, we discuss dynamics. The state machines are expressed in LSA+ language [VER 94b].

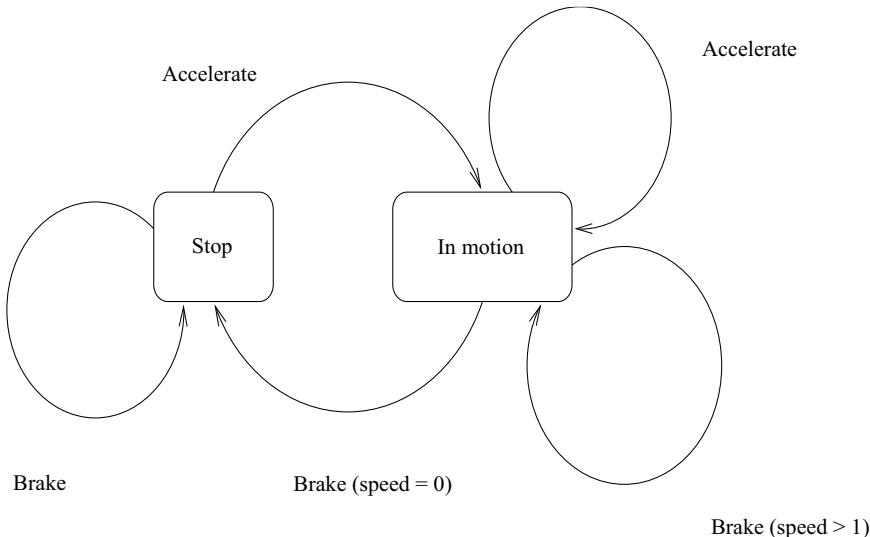


Figure 2.20. The state machine describing the behavior of the trains

The formalism of description of the extended communicating state machines use: the concept of state and transition for the state machines; the concept of a shooting interval associated with the transitions; the concept of emission and reception for the communications, and a subset of the PASCAL language to extend the possibilities of describing the state machines.

In the dynamic model of a leaf function, the states store the internal variables of the function and the variables exchanged with its environment. The transitions between states define the evolution of these variables.

Any transition can be conditioned (PROVIDED clause) by a reception or an emission of a named message, a launching conditioning in terms of temporal constraint and/or a Boolean expression. The launching of the transition involves the execution of a block of action able to change the environment of the state machine. The block action of each transition can use external functions and procedures from the state machine that can be written in PASCAL or C.

Figure 2.21 represents the graphic form of a state machine related to the movement of the trains on a restricted topology.

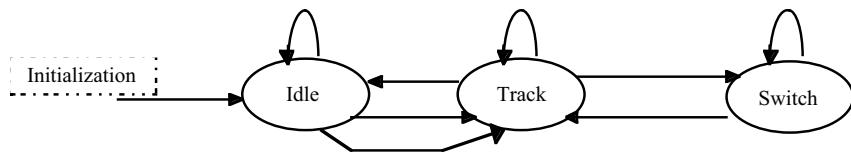


Figure 2.21. State machine describing the abstract movement of the trains

As shown in Figure 2.22, the textual form of the state machines is broken up into three parts:

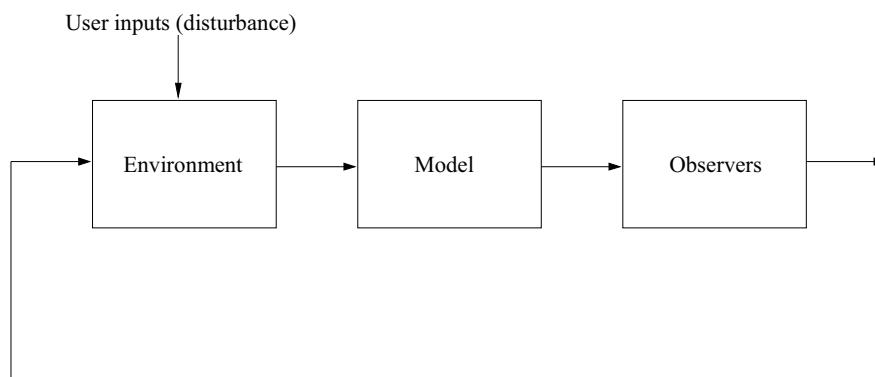
- declarations (we define the states of the state machine, the variables, and the functions or procedures);
- initialization (this makes it possible to position the state machine in a given state with a set of given values);
- description of the transitions allowing it to change from one state to another.

This presents two transitions related to the state *IDLE*, which make it possible to consume the message *Active_Train* intended for the state machine and thus to indicate that a train has just entered onto the track.

The safety properties were coded as observers [LAG 93]. Figure 2.23 introduces the structure of the model, which has three phases: handling of the environment; execution of the model; execution of the observers.

As shown in Figure 2.24, starting from a program *P*, of its specification *SP* and a set *A* of safety properties formulated as expressions of the variables of *P*, it is possible to build a model *M*, which has as an input *E* and as an output the Booleans *S* and *S'*. The formal verification of a program *P* then consists of simulating (interactively or exhaustively) the model with input to check that *S'* is always true.

<pre> TRANS FROM IDLE TO IDLE WHEN Active_Train(num) OUTPUT mode_control(num) PROVIDED (num = NB) and cycle_active and (speed_train(num)=0) NAME Input_Train_speed_nul: BEGIN cycle_active:= false END; </pre>	<pre> TRANS FROM IDLE TO TRACK WHEN Active_Train (num) OUTPUT mode_control(num) PROVIDED (num = NB) and cycle_active and (speed_train(num)<0) NAME Input_Train_with_acceleration: BEGIN cycle_active:= false accelerate_train(num) END; </pre>
--	---

Figure 2.22. Structure of the model**Figure 2.23.** Structure of the model

Setting up an environment is necessary to close the model (see Figure 2.18) and to limit the number of *out of context (spurious)* scenarios. The purpose of the environment is to react to the output of the functions to be verified and to produce new coherent input with the history in progress.

Figure 2.25 presents the dynamic behavior of such a model.

Since verification was carried out for a given piece of equipment and with respect to size, it was decided not to carry out a global model of the equipment but instead a model by function.

The link between the functions was carried out through traceability actions on the interface level.

To offer a greater re-utilizability, ASA+ supports several forms of genericity. The first form makes it possible to make a static module generic; the second allows a state machine to be used in several instances.

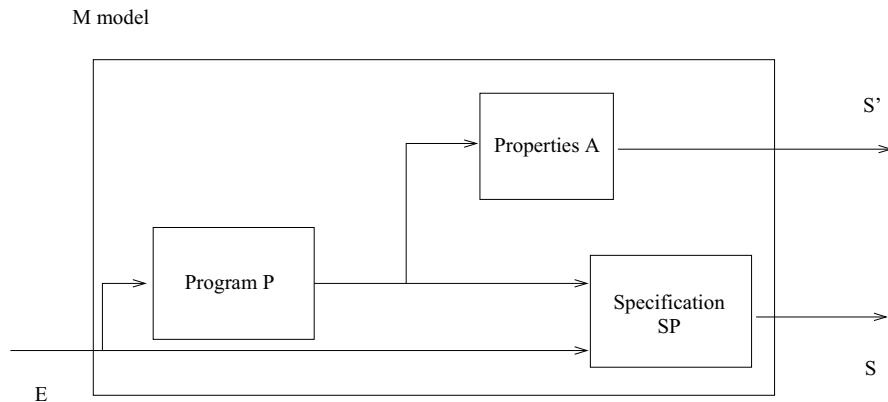


Figure 2.24. Principle of verification

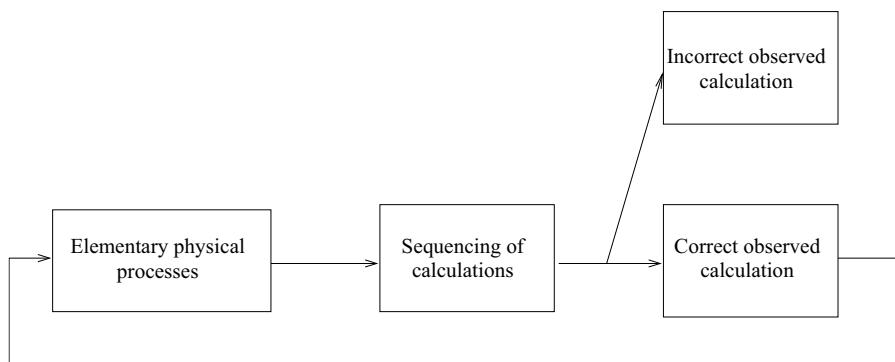


Figure 2.25. Principle of execution

In Figure 2.19, we have a generic module dealing with train management that was instantiated twice as well as the associated state machine. The *genericity* was used to introduce the behavior of the trains and the objects constituting the track (ND, CdV, Switch).

2.4.3.4. Structure of the model

The functions to be validated form a cyclic function, which gather data coming from the real world (state of the track's elements, train information, etc.), and which carries out processing and repeats the process. To have a closed model, we introduced the environment into the model.

This environment makes it possible to move (acceleration, deceleration, reversal) the trains and to introduce defects such as the disturbance of the behavior of the track's physical elements, loss of messages coming from the trains, and abnormal train movements such as the train moving backward.

2.4.3.4.1. Global structure

As shown in Figure 2.26, there is a model of the function's environment, a model of the function in term of properties, and a realization of each of these elements.

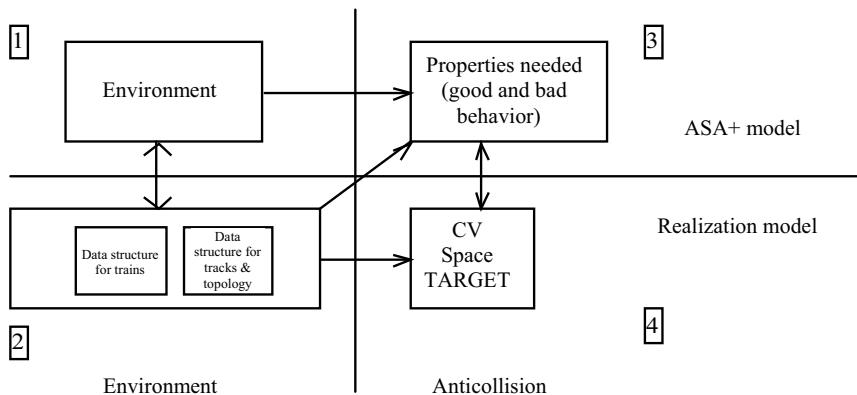


Figure 2.26. Structure of the model

The model is therefore structured into four parts:

- 1) specification of the environment, which includes the set of events that can intervene in a simulation (transition of the ASA+ model);
- 2) realization of the environment is deduced from the track plans and the operating procedures that define possible train movements;

3) specification of the *anticollision* function, which includes:

- a) the list of functions (transition of the ASA+ model),
 - b) rewriting of the functional specification of equipment as equations for the behavior and as Boolean functions for the properties;
- 4) the realization of the *anticollision* function, which is a rewriting of the technical specification of software needs in imperative language.

The ASA+ model, composed of models 1 and 3, is the driving force of validation. If a property is violated, the model falls into a *well* state; if not, it produces a new physical situation.

The realization model (2 and 4) is an independent module, written in a conventional programming language. It is interfaced with the ASA+ model, which remains independent of the possible modifications of the technical specification of software needs or the design choices such as the topological or kinematic parameters.

2.4.3.4.2. Modeling the environment

The environment is a sub-model whose dynamics can be controlled to guide the model of the anticollision function. The operation of the ATP-Ground is fixed for whatever section of track may be chosen. This is why the description of the tracks is outside the model and choosing a track is done during the initialization of simulation. The environment must thus make it possible to choose a track, to change the state of the elements constituting the track's topology (breakdown or repair), to position the routes, and to execute actions on the trains, such as activation, displacement, or change in the control mode.

The model (specification level) makes it possible to describe the behavior through equations and properties as state machines, (for example, the trains can advance or move back, but the backward movement can be done only starting from zero speed); the realization makes it possible to have an implementation via physical objects (the displacement of the trains involves updating the sensors) that construct the states of the environment.

The description of the environment must be sufficiently precise to describe the various configurations that must process the anticollision function. This is why the definition and establishment of the environment are two of the most difficult aspects of modeling.

2.4.3.4.3. Expression of the properties in the model

The set of properties that our system must verify is divided into three distinct sets: observation that collects the equations of behavior, evaluation that gathers Boolean properties (liveliness and safety) able to be expressed according to the variables of the system and the environment, and sequencing (behavior property) that collects a set of tree temporal properties.

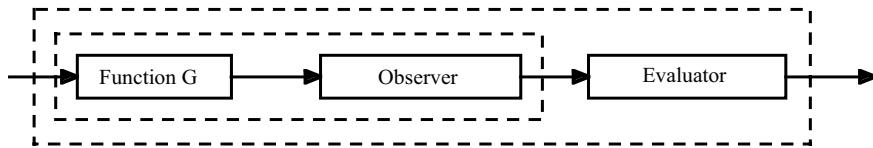


Figure 2.27. Observation and evaluation

Any elementary G anticollision function can thus be expressed in the form of an observer (subset of observation) and of an evaluator (subset of evaluation), according to the diagram in Figure 2.27.

Note that the observer does not redo the calculation but instead evaluates the validity of the result. The properties of the evaluator have a more global aspect when compared to those of the observer.

In the case of the function TRACKING, which is broken up into six sub-functions, we obtain the graph in Figure 2.28.

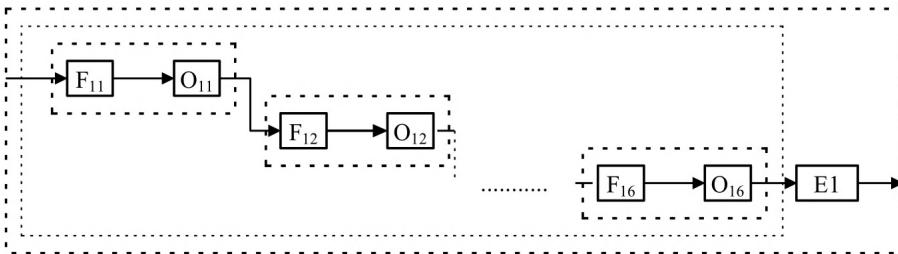


Figure 2.28. Decomposition of the tracking function

We have only one global evaluator for the TRACKING function, but we have six local observers for each sub-function. Each module F_{1i} represents the trigger and

the end of execution of a realization function whose result will be observed by the observer O_{li} .

The properties of behavior related to sequencing (expressed in several cycles or questioning the states achieved by various observers) cannot be introduced directly into the model, but will have to be verified by simulation.

2.4.3.4.4. Observers

The observers are state machines that make it possible to observe the behavior of the function. This behavior is defined by the properties of behavior that were expressed as equations.

Each equation is modeled by a transition of the observer whose states represent the observable states of the function. The observable states correspond to particular configurations for which the function produces identified output.

In our case, the observers are state machines based on the concept of a train and they establish that the calculation carried out near this train is correct. Figure 2.29 introduced the observer skeleton. Note that the starting state is always the IDLE state and that if all goes well, we always reach the state *Processing_Correct*, which is the return state.

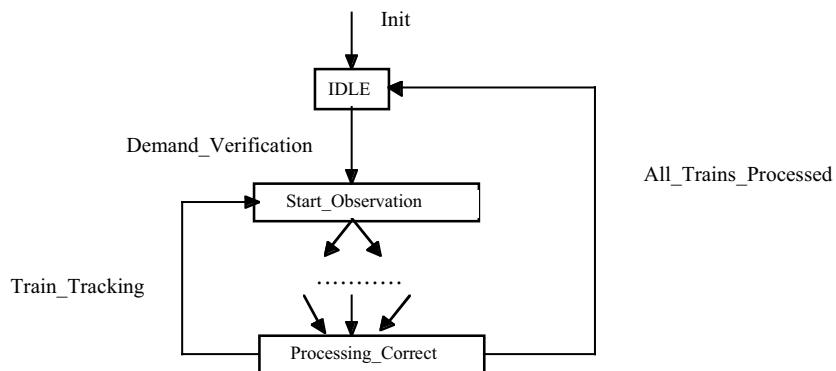


Figure 2.29. Structure of the observers

The task of the observers is to generate a history of the calculation carried out by each function. This makes it possible to validate the model and to locally trace the changes of states to a train. This local analysis allows the anomalies detected globally by the evaluators to be explained using a fine trace of the behavior of each train.

The transitions of observers must be discriminating to reconstruct a relevant history, which is to say that it is necessary to have only one path per situation and that falling into a well state characterizes an incoherent situation. A state machine whose transitions are all discriminating can be more easily validated and the coverage of the tests more easily measurable. This is also consistent with our need to obtain an exhaustive list of the unitary situations able to be met.

It is easy to show the total coverage of the transitions during validation. This coverage can be expressed in terms of coverage of real situations and not in terms of global states of the function.

2.4.3.4.5. The evaluators

As shown in Figure 2.30, the purpose of the evaluators is to evaluate the global properties (for example, the virtual occupation of the track must be more restrictive than the real occupation), these properties are coded as Boolean functions in the LSA+ language.

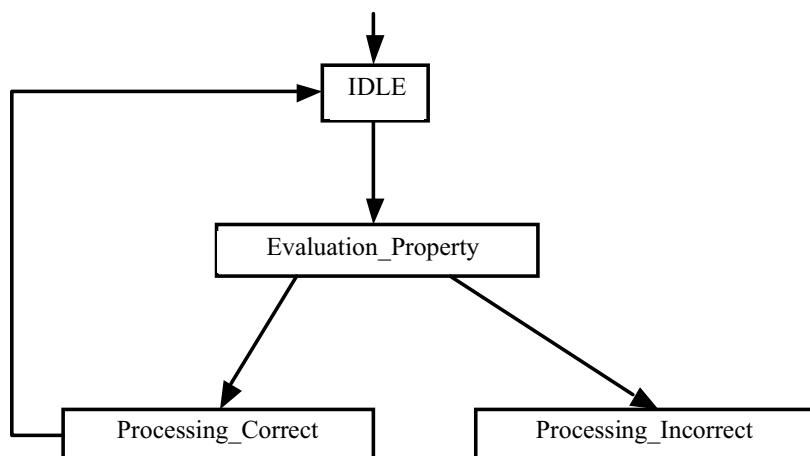


Figure 2.30. Structure of the evaluators

Coding a property amounts to constructing a Boolean function that has access to the environment's data (the position of the trains, the state of the physical elements) and to the data calculated by the analyzed function.

The action part of evaluators is therefore very simple (calling for Boolean functions) and the addition of properties involves few modifications.

Each property violated leads us into a well state; this behavior allows the simulation to be stopped as soon as possible. The properties introduced in the evaluators are the properties of cohesion, safety and correction that were introduced in Stage 1 of this study.

2.4.3.4.6. Model of the realization

The model of the environment in Figure 2.26 contains two parts:

- 1) the realization of the environment that makes it possible to calculate a physical configuration according to the transitions activated in the state machines;
- 2) the realization of the anticollision function described in the technical specification of software needs.

We had two options to set up the realization: to use the PASCAL extensions of the LSA+ language, which makes it possible to define types, variables and sophisticated processing on the state machine level; or to use the access to C language during the execution of a transition of the ASA+ model.

Since the set of processing of the environment and the functionality to be studied was relatively complex and contingent upon heavy data structures, we preferred to use C language as a development language. This choice has three advantages:

- the development of the program is simplified in C because in LSA+ the laying procedures of a point of observation or a trace requires the development of a state machine or a specific model. It is thus easier to build a C program or to use other facilities like a symbolic debugger;
- the environment described in C can be re-used for line simulation, which cannot rely on ASA+;
- the model of the technical specification of software needs can be replaced by the executable code effectively implemented, for example, by generating C code from the B method [ABR 96].

The algorithmic layer is broken into two parts: the environment and the realization of the anticollision function, the environment constituting the most important part (see Figure 2.26).

2.4.3.4.7. Diagram of model use

The dynamic behavior (Figure 2.25) is divided into three phases:

- calculating the transitions activated to the environment level in terms of physical processes. This means that the topology is updated by calling some specific functions and that the state reached by the physical process is obtained;

– activating the calculation of the analyzed function and obtaining the calculated state, which is the image that is created by the information processing system of the physical world;

– verifying the properties of behavior and safety. If the properties are verified, we return to the first stage, if not, we are blocked in a well state.

2.4.3.4.8. Global model

Figure 2.31 presents a complete description of the global model.

The anticollision function is established on a sequencer; the dynamic behavior of the model is cyclic (environment action, calculation/observation of the function, evaluation). With each cycle, the data concerning the new situation is acquired, and then the calculation is carried out without interruption. Initialization makes it possible to choose a section of track and to start the simulation.

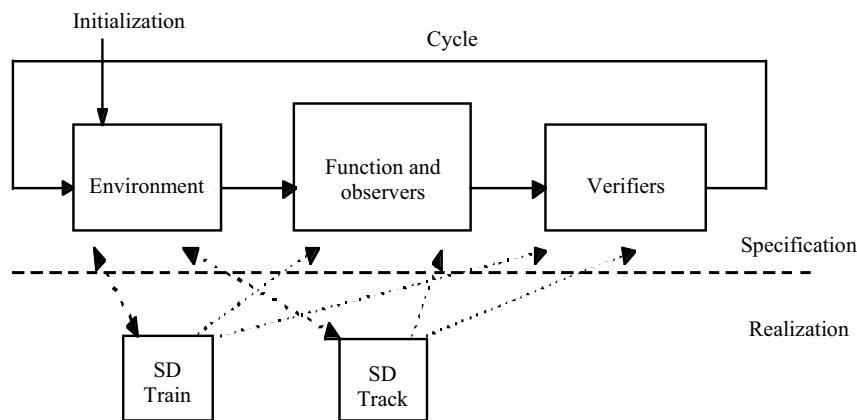


Figure 2.31. Global model

2.4.3.5. Technique for verifying the models

As shown in [VER 94a], the ASA+ environment has several mechanisms at its disposal for verifying a model.

DEFINITION 2.7.– SIMULATION. The simulation of a model M corresponds to carrying out a controlled execution of the model.

The model verification process implemented is broken up into several stages:

- *reading back*: a cross-checking read-back activity was implemented;
- *interactive simulation*: a first phase of simulation that aims to validate the behavior of the model;
- *exhaustive simulation*: starting from the generation of the entirety or part of the graph of the state of the modeled system, this phase of simulation makes it possible to verify the correction of temporal properties.

2.4.3.5.1. Verification by interactive simulation

The conventional principle of verification is simulation. A model of the system is executed in an environment and the observation of its behavior compared to the previous one makes it possible to decide if it behaves in accordance with its specification.

In the absence of a modeling of the environment, it is necessary to select the input by hand (randomly, at worst). There are two problems that arise:

- consistency between the input;
- link between the output and the state of the model.

Setting up an environment implies a complementary development and validation efforts to show the validity of the environment, but it makes it possible to limit the space of the executions, which is why we decided to work with closed models (see Figure 2.18).

The simulator makes it possible to carry out two types of simulation: a step-by-step simulation (see definition 2.7); or an automatic simulation by randomly choosing the transition to be executed.

These two types of simulation were first used to validate the model and the second time around to search for relevant scenarios.

In interactive simulation, all the states of the environment of the model executed cannot be explored. In verification, two strategies of exploration were used:

- a predetermination of the situations of the environment allowing an exhaustive path of the states and transitions of the *anticollision* function modeled;
- a guided simulation permitting the system to be led toward situations considered critical or toward rare events.

The first strategy depends on the specification tests generated during the validation phase of the model, the objective being to show that the safety properties are always respected.

The second strategy is based on the expert scenario concept introduced as transition sequences of the environment set by a railway expert. This strategy of simulation can be used to verify global scenarios and behaviors in extreme cases and to see unanticipated behaviors to evaluate the robustness of the functional principles.

Certain properties of the observers were violated:

- a lack of respect for certain hypotheses on the parametrization level of the model (poor adequacy between the speed of the trains and the topology of the track). The associated scenarios made it possible to highlight the limits of the model and constraints of exploitation;
- the fact that the application implements an algorithm that requires several cycles to stabilize it. We could show that this was not contrary to safety;
- defects were found in the specifications, all of which were corrected.

2.4.3.5.2. Verification by exhaustive simulation (model-checking)

Exhaustive simulation [BAI 08] aims to simulate a program for all the realistic configurations of its environment. It deals with a highly combinative problem.

The principal difficulty in implementing exhaustive simulation lies in controlling the memory size of the graph of the states of the system being analyzed. We are very often confronted with the problem of combinative explosion: the number of states of the system is also important.

There are several solutions:

- implementing optimization algorithms of the stored graph size: for example BDD (*binary decision diagram*);
- establishing an abstraction of the model;
- establishing an analysis strategy for the system based on trajectories (partial analysis);
- establishing other management strategies for the states; for example, we can no longer store the states but the properties of these states;

Within the context of the work completed by RATP, the verification objective was fixed as a partial correction proof for the system realized. There is no question of completely verifying its correct operation, but simply of making sure that it respects certain vital constraints of its observable behavior.

These constraints can be formalized as local tree temporal properties in sub-functions or global ones in the anticollision function.

Model for exhaustive simulation

The model is based on three views that belong to different levels of abstraction. Only the observers are abstract enough to allow exhaustive simulation to be implemented (see Figure 2.32). The exhaustive model is composed of the observers connected to each other. At the observer level, we should relax the constraints due to the environment and the objects of the model.

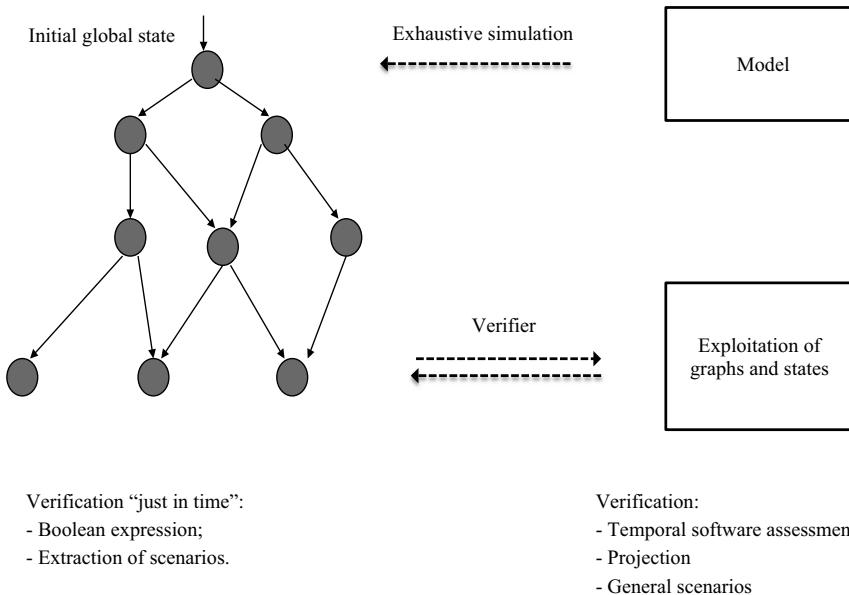


Figure 2.32. Implementation of exhaustive simulation

Diverse techniques were used to limit the combinations of the system to account for only physically feasible situations.

Exhaustive simulation can be restricted through the observers. This solution makes it possible to focus on a specific context. The set of contexts is representative of the situations to be analyzed.

Exploitation of the state graph

Once the graph of a state is generated, it is possible to raise questions that will result in a subgraph, as is the case with projection, or a set of scenarios, as is the case with searching for paths [VER 94a]. It is possible to combine the two approaches.

The concept of verifying property is based on a verification nucleus of logical formulae that can contain operators of tree temporal logic, which can apply to the states or the arcs. The concept of projecting a graph is founded on the concepts of bi-simulation and congruence compared to a relation of equivalence.

Within the context of searching for a path, the number of generated paths is too important to be exploitable. On the other hand, projection makes it possible to generate an average-sized state machine that can always be reduced by a new projection, but generating the path starting from the final graph is no longer possible.

In this study, projection makes it possible to highlight sequencing properties and correct operation (*liveliness*), between observers of the anticollision functions, but also to generate meta-scenarios. These meta-scenarios describe a situation in terms of a train, but without the concept of positioning on the track.

Example: “if two successive non-equipped trains enter a switch zone in the direct routes formed, then...”.

2.4.3.6. *Limit of verification by simulation*

The installation of significant test sets to validate the model by simulation is a difficult activity due to the fact that it is not possible to be exhaustive. This is why it should be coupled with a strategy and a metrology.

The strategy for selecting tests makes it possible to limit the field of the domain, but this is based on either knowledge of the problems or on choices.

Metrology is based on the analysis of the coverage of simulations (tests carried out), which makes it possible to evaluate the quality of executed work and can be seen as a stop indicator.

The technique of (pseudo-) random tests is an interesting solution because it does not require any preparation. It can be immediately implemented and it generates cases that may not have otherwise been considered; on the other hand, they are not very powerful (blind method, repetition, many failures).

2.4.3.7. *Validation of the model*

As shown in Figure 2.33, validation was carried out in two phases: validation of the realization and validation of the model.

The validation of the realization was done in relation to the technical specification of software needs through two activities: reading back the code by a third person and test program construction. The read-back activity revealed errors

and made it possible to increase legibility through questions formulated by the reader to the developer.

The test programs cover situations of references indexed in the various documents or derived from function principles. During this phase, the reader was tasked with generating documentation relating to the different modules of the realization. The process was repeated until the reader accepted the code.

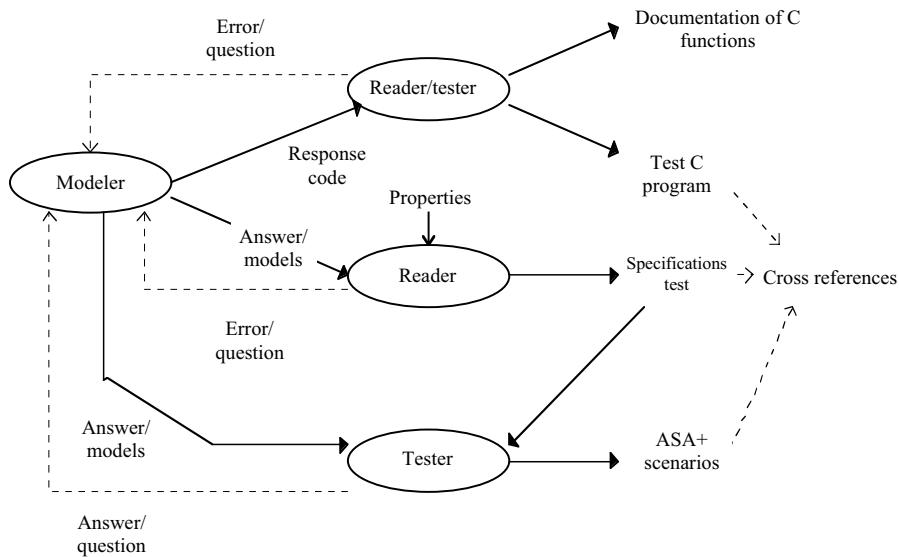


Figure 2.33. Development/validation process

We followed a similar process for validating the model. The test programs were replaced by ASA+ simulations which, for this phase, can be considered an animation of the specification.

The validation of the model was done in three phases:

- 1) C/ASA+ integration test;
- 2) validation of the behavior of the observers (not a well state, only one path, consistency between physical and calculated situations);
- 3) verification of the properties of the evaluators.

Alongside these activities, the set of situations able to be attained by the observers of the model was listed by a person outside the development and

validation processes. This whole situation allows the set of unitary tests that must have run in the model to be defined. A document made it possible to run this whole situation, the test programs, and the set of scenarios carried out in the model. This set of physical situations made it possible to define the test specification, in terms of situation and anticipated result. The definition of the test specification followed incremental development.

As shown in Figure 2.34, to be able to work quickly, we decided to carry out an incremental development of the functionalities of the model. With each new version of the model, we introduced a new element of topology, or a new anticollision function, and/or corrected software errors.

All the major modifications were indexed in a document and the set of tests were executed again to ensure non-regression of the model in progress. The chain of non-regression was maintained until the integration and validation of the entire model.

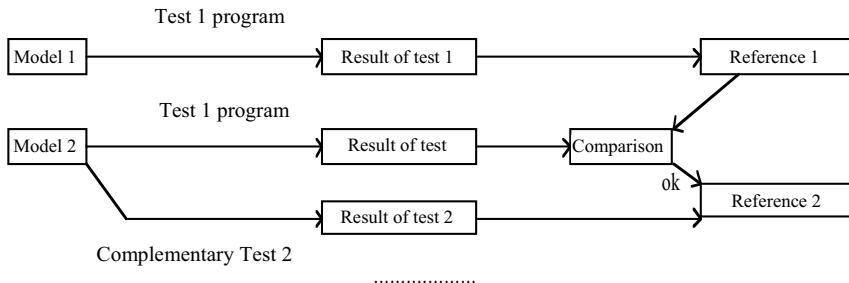


Figure 2.34. Non-regression chain

2.4.3.8. Assessment of modeling

It is important to note that to set up our model, it was necessary for us to write three views of the studied system: the observer view which is an equational view, the evaluator view which is a directed property view, and the algorithmic realization.

Our model rests on two important concepts: the environment to avoid having an open model able to accept incorrect situations and generate an unacceptable combination, and the observers/evaluators that permit having a directed property model.

The success of modeling owes much to the development/validation process implemented. This process made it possible to fully integrate the people of RATP into development and to begin the process of transferring knowledge early.

The 1,500 lines of code of the evaluators represent the coding of about 30 properties. The development, validation and verification of this model took 27 people/month.

	Environment	Observer	Evaluator
Specification	39 states and 135 transactions	65 states and 240 transactions	9 states and 6 transactions
Realization	8300 lines of code	2000 lines of code	1500 lines of code

Figure 2.35. Complexity of the model

2.4.4. Validation

2.4.4.1. Context

The purpose of validation is to show that the supplier product meets their functional objectives. The separation introduced into the development process between the code and the data makes it possible to set up a validation process for the parametrized system in two stages:

- validation of the generic application (functional aspect);
- validation of the specific application (parametrizing aspect).

First, there is a validation of the generic aspect of the equipment's safety software by functional tests, and second, there is validation of the set of data of the safety equipment being tested. Validation of the data enables the correction of each instantiation to be shown.

2.4.4.2. Validation of the generic software

Within the context of the SAET-METEOR, RATP decided to base its policy of double validation on the concept of a functional test.

Exhaustively testing an application, such as the control/command of a railway system, is impossible. Indeed, the number of possible track configurations is too large. This is why the concept of *generic software* is introduced.

The separation between the software and the data makes it possible to carry out as many configurations as required. The generic software is one of these versions regarded as representative of the use of the software. This configuration makes it possible to test the set of functionalities of the software for a reduced number of configurations.

2.4.4.2.1. Selection of test cases

Each model relating to a function could constitute a specification of functional tests. There are two methods for choosing test sets: either selecting the test input or generating the test input. We are in the step of functional test case selection (entries identification and output acquisition).

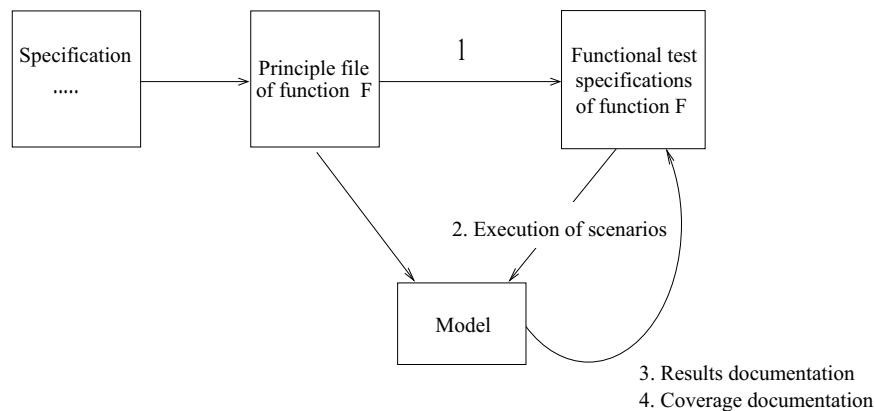


Figure 2.36. Link between the model and the specification tests

Starting from the analysis of the executed requirement specification, a first specification test is carried out. This specification test takes into account the set of functional requirements. As shown in Figure 2.36, the first specification test is implemented in the model to validate its contents. A measurement of coverage of the execution of tests in the model makes it possible to verify that there is no shadow area (dead code, inaccessible code, etc.) in the model.

The execution of each test in the model makes it possible to have the value of observables, but also makes it possible to verify that the unit is correct through property verification.

2.4.4.2.2. Execution of scenarios

The functional tests are carried out on target; this means that each test is carried out on an equipped version of the tested equipment, which is called a “test bay”. The

test bay is a hardware and software set that includes line equipment (ATP-Embedded, ATP-Line, or ATP-Ground) and a simulator of the environment.

As shown in Figure 2.37, starting from test plugs described in the preceding phase, the charge of validation writes test scenarios. The scenarios describe actions in the environment (train displacement, forcing of a physical state of an object, anticipation of an event, introduction of a delay, etc.).

The execution of the test scenario produces a file of results that contains the set of actions carried out, the set of exchanged messages, and a description of the state of the system in each cycle.

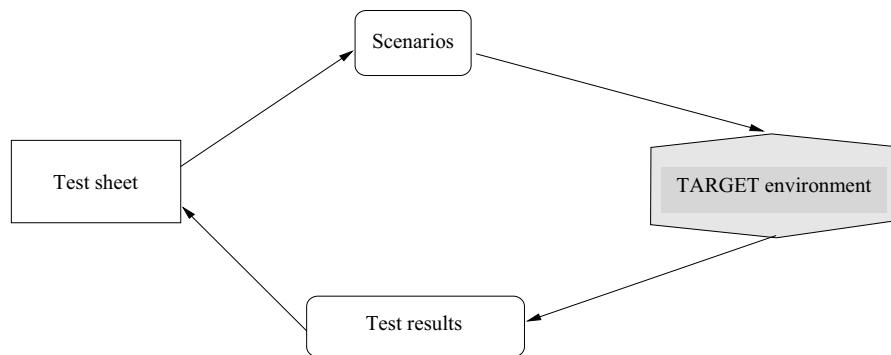


Figure 2.37. Implementation of the target tests

2.4.4.2.3. Analysis of results

There are two stages in the results analysis. In the first stage, the charge of validation must analyze the file of results produced during simulation and research the results anticipated during the development of the specification test. In the second stage, it is possible to use the previously executed model like an oracle test.

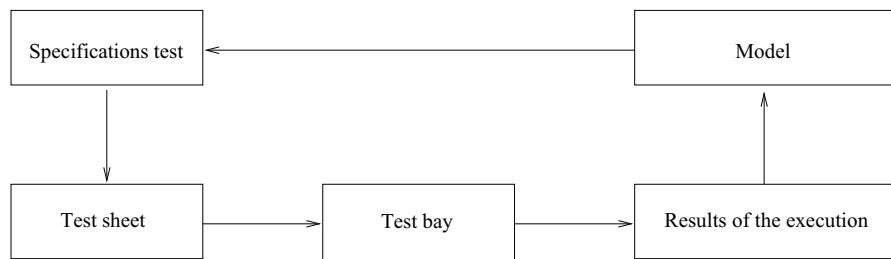


Figure 2.38. Verification and validation chain

Using the model as an oracle test (see Figure 2.38) makes it possible to have an automatic second look. The verification thus carried out is focused on respecting the properties.

As shown in Figure 2.39, the use of the model is based on replacing the functional aspect with a reading of an execution cycle on a test bay. The test plug is then supplemented with an analysis of the results and a judgment (OK or KO).

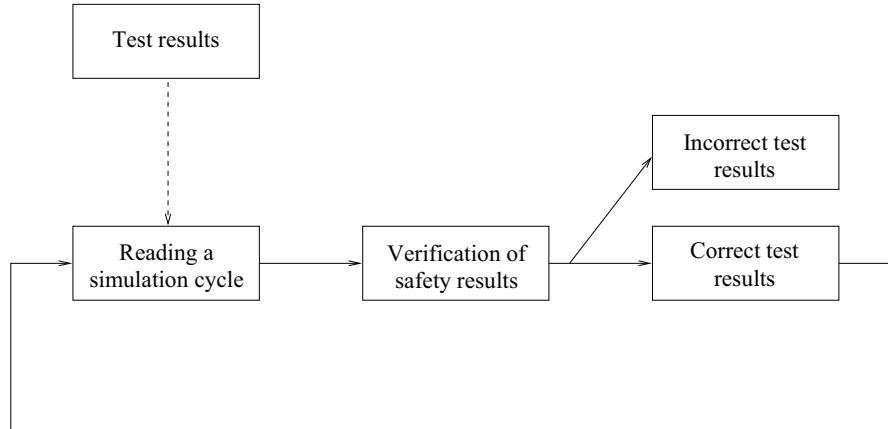


Figure 2.39. Verification of the execution

2.4.4.3. Validation of the data

Validation of the data is carried out in two stages:

- study of the generation process of the data (see Figure 2.16);
- effective validation of the data.

2.4.4.3.1. Traceability of the data

The traceability of the data is done through review activities, which consist of verifying:

- the existence of the data handled by the development process;
- the existence of the data handled by the software;
- the existence and correction of the generation process;
- the traceability of the requirements applied to the data.

2.4.4.3.2. Validation of the data

The control/command systems currently developed are becoming increasingly larger; they introduce the concept of communication and are data consumers.

A common feature of the control/command systems related to protection in the railway sector is the need to design each installation to fulfill the individual requirements of a specific application.

The establishment of a system configured by the data allows the establishment of generic software. There are two types of these *static* data:

- fixed data, known as invariant, which characterizes the environment (topology of the line, characteristic of the trains, etc.). They are known during system generation and define system requirements;
- data changing with the state of the system.

In an *equipped* generation process of the data, there are three strategies to demonstrate data correction:

– either we carry out all the tests making it possible to demonstrate that the generated data is correct. These tests must be executed while a train is circulating on the line. The number of tests will depend on the complexity of the line. This number is in general too important to make this solution acceptable. Especially as these tests are carried out on the real line, they are carried out at the end of the process and any detection of an anomaly can have an impact on the project plans;

– or we seek to demonstrate that the data generation tool is on the safety level required by the application (in general SSIL 3 or SSIL 4¹⁶ in our case). Demonstrating that a given tool is level SSIL 3 or SSIL 4 is an activity whose complexity varies with the amount of data, the complexity of each transformation, and the development process. The demonstration of this technique is dependent on the operating system and the tools used (compiler, etc.) although it relies on elements (certificates, documents, specific tests, etc.) that will have to be updated;

– or we carry out a verification and validation activity for data generation that would be based on the test and/or the formal proof. Considering the amount of data, this stage is based on using a tool.

¹⁶ The safety level of software in the railway sector is called Software SIL. SSIL is enumerated with five values 0, 1, 2, 3 and 4. The level SSIL0 means that the software does not have an impact on the safety of the system, level SSIL2 indicates that there can be impact such as damage and important hardware breakage, and the level SSIL3-4 indicates that there can be an impact on the system and people's lives. The level SSIL1 is not used and levels 3 and 4 are seen as only one level (even in practice).

For the SAET-METEOR, the last solution was selected. Recall that the verification and validation process, set up by RATP, is based on the concept of properties [BOU 00]. To have a homogeneous process, a similar process was set up for validating the data.

There are two strategies for setting up a validation of the data, either a descending validation strategy (see Figure 2.16), or an ascending validation strategy (see Figure 2.40). The descending strategy follows the same process as that of the development. It consists of carrying out a second data generation and in comparing it with that generated by the first chain.

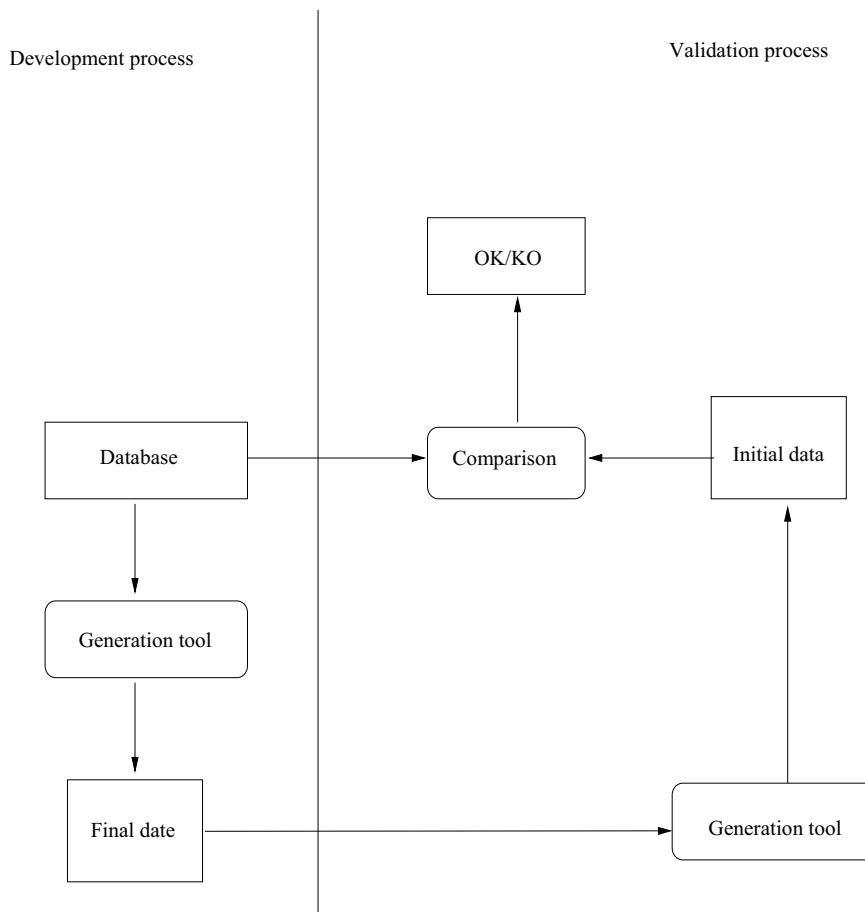


Figure 2.40. Generation process by inversions

This solution is what was followed by the industrial (see section 2.3.3.4.3). The ascending strategy starts from the executable program and consists of decompiling the data that it contains and comparing it with the initial data.

In both cases, there are two tools and two independent development teams. In the descending strategy, there is a potential common mode, which is the specification of the data generation process. In the ascending strategy, this common mode disappears because it is necessary to reverse the generation process.

For the SAET-METEOR, RATP chose to set up an ascending process because it means avoiding the common mode and creating work complementary to the industrial's work.

Remember that the starting point of data validation is the executable program that must be brought into service.

At this level, there are two strategies, either the validation of the data is performed starting from the executable program, which involves decompiling the executable program, or we demonstrate that the data sources were not altered during the generation of the executable program and that they are those found in the application (verification of *checksum*).

Within the context of the SAET-METEOR, it is possible to start with the source files for three reasons:

- the set of sources is managed in configuration;
- RATP carries out its own generation of the executable program and verifies that it conforms to that of the industrials by *checksum* verification;
- the application is carried out on a protected structure entitled *coded processor* [GEO 90], which guarantees the integrity of the program through code verification. These codes were generated starting from the source files, meaning that we can guarantee the correction of the compiler.

Concerning the processor based on safety code, in the event of a problem of inconsistency between the execution in progress and the code, the system is put in the safety state: stopping the trains. The code makes it possible to cover three groups of defects: operators' errors, operand errors and errors induced by handling data irrespective of the date.

RATP developed a specific environment [DEL 99b].

2.4.4.3.3. Inversion of the generation process

The validation of data [BOU 03], realized by RATP, is based on the idea that the process of generation can be reversed and thus shows the exactitude of the data by comparison (see Figure 2.40). As shown in Figure 2.41, generating the parametrized data is a process, that can produce one or more data tables through transformation functions of the topological data.

The transformation functions are injective, but the existence of objects known as virtual (they are not physical objects) makes it possible to introduce, in a table, objects that are not the product of the transformation function. A transformation function can require complementary topological data (a mark, etc.).

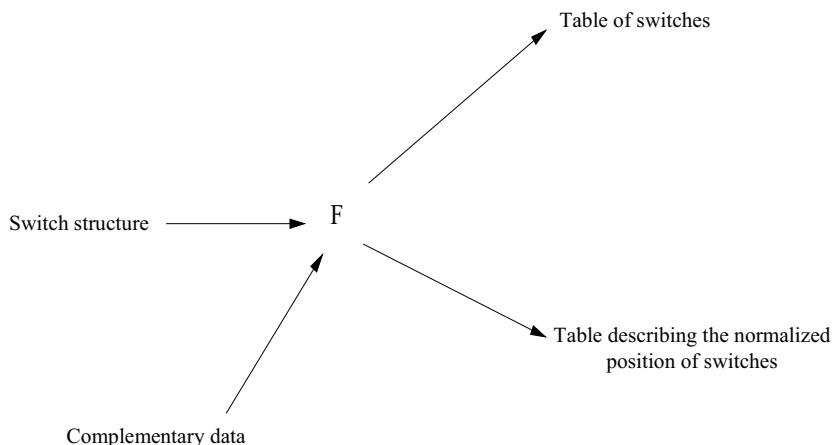


Figure 2.41. Example of transformation

As shown in Figure 2.42, the inversion of the generation process is done for each parametrized datum. The function F_1 makes it possible to initialize the set of switch structures; the function F_2 makes it possible to inform certain structure fields (position, etc.).

The person in charge of the inversion of the function F will have to show that the execution of the functions F_1 and F_2 is consistent with the function F^{-1} . In our example, the function F uses certain complementary data during the transformation process, functions F_1 and F_2 can require complementary data, if they do not exist on the parametrized data level, the opposite transfer functions cannot be completed and it will be necessary to introduce a constraint of completeness C for this parametrized data. It will then have to be shown that the ideal inverse transfer function F^{-1} is

equivalent to the execution of the supplemented transfer functions F_1 and F_2 completed with the verification of C .

With only this, the inversion of the generation process made it possible to identify a large number of errors (process scheduling problem in the tables, inconsistency in the specifications, etc.).

The tool for inversion makes it possible to generate the topological data starting from the parametrized data. The tool of comparison allows the verification of consistency between the initial topological data and the decompiled topological data. This comparison is not just an equality. Indeed, there can be information losses (compensated by the presence of properties of completeness), the decompiled data can contain the same data several times but with incomplete structures.

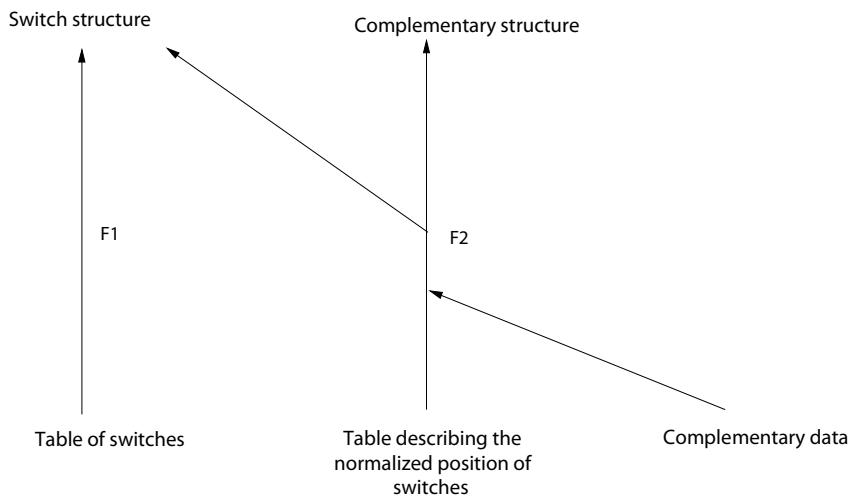


Figure 2.42. Example of inversion

The process of inversion can be compared to the process of writing a program starting from a specification. Demonstrating equivalence between the ideal inverse transfer function F_1 and the execution of the transfer functions F_1 and F_2 completed by verifying C , is carried out by the charge of the specification and is controlled by a second person. The introduction of three types of verification (inverse function, completeness constraint and comparison) makes it possible to guarantee that there cannot be a common mode between the tools (a datum cannot be incorrectly declared correct).

2.4.4.3.4. Control requirements

The process of controlling the requirements associated with the data is carried out with a specific tool [DEL 99b].

Validation by a tool which, on the basis of the topological invariants of the equipment, generates the data primitive by inverse transfer function, is completed by verifying the properties that are either extracted from the specifications, the development process, the safety analyses, or the inversion process.

In a formal development based on the B method, it is important to validate the properties introduced into the basic machines that processed the parametrized data. The validity of the evidence depends on these properties.

The properties related to the data can be properties of *safety* or others and are introduced within the verification tool with a set formal language (created for the occasion). This set language makes it possible to express the properties with the conventional mathematical operators (\forall , \exists , \Rightarrow , \vee , etc.). Verifying a property is carried out by exploring the space of possibilities (topological datum), which is equivalent to a proof by case. The tool for verifying the constraint is based on an algorithm similar to *model-checking*.

This verification tool makes it possible to formally check the data; to date, there are more than one hundred safety properties that have been verified.

As an example, we give two safety properties concerning the data:

- P1: the set of the track circuits forms a track partition (connectivity of the track);
- P2: the set of virtual blocks forms a track partition (connectivity of the track);
- P3: a switch is associated at the most with two routes.

The process described above conforms to the standard [CEN 01].

2.4.5. Assessment of RATP activity

The validation process (see Figure 2.43) following the properties is an external verification process that uses static and dynamic verification.

Static verification, without real activation, was controlled in a model of system behavior (a state machine in the finished extended communicating state), which leads to an analysis of the behavior.

Dynamic verification, which consists of activating the system by providing input values, was carried out as a functional test campaign for equipment in a test bay.

The proposed process of validation could be applied in the same manner to the safety data of each piece of equipment.

As indicated in [BOU 00], verifying the SAET-METEOR resulted in 20 main files that were associated with 23 models and the validation was supported by 30 specification tests that contain more than 5,000 functional tests.

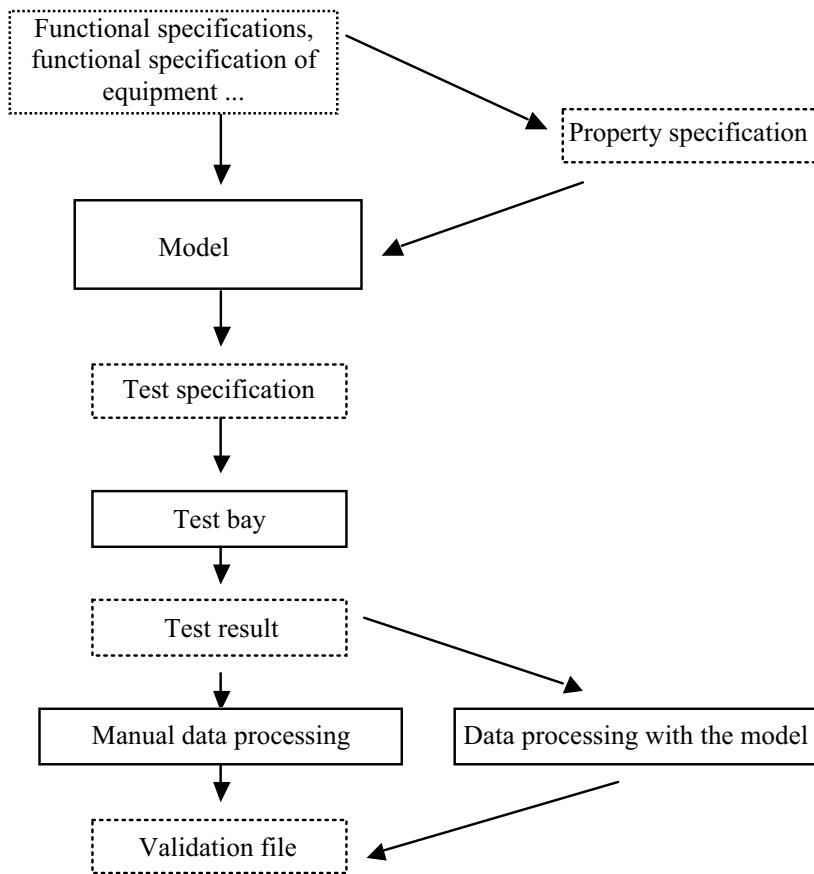


Figure 2.43. Process of validation

These activities made it possible to highlight 400 critical remarks for safety at the specification level and 110 anomalies relating to the set of versions of the safety software, three versions of the three applicative ATP-x.

Of course, all the anomalies relating to safety were corrected before service in the last version of the software.

2.5. Assessment of the global approach

This study enabled us to highlight the feasibility and the effectiveness of a validation process based on the concept of safety properties. The proposed process of validation could be applied in same manner to the safety data of each piece of equipment.

It should be noted that since the startup of the SAET-METEOR in 1998, the safety software has had no problems so that no change to the safety software has been necessary.

The robustness of the process, implemented with the SAET-METEOR, made it possible to reproduce the group of work for the extension that goes from Madeleine station to Saint-Lazare station. This extension has been open to the general public since December 16, 2003. It should be noted that in 2006/2007, RATP inaugurated the extension to Olympiades (in the 13th arrondissement of Paris).

We showed that it is possible to have a formal data validation process. In our case, we set up a validation based on a proof-by-case respecting a certain number of properties.

The work undertaken with the SAET-METEOR was the occasion for RATP to define a verification and validation process formalized through “trade referential”. This trade referential was successfully submitted to COFRAC accreditation¹⁷ in program 152 [COF 00]. As shown in [BOU 02, BOU 03], the AQL laboratory¹⁸ of RATP was the first and to date remains the only laboratory accredited to conduct reliability tests on software (all domains).

The SAET-METEOR project showed the effectiveness of formal methods. It should be noted that the renewal of Line 1 of the Paris metro was developed with a similar process (B method for design and SDL modeling¹⁹ [UIT 94] for RATP verification).

¹⁷ Comité Français d'Accréditation, French committee of accreditation. For more, see www.cofrac.fr.

¹⁸ AQL stands for *Atelier de qualification des logiciels*, a software qualification workshop.

¹⁹ SDL stands for *Specification and Description Language*.

This chapter shows that within the context of parameterized systems, it is necessary to grant as much importance to data validation as to software validation.

With the introduction of data processing in everyday life, the parameterizing aspect of control/command systems should assume greater importance.

2.6. Conclusion

This chapter shows the implication of RATP in the implementation of formal methods.

We note that not just one approach was set up, but a combination of approaches, such as proofs, *model-checking*, and testing, was implemented for the SAET-METEOR.

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
Formal methods including CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM, Z and B	–	R	R	HR	HR
Semi-formal methods	R	R	R	HR	HR
Structured method including for example JSD, MASCOT, SADT, SDL, SSADM and Yourdon.	R	HR	HR	HR	HR

Table 2.2. CENELEC EN 50128 - Table A.2

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
Formal methods including for example CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM, Z and B	–	R	R	HR	HR
Semi-formal methods	R	HR	HR	HR	HR
Structured method including for example JSD, MASCOT, SADT, SDL, SSADM and Yourdon.	R	HR	HR	HR	HR
...

Table 2.3. CENELEC EN 50128 - Table A.4

	SSIL0	SSIL1	SSIL2	SSIL3	SSIL4
Formal Proof	–	R	R	HR	HR
Static analysis	–	HR	HR	HR	HR
...

Table 2.4. CENELEC EN 50128 - Table A.5

The concept of generic software and the definition of the concept of parameterization is significant.

To demonstrate that generic software respects safety requirements is the first part of the answer, but it is necessary to show that the parameterization of generic software preserves the good properties.

The entirety of the work presented in this chapter was carried out between 1994 and 1998, and in 1998 the preliminary version of the standard CENELEC EN 50128 [CEN 01] was published. As Tables 2.2 and 2.3 show²⁰, the standard CENELEC EN 50128 recommends using formal methods for the specification and design phases.

As Table 2.4 shows, the proof was introduced as the verification technique within the standard CENELEC EN 50128.

2.7. Appendix

The goal of this appendix is to describe terms in railway practice so that the reader can understand the case studies presented in this chapter.

2.7.1. Object of the track

2.7.1.1. Track and track circuit (*CdV*)

Trains are assemblies of one or more operational vehicles. By operational vehicles, we mean “ground machine for transporting goods or passengers”, which is able to circulate on a guidance track (the railroad track).

²⁰ HR means “highly recommended” and R just “recommended”. Not implementing a HR technique must be justified and can be refused by the evaluator.

Any track of railroad consists of a set of rails which constitutes a continuous track. The first equipment allowing the physical detection of a train is called the *track circuit*.

A track circuit (noted CdV) is a portion of perfectly defined track (there is a beginning and an end) that allows the presence of at least one train to be detected.

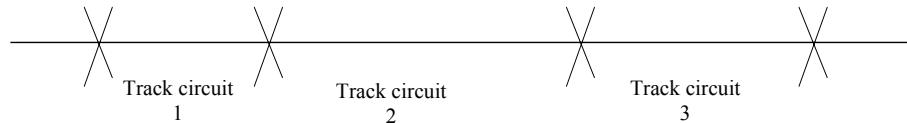


Figure 2.44. Any track is a succession of track circuits

The basic principle of the track circuit is related to the short-circuit (*shunting*)²¹ realized physically by the train. Any railroad track is thus a succession of track circuits (see Figure 2.44).

The track circuit is intrinsically safe equipment, which can be free or occupied. In the event of failure, it will be in the occupied state.

2.7.1.2. Switch and switching

Passing from one track to another is carried out by switch crossing. The switch is a blade.

Positioning the blade and controlling the position are performed through equipment.

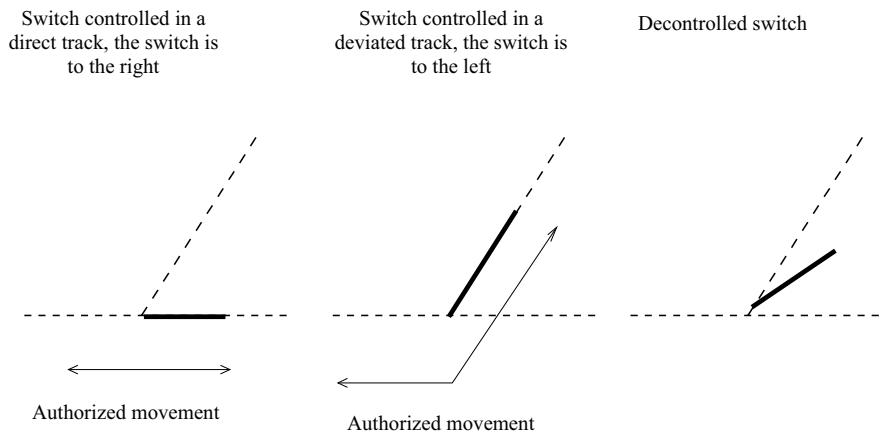
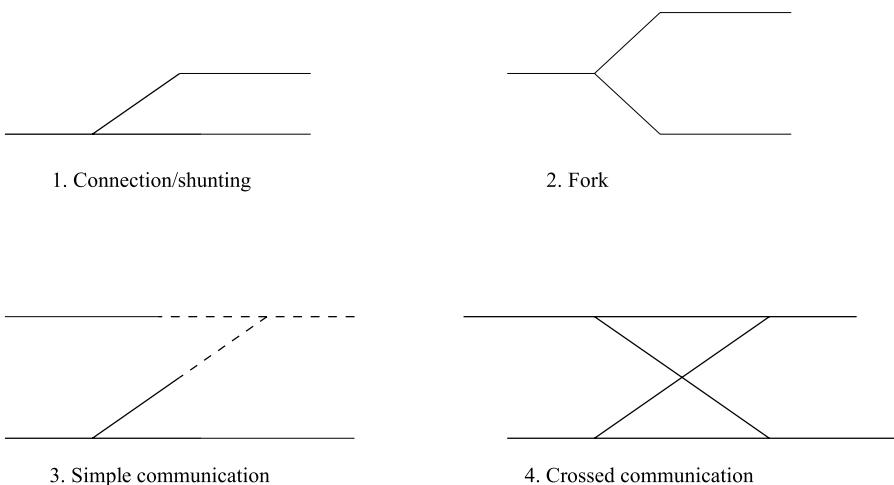
The term shunting includes the switch and the set of elements allowing its control/operation.

As shown in Figure 2.45, a shunt is equipment that makes it possible to go in two directions. The position of the blade is able to take three values: direct, deviated and decontrolled.

In order to be consistent with what was said above, it should be noted that the switch is a track circuit with three extremities.

Figure 2.46 presents the various combinations of shunting that can be implemented to construct a railway line.

²¹ The track circuit is fed energy. When the wheels of the same axle of a train are on the zone delimited by the track circuit, there is a short-circuit and it is said that the train *shunts* the track circuit. The principal function of the track circuit is to detect *shunting*.

**Figure 2.45. Switch****Figure 2.46. Placement of the switches**

It should be noted that we added the crossed half-heart, that is a double switch and thus a track circuit at four ends. The composition of two crossed half-hearts makes it possible to obtain the crossed heart.

With this set of configurations, it is necessary to add more complex constructions like the *bath tub curve* or the *passing track* presented in Figure 2.47.

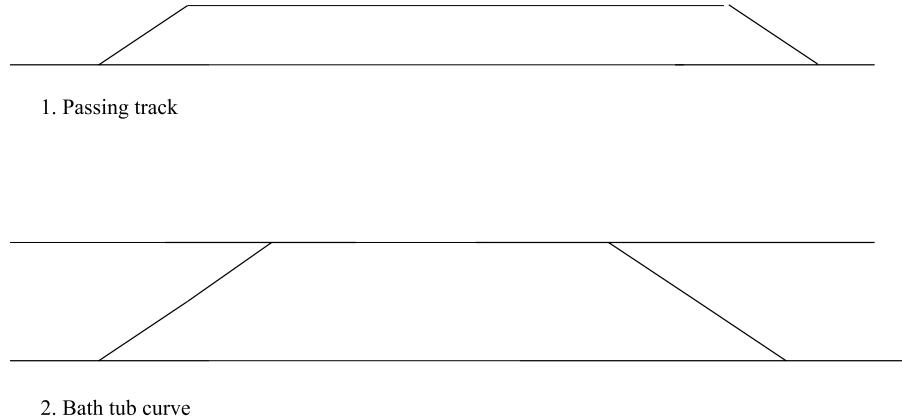


Figure 2.47. Other switch configurations

2.7.1.3. Negative detector (ND)

A negative detector is an optical barrier that allows the presence of a train in a short zone to be detected. There are two types of ND: ND 45° and ND 90°. The negative detector thus allows a specific detection.

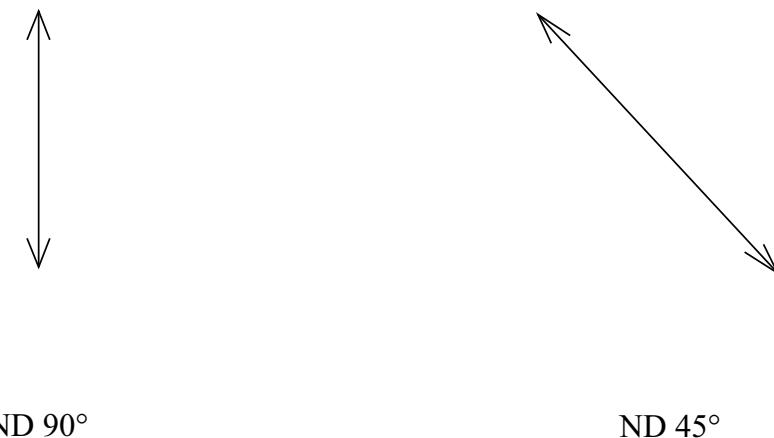


Figure 2.48. Negative detector (ND)

Figure 2.48 shows the representation of the two types of ND, the fundamental difference resides in the width covered by the optical beam. This width is a function of the direction of the optical beam in relation to the axis of the track.

2.7.2. Block logic

A rail-bound system of transport aims to organize train displacements on a given network. The management of these displacements must be guarded against potential accidents such as a collision between trains. Very early, a transport safety principle was defined based on the concept of *signaling*.

As trains are moving bodies of which the stopping distance is proportional to the square of the speed, managing spacing between trains is important.

Spacing between trains makes it possible to define a maximum speed and a limit of displacement.

Figure 2.49 presents these concepts and shows that starting from the limit of displacement, the braking coefficients, and the maximum speed, it is possible to define an envelope that the train's speed must respect.

To automate the spacing process, the track is divided into blocks. A *block* is a portion of track in which only one train is allowed. In the presence of a train the block is called occupied, if not it is called free. The blocks are protected by a stop signal. The stop signal can prohibit entry into the block (closed state) or authorize entry in the block (open state).

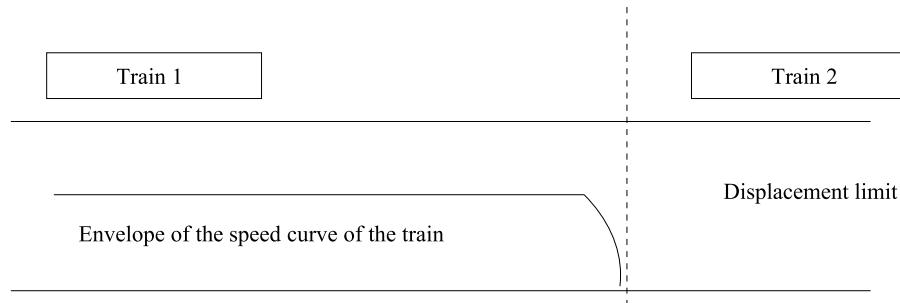


Figure 2.49. Two trains in the course of displacement

As soon as a block is occupied, the associated input signal must be closed. The purpose of signaling is to manage the state of the signals starting from the state of the blocks. As shown in Figure 2.50, it is sufficient to have a free block between two occupied blocks to have a safe system with respect to a collision.

The size of the block depends on the maximum speed of the trains authorized to use it and it thus permits rail traffic to be regulated. That means that a rail network will consist of a collection of blocks of different sizes.

There are downgraded modes where, with authorization, a train is allowed to penetrate an occupied block running at sight. The term *running at sight* indicates that the driver is the only person in charge of the movement of the trains and that he must proceed at a reduced speed.

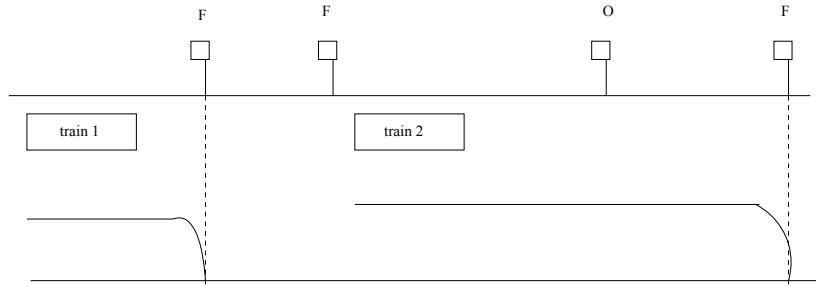


Figure 2.50. Signaling and the block system

Conventionally, the block is associated with a circuit track and the state of the track circuit is used directly to order signaling. The first optimization consists of not having a free block between two trains but of having a restricted light (Figure 2.51).

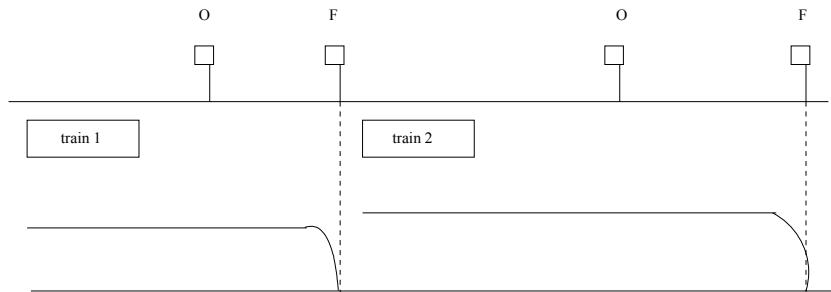


Figure 2.51. Virtual signaling and block system

The concept of a block has been refined (mobile block, deformable mobile block, virtual block, etc.) in order to propose more flexible and better modes of use. Indeed, in the version presented here, the lengths of the blocks can be important.

Concerning lateral indication, recall that controlling a train could be considered blind control since the stopping and braking distances are greater than human vision. Railway signaling thus has several goals: to indicate to the driver the downstream circulation condition, the crossings, and the specific situations. It must allow the drivers to regulate the speed of the train.

2.8. Bibliography

- [ABR 96] ABRIAL J.R., *The B-Book – Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.
- [ANS 83] ANSI. Norme ANSI/MIL-STD-1815A-1983. Langage de programmation Ada, 1983.
- [AUD 89] AUDUREAU E., ENJALBERT P., FARINAS DEL CERRO L., *Logique temporelle*, Masson, Paris, 1989.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, MIT press, Cambridge, 2008.
- [BEH 93] BEHM P., “Application d’une méthode formelle aux logiciels sécuritaires ferroviaires”, *Atelier logiciel Temps Réel, 6^e Journées internationales du Génie Logiciel*, 1993.
- [BEH 96] BEHM P., “Développement formel des logiciels sécuritaires de METEOR”, in HABRIAS H., *1st Conference on the B method, Putting into Practice Methods and Tools for Information System Design*, p. 3-10, IRIN (Institut de recherche en informatique de Nantes), Nantes, November 1996.
- [BEH 97] BEHM P., DESFORGES P., MEIJA F., “Application de la méthode B dans l’industrie ferroviaire”, *ARAGO ’20*, p. 59-88, Masson, Paris, 1997.
- [BIE 98] BIED-CHARRETON B., “Sécurité intrinsèque et sécurité probabiliste dans les transports terrestres”, *Synthèse INRETS no. 31*, November 1998.
- [BOU 00] BOULANGER J.-L., GALLARDO M., “Processus de validation basée sur la notion de propriété”, *Lambda Mu 12*, Montpellier, 27-30 March 2000.
- [BOU 02] BOULANGER J.-L., “AQL: un laboratoire pour la vérification et la validation des logiciels”, *Revue Savoir Faire*, 2002.
- [BOU 03] BOULANGER J.-L., “Validation des données liées à la sécurité”, *Institut de sûreté industrielle, Qualita 2003: 5^e congrès pluridisciplinaire Qualité et Sûreté de fonctionnement*, p. 28-35, Nancy, 18-20 March 2003.
- [BOU 06] BOULANGER J.-L., Expression et validation des propriétés de sécurité logique et physique pour les systèmes informatiques critiques, Thesis, university of Technology of Compiègne, 2006.
- [BOU 07] BOULANGER J.-L., “Etat de l’art de la validation des données dans le domaine ferroviaire”, *Revue REE*, no. 3, p. 25-31, March 2007.
- [BOU 09] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques – exemples concrets*, Hermès Lavoisier, Paris, 2009.
- [CEN 01] CENELEC, NF EN 50128 Applications Ferroviaires. Système de signalisation, de télécommunication et de traitement – Logiciel pour système de commande et de protection ferroviaire, July 2001.

- [CHA 90] CHAPRONT P., RUBAUX J.-P., "Le logiciel SACEM", *Revue générale des chemins de fer*, no. 6, Dunod, Paris, June 1990.
- [CHA 96] CHAUMETTE A.-M., LE FEVRE L., "Système d'automatisation de l'exploitation des trains de la ligne METEOR", *revue REE*, 8 September 1996.
- [COF 00] COFRAC – Programme 152. Evaluation en sûreté de fonctionnement des systèmes logiciels, version 01, December 2000.
- [CON 96] CONAN G., DEVILCHABROLLE M., "La logique de canton de METEOR", *Revue Générale des Chemins de fer*, no. 6, p. 35-36, June 1996.
- [COU 87] COURTIAT J.-P., DEMINBINSKI P., GROZ R., "Estelle, un langage pour les algorithmes distribués et le protocoles", *TSI*, 1987.
- [DEH 95] DEHBONEI B., MEJIA F., "Formal development of safety-critical software systems in railway signalling", in HINCHEY M.G., BOWEN J.P. (eds), *Applications of Formal Methods, Series in Computer Science*, p. 227-252, Prentice Hall International, Upper Saddle River, 1995.
- [DEL 99a] DELEBARRE V., NATKIN S., BOULANGER J.-L., "Validation des spécifications du métro automatique METEOR basée sur un modèle formel", *Revue des Transports et de la Sécurité (RTS)*, vol. 63, 47-62, April-June 1999.
- [DEL 99b] DELEBARRE V., GALLARDO G., JUPPEAUX E., NATKIN S., "Validation des constantes de sécurité du pilote automatique de METEOR", *ICSSEA '99*, Paris, 1999.
- [DES 96] DESFORGES P., BUSTANY F., BEHM P., "Utilisation de la méthode B pour la réalisation des logiciels de sécurité du SAET de METEOR", *Revue Générale des Chemins de fer*, vol. 6, 41-44, June 1996.
- [FOR 89] FORIN P., "Vital coded microprocessor principles and application for various transit systems", *IFAC – Control, Computers, Communications in Transportation*, p.137-142, 1989.
- [FOR 96] FORIN P., "Une nouvelle génération du processeur sécuritaire codé", *Revue Générale des Chemins de fer*, vol. 6, 38-41, June 1996.
- [GAL 99] GALLARDO M., "La validation des logiciels de sécurité de METEOR par la RATP", *Revue Générale des Chemins de fer*, vol. 10, 17-27, October 1999.
- [GEO 90] GEORGES J.-P., "Principes et fonctionnement du Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance (SACEM). Application à la ligne A du RER", *Revue Générale des Chemins de fer*, 6, June 1990.
- [GUI 90] GUIHOT G., HENNEBERT C., "SACEM software validation", *ICSE*, p.186-191, 26-30 March 1990.
- [HEN 94] HENNEBERT C., "Transports ferroviaires: le SACEM et ses dérivés", *ARAGO '15, Informatique tolérante aux fautes*, p.141-149, Masson, Paris, 1994.
- [HOA 69] HOARE C.A.R., "An axiomatic basis for computer programming", *Communication of the ACM*, vol. 12, p. 576-583, October 1969.

- [ISO 89] ISO, Estelle, a formal description technique based on an extended state transition model, Technical Report IS 9074, 1989.
- [ISO 95] ISO/IEC 8652, "Information technology – programming languages – Ada", 1995.
- [JON 90] JONES C.B., *Systematic Software Development using VDM*, Prentice Hall International, Upper Saddle River, Etats-Unis, 1990 (2^e édition).
- [LAG 93] LAGNIER F., HALBWATCHS N., RAYMOND P., "Synchronous observers and the verification of reactive system", *3rd International Conference on Algebraic Methodology and Software Technology AMAST '93*, June 1993.
- [LEC 96] LECOMPTE P., BEAURENT P.J., "Le système d'automatisation de l'exploitation des trains (SAET) de METEOR", *Revue Générale des Chemins de fer*, vol. 6, 31-34, June 1996.
- [LIS 90] LISSANDRE L., *Maîtriser SADT*, Armand Colin, Paris, 1990.
- [MAI 93] MAIRE A., "Présentation du système MAGGALY", *Symposium international sur l'innovation technologique dans les transports guidés, ITIG '93*, Lille, September 1993.
- [MAR 90] MARTIN J., WARTSKI S., GALIVEL C., "Le processeur codé: un nouveau concept appliqué à la sécurité des systèmes de transports", *Revue Générale des Chemins de fer*, vol. 6, 29-35, June 1990.
- [MAT 98] MATRA-RATP, "Naissance d'un Métro. Sur la nouvelle ligne 14, les rames METEOR entrent en scène. Paris découvre son premier métro automatique", *La vie du Rail & des transports*, no.1076, hors-série, October 1998.
- [MOR 90] MORGAN C., *Deriving Programs from Specifications*, Prentice Hall International, Upper Saddle River, 1990.
- [NAT 97] NATKIN S., BOULANGER J.-L., DELEBARRE V., OZELLO P., "Deriving safety properties of critical software from the system risk analysis, application to ground transportation systems", *2nd IEEE High-Assurance Systems Engineering Workshop HASE '97*, Maryland, 11-12 August 1997.
- [SAB 97] SABATHE J.-P., "Premier retour d'expérience sur le noyau de vérification formelle de l'outil asa+", *Génie Logiciel*, 45, September 1997.
- [SPI 89] SPIVEY J.-M., *The Z Notation – a Reference Manual*, Prentice Hall International, Upper Saddle River, 1989.
- [STU 93] STUPARU A.M., BARANOWSKI F., OZELLO P., "Inrets experience of the highly critical software validation of the maggaly subway", *ISSRE '93*, Colorado, November 1993.
- [STU 94] STUPARU A.M., BARANOWSKI F., OZELLO P., "La validation fonctionnelle des logiciels de sécurité du métro automatique Maggaly", *Recherche Transports Sécurité*, no. 42, March 1994.
- [STU 95] STUPARU A.M., "Métro automatique de Lyon, validation complémentaire des logiciels de sécurité", *Journée d'électronique*, 1995.

- [UIT 94] UIT Z-100: Ccitt – specification and description language: SDL, Technical report, UIT, Geneva, 1994.
- [VER 94a] VERILOG, Simulateur/vérificateur asa+ – manuel de référence, Technical report, 1994.
- [VER 94b] VERILOG, Langage LSA+- manuel de référence, December 1994.
- [VER 95] VERILOG, Base et références théoriques de l'environnement asa+, Technical report, 1995.
- [WAË 95] WAËSELYNCK H., BOULANGER J.-L., "The role of testing in the B formal development process", *6th International Symposium on Software Reliability Engineering (ISSRE '95) Toulouse*, p. 205-28, IEEE Computer Society Press, Los Alamitos, CA, 24-27 October 1995.

Chapter 3

The B Method and B Tools

3.1. Introduction

The use of formal methods [ARA 97, HIN 95, MON 00, BOU 11a, BOU 11b, BOU 11c] is increasing, especially in critical applications, such as nuclear power plants, avionics and rail transport. Ensuring maximum safety while operating the application is a significant challenge.

Formal methods present a mathematical framework for the development process, for producing correct software construction with a verifiable development process through validation techniques, such as proofs or explorations of the model.

They must describe specific properties that the computer system needs to possess. Formal methods come from various classes: algebraic specifications (PLUSS or PVS), equational specifications (LUSTRE [HAL 91, ARA 97]), and oriented specification models (B [ABR 96], VDM [JON 90], or Z [SPI 89]).

Contrary to an oriented validation by model exploration (*model-checking* [BAI 08] such as LUSTRE), the B method is based on proving a mathematical lemma (called proof obligation and noted PO), which guarantees the feasibility and coherence of the model (validity of refinement).

With formal methods [ARA 97, BOU 11a, BOU 11b] and in particular, the use of the B method [BOU 12] the software of these safety systems (railway signals and automatic operation) must meet very strict criteria of quality, reliability and robustness.

Chapter written by Jean-Louis BOULANGER.

One of the first applications of formal methods was made *a posteriori* in SACEM [GUI 90]. Projects such as CTDC, KVS, or SAET-METEOR¹ [BEH 93, BEH 96, BOU 06], CdG VAL, and the automation of Line 1 of the Paris metro that is currently in deployment, use the B method throughout the development process (from specifications to code).

3.2. The B method

The B method, developed by Jean-Raymond Abrial [ABR 96], is a targeted formal method *model* like Z [SPI 89] and VDM [JON 90], but it allows an incremental development in the specification up until the code using refinement [MOR 90] in a single formalism, the language of abstract machines. With each stage of the B development, the proof obligations are generated to guarantee the validity of refinement and the consistency of the abstract machine. In the example of Figure 3.1, we introduce the specification of the Hello program and its implementation.

MACHINE	IMPLEMENTATION
HelloWorld	HelloWorld_n
OPERATIONS	REFINES
Hello =	HelloWorld
skip	IMPORTS
END	BASIC_IO
	OPERATIONS
	Hello =
	STRING_WRITE("Hello World")
	END

Figure 3.1. The “Hello” program in B

3.2.1. The concept of abstract machines

The design of a B model is done by composition, decomposition, and refinement of abstract machines. The abstract machine is the basic component of the B method. An abstract machine is an initial component (not a refinement) or the refinement of another component. When an abstract machine cannot be refined, we call implementation.

¹ SAET-METEOR [MAT 98] has equipped Line 14 of the Paris metro since October 1998. The safety critical control-command architecture of the SAET-METEOR (French: *Système d’automatisation de l’exploitation des trains – Metro Est Ouest Rapide*, an automated system for train operation used by the Paris metro system) is presented in Chapter 2 of [BOU 09] and the development process, the V&V and the software safety demonstration is presented in Chapter 2 of [BOU 11a].

The concept of abstract machines is similar to the concept of module and/or the object found in traditional programming languages. The keyword being the encapsulation, the evolution of the state of an abstract machine should only occur through its behavior.

Abstract machines are divided into three levels:

- the *machines* that describe the highest level of specification;
- the *refinements* that combine all the intermediate stages between specification and code;
- the *implementations* that define coding.

The B method defines a single notation called *Abstract Machine Notation* (AMN), which makes it possible to describe these three levels of abstraction. It should be noted that to pass from a high-level machine to implementation, we pass by one or two refinements; this is the *chain of development* for the machine in question.

Starting from the last refinement (implementation), we find a level of detail that makes it possible to use an automatic code generator (C [ISO 99], Ada [ANS 83] and others) to obtain an executable code. The restrictions in AMN on the implementation level allow the construction of a translator.

Abstract machines are made up of three parts:

- the declarative part;
- the composition part;
- the executive part.

In the example in Figure 3.2, the declarative part is in italics, the executive part is in bold, and the composition part is in normal font.

The declarative part describes the state of an abstract machine through variables, constants, sets, and especially properties that should always verify the state of the machine. This part is based on set theory and first order predicates. We can then discuss a *model state*.

Within the framework of machine STACK in Figure 3.2, the state is composed of a variable *stack* sequence type whose elements are *object* and a constant *max_object*, introduced by the parameter setting, which enables parameterization of the maximum number of elements of this stack. The sequence is the second basic type after the sets.

```

MACHINE
  STACK ( max_object )

SEES
  OBJECT

CONSTRAINTS
  max_object ∈ NAT;
VARIABLES
  stack
INVARIANT
  stack ∈ seq(Object) ∧
  size(stack) <= max_object

INITIALISATION
  stack := <>
OPERATIONS
  PUSH (XX) =
    PRE XX ∈ Object ∧ size(stack) < max_object
    THEN stack := stack ← XX END;

  XX ← POP = PRE size(stack) > 0
    THEN XX,stack := last(stack),front(stack) END
END

```

Figure 3.2. Example of an abstract machine

As the example in Figure 3.2 shows, the B language uses a mathematical notation (\in , \wedge , ...) and a computer notation (., &, ...) allowing use of a traditional keyboard.

Figure 3.3 shows an example of a specification using sets; we use a total set PEOPLE that characterizes all the *people* able to exist in the specification. This set must be finite, where the constraint concerns the cardinal.

The set *people* characterizes the existing people and the set *men* characterizes a particular subset of *people*.

In the clause DEFINITIONS, we find that the set WOMEN is the complement of the set man. This clause makes it possible to have definitions in an “inline” sense of programming languages (a code *inline* is expanded with compilation). The clauses of composition (SEES, INCLUDES, IMPORTS and EXTENDS) make it possible to describe the various links between abstract machines, each clause introducing rules of visibility for the status and the operations of the abstract machine concerned.

```

MACHINE
  EXAMPLE
SETS
  PEOPLE
CONSTANTS
  max_people
PROPERTIES
  max_people : NAT
&   card(PEOPLE) = max_people
VARIABLES
  people, men
DEFINITIONS
  WOMENS == people - men
INVARIANT
  people < : PEOPLE
&   men < : people
...
INITIALIZATION
  people, men = {}, {}
...
END

```

Figure 3.3. Example of using sets

In the example of Figure 3.2, we introduce a visibility link (SEES) on the machine OBJECT, this link provides access to the set *Object*.

The executive part contains the initialization and the operations of the abstract machine; it is based on the generalized substitution languages (known as GSL). A subset of the GSL is presented in Figure 3.4. Generalized substitutions are an extension of the work of Dijkstra [DIJ 76] on substitutions.

Generalized substitutions are predicate transformers. $[S] P$ is written as the predicate obtained by applying substitution S to predicate P . In fact, as shown in Figure 3.5, substitutions are defined by their effect on a predicate, known as the *Weakest Precondition* (written as wp).

Complementary rules of calculation are used to handle generalized substitutions, such as distributivity:

$$[S](p \& q) = [S]p \& [S]q \text{ or } [S](\neg p) = \neg([S]p)$$

The behavior of our example (Figure 3.2) is summarized with two operations (*push* and *pop*), which are built around the behavior of the sequences. The operations *last*, *front*, and \leftarrow represent access to the last element, the sequence without the last element, and the concatenation on the right, respectively.

Simple substitution	$x := E$	$[x:=E] R \Leftrightarrow$ substitution of all the free occurrences of x in R by E .
Empty substitution	skip	$[\text{skip}] R \Leftrightarrow R$
Simultaneous substitution	$S \parallel T$	$[S \parallel T] R \Leftrightarrow [S] R_s \dot{\cup} [T] R_t$ with $R=R_s \dot{\cup} R_t$; T and S modify the sets of distinct variable V_s and V_t ; V_s (resp. V_t) is not free in R_t (resp. R_s).
Preconditions	$P S$	$[P S] R \Leftrightarrow P \dot{\cup} [S] R$
Limited choice	$S [] T$	$[S [] T] R \Leftrightarrow [S] R \dot{\cup} [T] R$
Kept choice	$P => S$	$[P => S] R \Leftrightarrow P => [S] R$
Choice not limited	$@ x . S$	$[@ x . S] R \Leftrightarrow "x.[S] R or x is not free in R."$

x describes a variable, E is an expression of set theory, P and R are predicates, S and T are generalized substitutions.

Figure 3.4. A subset of generalized substitutions

Substitution P is associated with the wp $[P]R$ for any R , which is translated by $[P]R \{P\} R$ in Hoare logic.

Example: Let P and R , respectively, be defined by $x := x + 1$ and $x : \text{NATURAL}$
we obtain $x + 1 : \text{NATURAL } \{x := x + 1\} x : \text{NATURAL}$

Figure 3.5. Use of predicate transformations

To know if the substitution of the body of the operation *push* can verify the invariant of machine STACK, we calculate the initial precondition so that substitution “stack := stack \leftarrow XX” satisfies the invariant of the abstract machine.

To ease the construction of abstract machines, a certain number of facilities, known as syntactic sugar, were introduced on the AMN level. All basic substitutions (see Figure 3.4) have a textual form, for example, the preconditioning $P | S$ is written PRE P THEN S END.

We calculate $[stack := stack \leftarrow XX] (stack \in seq(Object) \wedge size(stack) \leq max_object)$
 which gives $[stack := stack \leftarrow XX] (stack \in seq(Object)) \wedge [stack := stack \leftarrow XX] (size(stack) \leq max_object)$
 and is reduced to $(stack \leftarrow XX \in seq(Object)) \wedge (size(stack \leftarrow XX) \leq max_object)$

Figure 3.6. Application of the operation push on the invariant

A certain number of more advanced structures, which are found in the advanced languages, are introduced, but they can be rewritten by combining basic substitutions. Multiple substitution “ $y := E, F$ ” or the traditional conditional structure “IF P THEN S ELSE T END” are examples of actions that are found in the body of operations. They are rewritten in simultaneous substitution, respectively as “ $x := E \parallel y := F$ ” and in limited substitution as “ $P \Rightarrow S [] \text{ not}(P) \Rightarrow T$ ”.

As previously stated, AMN defines a single notation based on GSL, set theory, and first order logic. However, the set of substitutions is not usable on all levels, for example, the substitution ANY xx WHERE P(xx) THEN S END (which signifies that $\forall xx.P(xx) \Rightarrow [S]$) is usable at the abstract levels (machine and refinement) while substitution WHILE B DO S END cannot be accepted at the most abstract level.

The B method is based on encapsulation and the operations must provide the set of behaviors enabling the state of the machine to evolve. However, that is not always possible, as we will see later.

3.2.2. Machines with implementations

3.2.2.1. Refinement

As shown in Figure 3.7, the process of refinement [MOR 90] is normally represented as a succession of independent stages with associated verifications. A component $i+1$ (refinement or implementation) refines a component i (machine or refinement).

The process of refinement starts with the definition of a machine that contains the necessary abstract description. Refinements make it possible to concretize the need and to eliminate the non-deterministic and non-sequential elements. Implementation is a B component that uses a subset of the B language called B0.

The B0 sub-language is a language close to traditional languages (C, Ada, etc.). B0 is thus easily translatable into programming languages.

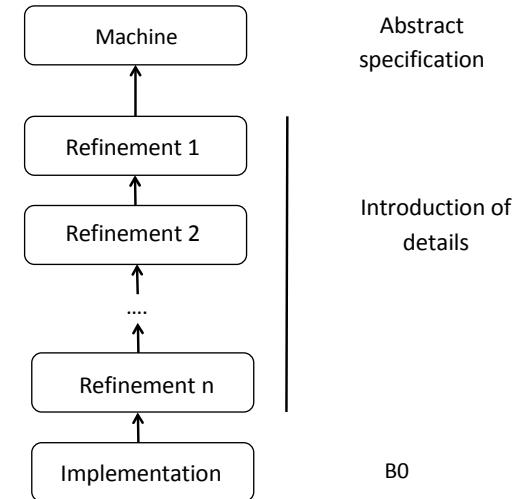
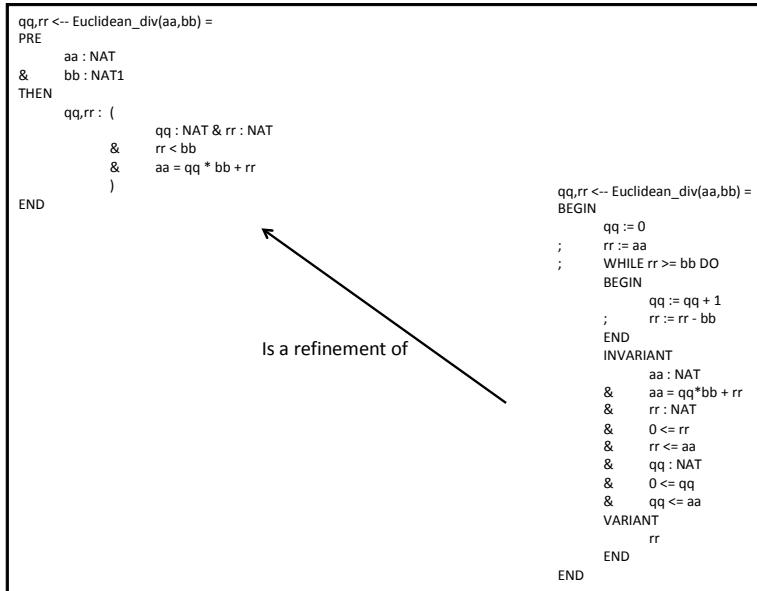
**Figure 3.7.** The refinement process**Figure 3.8.** Example of refinement

Figure 3.8 presents an example of refinement (for more on the refinement process, see [MOR 90]).

The specification indicates that the need is to find two numbers q and r , such that $a = q \cdot b + r$ and $r < b$. We see that the mathematical specification of Euclidean division can be replaced (in the sense of refinement) by an algorithm carrying out the calculation through successive subtractions.

The algorithm proposed uses the instruction WHILE DO END. For this instruction, the B language introduced two specific concepts: INVARIANT and VARIABLE. VARIANT makes it possible to show the end of the loop and INVARIANT makes it possible to verify the proper behavior of the loop.

3.2.2.2. Process

Figure 3.9 introduces the traditional process, which focuses on searching for a defect (“bug”) within the software application. This search for a defect is based on the concept of program execution. This approach seeks to show that the software is correct.

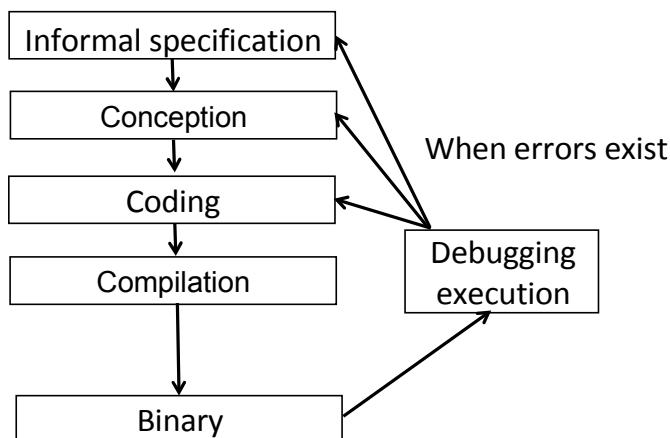
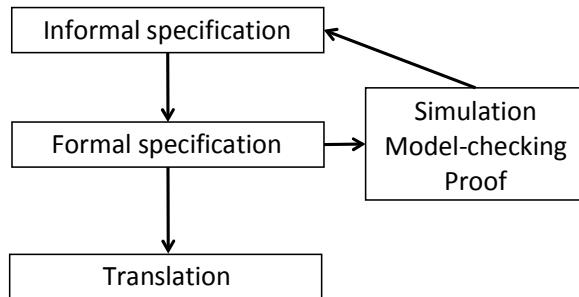


Figure 3.9. Cycle of development with the B method

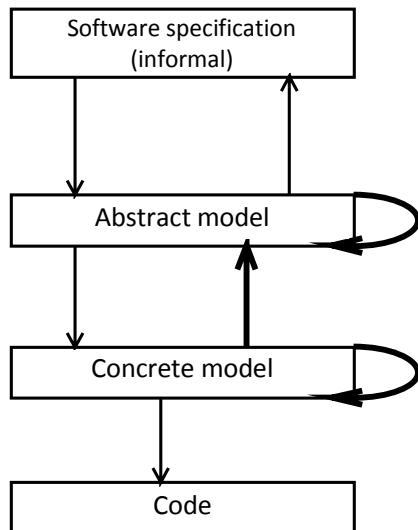
Within the framework of the formal methods, the paradigm is “software is correct by construction”.

The process is different because it focuses on analyzing the need and demonstrating that certain properties are true during all the executions (see Figure 3.9).

As shown in Figure 3.10, the approach consists of writing a model of the problem, simple and abstract, then progressively refining [MOR 90] more concrete and complex elements while proving coherence in the new models created.

**Figure 3.10.** Formal process

Implementation, which is the final stage, is free of abstract types of original data that have become programmable structures such as tables or files. The preconditions of subroutines, simultaneity and non-determinism presented in the abstract model are eliminated, and the desk-check structures, such as sequencing and loops, are introduced.

**Figure 3.11.** Cycle of development with the B method

It is at this stage that automatic conversion codes Ada or C, even assembler codes, for certain tools, are used because it is clear that the translation from a formal specification to a programming language is not easy and direct, since they have opposing thought processes.

The concept of proof (the bold arrows in Figure 3.11) is intimately linked to the B development and the specification is written based on these future obligations. Proof intervenes on all the levels of abstraction. After machine drafting, the proof of its internal coherence is carried out. If it succeeds, the development can then be continued.

The proof is thus made after each level of abstraction, checking its internal coherence and conformity with a higher level of abstraction.

3.3. Verification and validation (V&V)

3.3.1. Internal verification

As shown in Figure 3.12, the external verification (consistency of the specification) and the internal verifications (validity of refinements) are carried out through the proof obligations. The proof obligations form an integral part of the B method.

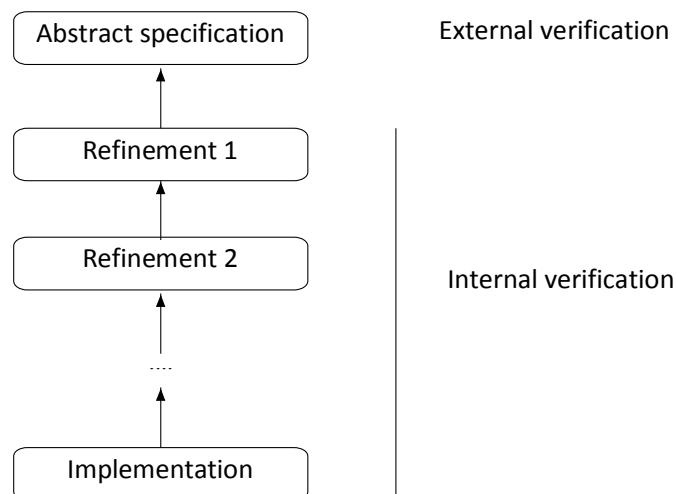


Figure 3.12. Internal and external verification

The B method introduces three phases of verification for all the abstract machines: syntactic analysis; type verification; generation of proof obligations. Within the framework of the implementations, there is a complementary verification dependent on the fact that the B component must respect the B0 language guaranteeing translation into the target language.

3.3.1.1. Syntactic and semantic analysis

Syntactic analysis makes it possible to check that the abstract machine is correctly built so as to ensure that it complies with the syntactic rules of AMN. One of the syntactic rules thus checked is related to the restriction of the use of certain substitutions. For example, simultaneous substitution, noted $S \parallel P$, cannot be used in an implementation.

The type checking makes it possible to detect faults related to:

- the objects not declared or badly declared (incompatible type, etc.);
- to the expression not able to typed;
- the inconsistencies between various definition and used of an operation;
- the violation of the visibilities rules introduced by the composition clauses;
- etc.

Any expression can be typed because the set theory used in AMN is a simplification of classical theory.

3.3.1.2. Generating the proof obligations (PO)

As we have presented, at each stage of development, there is a phase for generating the proof obligations. POs, are generated automatically by the tool.

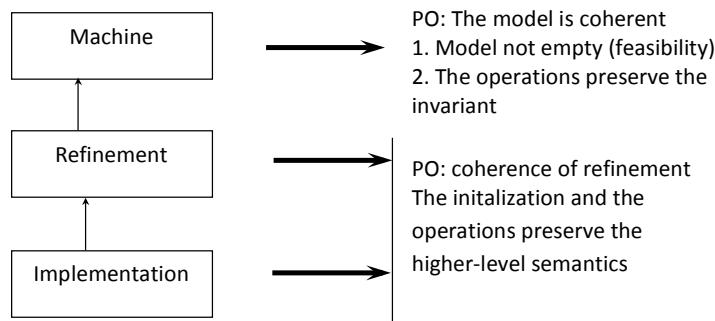


Figure 3.13. Objectives of the proof obligations

For a high-level abstract, machine-generated POs guarantee mathematical consistency. For refinements and implementations, POs guarantee the validity of refinement compared to the higher level machine in the chain of development. In general, the complexity of POs depends on the level of abstraction used (the more concrete it is, the more complex it is) and on the structure of the application in terms of links between machines.

All POs have information describing the context of the machine in their assumptions. The context contains a set of constraints defining the formal parameters and the properties relating to the sets and the constants of the abstract machine. It also takes into account that all the used sets are finished, not empty, and that all their elements are distinct. The context of an abstract machine is noted as $\langle \text{Context} \rangle$. The clauses of composition can add information to the context, the invariant, and initialization in agreement with the rules of visibility.

Verification of mathematical consistency introduces two types of PO. Initially one must show that the model is not empty; this is done by showing that initialization creates the invariant:

$$\langle \text{Context} \rangle \Rightarrow [\text{Initialization}] \text{ Invariant} \quad [3.1]$$

The second time, it should be proven that each operation of the abstract machine preserves the invariant. In the case of an operation OPE defined by a substitution S under the precondition Q is OPE = PRE Q then S END, an OP of the following form is created:

$$\langle \text{Context} \rangle \wedge \text{Invariant} \wedge Q \Rightarrow [S] \text{ Invariant} \quad [3.2]$$

Like the example, the result in Figure 3.6 can be used again; it shows that:

$$\begin{aligned} & (\text{stack} \in \text{seq(Object)} \wedge \text{size(stack)} \leq \text{max_object}) \wedge \\ & (\text{XX} \in \text{Object} \wedge \text{size(stack)} < \text{max_object}) \\ \Rightarrow & \text{stack} \leftarrow \text{XX} \in \text{seq(Object)} \wedge \text{XX} \in \text{Object} \wedge \text{size(stack} \leftarrow \text{XX}) \leq \text{max_object} \end{aligned}$$

To show that an abstract machine is a refinement of a higher level machine, relative to a chain of development, it is necessary for us to show that its initialization and operations preserve the semantics of their more abstract versions. The POs generated for the nth refinement are described by equation [3.3] for initialization and by equation [3.4] for each operation:

$$\langle \text{Context} \rangle \Rightarrow [\text{Init}_n] \rightarrow [\text{Init}_{n-1}] \rightarrow I_n \quad [3.3]$$

$$\langle \text{Context} \rangle \wedge I_1 \wedge I_2 \wedge \dots \wedge I_n \wedge Q_i \Rightarrow Q_n \wedge [[u_n := u_{n-1}] S_n] \rightarrow [S_{n-1}] \rightarrow (I_n \wedge u_n = u_{n-1}) \quad [3.4]$$

with Init_i , I_i , Q_i , S_i , u_i that are, respectively, initialization, the invariant, the precondition of the operation, the action of the operation, the formal parameter of the operation of the ith refinement.

We notice that there is no proof obligation concerning the feasibility of the predicates introduced into the invariant, the preconditions of the operations,

the constraints concerning the formal parameters, and the properties of the sets and constants. In fact, the feasibility is introduced by constructing the other evidence or the specific introduction of PO where necessary.

Verifying the existence of a variable satisfying the invariant is indirectly introduced by POs [3.1] and [3.3]. The existence of formal parameters that validate the precondition of an operation is carried out by POs [3.2] and [3.4]. It's necessary when an abstract machine include (clause INCLUDES) or import (clause IMPORTS) another abstract machine. All the abstract machine including or importing a parametrized machine should instantiate all the formals parameters, and the PO [3.5] is generated to guarantee that the actual parameters validate the constraints:

$$\langle \text{Context} \rangle \Rightarrow [\text{Formal_parameter} := \text{Actual_parameter}] (\text{A} \wedge \text{Constraints}) \quad [3.5]$$

with A as a predicate that indicates that the sets passed in parameter are finished and not empty, and with the predicate *Constraints* being associated with the clause of the same name.

The valuation of the sets and constants can be carried out at the highest level or at implementation (sets and constants submitted). The PO [3.6] guarantees that, on the implementation level, the properties are true for the values given to these objects:

$$\langle \text{Subset_of_Context} \rangle \Rightarrow [\text{DeferredSET}, \text{DeferredVAR} := \text{Values}] \text{ Properties} \quad [3.6]$$

With *Properties*, which is the predicate associated with the clause PROPERTIES, $\langle \text{Subset_of_Context} \rangle$, which concerns the part of the context that is used to create the *Values*.

Using constructive proof implies that certain verifications are carried out on the implementation level. In the case of parameterized machines, these verifications are reported until inclusion (machine or refinement) or importation (implementation) in a machine belonging to another chain of development.

We saw that we can use the structure WHILE on the final realizations level. As we are in a proof environment, a certain number of complementary POs are generated to guarantee that WHILE (see the example of Figure 3.8) and its termination are executed correctly.

To finish, we should note by analyzing POs that the number of predicates grow with the number of levels of refinement and with the number of links introduced by the clauses of compositions.

The proof obligations (POs) represented an aim to achieve under some assumptions. These are established starting from the preconditions and the definitions contained in the refined component, but also using the information read in the viewed or imported machines.

3.4. B tools

3.4.1. General principles

Any formal method must be based on precise syntax and semantics, and this holds good for the B method, too. See [ABR 96]. There is even a complete description of the various phases of internal validation of the abstract machines (syntactic analysis, typing, generating POs). This makes it possible to have a very precise view of what should be a workbench.

Figure 3.14 presents the framework of a workbench to use the B language.

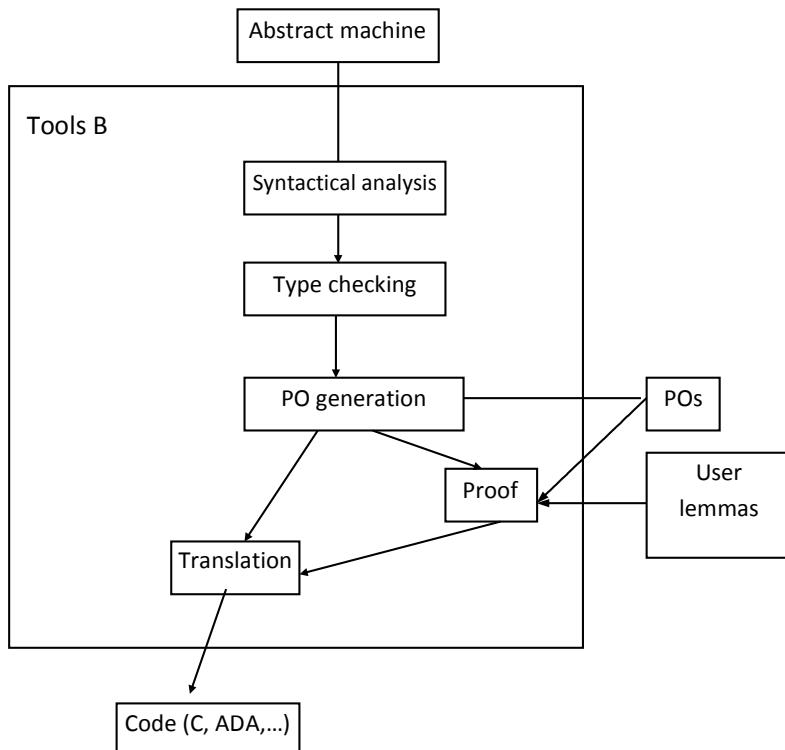


Figure 3.14. A tool for the B method

The arrows represent the activation order of the various phases, while the dotted lines represent access to input files: an abstract machine, PO, lemma users; or to output files: generated code, PO. The tool must take account the dependences introduced by the composition clauses (SEES, INCLUDES, etc.).

Contrary to other formal methods, there are two tools that completely cover the cycle of development in B (specification, refinement, code generation, and generation of the PO and assistance to the proof).

They are Atelier B and B-Toolkit marketed respectively by the company CLEARSY² and the company B-Core (UK) Ltd³.

3.4.2. *Code generation*

Atelier B in the commercial version places the code generators Ada and C, which were used in high-level safety projects (SSIL3-SSIL4⁴ in the sense of the standard CENELEC IN 50128 [CEN 01]) at your disposal.

Concerning the SAET-METEOR (see Chapter 2 of [BOU 11c]), the translation of the B model [ABR 96] in safe Ada language [ANS 83] is carried out automatically by the tool. The safety aspect is obtained through a subset of the code Ada and through the use of a processor based on safe code (PSC, see Chapter 2 of [BOU 09] and Chapter 1 of [BOU 11a]), which makes it possible to guarantee safe execution.

The formal methods have the following interests concerning code generation:

- the code produced conforms to the formal specifications, since it is used to make the proof. It is then possible to consider suppressing unitary and integration tests;
- the code obtained is coherent (not a useless variable, no typing problem, no piece of ineffective code, not an unvisited test, no infinite loop, no side effect, etc.), which makes it possible to detect a great number of traditional defects very early;
- the approach is rigorous and high quality.

3.4.3. *Prover*

The automatic prover applies its rules based on each obligation, rewriting the goal and the assumptions until they coincide. The major disadvantage of these rewritings

2 To know more about CLEARSY and AtelierB, visit www.clearsy.com.

3 In the past, the B-Toolkit was distributed by the company B-Core Ltd (United Kingdom).

4 The safety integrity level for a software is noted SSIL. The standard [CEN 01, CEN 11] identified five level from 0 (the low level of safety) to 4 (the highest level).

is that the workbench does not preserve a record of the previous goal, which sometimes involves a failure that should not take place.

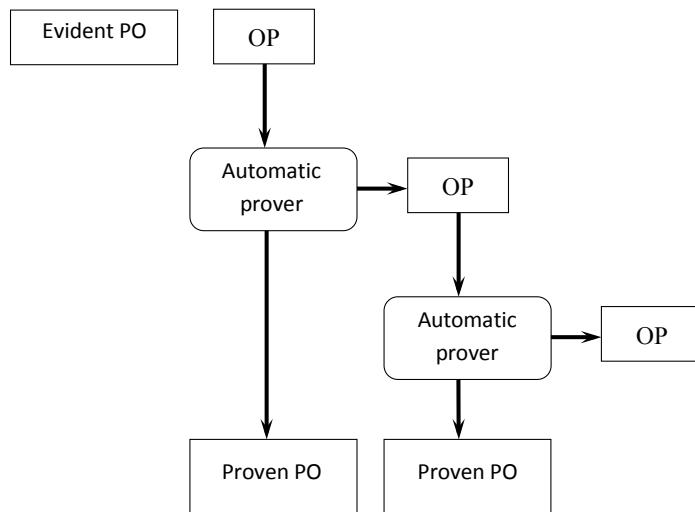


Figure 3.15. Implementation of the proof

In addition, a certain number of supplementary rules are available, but they are applied only if the user requires it when it launches the interactive proof option.

The prover can fail, in the demonstration of the proof obligations, for two reasons: either the proof obligation is false (sought goal: $100 < 50$), which indicates a design error; or the obligation is right, but the rule base is insufficient to be able to establish it.

In the latter case, it is sometimes necessary to add rules. The user has several possibilities.

First of all, the user can call upon the rules defined in the workbench. These rules are not observed systematically because they slow down the resolution time considerably. If the addition of these rules is not sufficient, the user can then write new rules.

Great care must be taken in drafting these rules because they cannot be validated, so it is possible to write a rule that seeks to solve a false obligation.

Besides, in most cases, the addition of the tactics defined in the workbench suffices for proving the set of obligations.

3.4.4. Atelier B

Atelier B is marketed by the company CLEARSY⁵. Atelier B⁶ since the version 4.0 is available free of charge.

Atelier B uses an IDE (*Integrated Development Environment*) as shown in Figure 3.16. This IDE allows us to create B projects and components (machine, refinement and implementation).

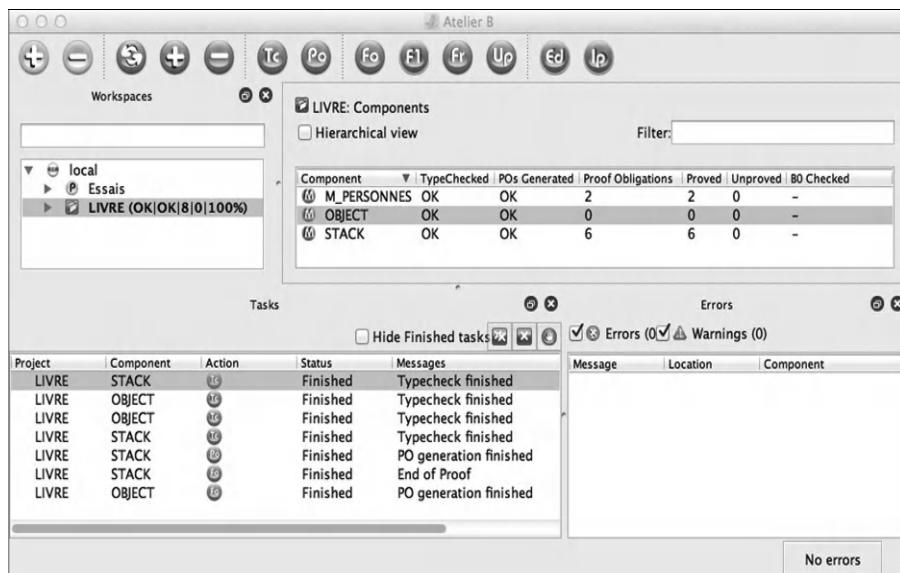


Figure 3.16. Atelier B

The *workspace* section lists the projects in progress. If a project is selected (for example BOOK), the list of components and the verification assessments appear: verification of the correct type, PO generation, numbers to prove PO, numbers of POs not proven, number of proven POs, and verification of B0 compatibility.

AtelierB uses a component editor as seen in Figure 3.17. This editor assists access to B models by offering a line between the mathematical symbols and the textual symbols in the right-hand portion.

⁵ To know more about CLEARSY and AtelierB, visit www.clearsy.com.

⁶ Visit www.atelierb.eu to retrieve AtelierB and associated information.

This editor carries out verifications (syntactic, *typechecking*, etc.) on the component during access.

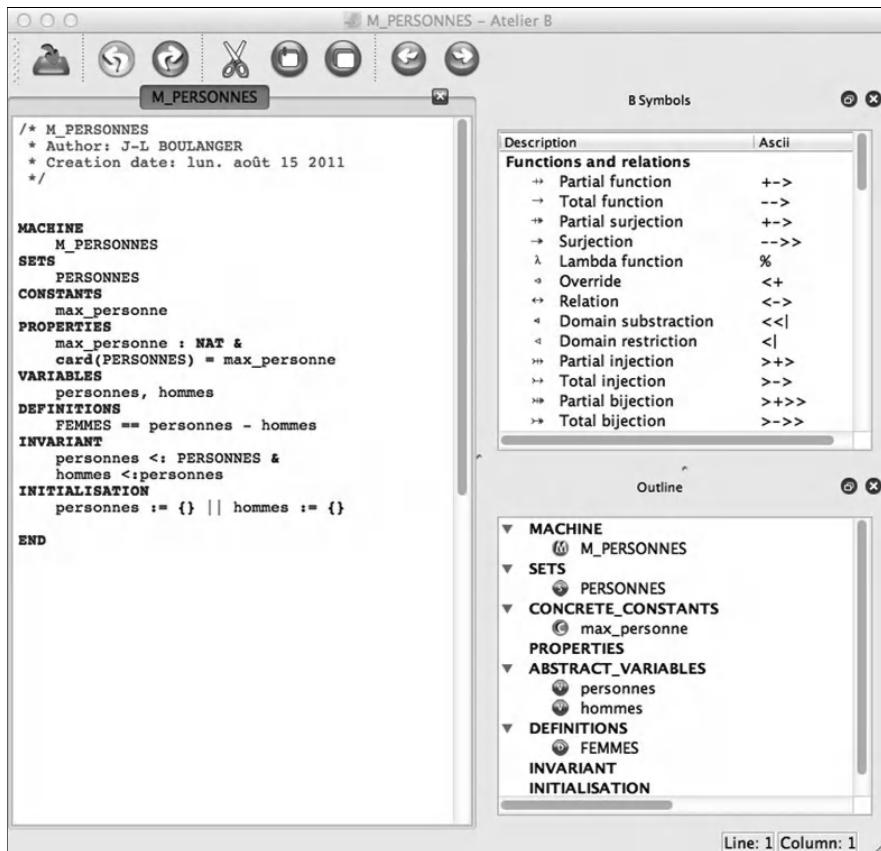


Figure 3.17. The component editor

Figure 3.18 presents the interface of the interactive prover. For a given component, once this interface is launched, the proof obligations can be manipulated. In the lower portion of the interface, the verification of the component **M_PERSONNES** is linked to the proof of two PO that are related to the initialization.

The PO1 is seen as *proven* and consists of showing that the empty set is included in the set of PEOPLE, which is an obvious proof.

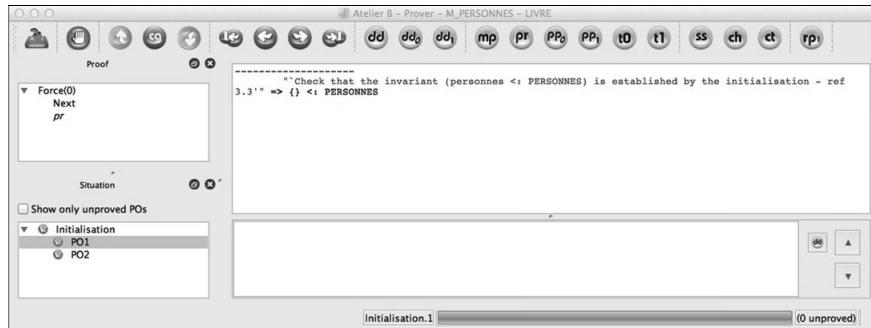


Figure 3.18. Interactive prover

3.5. Methodology

The development of an application in B is not directed code but directed toward the proof. This is why any development methodology of the application in B must be directed toward the proof. In fact, it is necessary that the generated POs are the smallest possible and more or less numerous, but it is also necessary that they are provable.

3.5.1. Layered development

Figure 3.12 represents the traditional development process using a formal method. We therefore begin by writing the system specification to be achieved and then execute successive refinements until we obtain the code.

To validate the specification, there is a single external verification at the level of specification and internal verifications with each refinement. The establishment of such a development process in B introduces, at the highest level of abstraction, an important network of links (SEES, INCLUDES, PROMOTES and EXTEND) between the machines (composition portion of the machines). Moreover, this quickly reveals variables and invariants characterizing those in high-level machines, which weighs down the predicates brought into play in the POs and thus making them more complicated.

B development must leave the highest level of possible abstraction and progressively introduce the details of implementation in the form of operation and data refinements. We then obtain a layered development (see Figure 3.19).

In fact, in a layered development [WAE 95], it passes from an abstract stage to a concrete stage that involves the appearance of new abstract stages. The concept of a chain of development and decomposition based on the services then appears.

A chain of development combines all the stages of the specification until a given model implementation (symbolized by the dotted rectangle). The structuring of the layered model is done through SEES and IMPORTS links, the SEES link (symbolized by a horizontal arrow) allows MACHINE and REFINEMENT to access the data structures, while the link IMPORTS (symbolized by a downward arrow) allows access to the operations during the final realization.

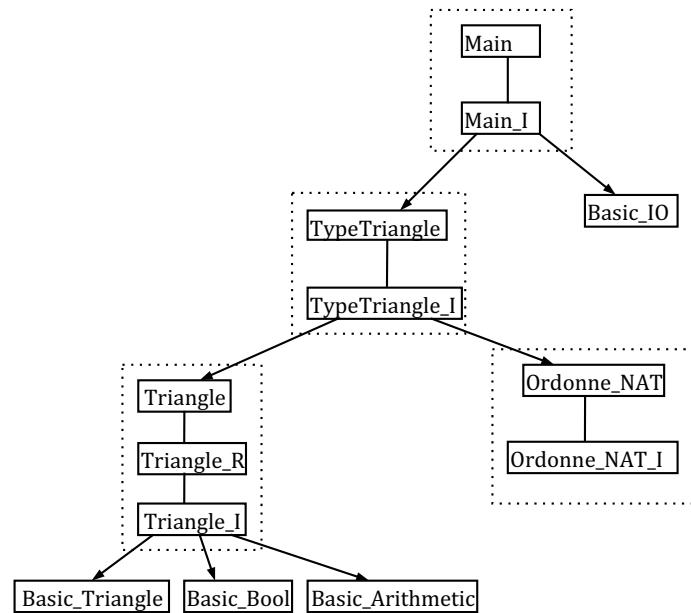


Figure 3.19. Layered development

The sheets of the graph thus obtained are basic machines that are already proven and that are realized in different target languages.

This type of development results in a model rather than a problem specification. As an immediate consequence, an external validation is found, which should be fragmented on the set of high-level abstract machines.

The layered development has three advantages:

- the first is related to the fact that to prove a chain of development, it suffices to use the high-level abstract machines that are used in the composition clause. This is very close to the concept of a module or a package found in languages such as Ada;
- the number of links between abstract machines is restricted, which decreases the complexity of POs and we limit ourselves to the bonds SEES and IMPORTS;

– the graph obtained is a non-transitive acyclic graph, which limits the impact of the modification of a high-level abstract machine. The modification of a refinement or an implementation does not have an impact outside of its chain of development.

3.5.2. Link between the project structure and the POs

In section 3.3.2.2, we presented the POs generated by the B method, and we remarked that the number of links between machines introduces a complexity in terms of the predicate placed in the PO. In fact, the links of composition are from two categories: links at the abstract level (machine and refinement); links at the concrete level (implementation).

Another aspect of the role of the decomposition relates to algorithmic refinement. If we start from a specification that is a bit abstract, the complexity of the POs increases very quickly. Like F. Meija pointed out in his articles, choosing the operation's structures of refinement is important. In the case of a conditional structure (IF THEN ELSE END or CASE OFF END), we should not try to make a situation disappear.

3.5.3. Cycle of development of a B project

Introducing the proof obligation within the development offers an important alternative to testing the program that is usually practiced during development. Figure 3.20 represents the cycle of development of an application in B; note that there is no compilation and thus no execution before the final phase.

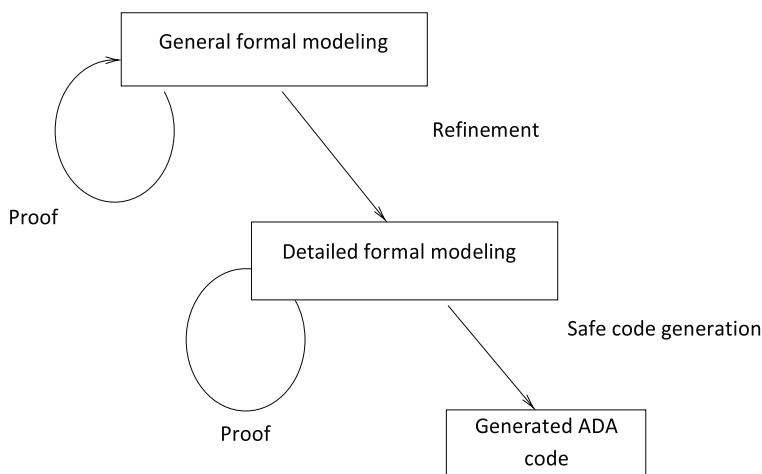


Figure 3.20. Cycle of B development

In fact, as we do not use an effective coordinator that evaluates the behavior of the B model under development, we cannot carry out the external verification of the model.

The presence of the formal proof of conformity of the code developed in relation to the specifications makes it possible to noticeably reduce the test phases of the cycle in V, which is schematized in Figure 3.21.

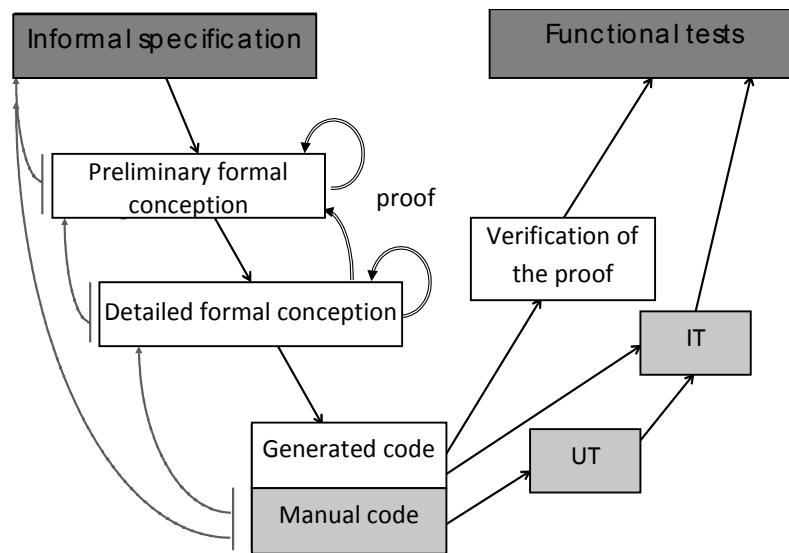


Figure 3.21. Cycle of B development

The placement of the B method in the SAET-METEOR (see Chapter 3 of [BOU 06]) allowed us to show that it was possible to replace test activities (UT and IT) with proof activities. Note: if it is required to create manual code (low-level function, link with components developed outside of B, etc.), it will be necessary to model these elements within B and to show that the component developed manually (or re-used) respects this specification.

This verification is shown by the dotted arrow that leads the manual code toward the higher levels and by the need to at least carry out unitary tests on these portions of code. It will then be necessary to carry out a phase of integration tests to show that the manual code and the generated code interact correctly.

Figure 3.21 shows an abstract specification; this specification will be taken partially into account by the B model and will be the input to execute the functional

tests. This shows that high-quality software was used. The phases of traceability are identified by the dotted arrows; it is necessary to show that all the needs in each stage were taken into account.

In the Figure 3.21, the proofs are indicated by double arrows and can require to add some lemmas during the interactive proofs. If we add some lemmas it is necessary to add a verification step for demonstrating that the proof and the added lemmas are correct.

However, without restarting the debate about the use of formal methods within the framework of software safety, [BEH 96] says that the unit and integration tests are redundant with the complete proof and the safe code generation. [WAE 95] recommends preserving all the test phases, in the expectation of returning to the methodology and validation of B tools; this discussion is always valid if the development is executed in a context outside a processor based on safe code (for PSC⁷ structure, see Chapter 2 of [BOU 09] and Chapter 1 of [BOU 11b]).

It is necessary to point out that creating and validating a B model does not guarantee that the code generator, the production tools of executable programs (compiler, linker, etc.), the installation tools, the management tools for configuration, material structure targets, etc. do not transform the execution and therefore nullify the evidence.

This has already been mentioned in various chapters of this book. It is necessary to build a qualification folder of the tools used. But constructing a qualification folder for a code generator and/or a tool of proof is not an easy task.

3.6. Feedback

3.6.1. Some figures

Table 3.1 provides information about the complexity of the B developments carried out in the railway industry.

It does not describe all the railway applications that were executed with the B method, but it makes it possible to realize the complexity of the developments carried out with the B method.

⁷ Processor based on safe code is given the acronym PSC because in French it is known as *processeur sécuritaire codé*.

System Name	Line of B code	Line of generated code	Language	Number of proof obligations
CDTC	5,000	3,000	Ada	700
KVB	60,000	22,000	Ada	10,000
KVB-SN	9,000	6,000	Ada	2,750
KVS	22,000	16,000	Ada	6,000
SACEM-simplified	3,500	2,500	Modula 2	550
SAET-METEOR	115,000	90,000	Ada	27,800
Eurocoder	10,000	4,500	Ada	4,200
CdG-VAL	PADS 256,653	186,440	Ada	62,056
	UCA 65,722	50,085	Ada	12,811

Table 3.1. Example of the complexity of a B model [BOU 06]

3.6.2. Some users

3.6.2.1. Focus on the situation

The B method was initially implemented in the railway industry. Contrary to SCADE (see Chapter 2), which is based on an equational language, the B method describes non-interruptible sequential programs.

One of the prominent features is the capacity to describe complex algorithms, knowing that the second prominent feature of the B method is making only one notation available that goes from specification to establishment.

The railway equipment responsible for managing the line occupation (called ground equipment) is modeled with difficult equations. It is necessary to describe algorithms that will handle complex structures describing line topology (see Appendix 2.7 of [BOU 11c]) and the characteristics of the trains and equipment.

The principal users of the B method are ALSTOM Transport System and SIEMENS Mobility, but AREVA, within the framework of a CBTC project, is beginning to implement the B method as well.

Note that there were several tests of implementing the B method in the automobile and space industries.

3.6.2.2. SAET-METEOR

The management of Line 14 of the Paris metro is carried out by SAET-METEOR and, more particularly, by the automatic piloting subsystem that is distributed along the line and in the equipped trains (see Chapter 3 of [BOU 11c], Figure 3.2).

The control system has complete control of the trains operating automatically (acceleration, emergency stop, etc.) On the other hand, the trains in manual operation are autonomous and can override a transmitted stop order through signaling. The line is managed by an *autopilot line* (AP-Line) and is cut out in the automation section [LEC 96].

The autopilot comprises three types of software developed with the B method and a level of SSIL3-SSIL4 with the standard CENELEC EN 50128 [CEN 01].

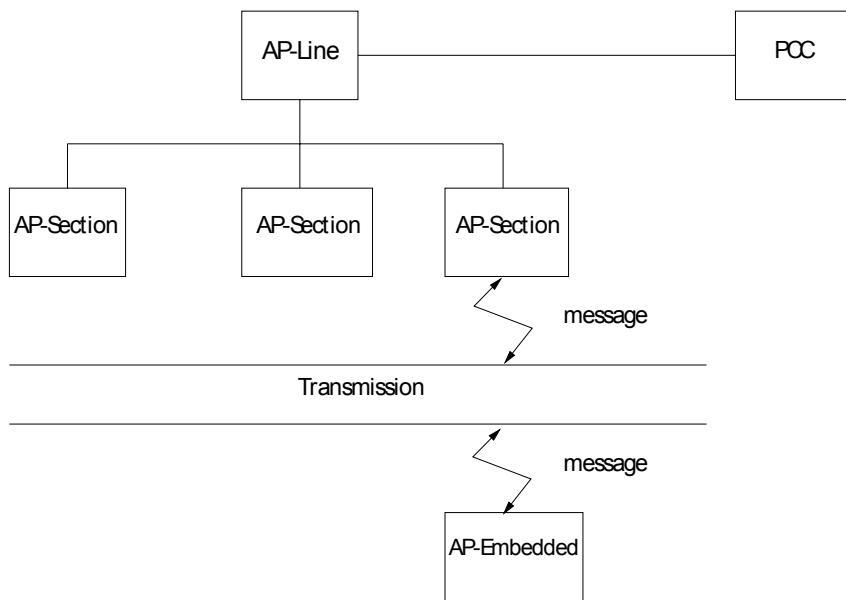


Figure 3.22. Decomposition of the SAET-METEOR autopilot

Since the startup of the SAET-METEOR in 1998, the safety software has had no problems. Therefore, no evolution of the safety software was necessary.

The robustness of the process, implemented within the SAET-METEOR's framework, made it possible to reproduce the group of work within the framework of the extension that goes from Madeleine station to Saint-Lazare station. This extension has been open to the general public since December 16, 2003. It should be noted that on June 26, 2007, the RATP inaugurated the extension to Olympiades (in the 13th arrondissement).

3.6.2.3. CdG VAL

The CdG VAL (Figure 3.23) is based on the VAL technology⁸ by Siemens⁹ and combines a traditional VAL (purely electronic) and a VAL controlled by a two-way autopilot and an alarm control unit (ACU).

The CdG VAL is composed of two lines (Line 1 and LISA) that have been in service since April 2007. One of the characteristics of the line is that it functions 24/7.



Figure 3.23. The CdG VAL at the platform¹⁰

Two-way autopilot (PADS) and ACU (Alarm Control Unit) were developed with the B method and a SSIL3-SSIL4 level within the standard CENELEC EN 50128 [CEN 01].

⁸ The VAL, the first entirely automatic driver-less metro in the world, inaugurated in Lille in 1983, circulates in the cities of Taipei, Toulouse (Lines 1 and 2), Rennes, and Turin. The VAL also equips Chicago O'Hare and Paris-Orly airports.

⁹ To learn more, see www.siemens.com.

¹⁰ Photograph by Jean-Louis Boulanger.

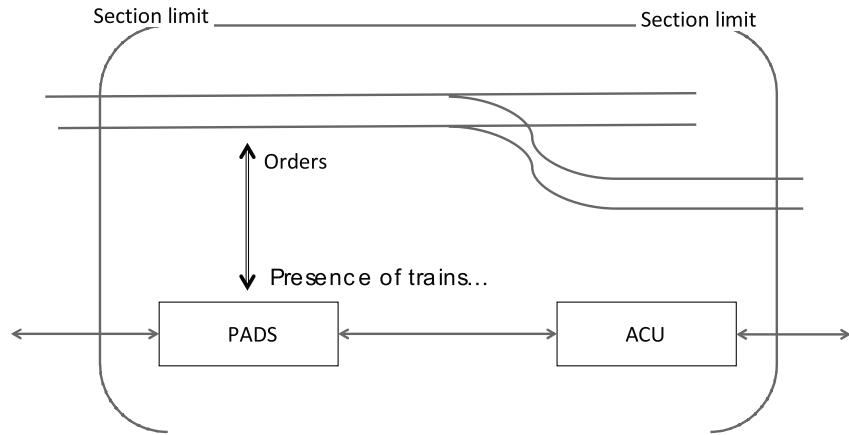


Figure 3.24. The structure of CdG VAL

3.6.2.4. Eurobalise coder

Like the third example, we discuss equipment in the campaign that establishes the link between a central station and the signals (the element that gives orders to the trains).

Within the framework of this application, there is only one formally developed software application with the B method [ABR 96] and one treatment unit. The software application was created to a SSIL3-SSIL4 level with the standard CENELEC EN 50128 [CEN 01].

While the B method guarantees (through mathematical evidence) that the software is correct with respect to the property, this does not cover the code generator, the chain of generation of the executable program (compiler, linker, etc.), and means of loading.

As shown in Figure 3.25, within the framework of this application, there are two code generators and two chains of generation for the achievable programs (two compilers). This makes it possible to have two different versions of the executable program.

It is thus possible to show that the address tables (variables, constants, functions, parameters, etc.) of the two executable programs are indeed different. Loading each version of the application is done in different memory capacities.

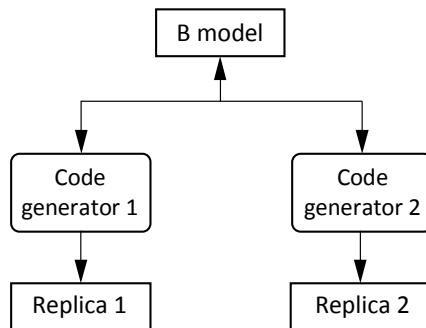


Figure 3.25. Diversification

3.7. Conclusion

Owing to space constraints, we could only demonstrate the guiding principles of the B method in this chapter. A formal method is based on syntax, semantics, and a principle of verification.

Within the framework of the B method, the proof is the principle of verification. The specifications thus obtained are free of the most traditional defects (bugs), and through the principle of specification refinement, it is possible to build algorithms and to show that these algorithms verify the specification. The fundamental difference with the traditional development is that, with the formal methods, software construction can be verified and validated.

The test therefore becomes an obsolete *activity*, which is good news when it comes to cost. But this is true only if a certain number of assumptions are checked, like the demonstration that the chain of generation of an executable program (code generator, compiler, etc.) does not introduce a defect and that it is possible to have confidence in the verification tools (proof obligation generator, prover, etc.). A following book [BOU 12] will be devoted to the B method and will present various examples of implementation.

3.8. Bibliography

[ABR 96] ABRIAL J.R., *The B-Book – Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.

[ANS 83] ANSI, nForme ANSI/MIL-STD-1815A-1983, Langage de programmation Ada, 1983.

- [ARA 97] ARAGO, “Applications des méthodes formelles au logiciel”, *Observatoire français des techniques avancées* (OFTA), vol. 20, Masson, Paris, June 1997.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, The MIT Press, Cambridge, MA, 2008.
- [BEH 93] BEHM P., “Application d’une méthode formelle aux logiciels sécuritaires ferroviaires”, *Atelier Logiciel Temps Réel, 6^e Journées Internationales du Génie Logiciel*, 1993.
- [BEH 96] BEHM P., “Développement formel des logiciels sécuritaires de METEOR”, in HABRIAS H. (ed.) *Proceedings of 1st Conference on the B Method, Putting into Practice Methods and Tools for Information System Design*, p. 3-10, IRIN Institut de recherche en informatique, Nantes, November 1996.
- [BOU 06] BOULANGER J.-L., Expression et validation des propriétés de sécurité logique et physique pour les systèmes informatiques critiques, thesis, University of Technology of Compiègne, 2006.
- [BOU 09] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques – exemples concrets*, Hermès Lavoisier, Paris, 2009.
- [BOU 11a] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques industrielles*, Hermès Lavoisier, Paris, 2011.
- [BOU 11b] BOULANGER J.-L. (ed.), *Utilisation industrielles des techniques formelles – interprétation abstraite*, Hermès Lavoisier, Paris, 2011.
- [BOU 11c] BOULANGER J.-L. (ed.), *Techniques industrielles de modélisation formelle pour le transport*, Hermès Lavoisier, Paris, 2011.
- [BOU 12] BOULANGER J.-L. (ed.), *Mise en œuvre de la méthode B*, Hermès Lavoisier, Paris, forthcoming 2012.
- [CEN 01] CENELEC EN 50128, Railway Applications – Communications, Signalling and Processing Systems – Software for Railway Control and Protection Systems, May 2001.
- [CEN 11] CENELEC 50128, Railway Applications – Communications, Signalling and Processing Systems – Software for Railway Control and Protection Systems, July 2011.
- [DIJ 76] DIJKSTRA E.W., *A Discipline of Programming*, Prentice Hall, Upper Saddle River, 1976.
- [GUI 90] GUIHOT G., HENNEBERT C., “SACEM software validation”, *Proceeding 12th IEEE-ACM International Conference on Software Engineering*, March 1990.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous dataflow programming language Lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305-1320, September 1991.
- [HIN 95] HINCHEY M.G., BOWEN J.P. (ed.), “Applications of formal methods”, *International Series in Computer Science*, Prentice Hall, Upper Saddle River, 1995.
- [ISO 99] ISO/IEC 9899:1999, Programming languages – C, 1999.

- [JON 90] JONES C.B., *Systematic Software Development Using VDM*, Prentice Hall International, 1990 (2th edition).
- [LEC 96] LECOMpte P., BEAURENT P.-J., “Le système d’automatisation de l’exploitation des trains (SAET) de METEOR”, *Revue Générale des Chemins de fer*, vol. 6, p. 31-34, June 1996.
- [MAT 98] MATRA-RATP, “Naissance d’un Métro. Sur la nouvelle ligne 14, les rames METEOR entrent en scène. PARIS découvre son premier métro automatique”, *La vie du Rail & des Transports*, hors-série no. 1076, October 1998.
- [MON 00] MONIN J.F., *Introduction aux méthodes formelles*, Hermès Lavoisier, Paris, 2000.
- [MOR 90] MORGAN C., *Deriving Programs from Specifications*, Prentice Hall International, 1990.
- [SPI 89] SPIVEY J.M., *The Z Notation – A Reference Manual*, Prentice Hall International, 1989.
- [WAE 95] WAESELYNCK H., BOULANGER J.-L., “The role of testing in the B formal development process”, *IISRE '95*, Toulouse, 25-27 October 1995.

Chapter 4

Model-Based Design Using Simulink – Modeling, Code Generation, Verification, and Validation

Modeling and simulation techniques, which have long been well established in many engineering domains, have recently gained importance in the development of embedded software as well. Models of embedded software enable (i) capture of functionality without the cognitive load of the accidental complexity introduced by programming, and (ii) capture of the temporal behavior of the embedded computation potentially in combination with the dynamics of the embedding physical world. Executable models further enable the design and refinement of conceptual models by simulating the behavior of the functionality to be realized (open/closed loop control, monitoring), possibly in concert with simulation of the physical environment. Once we arrive at a model with sufficient detail, an implementation of the functionality to be realized can be automatically produced using code generation. In addition to providing insight, executable models also enable verification and validation as the design progresses. Moreover, automatic code generation technologies further support verification and validation of the implementation.

4.1. Introduction

The increasing utilization of *embedded software* in application domains such as automotive, railway, aerospace and industrial automation has resulted in a staggering

Chapter written by Mirko CONRAD and Pieter J. MOSTERMAN.

complexity that has proven difficult to manage with conventional design approaches. Because of its capability to address the corresponding software complexity and productivity challenges, *Model-Based Design* [HEL 05, CON 05, CON 06, NIC 09, MAT 12] is quickly becoming the preferred software engineering paradigm for the development of embedded system software components across application domains. For example, practitioners report productivity gains of 4 to 10 fold when using graphical modeling techniques [HEL 05, KAM 07]

With the advent of Model-Based Design, a powerful artifact has been introduced into the software lifecycle between specification and the code: the *model*. In a typical Model-Based Design workflow, an initial *executable graphical model* represents the software component under development by including pertinent design detail while omitting the accidental complexity of the software implementation. The model is then refined and augmented until it is sufficiently detailed to serve as the blueprint for the final implementation through automatic *code generation*.

For a model to enable focus on a pertinent detail, it is crucial to support formalisms that are effective and efficient in describing as well as solving the design problem at hand. Operating in this problem space is in stark contrast to formalisms that are specific to the solution space of an implementation. In embedded software, the interaction with physics mandates time be a first class citizen in describing the problem space. Moreover, a declarative representation enables deference of implementation choices by only requiring specification of “what” the functionality should achieve as opposed to “how” it ought to do so. Furthermore, support for hierarchy is essential to manage complexity.

These *desiderata* are the characteristics of time-based block diagrams such as those supported by *Simulink*[®]. Time-based block diagrams consist of a set of blocks that are connected as the vertices of a graph. The edges of the graph capture input/output relations between the blocks, while the blocks represent either an elemental dynamic system or a hierarchical decomposition into another block diagram. At the leaves of the hierarchy are elemental dynamic systems only, which may be of a discrete time, continuous time, discrete event, or algebraic nature. Specifically, for discrete time blocks, a sophisticated clock calculus provides powerful support in the embedded systems’ problem space.

While a declarative approach can be very effective, some functionality may be better modeled in an imperative form. In particular, explicit state transition behavior has a long history of utility in modeling of widely varying systems. Therefore, it is essential that imperative modeling is supported as well, for example, by means of timed automata [ALU 94] or hierarchical state transition diagrams [HAR 87]. In Simulink, imperative modeling is supported in its breadth by (i) *Stateflow*[®] blocks

for hierarchical state transition diagrams with conjunctive and disjunctive state decomposition as well as by (ii) *MATLAB® function blocks* for imperative procedures written in an array-based language.

Design typically includes a creative element that derives from insight, which in the case of dynamic systems is to a large extent about insight into system behavior. In addition, the semantics of the formalisms used to represent a design can be made intuitive and unambiguous by studying the behavior of various syntactic elements in isolation. For example, time-based block diagrams may support a Gain block as a syntactic element, where the output of the Gain block is its input multiplied by a factor. The semantics of such a block in the case of multi-dimensional input may differ based on preference and executing the Gain operation with different input data provides an understanding of how the block behaves.

Additional value may be derived from studying behavior of a design, which highlights the importance of having executable models (i.e. models for which behavior can be generated). Execution of multi-formalism models such as those including both declarative elements as well as imperative elements in addition to discrete-time, continuous-time and discrete-event semantics requires complex execution engines. For example, discrete-time behavior may best be executed based on a static schedule, continuous-time behavior may rely on variable-step numerical integration, and discrete-event behavior may best be generated based on an event calendar [MOS 07]. Unifying all the various execution paradigms is a challenging endeavor that provides commensurate multiplier value as it unlocks the benefits of executable models to design at a system level.

As the semantics of the different modeling formalism are defined in a computational sense, an executable model can be transformed into various different representations while preserving its behavior. In embedded software design, a tremendous value proposition is the transformation of a model by automatically generating corresponding C/C++ code in various forms such as: (i) for accelerated execution of a model, (ii) for software-in-the-loop simulation, (iii) for deployed code, etc. A further value proposition derives from model transformation to a form that is amenable to static analysis methods such as model checking [AND 02] and abstract interpretation [FER 04]. The formal operations such as property proving and test generation that this enables would be difficult if not impossible for humans to achieve for models of a complexity that can be scaled by modern desktop computing power.

Finally, because of the computational representations throughout, all the various artifacts and operations can be interlinked with automated traceability, for example between code and model. As a result, design reviews, code reviews, requirements change propagation, etc. can all be further supported by information technology.

This chapter first presents Model-Based Design in more detail. It then presents an overview of the MathWorks products to support Model-Based Design. Next, the concrete case of the design of a throttle control system is provided. This system is then used to discuss verification and validation technologies in detail. Finally, conclusions about Model-Based Design in industry are presented.

4.2. Embedded software development using Model-Based Design

The complexity and scale of embedded software has dramatically increased over the past decades. The now prolific use of embedded computing power has enabled new and enhanced functionalities that were impossible or unviable before. Due to the ability to address software complexity and productivity challenges, modeling with formalisms that support both imperative and declarative notions as well as modularization and hierarchy is used extensively within the scope of Model-Based Design of embedded software in application areas such as automotive, railway, aerospace and industrial machinery [RAU 03, CON 05]. Typically, both an executable model of the control or monitoring software (*functional model*) and a model of the surrounding system (*plant model*) and its environment (*environment model*) are created early in the system lifecycle and are simulated together. This enables modeling of even complex systems with a high degree of detail at an acceptable calculation speed and to simulate their behavior closely to the modeled physics. During the course of development, the plant/environment model is gradually replaced by the physical system and environment. The functional model then serves as a blueprint for the implementation of embedded software on the control unit through code generation.

One characteristic of the Model-Based Design is that the functional model not only specifies the required function, but also provides design information and finally even serves as the basis of the implementation by means of code generation. In other words, such a functional model offers specification aspects as well as design and implementation aspects. In practice, these different aspects are reflected in an evolution of the functional model from an early specification model via a design model to an implementation model and finally its automatic transformation into C or C++ code (*model elaboration*). In comparison with traditional software development that has a clear separation of phases, in Model-Based Design a seamless integration of specification, design and implementation phases can be noted. Moreover, the same graphical modeling notation is used during the consecutive development phases. Such a seamless use of models facilitates highly consistent and efficient development.

Model-Based Design in industry relies heavily on automatic *code generation* technology. The need to obtain code for different use cases such as rapid prototyping, software-in-the-loop (*SIL*) simulation, processor-in-the-loop (*PIL*) simulation, or hardware-in-the-loop (*HIL*) testing and to be cost effective requires

flexible code generation capabilities from executable models and the automation of this code generation.

In various application domains, Model-Based Design has become established as the standard paradigm for the development of control/monitoring software. Significant advantages can be found in the consistency of the modeling notations and tools used, in the gains in efficiency that stem from using code generation [STU 05], and because testing can be started at model level [CON 04]. According to user reports, there are increases in efficiency of 20–50% for model-based development with code generation in comparison with traditional software development. There is also a rapid increase in the maturity level of developed functions [CON 06].

Modeling is frequently done with commercial modeling and simulation packages. The Simulink® modeling environment supports the different technologies that comprise successful Model-Based Design. It is being extensively used in different application domains. According to [HEL 05], for example 50% of the behavioral models for automotive controls are designed using this environment. Owing to its popularity, the Simulink environment is used in this chapter to illustrate Model-Based Design and code generation.

Simulink^{®1} is the platform environment for simulation and Model-Based Design. It supports multi-domain simulation and Model-Based Design for dynamic and embedded systems. Simulink provides an interactive graphical environment and a customizable set of block libraries to design, simulate, implement and test a variety of time-varying systems, including communications, controls, signal processing, video processing and image processing. Add-on capabilities extend Simulink software to multiple modeling domains and also provide tools for design, implementation and verification and validation tasks.

Graphic editors permit an intuitive development and description of system or software components, their connections, and interfaces. Simulink supports the description of both discrete- and continuous-time models in a graphical *block-diagram* language. Simulink models can further include state-transition diagrams, flow charts, or truth tables provided by *Stateflow*^{®2}, as well as imperative code written in *MATLAB*^{®3}. A model consists of interconnected function blocks. The function blocks describe the algorithmic or logical components of the model. Directed connections between the blocks represent the control and data communicated between the blocks by defining the output of one block as the input of another. Components within the block diagram hierarchy can aggregate other components or be elemental [JER 00, MER 00]. An important aspect of the

1 The MathWorks, Inc. Simulink® – www.mathworks.com/products/simulink.

2 The MathWorks, Inc. Stateflow® – www.mathworks.com/products/stateflow.

3 The MathWorks, Inc. MATLAB® Function Blocks – www.mathworks.com/help/toolbox/simulink/ug/f6-106261.html.

Simulink environment is that it provides extensive capabilities for simulation, which attribute executable semantics to Simulink models.

In addition, *Simulink Coder*^{TM4} facilitates C and C++ code generation from Simulink diagrams, Stateflow charts and MATLAB functions. The generated source code can be used for real-time and non-real-time applications, including simulation acceleration, rapid prototyping and HIL testing. *Embedded Coder*^{TM5} allows the generation of C and C++ code optimized for embedded systems. It extends Simulink Coder by providing configuration options and advanced optimization to generate code for on-target rapid prototyping boards and microprocessors used in mass production. Embedded Coder furthermore improves code efficiency and facilitates integration with legacy code.

4.3. Case study – an electronic throttle control system

This section illustrates modeling and automatic code generation with Simulink based on a simplified electronic throttle control system as found in modern automobiles.

4.3.1. System overview

Electronic throttle controls (ETC) [GRI 02] are replacing the mechanical link between the accelerator pedal and the throttle valve by using electronic signals. A typical electronic throttle consists of a throttle body with an electric motor, a throttle position sensor and an electronic control unit (ECU). The controller calculates the required throttle position based on the information from the accelerator pedal and other systems (such as cruise control) and sends the appropriate control signals to the motor to drive the throttle to the required position⁶. The algorithm contains a closed-loop control algorithm part as well as supervisory logic.

4.3.2. Simulink® model

Figure 4.1 shows the top level of the functional model describing the electronic throttle control system [MOS 06]. The system is structured into a Throttle Position Controller *subsystem* containing the feedback controller and a Throttle Sequencer subsystem containing the supervisory logic and triggering the execution of the Throttle Position Controller. The feedback controller consists of Simulink blocks, whereas the logic part is modeled by a Stateflow block.

4 The MathWorks, Inc. Simulink CoderTM – www.mathworks.com/products/simulink-coder.

5 The MathWorks, Inc. Embedded CoderTM – www.mathworks.com/products/embedded-coder.

6 The Clemson University Vehicular Electronics Laboratory. An experimental demonstration of electronic throttle control – www.cvel.clemson.edu/auto/systems/throttle_control_demo.html.

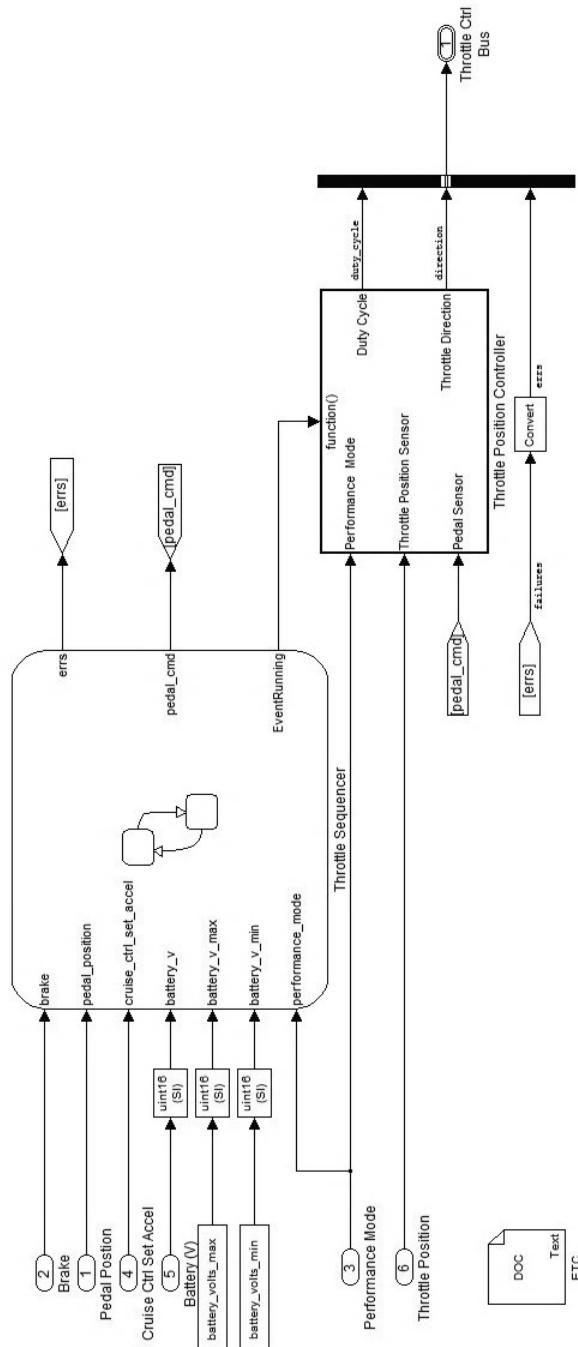


Figure 4.1. Electronic throttle control model—top level

Input to and output from the system is represented by numbered *ports* shown on the left- and right-hand side of the model, respectively. The input interface comprises brake and gas pedal positions (inports Brake and Pedal Position), the cruise control lever mode (inport Cruise Ctrl Set Accel), the battery voltage (inport Battery (V)), the performance mode (inport Performance Mode), and the throttle position sensor value (inport Throttle Position). The system calculates the duty_cycle signal, which determines the force with which the throttle is moved (outport duty_cycle), the throttle direction, which captures the direction of movement of the throttle (outport direction), and a 3D error signal comprising the proportional, integral and derivative errors (outport errs). All output signals are aggregated into the bus signal Throttle Ctrl Bus.

Ports and subsystems are connected by named or unnamed *signal lines*.

The top level of the block diagram also contains a documentation block (ETC) that can be used to capture system requirements and/or further textual description of the system.

In the following, we will dive down into the two major subsystems.

At a lower level in the model hierarchy, the Stateflow block Throttle Sequencer implements the supervisor logic of the system (Figure 4.2).

The Stateflow block Throttle Sequencer depicts the three main states of the system STARTUP, RUNNING and SHUTDOWN and the transitions between them. The STARTUP state for example, checks for potential input error conditions and zeros a counter variable. The input error check (MATLAB Function block Throttle Faults Check) uses MATLAB to calculate the errs signal (among other things) based on a range check of the battery voltage. For that purpose, the actual battery voltage (battery_v) is compared with the permissible minimum (battery_v_min) and maximum (battery_v_max) voltages. The actual voltage is provided via the Battery (V) inport, whereas the minimum and maximum values are calibration data items represented by the constant blocks battery_v_min and battery_v_max, respectively. All three values are converted into signals of type uint16 before being consumed by the Stateflow block. The corresponding MATLAB code excerpt looks as follows:

```
function ThrottleFaultsCheck ( )
if (battery_v > battery_v_max)
    errs (1) = 1;
end
if (battery_v < battery_v_min)
    errs (2) = 1;
end
...
```

Depending on the value of the errs variable, the throttle sequencer transitions into either the RUNNING or SHUTDOWN mode.

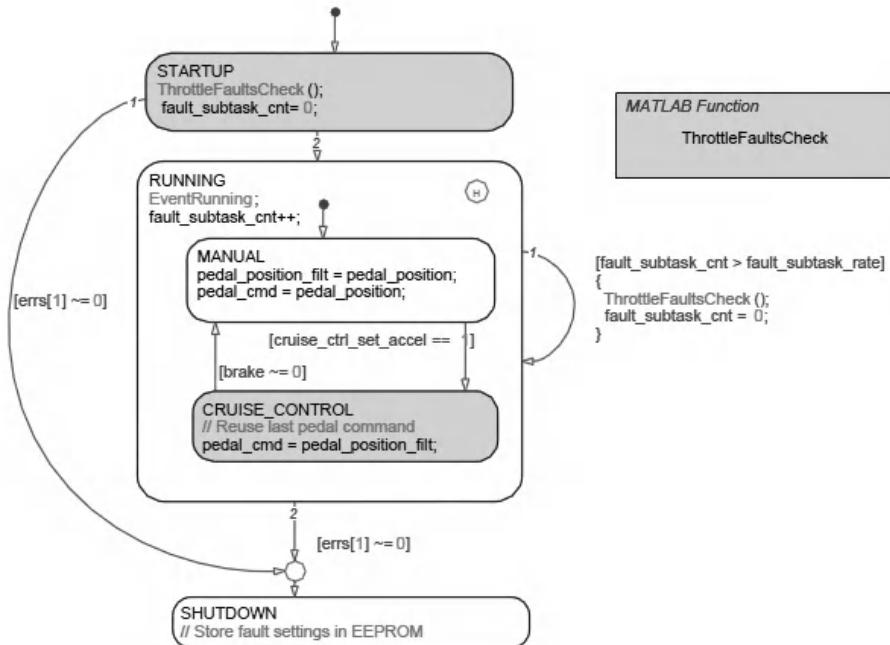


Figure 4.2. State machine throttle sequencer

The state machine Throttle Sequencer is also used to schedule the execution of the Throttle Position Controller task. Such explicit execution control requires an imperative model of computation as indicated by the dashed/dotted function-call connection from the Throttle Sequencer to the Throttle Position Controller. The Throttle Sequencer executes the Throttle Position Controller in the RUNNING mode. When failures occur as well as during STARTUP and SHUTDOWN, execution of the Throttle Position Controller may be suspended.

The Simulink subsystem Throttle Position Controller encapsulates the closed loop control portion of the algorithm; see Figure 4.3. The Throttle Position Controller embodies the low-level feedback control to position the throttle according to a set point. In this example, position, integral and derivative (PID) control is employed. This type of control takes in the difference between the set point and the actual position of the throttle and computes a control force as a weighed sum of (i) the magnitude of the difference, (ii) the time integral of the difference and (iii) the time derivative of the difference.

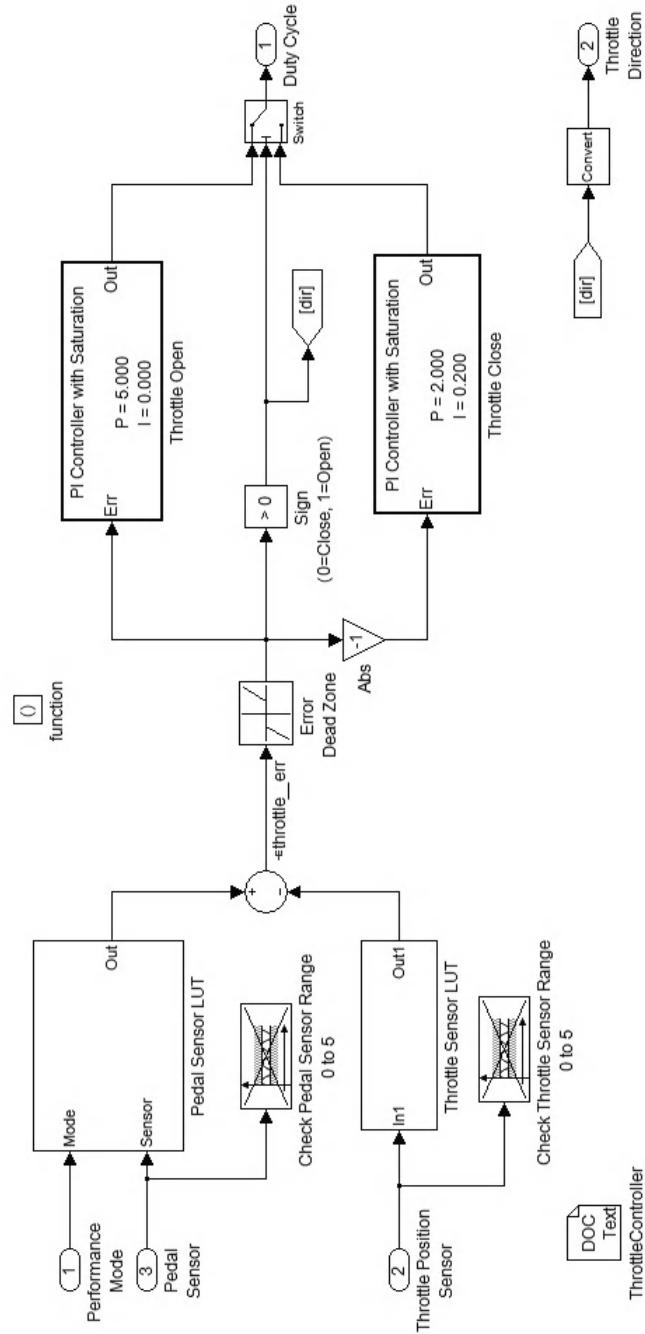


Figure 4.3. *Simulink® subsystem throttle position controller*

In Figure 4.4, a model of the position and integral computation of the PID control is shown where a limiter is applied to the computed control signal to prevent excessive control actions.

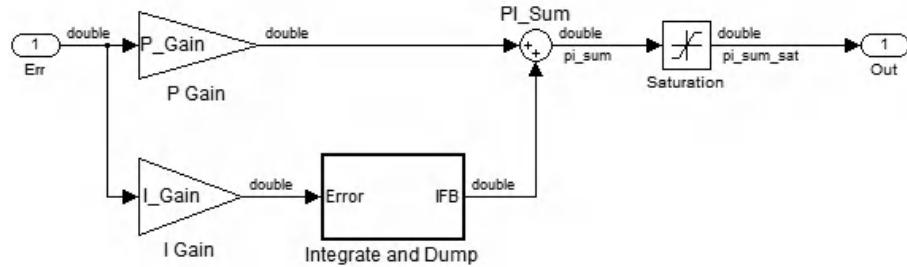


Figure 4.4. The position and integral PID control part

4.3.3. Automatic code generation

When the C source code is automatically generated for the position and integral computation of the PID control of the model, the following code fragment represents the computations of the P Gain and PI_Sum blocks:

```

...
/* Sum: '<S1>/PI_Sum' incorporates:
 * Gain: '<S1>/P Gain'
 */
u = rtp_P_Gain * rtu_Err + rtb_ifb;
...
  
```

Inspection of the code to review whether it properly reflects the model in Figure 4.4 is aided by the generated comments that state which block is responsible for each line of code. The Embedded Coder provides an HTML report of the code that hyperlinks block references such as *<S1>/PI_Sum* directly back to the original Simulink model.

In the above code fragment, notice the use of standardized data structures to store input variables, parameters and block variables, here prefixed by *rtu_*, *rtp_* and *rtb_*, respectively. In general, however, these prefixes can be customized.

In typical automotive applications, to save on stack space, function arguments are often defined as globals and not explicitly passed in through the argument list. Alternatively, local variables could be used to increase modularity, which renders a code review that is more transparent.

Automatic code generation can be configured by subsystem options to generate local variables that are passed into a function by means of an argument list. For the PID control in Figure 4.4, if the PID functionality is selected to be a *re-usable function*, the following function header is automatically generated:

```
void PI_Ctrl(real_T rtu_Err, rtDW_PI_Ctrl *localDW,
real_T rtp_I_Gain, real_T rtp_P_Gain, real_T rtp_SatUpperLim)
```

The corresponding data structures are defined in the accompanying header file.

4.3.4. Code optimization

To obtain code with a smaller memory footprint and lower computational complexity, the user may choose to apply different types of *optimization*, such as block reduction or conditional input branch execution. A *Code Generation Advisor* allows the user to specify and prioritize objectives for code generation, such as RAM, ROM use, execution efficiency, safety precaution and MISRA-C compliance.

4.3.5. Fixed-point code

Another important aspect of control systems in certain industries is their implementation using *fixed-point computations* because fixed-point microprocessors are less expensive than their floating-point counterparts. Some control functionality may be entirely implemented in fixed point, whereas in other cases, a floating-point implementation is employed with a fixed-point back-up system. In the latter case, this results in optimal behavior in nominal modes of operation, while in the face of faults, a rudimentary fixed-point control is used as a backup.

The PID control in Figure 4.4 can be transformed into a fixed-point version by specifying the required fixed-point data type. The polymorphic behavior of the model elements such as the gain and sum blocks causes the fixed-point functionality to be automatically implemented.

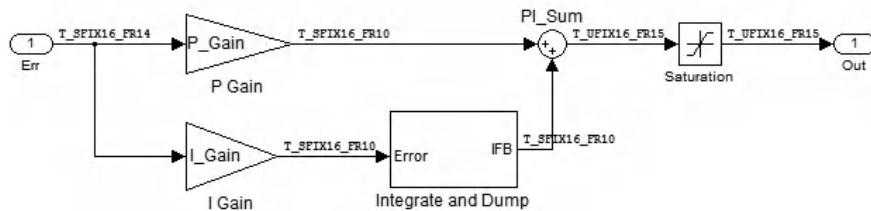


Figure 4.5. PID control in fixed point

In Figure 4.5, a fixed-point version of the PID control in Figure 4.4 is shown. The fixed point data types are shown on the signal connections as, for example, T_SFIX16_FR14, which is shown as the fixed-point data type of the input signal of the Err port. To determine the fixed-point characteristics of this signal, its details can be requested at the MATLAB[®]⁷ command prompt:

```
>> T_SFIX16_FR14

T_SFIX16_FR14 =
Simulink.NumericType
  DataTypeMode: 'Fixed-point: binary point scaling'
  Signedness: 'Signed'
  WordLength: 16
  FractionLength: 14
  IsAlias: true
  DataScope: 'Auto'
  HeaderFile: 'eng_ctrl_prj_types.h'
  Description: ''
```

This shows that the signal value is signed and stored in a 16-bit word where 14 bits are used to represent the fraction of the signal value. This data type is mapped onto a signed short in the eng_ctrl_prj_types.h header file:

```
typedef signed short T_SFIX16_FR14;
```

A fixed-point version of the PID control code that includes *expression folding* can now be automatically generated again. The part that reflects the PI_Sum and P Gain blocks takes the form:

```
/* Sum: '<S1>/PI_Sum' incorporates:
 * Gain: '<S1>/P Gain'
 */
rtb_PI_Sum = (uint16_T)((uint16_T)((int16_T)(rtp_
    P_Gain * rtu_Err >> 14) + rtb_ifb) << 5);
```

Here the left and right bit shift operators, << and >>, respectively, are used for optimal multiplication and addition in a fixed point.

⁷ The MathWorks, Inc. MATLAB[®] – www.mathworks.com/products/matlab.

4.3.6. Including legacy code

Embedded software has been extensively used for decades in the automotive and aerospace industries. Therefore, automatically generated code must often be integrated with existing code, the so-called *legacy code*, which has proven its quality and value.

4.3.7. Importing interface definitions

In legacy code, interface structures are typically specified in header files. These header files can be read by Simulink to import data structures that are defined in legacy code with which the automatically generated code must integrate. For example, in the eng_ctrl_prj_types.h header file, the throttle control data structure T_THROTTLE_CTRL is defined as:

```
typedef struct {
    T_UFIX16_FR15 duty_cycle;
    T_BOOLEAN direction;
    T_BOOLEAN errs[3];
} T_THROTTLE_CTRL;
```

The elements that it contains are the duty_cycle, the direction, and the three-dimensional errs signal.

After importing this structure definition from the header file, it becomes accessible in the MATLAB workspace as a regular MATLAB expression:

```
>> T_THROTTLE_CTRL
T_THROTTLE_CTRL =
Simulink.Bus
Description: ''
DataScope: 'Auto'
HeaderFile: 'eng_ctrl_prj_types.h'
Alignment: -1
Elements: [3x1 Simulink.BusElement]
```

This shows the header file that contains the structure source. Viewing one of the elements in the Simulink Bus provides detailed information about the characteristics of that signal:

```

>> T_THROTTLE_CTRL.Elements(1)

ans =
Simulink.BusElement
    Name: 'duty_cycle'
    Complexity: 'real'
    Dimensions: 1
    DataType: 'T_UFIX16_FR15'
    Min: []
    Max: []
DimensionsMode: 'Fixed'
SamplingMode: 'Sample based'
SampleTime: -1

```

This shows the duty_cycle bus element to be a fixed-point signal (unsigned 16-bit word with a 15-bit fraction) that only has a real term (i.e. not an imaginary term of a “complex” type), is a scalar, and has a sample time that is inherited from its context.

4.3.8. Importing algorithms

Legacy algorithms tend to be tried and tested in the field and have proven themselves for many years in products. Therefore, rather than re-design them, the legacy code may have to be re-used. To support this re-use, in addition to interface definitions, algorithms implemented in legacy code can be integrated into automatically generated code as well. Details on how the Simulink environment facilitates the integration of model generated code and handwritten legacy code are documented in previous work [MOS 06].

4.4. Verification and validation of models and generated code

Stringent software development workflows are required to satisfy customer expectations and to ensure the essential quality and reliability of embedded software in many application domains. However, given the high-integrity nature of many state-of-the-art systems, application of verification and validation techniques over and above existing software development practices must be considered [CZE 04].

4.4.1. Integrating verification and validation with Model-Based Design

For high-integrity applications – irrespective of the software engineering paradigm used – it is crucial that the software lifecycle includes a relevant combination of

verification and validation techniques to detect errors and promote confidence in the correct functioning of the software.

In principle, verification and validation of embedded software developed using Model-Based Design and code generation could be carried out in the same way as manually written code. This, however, would fail to capitalize on the benefits of Model-Based Design to improve process efficiency.

Introducing executable models into the design process for embedded software enables the application of automatic verification and validation techniques earlier in the software lifecycle, and at a higher level of abstraction. As such, Simulink models are an excellent source of information that can be used for various verification and validation techniques.

Verification and validation activities are used to improve the quality of the embedded software. They cannot be undertaken in isolation, but to be effective and efficient must be integrated into the software development lifecycle. Therefore, it is desirable to integrate verification and validation approaches well with the constructive activities of Model-Based Design. In particular, there is a high demand for methods and tools to verify and validate systems that include generated code and that are easy to incorporate into engineering workflows.

Various authors consider translation validation of generated code as a suitable means to carry out verification and validation of generated code in an engineering context [EDW 99, TOE 99, BUR 04]. In the scope of this chapter, the notion of translation validation that is adopted is as introduced by [PNU 98]. In this interpretation, translation validation refers to application-specific verification/validation approaches where each individual model-to-code translation is followed by a validation phase that verifies that the target code produced by this translation (i.e. code generation/compilation) properly implements the submitted source model. Note that it is not implied that formal verification techniques are used to perform this validation. Instead, an approach is described, where dynamic testing for numerical equivalence between models and generated code constitutes the core part of the translation validation process [CON 11]. An important benefit of this approach in the context of its industrial applicability is its scalability.

When using Model-Based Design and production code generation, application-specific verification and validation can be divided into two tasks:

Design verification combining verification and validation techniques at the model level to demonstrate that the model is well-formed, meets its requirements, and does not contain unintended functionality.

Code verification using equivalence testing and other techniques to demonstrate equivalence between the model and the generated code compiled into an executable model.

The following sections summarize such a two-phase approach to verify and validate models and generated code.

4.4.2. Design verification

The goal of design verification is to gain confidence in the model that is being used for production code generation.

The design verification part comprises a combination of reviews, static analyses and comprehensive functional testing activities at the model level, before the code is generated [CON 08, CON 09]. These activities show that the design satisfies the associated requirements, does not contain unintended functionality, and complies with defined modeling guidelines.

4.4.3. Reviews and static analyses at the model level

Model components (i.e. model subsystems considered as modules) should be reviewed. If feasible, manual model reviews should be automated using static analyses of the model using tools such as *Model Advisor* [BEG 07].

To facilitate efficient and effective review, *modeling guidelines* [CON 05b, ERK 05] should be used and adhered to. Modeling constructs that are not suited or not recommended by the tools vendor for production code generation should be avoided.

Static analyses at the model level can be augmented by the use of formal verification techniques. *Simulink Design Verifier*^{TM8} [HAM 08] (see also Chapter 5) can be used to detect design errors such as divide-by-zero or overflows/underflows and to formally prove properties of the model.

4.4.4. Module and integration testing at the model level

Model components should be functionally tested using test cases that are systematically derived from the software requirements specification. The objective of module testing is to demonstrate that each model component performs its intended function and does not perform unintended functions.

⁸ The MathWorks, Inc. *Simulink Design Verifier* – www.mathworks.com/products/sldesignverifier.

Once model component testing is completed, components can be integrated. The model integration stages shall be tested in accordance with the dedicated integration tests. These tests should show that all model components interact as specified to perform their intended function and do not perform unintended functions.

A more detailed discussion on testing at the model level can be found in [CON 09b] while integration at the model level has been the topic of previous work [MOS 05].

The result of the design verification is a *golden model*, that is, a verified and validated, well-formed model that implements the requirements and does not contain unintended functionality. The golden model serves as a reference for the subsequent code verification activities.

4.4.5. *Code verification*

A targeted combination of verification, validation and testing activities at the code level will demonstrate that the semantics of the golden model is being preserved during code generation, compilation and linking. Furthermore, it needs to be shown that no unintended functionality is being introduced during code generation.

4.4.6. *Back-to-back comparison testing between model and code*

Back-to-back testing or equivalence testing [CON 09a, CON 11] i.e. translation validation using systematic testing, can be employed to demonstrate numerical equivalence between the model and the generated source code. Each run of the code generation tool chain is then augmented with a back-to-back testing phase to verify that the executable compiled from the generated C or C++ code (object code) numerically matches the golden model.

Back-to-back testing for numerical equivalence is well suited to automation because the expected output for the test vectors does not have to be provided [ALD 02]. Detailed discussions of back-to-back testing procedures are available in [STU 07, CON 10].

4.4.7. *Measures to prevent unintended functionality*

Back-to-back testing alone may not be enough to demonstrate the absence of unintended functionality in the generated source code. Therefore, back-to-back testing to assess numerical equivalence must be augmented with additional measures to demonstrate the absence of unintended functionality. This second activity in the code verification process demonstrates that the generated C code does not perform

unintended functions (i.e. the code does not contain functionality that is not embodied by the model).

Alternative techniques are available to achieve this objective including (a) model versus code coverage comparison and (b) traceability analysis. These techniques serve to demonstrate structural equivalence between the model and the source code. If model versus code coverage comparison is used to show the absence of unintended functionality, model and code coverage are measured during back-to-back testing and compared with each other. Discrepancies with respect to comparable coverage metrics must be assessed. Alternatively, a traceability analysis can be performed to demonstrate that all parts of the generated C source code can be traced back to the golden model. In this case, the generated code is subjected to a limited review that exclusively focuses on traceability aspects and the non-traceable code shall be flagged and assessed.

4.5. Compliance with safety standards

The benefits of Model-Based Design apply to a variety of embedded systems, including high-integrity applications. In the recent past, engineers have successfully employed graphical modeling with Simulink and code generation to produce software for high-integrity applications [POT 04, JAB 08, FEY 08, KRI 10]. Further consideration and rigor is necessary to improve the efficiency with which to address the constraints imposed by functional safety standards and to produce the required evidence for compliance demonstration. Modern functional safety standards such as IEC 61508 [IEC 10], EN 50128 [EN 11], and ISO 26262 [ISO 11] discuss concepts such as modeling, model-based development and code generation and provide – depending on the standard at hand – more or less detailed objectives for implementing them.

The workflow for verification and validation of models and generated code that is outlined in this chapter can – with small standard specific variations – be useful in meeting the requirements of these standards with respect to core Model-Based Design artifacts, that is, executable models and generated code.

To facilitate the utilization of Model-Based Design for high-integrity applications, the IEC Certification Kit⁹ provides an extended version of the workflow for verification and validation of Simulink models and generated C/C++ code as well as checklists and templates to document the compliance with applicable functional safety standard. The mapping of this reference workflow onto IEC 61508 and ISO 26262 has been discussed in (Conrad 2009) and (Conrad 2012), respectively and can be adapted to other standards as well.

⁹ The MathWorks, Inc. IEC Certification Kit – www.mathworks.com/products/iec-61508.

4.6. Conclusion

This chapter presented an overview of Model-Based Design with Simulink in industry. In particular, it discussed modeling and code generation. An electronic throttle control system has been used to illustrate a number of aspects.

In addition, an approach to augment modeling and code generation with verification and validation activities has been outlined and its use to satisfy the requirements of functional safety standards has been discussed.

4.7. Bibliography

- [ALD 02] ALDRICH W.J., "Using model coverage analysis to improve the controls development process", *AIAA Modeling and Simulation Technologies Conference, Monterey, CA, USA*, 2002.
- [ALU 94] ALUR R., DILL D.L., "A theory of timed automata", *Theoretical Computer Science*, vol. 126, p. 183-235, 1994.
- [AND 02] ANDERSSON G., BJSSE P., COOK B., HANNA Z., "A proof engine approach to solving combinational design automation problems", *39th Design Automation Conference (DAC 2002)*, New Orleans, LA, USA, June 2002.
- [BEG 07] BEGIC G., "Checking modeling standards implementation", *The MathWorks News & Notes*, June 2007.
- [BUR 04] BURNARD A., "Verifying and validating automatically generated code", *International Automotive Conference (IAC '04)*, p. 71-78, Stuttgart, Germany, 2004.
- [CON 04] CONRAD M., *Modell-basierter Test eingebetteter Software im Automobil - Auswahl und Beschreibung von Testszenarien*, Deutscher Universitätsverlag, Wiesbaden, Germany, 2004.
- [CON 05a] CONRAD M., FEY I., GROCHTMANN M., KLEIN T., *Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler*, Informatik Forschung und Entwicklung, vol. 20, no. 1-2, p. 3-10, 2005.
- [CON 05b] CONRAD M., DÖRR H., FEY I., POHLHEIM H., STÜRMER I., *Guidelines und Modellreviews in der Modell-basierten Entwicklung von Steuergeräte-Software*, 2nd Tagung Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik, Berlin, Germany, 2005.
- [CON 06] CONRAD M., DÖRR H., "Model-based development of in-vehicle software", *Conference on Design, Automation and Test in Europe (DATE '06)*, Munich, Germany, p. 89-90, 2006.
- [CON 08] CONRAD M., "Model-Based design for iiec 61508: towards translation validation of generated code", *Workshop Automotive Software Engineering: Forschung, Lehre, Industrielle Praxis*, Munich, Germany, 2008.

- [CON 09a] CONRAD M., “Testing-based translation validation of generated code in the context of IEC 61508”, *Formal Methods Systyem Design*, vol. 35, no. 3, p. 389-401, 2009.
- [CON 09b] CONRAD M., FEY I., “Testing automotive control software”, in: NAVET N., SIMONOT-LION F. (eds), *Automotive Embedded Systems Handbook*, CRC Press, Boca Raton, FL, USA, 2009.
- [CON 10] CONRAD M., ERKKINEN T., MAIER-KOMOR T., SANDMANN G., POMEROY M., “Code generation verification – assessing numerical equivalence between simulink models and generated code”, *4th Conference Simulation and Testing in Algorithm and Software Development for Automobile Electronics*, Berlin, Germany, 2010.
- [CON 11] CONRAD M., “Testing-based translation validation of generated code”, in: ZANDER J., SCHIEFERDECKER I., MOSTERMAN P.J. (eds), *Model-Based Testing for Embedded Systems*, CRC Press, Boca Raton, FL, p. 579-599, USA, 2011.
- [CON 12] CONRAD M., “Verification and validation according to ISO 26262: a workflow to facilitate the development of high-integrity software”, *Embedded Real Time Software and Systems (ERTS² 2012)*, Tolouse, FR.
- [CZE 04] CZERNY B.J., D’AMBROSIO J. G., JACOB P.O., MURRAY B.T., SUNDARAM P., “An adaptable software safety process for automotive safety-critical systems”, *SAE Technical Paper*, 2004-01-1666, SAE World Congress, Detroit, MI, USA, 2004.
- [EDW 99] EDWARDS P.D., “The use of automatic code generation tools in the development of safety-related embedded systems”, *Vehicle Electronic Systems*, ERA Report no. 99-0484, 1999.
- [EN 11] EN 50128:2011, Railway applications - Communication, signalling and processing systems – Software for railway control and protection systems, European Standard.
- [ERK 05] ERKKINEN T., “Model style guidelines for production code generation”, *SAE Technical Paper*, 2005-01-1280, SAE World Congress, Detroit, MI, USA, 2005.
- [FER 04] FERET J., “Static analysis of digital filters”, *Lecture Notes in Computer Science* 2986, Springer-Verlag, Berlin, Germany, p. 33-48, 2004.
- [FEY 08] FEY I., MÜLLER J., CONRAD M., “Model-Based design for safety-related applications”, *SAE Technical Paper*, 2008-21-0033, Convergence 2008, Detroit, MI, USA, 2008
- [GRI 02] GRIFFITHS P.G., Embedded software control design for an electronic throttle body, Masters Thesis, University of California, Berkeley, CA, USA, 2002.
- [HAM 08] HAMON G., “Simulink design verifier – applying automated formal methods to simulink and stateflow”, *3rd Workshop on Automated Formal Methods (AFM 08)*, Princeton, NJ, USA, July 2008.
- [HAR 87] HAREL D., “Statecharts: a visual formalism for complex systems”, *Science of Computer Programming*, vol. 8 p.231-274, 1987.

- [HEL 05] HELMERICH A., KOCH N., MANDEL L., Study of worldwide trends and R&D programs in embedded systems in view of maximising the impact of a technology platform in the area, Report For The European Commission, Brussels, Belgium, 2005.
- [IEC 10] IEC 61508:2010, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, International Standard, 2nd edition.
- [ISO 11] ISO 26262: 2011, Road vehicles – Functional safety, International Standard.
- [JAB 08] JABLONSKI T., SCHUMANN H., BUSSE C., HAUSSMANN H., HALLMANN U., DREYER D., SCHÖTTLER F., *Die neue elektromechanische Lenkung APA-BS*, ATZelektronik, vol. 3, no. 1, p. 30-35, 2008.
- [JER 00] JERSAK M., CAI Y., ZIEGENBEIN D., ERNST R., “A trans-formational approach to constraint relaxation of a time-driven simulation model”, *13th International Symposium on System Synthesis*, Madrid, Spain, 2000.
- [KAM 07] KAMGA J., HERRMANN J., JOSHI P., “Deployment of model-based technologies to industrial testing”, *D-MINT automotive case study – Daimler*, ITEA 2 Project Deliverable 1.1. 2007.
- [KRI 10] KRISTAN S., ZIMMERMANN J., “Evaluating costs and benefits of model-based development of embedded software systems in the car industry – results of a qualitative case study”, *Altran Technologies*, Munich, 2010.
- [MAT 12] MATHWORKS, *Model-Based Design*. www.mathworks.com/model-based-design, 2012.
- [MER 00] MERZ R., LITZ L., Objektorientierte *Mathematische Modellierung - Generische Methoden bei komplexen dynamischen Systemen*, Informatik Spektrum, vol. 2, no. 23, p. 90-99, 2000.
- [MOS 05] MOSTERMAN P.J., GHIDELLA J., FRIEDMAN J., “Model-based design for system integration”, *2nd CDEN International Conference on Design Education, Innovation, and Practice*, p. TB-3-1 - TB-3-10, Kananaskis, Alberta, Canada, July 2005.
- [MOS 06] MOSTERMAN P.J., “Automatic code generation: facilitating new teaching opportunities in engineering education”, *36th Annual ASEE/IEEE Frontiers in Education Conference*, p. 1-6, San Diego, CA, USA, October 2006, IEEE Press
- [MOS 07] Mosterman P.J., “Hybrid dynamic systems: modeling and execution”, Chapter 15 of *Handbook of Dynamic System Modeling*, FISHWICK P.A. (ed.), CRC Press, 2007.
- [NIC 09] NICOLESCU G., MOSTERMAN P., *Model-Based Design for Embedded Systems*, CRC Press, Boca Raton, FL, USA, 2009.
- [PNU 98] PNUELI A., SIEGEL M., SINGERMAN E., “Translation validation”, *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, Lisbon, Portugal, p. 151-166, 1998.
- [POT 04] POTTER B., “Use of the mathworks tool suite to develop do-178b certified code”, *ERAU / FAA Software Tools Forum*, Daytona Beach, FL, USA, 2004.

- [RAU 03] RAU A., Model-based development of embedded automotive control systems, PhD thesis, Tübingen University, Germany, 2003.
- [STU 05] STURMER I., WEINBERG D., CONRAD M., “Overview of existing safeguarding techniques for automatically generated code”, *2th International ICSE Workshop on Software Engineering for Automotive Systems (SEAS '05)*, St. Louis, MO, USA, 2005.
- [STU 07] STURMER I., CONRAD M., DÖRR H., PEPPER P., “Systematic testing of model-based code generators”, *IEEE Transactions on Software Engineering*, vol. 33, no. 9, September 2007.
- [TOE 99] TOEPPE S., RANVILLE S., BOSTIC, D., WANG Y., “Practical validation of model based code generation for automotive applications”, *18th, AIAA/IEEE/SAE Digital Avionics System Conference*, 1999.

Chapter 5

Proving Global Properties with the Aid of the SIMULINK DESIGN VERIFIER Proof Tool

5.1. Introduction

This chapter has the objective of highlighting, using a significant example, the contributions of model verification and formal proof techniques for model-driven design processes applied mainly to complex embedded systems.

Models described in this chapter are command and control models specified in Simulink¹ (© The Mathworks) and SCADE² (© Esterel Technologies), comprising various data and complex temporal problems.

To begin with, we will recall the principal application cases of model verification methods so as to position our work in relation to academic and technological state of the art reviews and industrial practice. We will afterwards specify the different verification levels applicable for a complex model as well as the property classes verifiable at each level. The third part of this chapter is devoted to the study realized at SafeRiver³ on a complex function for a rail system. This study illustrates the implementation of a methodology for the modeling and verification of behavioral/

Chapter written by Véronique DELEBARRE and Jean-Frédéric ETIENNE.

1 To find out more, visit: www.mathworks.com/products/simulink.

2 SCADE is distributed by ESTEREL TECHNOLOGIES; to find out more, visit www.esterel-technologies.com.

3 For more details, visit: www.saferiver.com

safety properties. The contributions of this approach for the setting up of safety cases as well as the positioning of these activities in normative environments will also be discussed.

5.2. Formal proof or verification method

Formal methods cover most specification, verification and testing techniques, which are generally based on mathematical theories such as logic, automata, or graph theory.

The activity of specification consists of obtaining a system description at different stages of the development lifecycle and a description of the system properties.

The term verification includes any activity, that consists of showing that a given code satisfies a property or conforms to a description at the highest level. In general, a distinction is made between deductive verification and model verification, both of which aim at establishing the correctness of a system description.

Among verification techniques are also considered the class of methods aiming at detecting errors in specifications. In particular, we will present methods that capture specification and design errors in the form of counter-examples, and others that target specific implementation errors highly dependent on the specification language. Static code analysis belongs to the latter category and will be developed in the state of the art review.

Methods, which are used to prove properties or detect behaviors violating properties, operate on descriptions classified into four categories:

- formal specification methods grounded on mathematical notations derived from logic theory, with a verification process based on formal proof techniques;
- system specification languages based on automata theory, with a verification process implementing model verification, state graph exploration, and testing techniques;
- semi-formal specifications and notations that can be formalized using one of the previously mentioned specification languages, when properly constrained;
- programs, which are specifications can directly be compiled and executed on a given platform. Program verification implements static or dynamic analysis techniques such as testing or assertions taken into account at runtime.

Industrial interest in formal methods hinges fundamentally on the verification and “proof” of system correctness. Three categories of formal methods have been tested and received experience feedback:

- methods based on modeling and model verification;
- methods based on proof;
- static code analysis based on abstract interpretation.

Model verification techniques mainly consist of constructing a model M of a given system, of specifying a set Φ of properties, and of establishing that “ M satisfies Φ ” (see Figure 5.1).

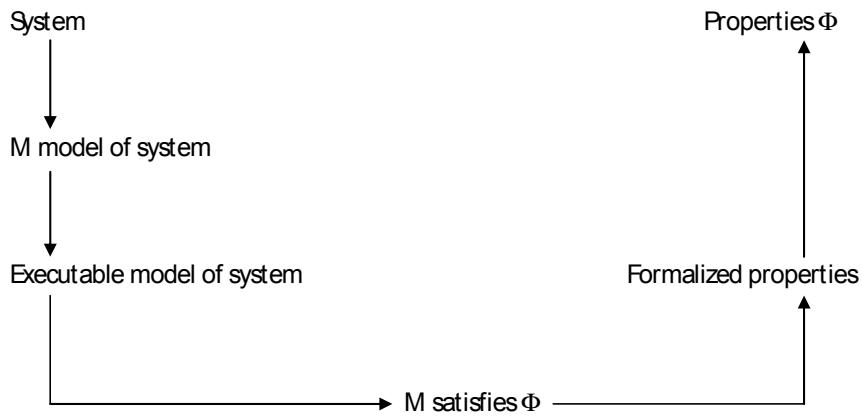


Figure 5.1. Model verification

Executable models are generally formalized in terms of state/transition systems and model verification consists of verifying properties on the graph state space.

*Model-checking*⁴ techniques [BAI 08, GRU 08] are sensitive to the size and complexity (rarity, loops, and domain of state variables) of the graph state space and are more suitable for control-oriented problems manipulating relatively simple data structures.

The model verification approach is considered sufficiently mature and “simple”, especially for communication protocols and controller design in the world of embedded systems, as well as for the design of hardware circuits. However, this approach is limited as it relies on an exhaustive enumeration of states or “equivalence classes” of system states, and cannot manipulate complex data structures.

⁴ Grouping together a set of model analysis and verification techniques to statically evaluate some behavioral properties, generally expressed using temporal logic formulae (LTL or CTL or CTL*).

For problem solving that manipulates structured data types such as lists (or trees) or that deals with potentially “infinite” systems, proof-based approaches are more suitable as they promote induction reasoning. The basic idea behind program verification (see Figure 5.2) consists of specifying hypotheses or expressing constraints on the program’s inputs and states before execution, of specifying the constraints or properties expected on the program’s outputs, and of proving that the states obtained after execution satisfy the expected properties.

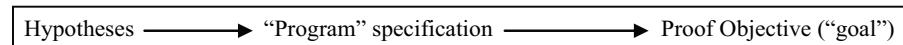


Figure 5.2. From hypotheses to proof objectives

The formal proof size of a program/function is generally significant. In fact, the number of lemmas required for the formalization and proof of a given goal can be very substantial. This can have a negative impact on the readability of the proof outline and can even impede the proof of the proof process (i.e. verifying the correctness of the proof outline). Moreover, the gap between the specification level of properties to be shown and the level at which the proof is carried out is a serious obstacle to overcome when using such methods.

The key points to consider for the practical use of formal proof methods and tools are:

- ability to formalize application problems in a high-level language and to carry out the proofs in an automated manner on the high-level formalization;
- use of proof assistants or interactive proof;
- assistance for the comprehension and control of the proof outline

5.2.1. Model verification

5.2.1.1. Benefits and positioning of model verification methods with respect to the development cycle

Model verification methods are naturally integrated into the “model-based design” flows applied to the field of command and control systems. SCADE and SIMULINK⁵ formalisms are examples of such models.

⁵ In this chapter, we only deal with models of this type, and we have in particular avoided tackling development cycles based on SysML or UML [OMG 07], increased by rules or description formalisms arising from formal methods.

We identify three levels of descriptions or models:

- system-level models, which show some functional principles (including the safety aspects) of the target system within its environment. The environment is most often constituted of the controlled system and of events influencing system functioning conditions. These high-level models are often complex, especially due to asynchronisms between system components or due to events in the environment. Several formalisms are suggested for dealing with this level, from UML⁶ [OMG 07] and SysML [FRI 08] notations to languages based on communicating controllers. The SIMULINK language is largely used for embedded systems, notably in the automobile industry. This phase continues to be described largely in textual language, in the form of more or less formal clauses;

- at the operational specification and software design level, formalisms enabling code generation to be embedded will mainly be considered. B models [ABR 96], SCADE models [DOR 08], SIMULINK or related models like TargetLink are the descriptions used most often for development of control systems in the railway, aeronautical, and automobile industries. These models allow verification that a) the model is correct in relation to behavioral properties, especially the functional safety requirements; and b) execution errors are absent in the design. Subject to use of appropriate code generation options, the conservation of these properties in the generated code can thus be guaranteed;

- at the level of “programs” due to the many verification activities related to proof or model verification activities; static analysis, dependency calculations, execution path calculations, assertions, etc. Code verification activities differ according to whether a manual or generated code is considered from a design model.

Model verification techniques have been in use in communication protocols for several decades. Some formalisms have been developed especially to address these categories of problems (that is to say, Promela/SPIN [HOL 04, MAG 03]; models based on the SDL language⁷ or even Petri nets).

They enable verification of generic properties such as re-initialization, the absence of deadlock, or black hole states, and of specific communication protocol properties such as message sequencing or conformity to expected service delivered. Verification of properties relies most often on their formalization by temporal logic clauses (LTL, CTL⁸).

⁶ UML is a set of notations for description of systems and software.

⁷ SDL is a language standardized by the ITU for specification and description of distributed systems.

⁸ LTL: Linear Temporal Logic, CTL: Computation Tree Logic.

Finally, model verification applies to hardware circuit design at *Electronic System Level*, described in the languages System C, C (with extensions), or RTL level languages.

For many years, languages for description of properties have been developed (for example, PSL⁹ [EIS 06]), which generally rely on temporal logic (LTL or CTL, or CTL*). Editors provide model verification and equivalence verification tools integrated into circuit design flows.

There is a lot of literature on this, and a list of references can be found at the end of this chapter, which presents some case studies in various fields of application.

5.2.1.2. Target properties

Generally, properties are classified as:

- system state or invariant properties;
- temporal properties applying to possible system trajectories and behaviors;
- safety properties;
- liveness properties, aiming to show that the system *progresses*;
- time-bound properties;
- execution integrity properties;
- robustness properties, consisting of showing that the behavior of the system complies with its specification in the presence of events, which may disturb software execution.

The first four classes of properties are essentially logical and can be verified using models of different abstraction levels. In practice, expected system behavior properties are formalized by logic clauses, which apply to objects described in the model: properties of system interface states, internal variables or sequence properties between observable system states, for example. Some properties can also be formalized at the programming level by means of assertions or “track systems” integrated into the code.

On the other hand, the last three classes are applicable only to executable models and must take into account data on the execution conditions: processor speed and

⁹ PSL: *Property Specification Language*. PSL is a standardized language for specification of temporal logic properties (IEEE 1850) used in the field of hardware circuit design (see www.accellera.org and www.eda.org/ieee-1850).

conditions of random hardware failures, which may impact software execution. In general, these properties are established at the software level through the introduction of some characteristics of the target processor.

5.2.2. Formal methods and proof of correction

These methods are based on a rigorous abstraction, which does not authorize the subtext or implicit hypotheses. Sometimes termed deductive methods, they use a mathematical logic as a formal language to describe a system and prove correction of this description. They offer two main approaches:

- *automated proof*: description of the system and its properties is made in the language of a proof tool and this tool is used to prove correction of these properties;
- *program transformation*: the objective of this approach is correction by construction. It enables a correct implementation (concrete expression) by transformation of the model and its specification (abstraction). Transformations can be made automatically by rewriting or refinement.

5.2.2.1. Automated proof

The objective of automated proof is to free mathematicians from the burden of sketching proof outlines: i.e. providing the reasoning steps necessary for deriving the validity of a given formula based on some axioms described in a formal language. The architecture of the automated proof relies on some core mathematical definitions that are deemed to be sound and that guarantee the validation of only putative proofs.

The first stage of using a proof tool is formalization of the problem and properties in an adapted language. This specification stage, judged to be very costly, consists first of all, of modeling the entities of the problem, most often by means of “types” and identifying the “invariants”, which are properties which must be true for all system states or even for all possible program executions. It is also necessary to define operations as functions taking types in arguments or as predicates applying to defined types. These functions operate transitions in the system state and must guarantee invariants.

The objective of the third stage is to make the actual proof. A proof objective (*goal*) is formalized, which is a root objective. Making the proof consists of constructing a proof tree, breaking the proof objective into sub-objectives in an iterative manner up to the point where all nodes of the proof tree can be shown. Construction of this proof tree may be automated and more or less easy to implement. Indeed, proof tools have some axioms and reasoning modes at their disposal to reduce the proof objective to sub-goals until resolution. Often, however,

intermediate lemmas must be added by the user to specify the sub-goals. The most well-known proof tools are ACL2, HOL, HOL-Light, Isabelle, PVS and Coq.

Among the tools previously mentioned, the PVS environment is without doubt the most suited for industrial applications. It implements a very pragmatic proof approach, by means of commands, which aid construction of the proof tree.

Two types of assistants can be differentiated: those based on logic and those based on meta-logic. The latter can be instantiated by choosing logic to construct a tool of the former class: Isabelle was declined in Isabelle/HOL, Isabelle/ZF and Isabelle/FOL¹⁰.

There are differences between assistants at several levels:

- the underlying logic: higher-order logic (Isabelle, HOL, PVS), structured typing (Coq);
- the automation level and decision-making ability (that is to say, PVS commands);
- generation of an executable specification;
- construction of proofs, which survive verification: Isabelle, Coq;
- proof of the tool itself: Coq is proofed by Coq.

Proof assistants prove proofs supplied by other systems (including other assistants).

5.2.2.2. Rewriting

Term rewriting is a method of proof by equational logic. It enables one term to be transformed into another by applying a rewriting rules system. One stage of this transformation replaces the instances of a motif (*pattern*) of the left member of a rule with the instance corresponding to the motif (*pattern*) of the right member of this rule in the term.

The method that determines how to apply rewriting rules is implicit: the rules are applied to a term until no rule can be applied any longer. If the rewriting system is confluent and terminating, the order will have no effect on the result, but it will have an impact on the effectiveness of the calculation.

If the set of rewriting rules concerning a given problem is not confluent and not terminating, it is necessary to control the rewriting rule system application by adding strategies: *top-down*, *bottom-up*, etc.

¹⁰ To find out more about Isabelle, visit the University of Cambridge site at www.cl.cam.ac.uk/research/hvg/Isabelle.

Rewriting can be used as a specification execution or calculation tool. For example, if the system expresses concurrence or transition between system states, it can simulate progress of a reactive system.

There are several specification systems based on rewriting logic. These systems offer a language based on rewriting rules, which enables the specifications to be executable. ASF-SDF, CafeOBJ (OBJ), ELAN, Maude, Stratego and Tom are languages of this type.

The ASF-SDF language can be transformed into C, which can be compiled using gcc, for example.

The Tom language is an extension of a target language (C, Java or Eiffel). The Tom tool can thus be seen as a pre-processor of the target programming language because it transforms extensions of the target language (Tom constructions) into a program written entirely in the target language.

5.2.2.3. Formal description and refinement methods

The objective of the refinement process is incremental program development from specifications written in a formal language. The most well-known methods in this family of formal methods are B [ABR 96], Z [SPI 89] and VDM [JON 90], which have benefitted from industrial experience feedback. In France, the B method has notably been implemented for development of the SAET-METEOR¹¹ metro line autopilot [BOU 06, BOU 99].

The first stage of the refinement process consists of defining the abstract (non-executable) model of the system: the specification defines the invariants; the model that describes system behavior is verified, and then proven correct in relation to the invariants.

The second stage consists of refining this model toward a less abstract model (memory and time). Invariant refinement is achieved by transforming abstract types into data types; model refinement is achieved by breaking each operation down into algorithms; the refined model is verified, and then proven correct.

This second stage can be reiterated until an executable model written in an executable subassembly of the language (B0 for the B language) is obtained or a subassembly is automatically translatable into a programming language (C or Ada for B, Ocaml for FoCal). Different proofs enable verification of the coherence of the abstract model and conformity of each refinement to the higher model (thus proving compliance of the set of actual installations with the abstract model).

¹¹ SAET-METEOR: *système d'automatisation de l'exploitation des trains – Metro Est Ouest rapide* (train operation automation system - East-West high-speed metro); see [MAT 98] for more information.

Several languages enabling application of this development model have been devised: B, FoCal, LOTOS, Unity, VDM (RAISE, VDM-SL, and VDM++), and Z.

The B language uses an industrialized development environment, “Atelier B” as well as an “open source” version, B4free. The B system additionally enables production of Ada code for the target processor, “coded processor” sometimes called SACEM. In addition, JEdit, Zlive, Logica, Zeus, and Z/Eves are tools associated with the Z language.

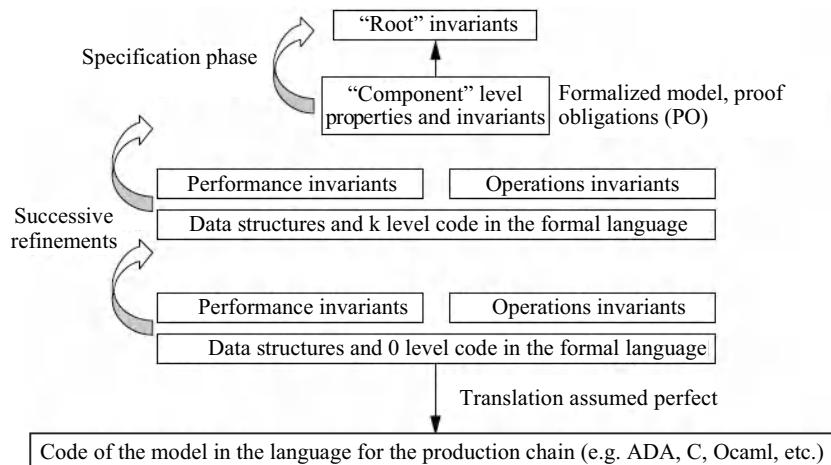


Figure 5.3. Formal process

5.2.3. Combining models and proof tools

The previous sections illustrate the different advantages and drawbacks of each of the two approaches: model verification and proof of correction. However, both methods are now implemented in a complementary manner, which has enabled the development of more effective and easily implemented tools.

This is the case with the Prover tool (© Prover Technology) especially, which implements the following techniques:

- a model verification technique: *Bounded Model Checking*;
- a proof-by-induction technique, which enables verification of invariant properties of states;
- an underlying technique of *satisfiability*, which serves as a decision procedure with respect to reachability of states.

Naturally, the Prover tool requires a description of the system in the TECLA language. However, two modeling environment editors have integrated this proof tool into their respective modeling and code generation environments. Mathworks has integrated the Prover tool into the Matlab environment (© The Mathworks), and Esterel Technologies has integrated it into the SCADE environment (© Esterel Technologies).

Method	Advantages	Inconveniences
Model verification	Ease of description, generally in a development language (possibly with restrictions) Ease of positioning in the development cycles Ease of expression of state properties and properties in the system trajectories Test generation possibilities	Sensitivity to the size of the model and risks of combinatorial explosion Difficulty of including complex data structures
Proof of correction	Processing of infinite state spaces Capability of direct reasoning on the principles and rules of system functioning Capability to take into account notions of “learning” by introduction of particular rules	Specific formalism (except B) Difficulty of implementing proof tools, especially interactive proof tools

Table 5.1. Comparison between verification and proof

Design flow can thus easily be increased by verification and proof-of-property activities, and the designer is freed of the work of translation into formal language. This is the process that we tested in the Matlab/SIMULINK environment and, which will be described in detail in section 5.3.1.

This approach is a significant step for a better appropriation of verification and proof technologies by manufacturers and is helpful in the reduction of the total cost of ownership for such tools.

5.3. Implementation of the SIMULINK DESIGN VERIFIER tool

The MATLAB environment (© The Mathworks) provides a wide variety of tools that can be applied at the different phases of the development lifecycle process. For

in our case study, we essentially made use of SIMULINK/EMBEDDED MATLAB for modeling purposes and SIMULINK DESIGN VERIFIER for the formal verification and validation process. These are presented in the first subsequent subsections. We afterwards introduce our case study and mainly focus on the various factors affecting the complexity of the “Train Tracking” function in a Communication-Based Train Control (CBTC - [IEE 04]) project. We next show how the “Train Tracking” function was formalized in SIMULINK and outline the various proof strategies employed to counteract the state space explosion problem when proving the high-level safety properties on the model.

5.3.1. Reminders of the MATLAB modeling and verification environment

5.3.1.1. Simulink/Embedded Matlab

SIMULINK is a synchronous *data flow* language, which has the particularity of being graphical. In SIMULINK, data flows are said to be synchronized as they are controlled based upon a unique global clock. The clock tick determines when the data flows change value. Each data flow is also typed (e.g. integers, Boolean, unsigned integers, etc.) and new data flows may be derived from existing ones using primitive constructs called *blocks*. As such, SIMULINK can be also considered as a functional language.

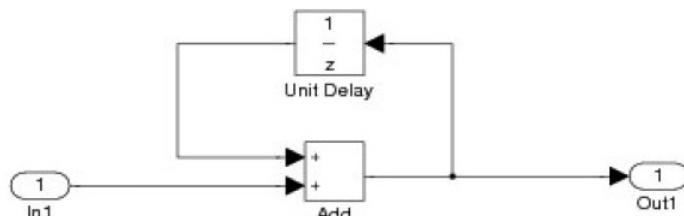


Figure 5.4. Formal process

SIMULINK provides different types of primitive *blocks*, namely: arithmetic, Boolean and relational blocks; control flow blocks such as loops and conditionals; temporal blocks based on the *unit delay* block, which is a reminiscent of the *pre* operator of the LUSTRE language [HAL 91]. The *unit delay* is generally used to specify temporal loops. An example of such a loop is given in Figure 5.4, where the output data flow (Out1) at t_i (current time step) is the sum of the input data flow (In1) at t_i and the output data flow (Out1) at t_{i-1} (previous time step). It should also be noted that the *unit delay* block is initialized to 0 at time t_0 .

EMBEDDED MATLAB (EML) is a high-level programming language that may be considered as textual form of SIMULINK. In essence, with the exception of

the *unit delay*, each primitive SIMULINK block has a corresponding construct in EML. However, the behavior of the *unit delay* block can be coded in EML using *persistent* variables. In an EML function, a variable declared as *persistent* retains its assigned value from the previous function call. As such, the temporal loop example given in Figure 5.4 can be specified in EML as follows:

```
function out1 = sum_loop(in1)
%#eml

persistent sum;

if isempty(sum)
sum = int32(0);
end

out1 = sum + in1;
sum = out1;

end
```

where *is empty* is a predicate, which is used to initialize the *sum* variable when function *sum_loop* is called first.

It is important to note that *for* loops cannot be of variable size in EML. In fact, the number of iterations to be performed cannot be determined at runtime. It should always be specified by a constant value. This restriction is mainly due to the synchronous hypothesis. Finally, it is also possible to reference an EML function within a SIMULINK model via a *user-defined function* block.

5.3.1.2. Simulink Design Verifier

SIMULINK DESIGN VERIFIER (DV) is a plug-in to Prover, which is a formal verification tool that performs reachability analysis tool based on the principle of *bounded model-checking* (BMC) and satisfiability (SAT¹² Solver). With DV, one can generate tests for SIMULINK, EML and Stateflow models according to model coverage and user-defined objectives. The underlying Prover engine also allows the formal verification of properties for a given model. Moreover, any failed proof attempt ends in the generation of a counterexample, which represents an execution path to an invalid state. A harness model is also generated to exploit the counterexample for debugging purposes. The underlying concepts are presented in the subsequent subsections.

¹² SAT is a technique enabling the satisfiability of a Boolean expression to be decided.

5.3.1.2.1. Transition systems

In Prover, a given problem is transformed into a transition system represented by the triplet (S, S_0, T) , where:

- S denotes the set of states;
- $S_0 \subseteq S$ is the set of *initial* states;
- $T \subseteq S \times S$ represents the transition relation.

As such, a property P to be satisfied is represented by a set of states denoting the valid states. The satisfiability of the property P for a given transition system is determined by showing that all states reachable from the initial states are included in P , i.e.:

$$\text{Reachable}_T(S_0) \subseteq P$$

where $\text{Reachable}_T(S_0)$ denotes the states reachable from S_0 using the transition relation T .

5.3.1.2.2. Model-checking and K-induction

Constructing an explicit representation of the reachable states space is in general difficult to achieve (in some cases even impossible). In Prover, the sets of states are therefore represented symbolically using predicates. As such, checking the non-reachability of a set of invalid states is reduced to checking the non-satisfiability of Boolean and linear arithmetic formulae.

For a sequence of states, s_0, \dots, s_n , $\text{path}(s_0, \dots, s_n)$ is a predicate determining that the sequence corresponds to a path through the graph of the transition relation:

$$\text{path}(s_0, \dots, s_n) = \forall i \in \{0, \dots, n-1\} : T(s_i, s_{i+1})$$

where T is the transition relation as described previously. Based upon this definition, the reachability problem for a transition system (S, S_0, T) and a given property P can be formalized as follows:

$$\forall n \geq 0 : \forall s_0, \dots, s_n : \text{path}(s_0, \dots, s_n) \wedge S_0(s_0) \rightarrow P(s_n)$$

where $S_0(s_0) \leftrightarrow s_0 \in S_0$. As can be seen, the reachability problem is reduced to showing that every state reachable from the initial state s_0 satisfies property P . However, it can be noted that the search space is limited so as to only consider paths of size n accessible from s_0 .

To resolve such types of problem, two methods are available, namely: *Bounded Model Checking* [BAI 08, BIE 03] and *Induction over time* [SHE 00].

The first method is generally used as a refutation technique, where counterexamples of size n are systematically sought. This technique is suitable for debugging purposes, i.e. finding errors in an invalid system. This can be formalized as follows:

$$bmc_n(s_0, \dots, s_n) = path(s_0, \dots, s_n) \wedge S_0(s_0) \rightarrow P(s_n)$$

This predicate is iteratively applied, whereby n is increased systematically until a counterexample is found (i.e. $bmc_n(s_0, \dots, s_n)$ is falsifiable). As a result, a counterexample represents the shortest path leading to an invalid state. However, this method will never terminate for a valid system (i.e. Property P is always satisfied). Indeed, n will be incremented until a completeness threshold is reached. However, determining a completeness threshold is costly. Furthermore, for relatively significant completeness thresholds, the exploration of the search space can be inefficient. Finally, these completeness thresholds do not exist for infinite state systems.

The second method uses a combination of *Bounded Model Checking* (BMC) and K-induction rule. By applying this induction rule, BMC is used to determine the invariance of property P in the first n states of any execution. As such, counterexamples of size n can still be detected. The induction rule is afterwards generalized to show that if P holds in every state of every execution of length n , then every successor state also satisfies P . This is formalized as follows:

– *Base case*:

$$\forall n \geq 0: \forall s_0, \dots, s_n: bmc_n(s_0, \dots, s_n) \text{ is a tautology};$$

– *Induction hypothesis*:

$$\forall n \geq 0, k \geq 0: \forall s_k, \dots, s_{k+n}: ih_n(s_k, \dots, s_{k+n}) = path(s_k, \dots, s_{k+n}) \wedge \forall 0 \leq i \leq n: P(s_{k+i})$$

– *Induction step*:

$$is_n(s_{k+n}) = \forall s_{k+n+1}: T(s_{k+n}, s_{k+n+1}) \rightarrow P(s_{k+n+1})$$

As in BMC, the bound n is increased until either a counterexample is detected in the first n states of an execution or the given property is shown to satisfy the induction schema. In other words, n is increased, starting from 0, until

$$(\forall s_0, \dots, s_n: bmc_n(s_0, \dots, s_n)) \wedge (\forall k \geq 0: \forall s_k, \dots, s_{k+n}: ih_n(s_k, \dots, s_{k+n}) \rightarrow is_n(s_{k+n}))$$

is satisfied. If this is the case, then the system is considered valid. Otherwise, the base case provides a counterexample.

The k-induction rule is considered to be *sound*. However, it is considered to be *complete* only when restrained to loop-free paths. Predicate *path* is therefore modified as follows:

$$\text{path}(s_0, \dots, s_n) = \forall i \in \{0, \dots, n-1\} : T(s_i, s_{i+1}) \wedge \forall j \in \{0, \dots, n\} : i \neq j \rightarrow s_i \neq s_j$$

5.3.1.2.3. Satisfiability

In Prover, the non-reachability problem for a set of invalid states is transformed into deciding whether a mathematical formula is satisfiable or not. The formulae manipulated consist of Boolean propositions and linear arithmetic predicates, namely:

- a constant $c \in Q$ is a mathematical term;
- a variable v over Q is also a mathematical term;
- $c \cdot v$ is a mathematical term (multiplication);
- if t_1 and t_2 are mathematical terms, then $t_1 + t_2$ and $t_1 - t_2$ are also mathematical terms;
- a Boolean proposition $A \in \{\perp, \top\}$ is a mathematical formula;
- if t_1 and t_2 are mathematical terms, then $t_1 \diamond t_2$ is a mathematical formula, where $\diamond \in \{=, \neq, <, >, \leq, \geq\}$;
- if ψ_1 and ψ_2 are mathematical formulae, then $\neg \psi_1$ and $\psi_1 \wedge \psi_2$ are mathematical formulae.

The logical connectors \vee , \rightarrow and \leftrightarrow are derived from \neg and \wedge according to standard definitions.

An approach for deciding the satisfiability of a mathematical formula is to examine all the satisfying assignments of the Boolean variables in the formula and to resolve the resulting linear constraints for each of these assignments. This approach is a NP-hard problem. The proof engine implements an efficient solver, which combines different SAT techniques such as the Stålmarck saturation method [STA 00], Reduced Ordered BDDs¹³ (*Binary Decision Diagrams*) [BRY 86], linear

¹³ Canonical representation of state graphs and properties in Boolean form. This method of representation has enabled the development of very efficient verification algorithms for temporal tree logic, and a significant increase in the size of problems processed.

programming techniques, and constraint propagation. In practice, even if a satisfiable formula contains non-linear predicates, the proof engine often manages to make it decidable. In Prover, integers restrained to finite domains are converted to bit vectors and binary arithmetics are used to perform all operations. This method is able to handle non-linear arithmetic over finite domains.

5.3.1.3. Implementation through plug-in

For the model designer (see Figure 5.5), the proof operative mode for a given property is as follows: Let F be a function, represented by a SIMULINK block for which we would like to prove a certain property P :

- the output of F function is connected to a block, which formalizes property P in SIMULINK or EML;
- P is defined as a predicate such that it always returns *true* when hypotheses H specified on the input data flows of the model are satisfied. By construction, P is connected to a specific *Assertion* block of the SIMULINK DV environment;
- H is a predicate defined as a SIMULINK block, which enables the description of complex constraints: value domain specification, list of values at each cycle, etc.). By construction, H is connected to a *Proof Assumption* block;
- *Proof assumption* blocks are used to constrain the input data flows of the model during the proof construction process;
- an *Assertion* block verifies that its input is always true, i.e. that the predicate P is always satisfied.

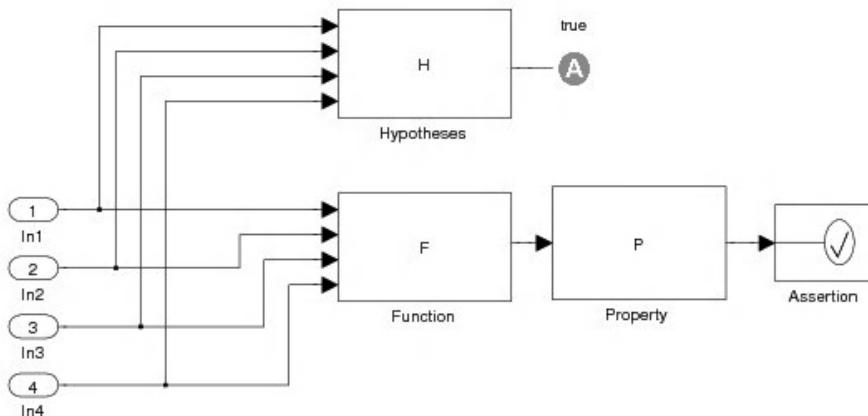


Figure 5.5. General proof outline

In principle, this operative mode is very simple and relates to the formalization and connection of an observer to the inputs/outputs of function F to be proved. It therefore facilitates access to proof technologies.

However, this simplicity is sometimes only apparent. In fact, the writing of assumptions often comes back to precisely modeling environmental constraints, for example kinematic constraints for movement of trains. The verification flow can be “open” or with “loopback” (see Figure 5.5).

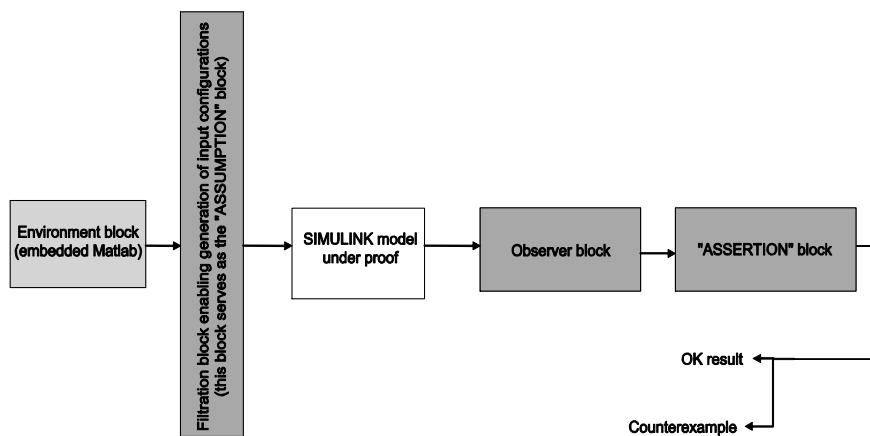


Figure 5.6. Verification flow without looping

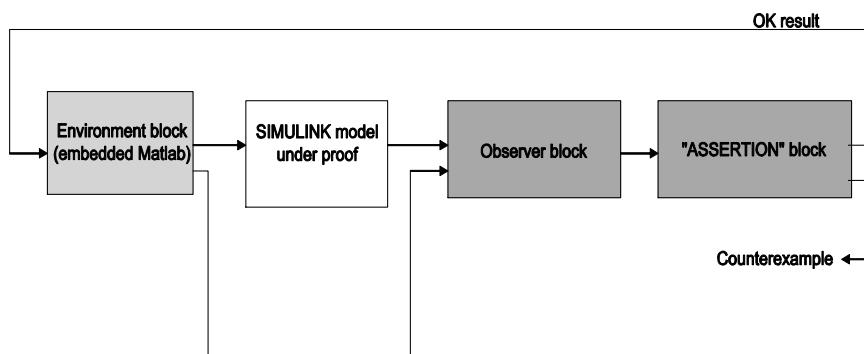


Figure 5.7. Verification flow with loopback

When the flow is open, the environment block describes some conditions and possible train movements.

The proof tool enables positive determination of respect of properties for all these configurations or counterexamples to be found.

Use of loopback enables temporal sequences to be modeled.

5.3.2. Case study

Our case study concerns a “Train tracking” function for an automatic train protection system (ATP) developed by THALES Railway Systems. The target system is a subway line located in Paris (France), which is considered to be even more complex than the “METEOR” system [MAT 98]. On the one hand, it has to manage fully automated trains communicating via a dedicated network (Communication Based Train Control). On the other hand, it has to handle a traditional signaling system for non-automated and non-communicating trains via classic Track Vacancy Devices (TVD).

More generally, the global system, which includes the protection system to be specified and developed, signaling, equipped trains and communication, is in fact a distributed system in which these are numerous random events to take into account:

- “entrance” of a train in the system (or in the train tracking zone for our function);
- location message broadcasting on the track by equipped trains;
- occupancy/freeing-up of track circuits by unequipped trains;
- movement of switches;
- loss of transmission linked to network latency or transient errors on equipped trains, which can become “mute” and re-enter the system.

Also, it is necessary to take into account chance phenomena, which are due to the technology of the trains and location especially, as well as “unexpected” movements such as reversal, etc.

The “train tracking” function must first of all construct a coherent and safe image of train positions and TVD occupancy, taking into account network delays relative to transmission, faults, or train location failures. The image is called “cartography”. This informal notion overlaps with that of “maintenance of the system in a safe state” and must be formalized by some properties to be proven. These properties are not trivial. It is moreover necessary to note that asynchronism will determine the size of the temporal window within which the system must be observed. These points are detailed in the section dealing with formalization of target properties (section 5.3.3.4).

In addition, the tracking function fulfills another system functionality: calculation of movement authorizations for a number of equipped trains at each cycle. At each cycle, an equipped train receives a maximum distance (or a target point according to the system concepts implemented), which determines its movement authorization zone or Limit of Movement Authorization (LMA). The LMA is calculated by taking into account the series of obstacles, which can be found on the track: another train, a restrictive signal, a fixed obstacle, or even an uncontrolled switch. The number of conditions that the model will have to integrate is ascertained to show that the Limit of Movement Authorization is always limited by an obstacle.

An additional complexity factor lies in the track topology and management of possible paths, which withstands train movement, and also in the distributed nature of the track, which is organized in tracking sections. Sections “swap” the lists of trains that they track. The train tracking function must also be capable of simultaneously managing 32 trains for a given section.

Train tracking is an SSIL4¹⁴ (Software Safety Integrity Level) function. Our study has been conducted as a safety principles validation activity in the system specification phase.

The process that we have proposed is organized into three activities: the first activity, conducted by the system designers, consists of setting out the principles of functioning and maintenance of the global system in a safe state along with high-level properties.

Functioning principles at this stage are often expressed in the form of scenarios, structured by large functional blocks.

The second activity is mathematical formalization of functioning principles and properties. This task of formalization is necessary to characterize objects, which are managed by the system, as well as operations on these objects, without ambiguity. In the case study, the mathematical specification has been rewritten in terms of set theory and first-order logic and properties are expressed in this standard.

By way of an illustration, large objects of the tracking function are elements of the calculated cartography, including train positions, signal positions and states, with accuracy constraints and Limit of Movement Authorization.

¹⁴ The safety level of software in the rail industry is named Software SIL. The SSIL is an enumeration of five values: 0, 1, 2, 3 and 4. SSIL0 signifies that the software has no impact on system safety, SSIL2 that it may have an impact, such as injuries and important hardware damage, and SSIL3-4 indicates that there may be an impact on the system and people's lives. It should be noted that SSIL1 is not used and that levels 3 and 4 are seen as one single level (practice even).

The properties retained for study must be expressed in this referential.

Thus have been formalized:

- coherence properties between the calculated cartography and the real situation, taking into account transmission delays and the set of events listed above. For example, at any moment, the cartography of a tracking section must include the positions of all equipped and communicating trains physically in the section. In addition, every non-equipped train must be tracked. Finally, no train must be left out. This last property expresses the maintenance of safety of the cartography, in the presence of events such as transmission loss, loss of location of a train, and crossing over switch zones. The cartography must also be correct in terms of train positions in the associated reference point, taking into account all measurement errors and transmission delays. Formalization of tracking properties was very complex because it involved finding applicable safety criteria. Formalization and proof activities were useful for showing the coherence of cartography with the real situation at any moment as well as the completeness of situations taken into account by the tracking function;
- fundamentally, the tracking function constructs an order of trains, which must be maintained at all times on a complex topology (meaning authorized movement on the segments and switch positions);
- safety properties associated with the LMA calculation are easier to identify. The limits of movement authorization must be limited by obstacles either fixed (for example, signals) or mobile (equipped train or non-communicating train).

To guarantee system progression, it will be necessary to show that every equipped train receives a limit of movement authorization (liveness property) and that if this limit of movement authorization is strictly less than the maximum distance that a train can travel, then the limit respects the safety property stated above (coherence property).

The third activity is constituted of modeling and proof of properties on the model. This activity includes several sub-activities in reality. First of all, it involves obtaining a traceable model with the specification in mathematical language, and then validating it by simulation in such a way as to ensure the suitability and good functioning of the model with functional scenarios.

Verification and proof of property activities are led by the model already “validated” by intensive simulation.

NOTE 5.1.– The process is iterative. The process has been run integrally with design studies and a key point is traceability between the textual description and the mathematical description. It would be possible to have the same approach in taking a

structured analysis type functional breakdown as input. In both scenarios, the links between the two levels of description must be established by construction and rigorously maintained. Ideally, the design and safety assurance teams must determine the pivot description, on which the functional safety analyses notably will be carried out. *Ultimately*, the SIMULINK model must be able to be traced with the pivot description – observable objects (definitions, types) and objects, scenarios especially.

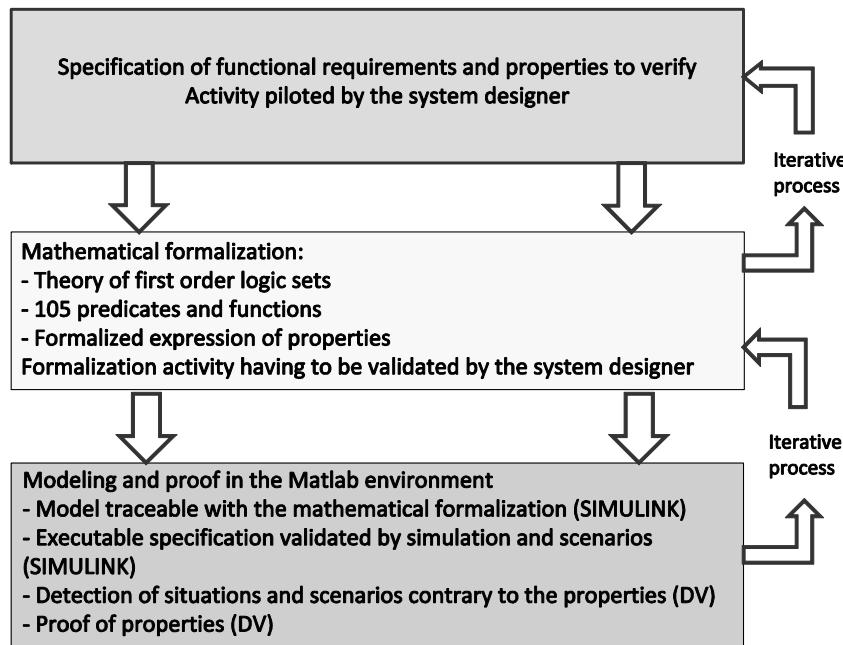


Figure 5.8. Global process

5.3.3. Modeling

The model designed in SIMULINK is very significant. In essence, it is composed of about 242 proprietary basic blocks for custom-made operators, of 60 *Assertion* blocks for the proved properties and all *Proof Assumption* blocks necessary for the hypotheses and intermediate lemmas required during the formal verification process. The model complexity mainly resides in the choice of data representations to properly capture the different concepts elicited in the mathematical referential. This also has an impact on the operators as well as the required properties necessary to characterize the chosen data structures. The following sections describe the formalization carried out to transform the mathematical referential into an executable specification.

5.3.3.1. Data representation

In the referential, various mathematical principles are introduced to formulate in an unambiguous way the different concepts relative to the “Train Tracking” function. This involves the specification of the basic notations that allow to:

- represent all the data relative to the tracking in the form of very simple data structures (fixed size integer vectors and matrices) so as to guarantee compatibility with the proof engine;
- describe the locations of fixed obstacles;
- describe train locations, while considering all possible errors (errors relative to the physical location mechanism, to the measurement precision and to message latency);
- describe the limits of movement authorities (LMAs) computed for each automated train.

We here show how these concepts are coded in the SIMULINK model.

Oriented graph. The topology of the railway track is formally specified as an *oriented* graph. In SIMULINK, the oriented graph is defined as a combination of scalars and vectors of unsigned integers. The *basic* definition may be denoted as $G = (V_\omega, S_\omega, H_\omega, T_\omega)$, where V_ω represents the number of *vertices* (or *nodes*) and S_ω the number of *arcs* (or *directed edges*). To stick to the railway jargon, *arcs* are referred to as *segments*. Each vertex v of the graph is represented by a strictly positive integer s.t. $v \leq V_\omega$. Similarly, each segment a is a strictly positive integer s.t. $a \leq S_\omega$. H_ω and T_ω are vectors of size S_ω and are indexed according to the segment identifier a . In fact, each segment a is associated with an ordered pair (x, y) of vertices and is considered to be directed from x to y (where $x, y \leq V_\omega$). As such, x is called the *tail* and y is called the *head* of the segment. Vectors H_ω and T_ω are therefore used to store the set of ordered pairs (x_a, y_a) s.t. $H_\omega = [x_1, x_2, \dots, x_n]$, $T_\omega = [y_1, y_2, \dots, y_n]$ and $n = S_\omega$. We write $H_\omega(a)$ and $T_\omega(a)$ to respectively denote the head and tail of segment $a \in [1..S_\omega]$. Oriented graph G is also subjected to the following well-formedness constraints:

1. $\forall a \in [1..S_\omega], T_\omega(a) \neq H_\omega(a)$
2. $\forall a_1 \in [1..S_\omega], \exists a_2 \in [1..S_\omega],$

$$(H_\omega(a_1) = T_\omega(a_2) \wedge T_\omega(a_1) = H_\omega(a_2)) \vee$$

$$(H_\omega(a_1) = H_\omega(a_2) \wedge T_\omega(a_1) = T_\omega(a_2)))$$
3. $\forall v \in [1..V_\omega], \text{card}(\{a \in [1..S_\omega] \mid H_\omega(a) = v \vee T_\omega(a) = v\}) \leq 3$

Condition (1.) specifies that G must not contain loop transitions (i.e. segment which starts and ends on the same vertex). Condition (2.) states that symmetric transitions as well as cycles of less than two vertices are not allowed. Condition (3.) prescribes that a vertex can be connected to at most three segments. With this last restriction, railway switches present on the track can be modeled correctly. In fact, a *commutation point* is modeled as a vertex incident to three segments. We therefore have a triple of segments $(p, h_l, h_r) \in [1..S_\omega] \times [1..S_\omega] \times [1..S_\omega]$, where p is being referenced as the *point of switch*, while h_l and h_r are the *left-hand heel* and *right-hand heel* respectively. As such, the definition of graph G is extended with: SW_ω to represent the number of railway switches on the track; vectors SW_P , SW_{HL} and SW_{HR} to store the set of triples (p_i, h_{l_i}, h_{r_i}) s.t. $SW_P = [p_1, \dots, p_n]$, $SW_{HL} = [h_{l_1}, \dots, h_{l_n}]$, $SW_{HR} = [h_{r_1}, \dots, h_{r_n}]$ and $n = SW_\omega$. We write $SW_P(s)$, $SW_{HL}(s)$ and $SW_{HR}(s)$ to respectively denote the point, the left-hand heel and the right-hand heel of a railway switch $s \in [1..SW_\omega]$.

Undirected path. The notion of *undirected* path is mainly used to designate the different possible routes that can be followed by a given train while moving along the railway track. An *undirected* path is basically a path in which the segments (or edges) are not all oriented in the same direction. In the SIMULINK model, an *undirected* path (or path for short) of n segments is represented by an unsigned integer vector of fixed size m denoted as P_s , where $n \leq m$. In fact, even though it is possible to specify vectors of variable size in SIMULINK, they are not supported by DV due to the synchronous hypothesis. The fixed size m is determined according to the length of the train and the location errors or according to the maximum distance that can be covered by the train for a period of seven seconds (the maximum authorized delay), whichever is greater. Hence, $P_s(1), \dots, P_s(n)$ corresponds to a sequence of adjacent segments, while $P_s(n+1), \dots, P_s(m)$ are padded with zero whenever $n < m$. Moreover, each path P_s is assigned an orientation to reflect the intended direction movement. This orientation is transposed on each of its corresponding segments such that an additional unsigned integer vector P_o of fixed size m is required. Each orientation $P_o(i)$ is determined according to whether or not the corresponding segment $P_s(i)$ is oriented in the same direction as the path: $P_o(i) = 1$ if $P_s(i)$ is in the same direction; $P_o(i) = 2$ if $P_s(i)$ is in the opposite direction; $P_o(i) = 0$ if $i > n$ and $i \leq m$. In the following, the given set $\Omega = \{0, 1, 2\}$ denotes the domain values for a given orientation o . For graph G , a path is therefore modeled as a pair (P_s, P_o) and the corresponding given set is denoted as \mathbb{P}_{path} .

Finally, the well-formedness of an *undirected* path is also limited by the fact that adjacent segments cannot correspond to the left-hand and right-hand heels of the same switch simultaneously. In particular, a train can traverse a railway switch from point to heel (diverging movement) or heel to point (converging movement), but not from the left-hand heel to right-hand heel (or vice versa). Based on how the railway switches are commuted, the set of all possible routes for graph G may therefore

change from one application cycle to another. To represent the commutation status of the railway switches on the track, two Boolean vectors K_{HL} and K_{HR} of size SW_ω are introduced s.t. $K_{HL}(s)$ and $K_{HR}(s)$ denote the status for the left-hand and right-hand heels of switch $s \in [1..SW_\omega]$ respectively. It should also be noted that $K_{HL}(s)$ and $K_{HR}(s)$ are mutually exclusive.

Point on segment. As its name suggests, the notion of *point on segment* describes a location on the graph according to a given segment. It is characterized by an *abscissa* and an orientation that determines whether or not it is defined in the same direction as its segment of reference. A point on segment is thus modeled as a triple $(d, a, o) \in \mathbb{N} \times S_\omega \times \Omega$, where d denotes the abscissa, a the segment of reference and o the corresponding orientation. The *tail* vertex (i.e. vertex from which the segment emanates) of segment s is referenced as origin to determine the positioning of abscissa d s.t. $d \leq Lg_\omega(s)$. In the SIMULINK model, vector Lg_ω of size S_ω is introduced to store the length (i.e. between 15,000 and 100,000 cm) associated to each segment of graph G . We write $Lg_\omega(s)$ to denote the length for segment s . The given set of all points on segment is defined as $P_{seg} \subseteq \mathbb{N} \times S_\omega \times \Omega$.

Location area. The first model created for the “Train Tracking” function was based on the notion of *location areas*. The notion of *location area* is introduced to represent where signaling lights and trains are located on the track. Location areas are also used to consider the LMAs assigned to each automated train. A location area is a path limited by two *abscissas* to mark the beginning and end of the area. In the SIMULINK model, a location area is therefore designed as a triple $(P, Abs_s, Abs_e) \in P_{path} \times \mathbb{N} \times \mathbb{N}$, where P denotes the associated path as described previously. The unsigned integer scalars Abs_s and Abs_e specify the corresponding abscissas such that the beginning and end of the area are characterized by $(Abs_s, P_s(1), P_o(1)) \in P_{seg}$ and $(Abs_e, P_s(n), P_o(n)) \in P_{seg}$ respectively (where $P_s = \prod_l(P)$, $P_o = \prod_l(P)$ and n the number of segments in path P_s)¹⁵. The given set for location areas is noted as L_{area} .

Signaling light. A signaling light is modeled as a pair $(l, b) \in L_{area} \times B$, where l denotes a small location area and b a Boolean value that determines whether it is in a permissive (i.e. green signal) or a restrictive (i.e. red signal) state. We write $S_{sig} \subseteq L_{area} \times B$ to denote the given set for signaling lights.

Train. Finally, a train is defined as a tuple $(l, v_s, b_d, e_d) \in L_{area} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N}$, where l denotes its location area, v_s its actual speed (in cm.s^{-1}), b_d a Boolean value determining its direction movement (i.e. whether it is moving forward or backward), and e_d a message latency (in application cycle) representing the aging factor associated to the information received from the train. Notation $T_{train} \subseteq L_{area} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N}$ is used to denote the given set for trains.

¹⁵ Notation $\prod_i(T)$ is used to access element i of a given tuple T .

As can be seen, each object present on the track is characterized by a location area: i.e. location areas for all fixed or tracked objects (signaling lights, TVD for non-automated trains or for automated trains that become mute, etc.) as well as the limits of movement authorities (LMAs) for automated trains. The location area is, by construction, well-formed and safe compared to the real position of the object. In particular, uncertainties due to measurements and latencies are catered for by an increase in the size of the location areas associated with each automated trains.

This relatively simple data representation choice allows the formalization of non-collision properties in terms of empty intersections between location areas allocated to automated and non-communicating trains. For optimization reasons, this principle was modified afterwards and it was necessary to handle a temporal observation window for the periodic locations of each tracked objects. This point has an impact on the proof construction process. In essence, for a data representation based on location areas, it comes back to proving an invariant in the zones allocated to each cycle, while in the temporal version, it comes back to proving the non-reachability of dangerous situations.

5.3.3.2. Model operators

Based on the concepts introduced in the previous section, the “Train Tracking” can essentially be seen as a set of basic operators for resolving graph problems. In the SIMULINK model, a set of basic operators is therefore defined to properly encode the high level functionalities and properties. These operators mainly work on the different data structures defined in previously. In particular, they allow to compute new location areas/points on segment, to infer a certain number of characteristics associated to these data structures (e.g. length of a location area) and to provide predicates that can be used when evaluating properties (e.g. well-formedness constraints). For instance, the following is a non-exhaustive list of the operators coded in SIMULINK/EML:

- LA_Inter(l_1, l_2): determines whether location areas l_1 and l_2 intersect;
- LA_Direction (l_1, l_2): determines whether location area l_1 is in the same direction as location area l_2 ;
- Pseg_shift (e, d, K_{HR}, K_{HL}): translates point on segment e according to distance d in the direction pointed by $\Pi_3(e)$. The translation is also performed w.r.t. the status of the railway switches on the track;
- LA_Coherence(l): determines whether location area l is the well-formed;
- LA_Comm(l): determines whether location area l only traverses commuted railway switches (if any).

In the SIMULINK model, only discrete state operators were used to code all the functions necessary for the “Train Tracking” system. *Selector* and *Assignment*

blocks were extensively used in conjunction with *For-Iterator* blocks to efficiently manipulate vectors/matrices of relatively large size. Conditional blocks such as *If-Then-Else*, *Merge* and *Switch* were generally applied to exert a certain control flow in basic operators. The use of *unit delay* blocks and temporal loops allowed the proper modeling the temporal features inherent to train tracking.

5.3.3.3. Environment modeling

It was necessary to model the environment so as to be able to simulate the model in realistic conditions and especially to formalize the assumptions for proof of properties with the help of constraints applying to:

- train movements, which were constrained by modeling the kinematic rules and taking into account train capacities (for example, the possibility of reversing). They are calculated at each cycle of the model;
- changes of state of signaling elements, which have also been modeled: switches, CDV and lights;
- hardware failures (equipped train becoming mute, failure of a CDV, uncontrolled switch state) are taken into account in the environment;
- transmission errors (failures of omission and latency) are also modeled.

In addition, the control model interacts with the environment model (looped model). The environment model must apply the orders or constraints resulting from the calculation.

For example, the running of a train is constrained by the kinematics, and by the LMA calculated at each cycle.

5.3.3.4. Modeling of properties

A property can be modeled by a SIMULINK block or EML function.

The clauses that formalize the properties generally comprise some logic connectors and quantifiers.

The variables that are quantified are “translated” by the tool in value ranges for these variables, modulo the constraints and hypotheses expressed in the *Proof Assumption* blocks.

The properties that we have modeled are complex. However, it is also necessary to completely describe all hypotheses under which these properties must be verified.

Hence, the requirement “an LMA allocated to a train must not contain an obstacle” is not enough.

The formalization must be completed as follows:

HYPOTHESIS 5.1.– For every equipped train, the location zone of the train is coherent (that is to say – it only contains connecting segments and crosses the commuted switches).

HYPOTHESIS 5.2.– For every equipped train, the location zone of the train is in empty intersection with location zones associated with other trains.

HYPOTHESIS 5.3.– Every equipped train respects the restrictive signals.

Proof objective: the LMA contains no obstacle.

We do not describe the SIMULINK/EML formalization of this property. Formalization of the proof objective requires breakdown of the proof schematic into smaller properties and choosing an appropriate order when running the proof of smaller properties.

In the previous example, it is of interest to prove first of all that the subassembly of the model, which calculates the location zones is correct compared to hypothesis 5.1.

Likewise, it must be proved that the subassembly of the model, which calculates the movement authorizations, only produces zones that are coherent and do not intersect mutually. Hypothesis 5.3 must be shown by the environment model. In reality, the process of proof of complex properties is that the breakdown is proved.

In other words, the proof schematics that the designer intuitively constructs can thus be validated.

In our case study, this was crucial because the principles of carrying out train tracking defined by the designer were new. Another feature of this case study was that the properties sought were high level and had been formulated independently of the modeling choice.

The property previously described is formally specified in the following manner:

$$\begin{aligned}
 & \left. \begin{aligned}
 & (\forall t_0 \in \mathcal{T}_{train}, \forall l \in \mathcal{L}_{area}, \\
 & l = \Pi_l(t_0) \implies \\
 & \text{LA_Coherence}(l) \wedge \text{LA_Comm}(l)) \implies
 \end{aligned} \right\} \text{Hyp 1} \\
 & \forall t_1 \in \mathcal{T}_{train}, \forall l_1, l_{LMA} \in \mathcal{L}_{area}, \\
 & l_1 = \Pi_l(t_1) \implies \\
 & \left. \begin{aligned}
 & (\forall t_2 \in \mathcal{T}_{train}, \forall l_2 \in \mathcal{L}_{area}, \\
 & t_1 \neq t_2 \wedge l_2 = \Pi_l(t_2) \implies \exists \text{LA_Inter}(l_1, l_2)) \implies
 \end{aligned} \right\} \text{Hyp 2} \\
 & \left. \begin{aligned}
 & (\forall s \in \mathcal{S}_{sig}, \forall l_s \in \mathcal{L}_{area}, \\
 & l_s = \Pi_l(s) \implies \\
 & \Pi_l(s) \vee \exists \text{LA_Direction}(l_s, l_1) \vee \exists \text{LA_Inter}(l_s, l_1)) \implies
 \end{aligned} \right\} \text{Hyp 3} \\
 & l_{LMA} = \text{Train_LMA}(t_1) \implies \\
 & \nexists l_{OBS} \in \text{Obs}_{\omega}(t_1), \text{LA_Inter}(l_{OBS}, l_1)
 \end{aligned}$$

where $\Pi_i(T)$ denotes the i element of a given tuple T and $\text{Obs}_\omega(t_1)$ is defined as follows:

$$\begin{aligned} \text{Obs}_\omega(t_1) \equiv \{ l \in \mathcal{L}_{area} | & (\exists t_2 \in \mathcal{T}_{train}, t_2 > t_1 \wedge \Pi_1(t_2) = l) \vee \\ & (\exists t_2 \in \mathcal{T}_{train}, t_2 < t_1 \wedge \text{Train_LMA}(t_2) = l) \vee \\ & (\exists s \in \mathcal{L}_{sig}, \Pi_1(s) = l \wedge \lceil \Pi_2(s) \wedge \\ & \text{LA_Direction}(\text{Train_LMA}(t_1), l)) \} \end{aligned}$$

5.3.4. Modeling

The highest level properties that have been proved are as follows:

- the allocated movement authorization zones guarantee the property of non-collision and are of a length, which is only limited by the achievement of the maximum length authorized or by an obstacle (other train, restrictive light). This property guarantees the progress and liveliness of the system;
- the train tracking function does not lose any trains. This property has been very difficult to demonstrate. It has been broken down according to a principle of construction of an order of trains and demonstration of the conserving of the order of trains, in taking into account withdrawal and reinsertion of trains linked to latency time and communication faults.

A sum total of 57 properties, relating to objects modeled and train tracking operators have been demonstrated with the aid of SIMULINK DV:

- proofs of correction of operations and manipulations applying to zones;
- proofs of coherence between the physical situation (position and order of trains, position and state of track equipment) and cartography calculated by the train tracking function;
- proof of restrictiveness of the cartography calculated in relation to the physical situation and communication loss or latency events;
- proofs of detection of dangerous situations and emergency control of the system.

5.4. Experience feedback and methodological aspects

5.4.1. Modeling rules and convergence control

The aim of modeling rules is to guarantee compatibility of the model with the proof process. The tool editor (The Mathworks) provides rules of syntactical order, but we were led to complete them and develop a method of incompatible block detection.

To detect problematic blocks, the idea is to duplicate the functions to analyze and prove the equality of values returned. Figure 5.8 illustrates this principle.

The function “Function1” shows the method applied for numerical values, just as function “Function2” shows the method used for Boolean values.

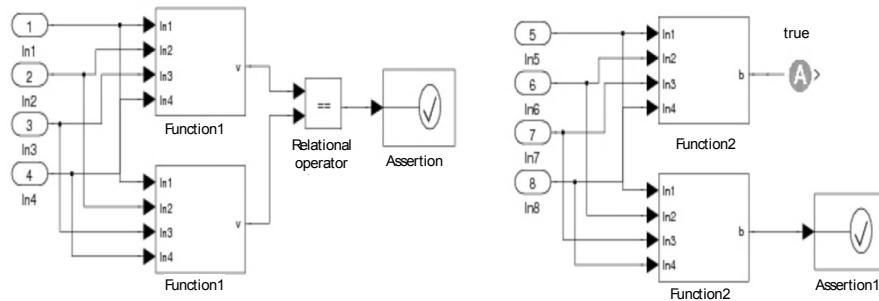


Figure 5.9. Example

Detection of problematic blocks

Generally, for compatible blocks, the proof must be carried out in a reasonable time. In the opposite case, the function tested is considered as problematic.

For problematic functions, the same process is applied to each of their sub-blocks. These tests are reiterated up to chancing upon atomic blocks (that is to say, blocks of Simulink libraries) and detecting incompatible atomic blocks.

This strategy has enabled identification that the following blocks are partially compatible:

- selectors;
- Boolean operators;
- constants encoded directly in the selectors;
- “Switch” block.

To get around these problems, selectors are improved with *Assumption* blocks in the index and output. These *Assumption* blocks specify the range of values taken by the index and intervals of values that must respect the output. The Boolean operators have been rewritten with the help of bit manipulation operators on integers (“BITWISE” blocks).

Likewise, the “Switch” blocks are also rewritten with the help of “BITWISE” blocks. Involving the constants encoded directly in the selectors, they are externalized through the use of “CONSTANT” blocks, and then re-linked to the selectors through explicit index ports.

5.4.2. Modular proof phase

The objective of this phase is to verify some properties on the functional blocks of the model either at the leave level (basic blocks) of the model or on the intermediate level blocks. In the context of the experiment, we have shown the coherence of custom blocks:

- zone length;
- zone inversion;
- zone reduction;
- zone extension;
- path length;
- path inversion;
- path reduction;
- path extension.

This phase has enabled it to be shown that zone coherence properties are conserved by these operators. The proof has been carried out on all possible zones of the graph which models the topology.

This mode of use is very interesting: it is possible to construct proven operators, which are free of execution error (access to data manipulated by these operators).

NOTE 5.2.– This method should be used for all critical operators.

The verification procedure for critical operators can be organized as follows:

- (1) identify the critical operators (specification level AMDE procedure);
- (2) for each critical operator:
 - formalize the target correction properties (clauses),
 - translate the properties into “Embedded MatLab or Simulink” observers for DESIGN VERIFIER,
 - identify the hypotheses or constraints applicable to block input,

- formalize the constraints in the form of a block, which produces input (the equivalent of an “Assumption” block) in Embedded MatLab (or Simulink),

- connect the block input to prove to the hypotheses block output,
- connect the block output to prove to the observer block,
- connect the observer block to the corresponding assertion block;

(3) launch the proof with DESIGN VERIFIER:

- the results are either a positive proof result, or a counterexample,
- if needed, modify the module until a positive result is obtained.

In the case of non-convergence, identify the constraints or properties to modify.

5.4.3. Proof of global properties

During analyses undertaken on the Simulink model, several proof strategies have been used:

- a “raw” strategy, which consists of directly proving high-level properties with DV without the introduction of supplementary hypotheses or possible optimizations;
- a “breakdown of properties” strategy, where the high level properties are proved with DV with the aid of intermediate lemmas placed in hypotheses. The veracity of these lemmas is also established with DV;
- a “proof by induction” strategy, where the proof is carried out by taking inspiration from the proof by induction schematic. For a given property, DV is used to show the basis case, and then the inductive case.

The “raw” strategy is not very satisfying because during demonstration of global properties, it requires significant memory consumption (use of more than 8GB of RAM) with very long proof times (indeterminate, even).

This section presents the techniques developed to validate the model in its entirety. The methods that are used to reduce proof times and memory consumption during the demonstration of global properties are indicated.

5.4.3.1. Breakdown of properties into sub-lemmas

As previously mentioned, some global properties are relatively complex and generate a considerable memory consumption and very long proof times (number of states too significant for DV). To remove this point, the property to prove is broken

down into intermediate lemmas, the goal of which is to guide DV during the proof demonstration process.

These intermediate lemmas are put in “Assumption” during proof of global properties. They are also the object of a demonstration to guarantee their veracity. For example, to prove the coherence of an inverted zone, coherence is demonstrated first on path inversion, then on the abscissas characterizing the beginning and end extremities of the zone.

Figure 5.9 illustrates the proof schematic thus used. The “Hypotheses” block contains all the lemmas used, whilst the “Property” block encapsulates the global property to prove.

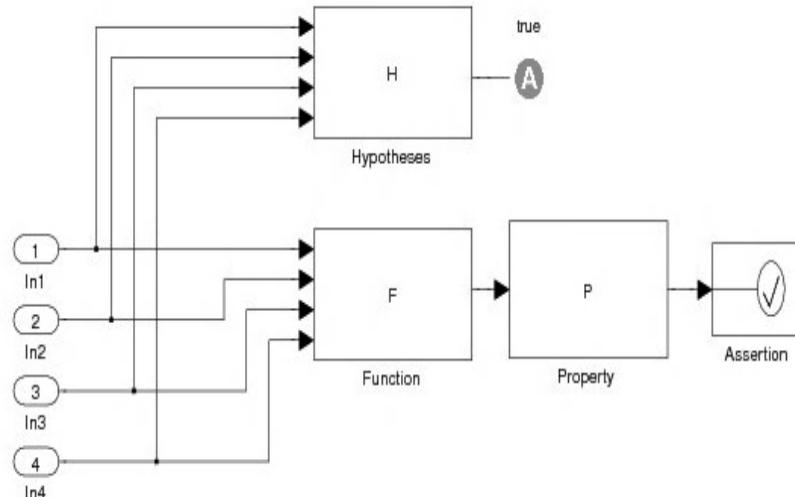


Figure 5.10. Proof schematic

5.4.3.2. Proof by induction

The additional technique adopted to reduce proof time relies on the principle of proof by induction used in mathematics. Indeed, even if DESIGN VERIFIER implicitly carries out the induction, installing an inductive schematic enables the initial property to be broken down and other intermediate lemmas to be identified.

Implementation of this approach requires two models derived from the initial model to be implemented: one for the basis case (Figure 5.10), the other for the inductive case (Figure 5.11).

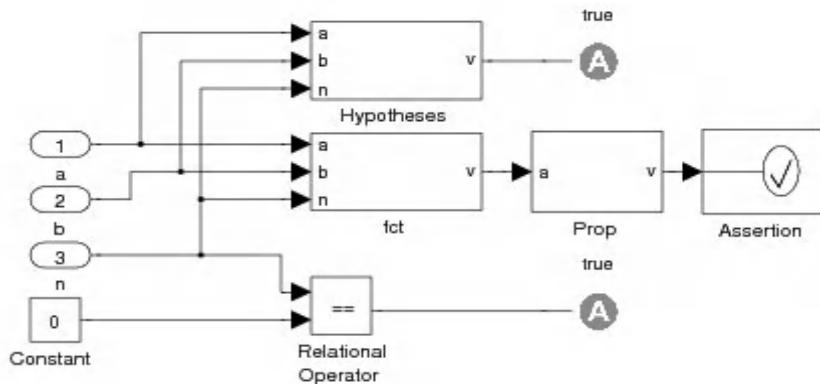


Figure 5.11. Basis case

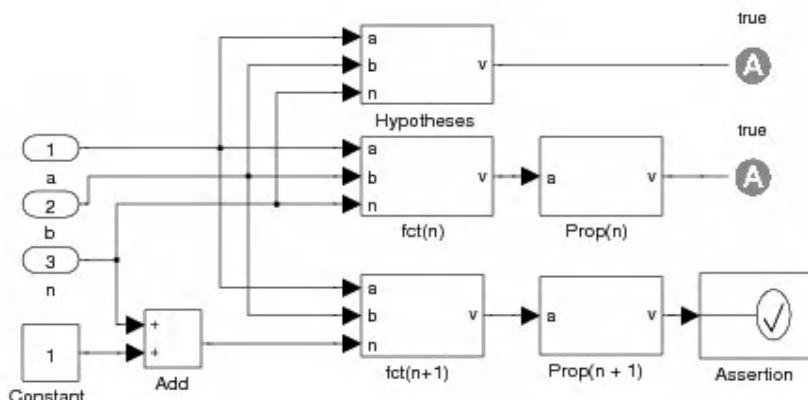


Figure 5.12. Inductive case

5.4.3.3. Other additional techniques

As the SIMULINK model created is an executable implementation, the data structures manipulated are quite complex and involve different types of representation (for example, tables, matrices, uint32, Boolean, etc.), necessitating the use of a large variety of primitive operators.

As a result, by specifying the basic properties that characterize these primitive operators such as intermediate lemmas, memory consumption, and even proof times may be reduced.

The basic properties used are as follows:

- properties on the arithmetic operators;
- properties on the relational operators.

5.4.3.4. Model modification

With a view to optimizing memory consumption and proof times, the SIMULINK model has also undergone some modifications to simplify the processing of manipulated data. The model thus brings pre-calculated data sets into play, the objective of which is to reduce the complexity of the calculation required. For example, zone extension is achieved from a set of pre-calculated paths contingent upon the directed graph considered.

Considering the proof process, the zone extension function benefits from properties relating to the graph concerned, to induce the properties in zone extension. In effect, the pre-calculated paths for a given graph respect the following properties:

- each path is single;
- each path is coherent;
- each path is linear (no repetition at segment level);
- all paths equal to or of a smaller size than n are considered, where n represents the maximum number of support segments associated with a zone;
- all paths cover the track.

5.4.4. Detection of counterexamples

SIMULINK DV can be used in two modes: for verifying properties or for detecting counterexamples to a given property.

In the “counterexample detection” use mode, SIMULINK Design Verifier discovers and provides test scenarios, which can cover several cycles. It provides a set of control values, which can then be replayed by simulation. In particular, this functioning mode has enabled detection of inconsistencies or deficiencies in the control/command model compared to the environment assumptions.

Some users of the tool use the counterexample functionality to decide the correction of the model, going by the search time of a counterexample as a stop criterion. In these cases, it is desirable to combine this technique with a phase of intensive simulation of the model to assess the coverage rates of transitions.

5.5. Study case feedback and conclusions

5.5.1. SIMULINK model

This experiment has established the feasibility of an executable model of the mathematical formalization, which has enabled design choices to be specified and predicates to be implemented. The executable specification obtained can be operated by intensive simulation and by testing.

It is also important to emphasize that to make the model executable, the abstractions used in the mathematical specification have been described using actual types of representation. Furthermore, the SIMULINK model is created in such a way that all universal and existential quantifiers are completely eliminated. Lastly, the genericity of the model enables any type of directed graph respecting the good formation hypotheses described in the formal specification to be taken into account.

The choice of non-complex data structures (structure tables have not been used) is due to taking into account compatibility constraints with the DESIGN VERIFIER proof tool (and *ultimately* with the proof engine Prover). If the constraints are relaxed, the modeling and intensive simulation stage is facilitated.

5.5.2. Proofs achieved

The properties shown are of different natures but some emphasize that the SIMULINK model is functionally correct, while others ensure that the availability and safety of the system are preserved if the good functioning hypotheses are guaranteed.

Proofs on the SIMULINK model have enabled identification of inconsistencies/omissions in the formal specification, which were not detected during simulation of the model run in 100,000 cycles. This shows that the proof guarantees an exhaustive exploration of all possible cases that function well.

The proof has also enabled the compliance of the SIMULINK model with the formal specification to be established. In fact, even if the methodology adopted during production of the SIMULINK model enables some traceability with the formal specification to be ensured, some disparities always remain. However, reconciliation with the mathematical specification is obtained during the demonstration of functional properties, availability, and safety deduced from the specification. Lastly, the counterexamples supplied by DESIGN VERIFIER have revealed some bugs introduced during production of the model.

Phase	EN50128 reference	Coverage	Comments
<i>Software requirements specification</i>	Table A*		
	<i>Formal methods</i>	<i>Mathematical specification + restriction of the language to a SIMULINK subassembly</i>	<i>A proof of coverage could be based on demonstration of the equivalence between the execution semantic of the restrained SIMULINK language and that of a development language accepted in SIL3 or SIL4 processes (for example, in self cartography on SCADE and on the associated rules of use)</i>
		<i>Mathematical specification</i>	<i>V&V activities make use of a model to trace properties associated with the predicates (possibly to the members of predicates) and to identify the safety demonstration schematic (proof tree). Identification of critical predicates (in the safety demonstration tree).</i>
		<i>Covered by the data flow modeling of the SIMULINK model</i>	<i>Possibility of using the formal specification or its SIMULINK trace to identify predicates and critical operators by MADE</i>

Table 5.2. Conformity to Table A8 of the CENELEC EN 50128 standard

5.5.3. Incremental verification approach

The critical function is defined and modeled in SIMULINK following an incremental approach. As a result, the verification approach is also guided by definition of properties at each stage of incrementation. Some additional proofs are carried out for new functioning contexts; non-regression of the model is shown through insertion of equivalence proofs between each stage for common functioning conditions (DESIGN VERIFIER is ideal for this type of proof) and verification of the mutual coherence of properties (old and new) is demonstrated.

5.6. Contributions of the methodology compared with the EN50128 [CEN 01] normative referential

This section is written in the context of the establishment of a process based on models, with implementation of a SCADE type development language for the software design level.

We present a proposition here to implement formal methods in conformity to Tables A8 and A10 (see Tables 5.2 and 5.3) of the CENELEC EN 50128 standard.

Phase	EN50128 reference	Coverage	Comments
<i>Software design and implementation</i>	Table A10		
	<i>Formal methods</i>	<i>Mathematical specification + restriction of the language to a SIMULINK subassembly</i>	<i>A proof of coverage could be based on demonstration of the equivalence between the execution semantic and that of a development language accepted in SIL2 or SIL4 processes (for example in self cartography on SCADE and on the associated rules of use)</i>

	<i>Modular approach</i>	<i>Model architecture Traceability with the predicates “Incremental” development of the specification: the predicates can be broken down (taking into account the communicating trains, ONIs, signals and degraded modes)</i>	<i>V&V activities make use of a model for verification of dataflow coherence</i>
	<i>Design and coding standards</i>	<i>SIMULINK modeling rules</i>	<i>Critical analysis of modeling rules by the V&V team</i>
	<i>Strong typing programming language</i>	<i>Use of typed data structures + clear dissociation of data and table indexes</i>	
	<i>Language subassembly</i>	<i>SIMULINK modeling rules</i>	
	<i>Verified translator</i>	<i>Partial</i>	<i>Subject to manual or automated crossing towards an environment with a verified translator</i>
	<i>Library of verified modules and components</i>	<i>Proven operators</i>	
	<i>Functional tests</i>	<i>Intensive simulations of the SIMULINK model</i>	
	<i>Performance testing</i>	<i>Profiling of the SIMULINK model, control of model cycle times</i>	

Table 5.3. Conformity to Table A10 of the CENELEC EN 50128 standard

5.7. Bibliography

- [ABR 96] ABRIAL J.-R., *The B-Book - Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, The MIT Press, Cambridge, 2008.
- [BIE 03] BIÈRE A., CIMATTI A., CLARKE E., STRICHMAN O., ZHU Y., “Bounded model checking”, *Advances in Computers*, vol. 58, Academic Press, Maryland Heights, 2003.
- [BOU 99] BOULANGER J.-L., DELEBARRE V., NATKIN S., “METEOR: validation de spécification par modèle formel”, *Revue RTS*, no. 63, p. 47-62, April-June 1999.
- [BOU 06] BOULANGER J.-L., Expression et validation des propriétés de sécurité logique et physique pour les systèmes informatiques critiques, thesis, Compiègne University of Technology, 2006.
- [BRY 86] BRYANT R., “Graph based algorithms for Boolean functions manipulation”, *IEEE Transactions on Computers*, 1986.
- [CEN 01] CENELEC, NF EN 50128 Applications Ferroviaires. Système de signalisation, de télécommunication et de traitement – Logiciel pour système de commande et de protection ferroviaire, July 2001.
- [CLA 96] CLARKE E.M., WING J.M., “Formal methods: state of the art and future directions”, *ACM Computing Surveys*, 1996.
- [DOR 08] DORMOY F.-X., “Scade 6 a model based solution for safety critical software development”, *Embedded Real-Time Systems Conference*, 2008.
- [EIS 06] EISNER C., FISMAN D., *A Practical Introduction to PSL*, Springer, New York, 2006.
- [FRI 08] FRIEDENTHAL S., MOORE A., STEINER R., *A Practical Guide to SysML*, Elsevier, Kidlington, 2008.
- [GRU 08] GRUMBERG O., VEITH H., *25 Years of Model Checking*, Springer Verlag, Heidelberg, 2008.
- [HAL 91] HALBWACHS N., CASPI P., PASCAL R., PILAUD D., “The synchronous dataflow programming language Lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305-1320. September 1991.
- [HOL 91] HOLZMANN G.J., *Design and Validation of Computer Protocols*, Prentice Hall International, Upper Saddle River, 1991.
- [HOL 04] HOLZMANN G.J., *The SPIN Model Checker*, Addison Wesley, Boston, 2004.
- [IEE 04] IEEE 1474.1, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements, 2004.
- [JON 90] JONES C.B., *Systematic Software Development using VDM*, Prentice Hall International, Upper Saddle River, 1990 (2nd édition).

- [LAM 05] LAM W.K., *Hardware Design Verification*, Prentice Hall International, Upper Saddle River, 2005.
- [MAG 03] MAGGI P., SISTO R., *Using SPIN to Verify Security Properties of Cryptographic Protocols*, 2003.
- [MAT 98] MATRA-RATP, “Naissance d’un Métro. Sur la nouvelle ligne 14, les rames METEOR entrent en scène. PARIS découvre son premier métro automatique”, *La vie du Rail & des transports*, no. 1076, hors-série, October 1998.
- [OMG 07] OMG, Unified Modeling Language (UML) – v.2.1.1. 2007.
- [PVS 01a] PVS Language Reference, Technical report, November 2001.
- [PVS 01b] PVS System Guide, Technical report, November 2001.
- [RUS 07] RUSHBY J., “Automated formal methods enter the mainstream”, *Communications of the Computer Society of India*, 2007.
- [RUS 08] RUSHBY J., “An evidential tool bus”, *PVS, SAL and the Tool Bus*, 2008.
- [SHA 00] SHANKAR N., “Combining theorem proving and model checking through symbolic analysis”, *Invited Paper CONCUR 2000*, 2000.
- [SHE 00] SHEERAN M., SINGH S., STÅLMARCK G., “Checking safety properties using induction and a SAT-solver”, *3rd International Conference on Formal Methods in Computer Aided Design, LNCS*, Springer Verlag, Heidelberg, 2000.
- [SPI 89] SPIVEY J.M., *The Z Notation – a Reference Manual*, Prentice Hall International, Upper Saddle River, 1989.
- [STA 00] STÅLMARCK G., “A tutorial on Stålmarck method”, *Formal Methods in System Design*, January 2000.
- [WOO 09] WOODCOCK J., LARSEN P.G., BICARREGUI J., FITZGERALD J., “Formal methods, practice and experience”, *ACM Computing Surveys*, 2009.

Chapter 6

SCADE: Implementation and Applications¹

6.1. Introduction

This chapter presents the use of the SCADE² formal notation and its tools for the development of safety-critical embedded applications.

After a presentation of the needs, history, and foundations of the SCADE formal notation, we will make a presentation of the applications (see sections 6.7, 6.8 and 6.9) of SCADE, which have been possible in different industries such as railways, aeronautics, nuclear, and industrial applications.

6.2. Issues of embedded safety-critical software

6.2.1. *Characteristics of embedded safety-critical software*

Embedded computing has specific requirements, which are different from server-networked computing or micro-computing. These are mainly:

– *reactivity*: the system must react to its environment and at a rhythm imposed by the real world. It is not a matter of functioning quickly “on average”. For a given function, the *Worst Case Execution Time* (WCET), must never exceed a time that is fixed in advance and dependent upon physical constraints;

Chapter written by Jean-Louis CAMUS.

1 SCADE stands for *Safety Critical Application Development Environment*.

2 In this document, the term SCADE designates the modeling language and the term SCADE designates the tool chain reliant on the SCADE language.

– *criticality*: we will limit ourselves to faults in embedded safety-critical software, which can cause damage or harm to goods or people and jeopardize missions;

– *robustness*: software must react to failures of its environment (for example, sensors and other computers) in the safest way.

Although security aspects (such as resistance to malicious attacks) are of growing importance owing to terrorism-linked risks, they will not be dealt with here.

6.2.2. Architecture of an embedded safety-critical application

The architecture of an embedded safety-critical real-time system is typically composed of several components. These are in charge of input-output, scheduling and of the software application itself: the controller.

The controller exchanges data with the hardware and software environment through this architecture, which is illustrated in Figure 6.1.

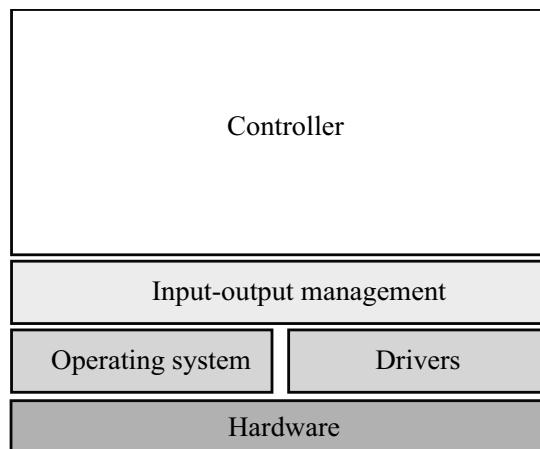


Figure 6.1. Architecture type of an embedded safety-critical real-time system

6.2.3. Criticality and normative requirements for embedded safety-critical applications

Development of embedded safety-critical real-time applications requires rigorous development processes, methods and tools adapted to guarantee the high degree of quality and safety that is required.

In some industrial sectors, government authorities require the software to be produced according to safety standards in force, for example [RTC 11] for avionics, [CEN 01] for rail, [IEC 98] for nuclear, or more recently [ISO 09] for automotive. The requirements of the aeronautical, industrial and rail sectors are summarized in the introductions to the sections dealing with these sectors.

Certification is the process that enables verification of the whole of the development process and that the software conforms to these standards.

Beyond their differences, these standards comprise many commonalities [BAU 10]. The software development process is broken down into elementary processes or phases, each with well-defined objectives. It is consequently necessary to verify that output of each of these processes or phases conforms to input of the same phase.

The objective of these verification activities is to ensure that a given phase does not introduce errors into its implementation.

In this context, the major challenge consists of limiting the cost of verification, which for certified applications, represents the major part of the total project cost, all the while ensuring the required degree of quality, safety and reliability.

6.2.4. Complexity, cost and delays

While the criticality and the corresponding rigor of development have been established for nearly 30 years, two categories of difficulties have been added over the years: complexity and the constraints of cost and delays.

In terms of the growth of complexity, two factors are combined:

- complexity of each function in itself;
- advanced integration of various functions among themselves.

For example, in the aeronautical industry, it is possible to go from:

- 200 KLoCs³ for the B767 (beginning of the 1980s) to 4,000 KLoCs for the B777 (middle of the 1990s);
- 800 KLoCs for the A320 (middle of the 1980s) to 10,000 KLoCs for the A380 (2005).

³ LOC for *Line of Code*.

At the same time, the constraints of cost and delays have intensified and it is common to have to develop software, which is simultaneously safety-critical and complex, in less than two years.

6.3. Origins of SCADE

6.3.1. Introduction

SCADE arose from the need to bridge the gap between the control engineering and software engineering views in the field of embedded safety-critical systems.

Upstream, control engineers characterize functioning modes and the transition criteria between these modes (for example, in the form of controllers) and design control laws for these modes (for example in the form of *block* diagrams and Z-polynomials).

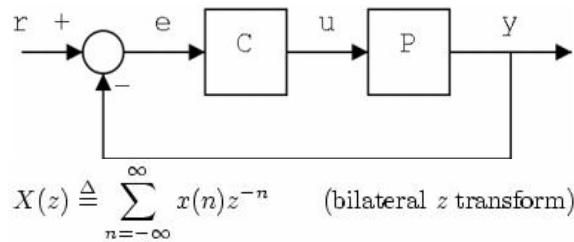


Figure 6.2. The world of control engineering

Computer scientists responsible for the implementation of these functionalities describe the world in terms of data structures, tasks and algorithms. Owing to differences between these visions of the world, the path from a control engineer's specification to a software design represents a costly and error-prone "impedance change".

6.3.2. Initial industrial approaches

During the 1980s, some industrial companies felt the need to bridge the gap. The first stage generally consisted of standardizing (within the company at the least) some notations used by system designers.

Tool chains supporting simulation and code generation were developed alongside these notations.

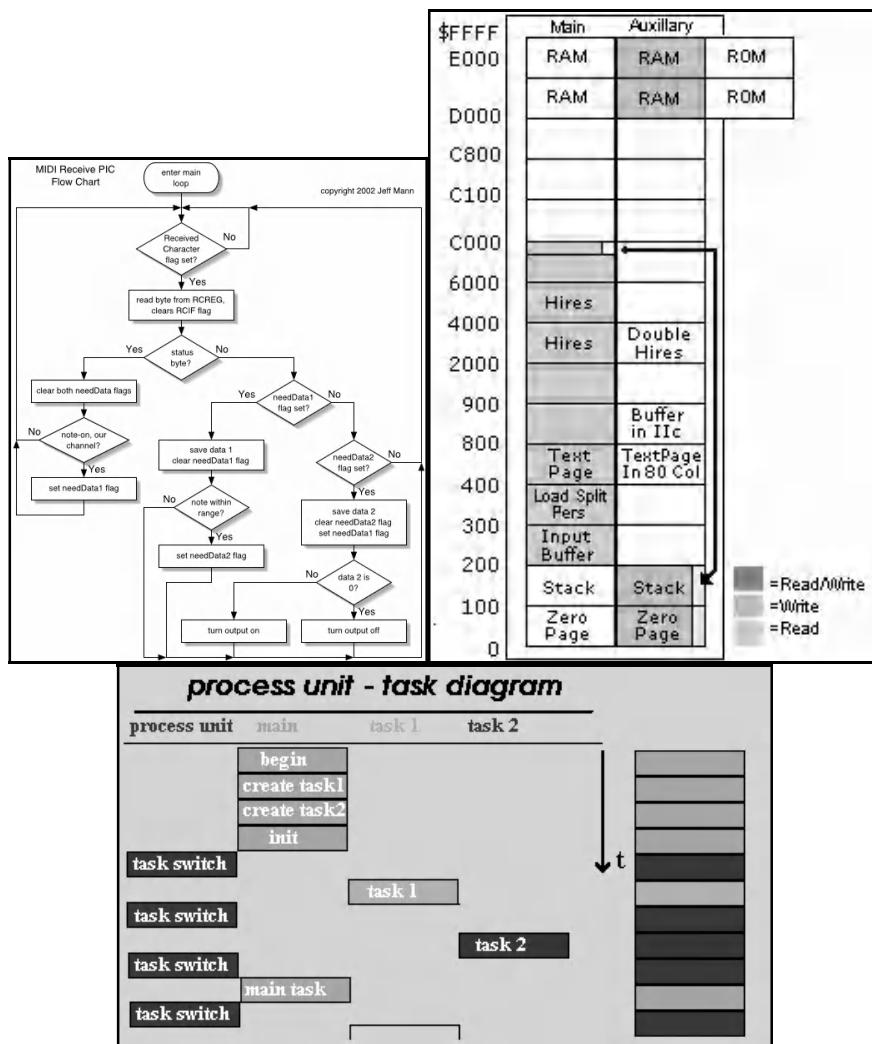


Figure 6.3. The software engineering world

This phenomenon has been as observable in the aeronautical or nuclear industries with some “data-flow” type notations, such as those in the process industries, for example with GRAFCET (a sort of industrial variant of Petri nets) or those of telecommunications with the SDL language⁴.

4 SDL for Specification and Description Language, see [UIT 94] and www.itu.int.

Some examples in the field of safety-critical real-time software are:

- SAO (*spécification assistée par ordinateur*) (computer-aided specification): designed and used by the French aeronautical community around Aérospatiale and Thales Avionique. The SAO notation was used by Airbus for flight control for airplanes in the A320 and A340 family;
- SAGA (*spécification d'applications et génération automatisée*) (automated application and generation specification): designed and used by Merlin Gérin for nuclear instrumentation software.

6.3.3. “Real-time” extensions of current languages

Some “real-time” extensions or variants have been created for most of the current languages, from C to Java. The case of the Ada language [ANS 83] is somewhat specific given that it had built-in “real-time” features from its conception.

These languages remain generalist languages and thus mainly retain the characteristics of their origin:

- advantages: flexibility, the capacity to address varied problems;
- disadvantages: relatively low abstraction levels, with always a “computer scientist” vision.

6.3.4. Synchronous formal languages dedicated to “real-time” created in laboratories

At the same time, several research laboratories devoted themselves to language creation approaches specifically adapted to the description and production of real-time software.

One language family particularly stands out: the synchronous languages family, of which the principal representatives are:

- Lustre [HAL 91]: “data-flow” type declarative synchronous language. It constitutes the main basis of SCADE;
- Signal [LEG 91]: system level “data-flow” type declarative multi-clock synchronous language;
- Esterel [BER 00]: imperative synchronous language. It has mostly served as a basis for the Esterel Studio system, support for complex SOC (*System On a Chip*) development. Some of its concepts have moreover served as a basis for the introduction of state machines in SCADE.

An excellent summary of the origins and state-of-the-art review of synchronous languages can be found in [BEN 03].

Given the context of their creation in laboratories, these languages have had formal bases from the very beginning.

6.3.5. Birth of SCADE

SCADE is the fruit of a joint venture between:

- two major manufacturers, Aérospatiale and Merlin Gérin;
- the VERIMAG laboratory, having designed the Lustre language;
- Verilog software engineering tools editor.

The manufacturers' objectives included:

- externalizing activities not constituting their core business and sharing their costs with other users;
- making these technologies accessible to their suppliers, to facilitate exchange of software requirements, and improve the global productivity of their ecosystem. In effect, SCADE is more than a tool; it is the *lingua franca* of the Airbus ecosystem. Airbus designs and communicates its software specifications to its internal and external equipment manufacturers and even to the in-flight and maintenance testing services, in the form of SCADE models. In addition, a tool editor, whose job is devoted to development, maintenance and support of the tool is in a better position to make available this service to a wider community of users;
- use of the formal basis Lustre enables freedom from the risks of dialects or ambiguities of “in-house” products. Moreover, this opens up access to the formal proof (this, however, having had no decisive importance at the time of the choice).

6.4. The SCADE data-flow language

6.4.1. Introduction

SCADE is used for developing software controllers. On a practical level, SCADE is used for implementing the informal approach of control engineers based on a function of cyclical reaction to the environment.

The main function generated by SCADE is called in a cyclical way, be it at regular intervals or on the basis of particular events.

A section of manually developed code typically complements the component covered by SCADE with some functions that manage and prepare input and output data, thereby interfacing with low-level software layers and the hardware, as illustrated by Figure 6.4.

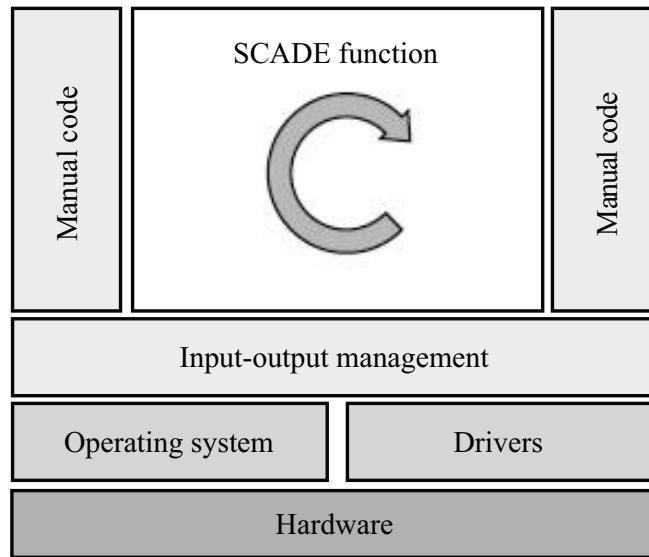


Figure 6.4. The component developed with SCADE in a real-time system

6.4.2. Synchronous language

SCADE is a synchronous language. The frequently-given synchronous hypothesis definition is that calculation takes *zero-time*.

It can equally be viewed as a model of calculation in *purely discrete-time*, where the notion of physical time is ignored. This world is structured by an ordered sequence of *moments*, inappropriately called *cycles*.

A SCADE model formally defines how, for each instant, output and the new state are calculated from the current moment's input and the state resulting from the previous moment.

It is assumed that there will be no interaction with the environment during the computation of an instant; in other words, the calculation of an instant is atomic; input, especially, must not change.

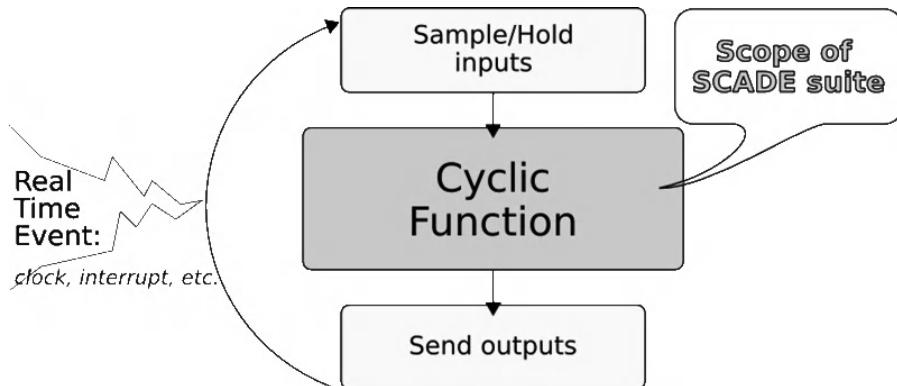


Figure 6.5. The synchronous calculation model

6.4.3. “Data-flow” functional language

The SCADE language, termed “data-flow”, which is the historic kernel, is a functional language based on the Lustre language.

It appears to most users in the form of box-and-arrow diagrams (*block diagrams*), such as those frequently used by control engineers.

A box depicts processing; an arrow connects the output of a block to the input of another block (or symbol).

The nesting of boxes makes it possible to describe arbitrarily complex processing in a structured way.

More precisely, a SCADE model is a hierarchy of operator instances, right down to the predefined language operators such as logic or temporal operators.

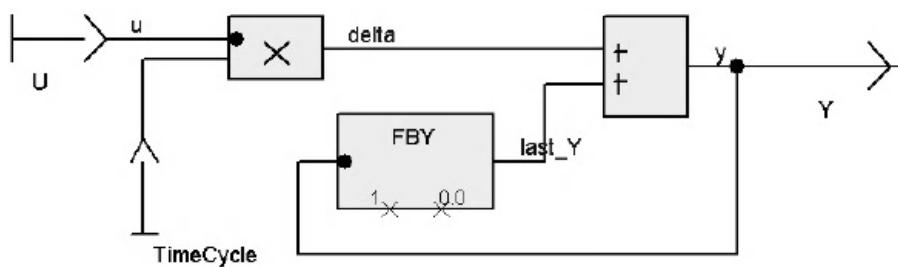


Figure 6.6. Graphic of an integrator in SCADE

A SCADE operator is a module, which comprises three parts:

- 1) a formal interface:

```
node IntegrFwd( U: real ; hidden TimeCycle: real)
    returns ( Y: real);
```

- 2) some declarations of local variables:

```
var
    delta : real ;
    last_Y : real;
```

- 3) a set of equations:

```
delta = u * TimeCycle ;
y = delta + last_Y ;
last_Y = fby(y , 1 , 0.0) ;
```

It is worth noting that these are equations in the mathematical sense of the term and not assignment statements.

Their position in the text (or in the graphic) is not significant; only the functional dependencies count.

6.4.4. Scalar data types

The language supports some types of scalar and structured data. The scalar types are:

- Boolean (bool);
- Integer (int);
- Real (real);
- Enumerated (enum).

The language includes a set of operators predefined on Booleans (AND, OR, XOR, NOT).

With respect to the numerical type *integer*, it supports the +, -, * operators, integer division, and modulo.

For the real type, the operators are +, -, *, and real division. Lastly, conversions from *real* to *integer* and *vice versa* enable switching from one type to another.

With the current versions of SCADE, the “int” and “real” types represent some ideal mathematical types, which are implemented in “int” type form, “float” or double C language at the moment of compilation.

Future versions of SCADE will support some definitions of types with finite intervals such as “int16” and “uint32”.

The principal use of enumerated types involves choice operators (see selectors or imperative structures in sections 6.4.7 and 6.4.8).

Lastly, it is possible to use types known as “imported”, which are defined with the help of an external programming language (C, Ada), the instances of which are seen as “black boxes” and to whom only imported or completely generic predefined operators can be applied.

6.4.5. Structured data types

The structured types are:

- named field structure, for example: “Type Sens Temperature {Valid: bool, Value: real}”;
- arrays, containing an N-ordered set of data of the same type.

The tables are mono-dimensional and a matrix is a table of tables. It is possible to arbitrarily nest the structured types.

The language offers composition, breakdown and projection operators of structured types. For the tables, the language also supports transposition, reversal, dynamic projection and *slicing* (subsection of the table).

6.4.6. Clocks, temporal operators, and causality

In a SCADE model, every flow is linked to a “clock”, which defines when this flow is available (either because of the environment that provides it or because of calculation conditions within the model).

Although the clocks may be accessible to any users who want them, they are used more often in “packaged”, easier-to-use forms such as *activate*-type structures and state machines (see following sections). We will therefore not go into the subject in detail here.

The “pre” operator enables to delay a flow by one cycle, which corresponds to the $z-1$ operator of the automatic. The “ \rightarrow ” operator enables the definition of the initial value of a flow, notably for a delayed flow.

Figure 6.7 presents an example of a rising edge detector.

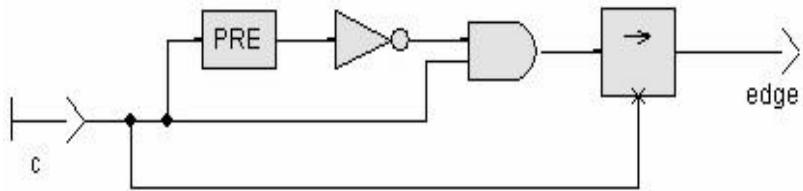


Figure 6.7. Rising edge detector

The semantic rules of the language prohibit non-causal loops such as that of Figure 6.8. This is detected as a semantic error, which inhibits code generation.

In the context of Figure 6.9, the delay operator (*fby*) enables the removal of ambiguity in clearly indicating that it is the throttle value at the preceding cycle, which is an input to *ComputeTargetSpeed*.

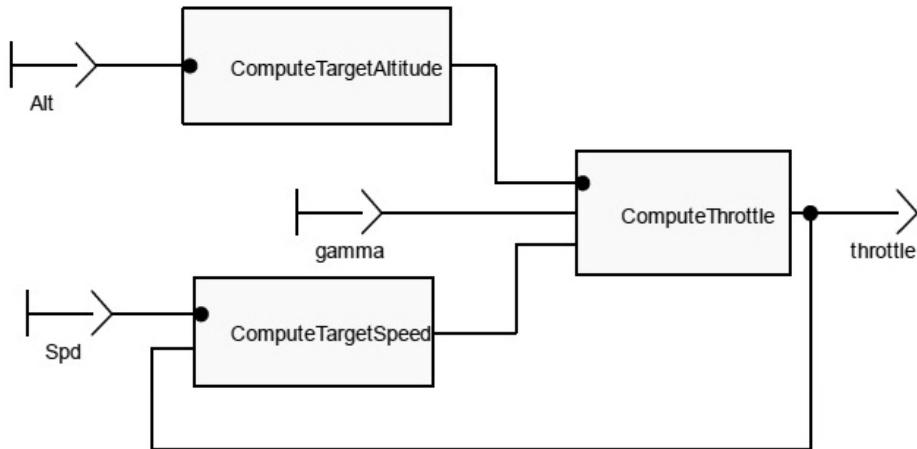
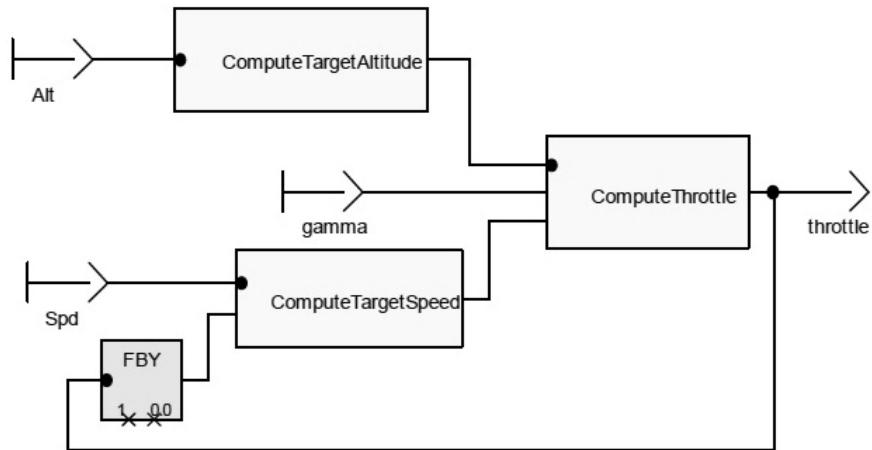


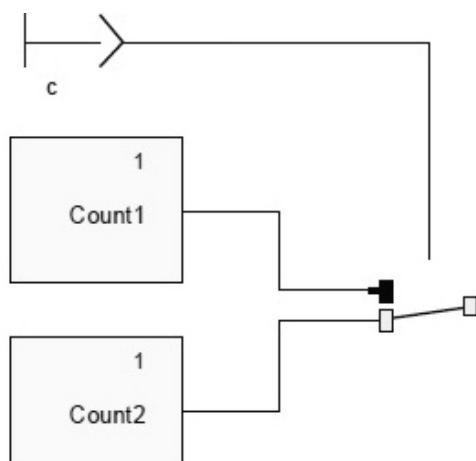
Figure 6.8. Non-causal loop

**Figure 6.9.** Causal loop

6.4.7. Selectors

The selectors “if” (Boolean) and “Case” (multiple choice) enable one flow among several to be selected.

These need to be distinguished from imperative control structures: in the example of Figure 6.10, the two counters (Count1 and Count2) are raised at each cycle, whatever the value of the selection Boolean.

**Figure 6.10.** Example of “If” selector

6.4.8. Imperative structures

Contrary to selectors, imperative structures determine the conditions in which a calculation takes place. This is close to C/Ada imperative structures such as “if/then/else” or “switch/case”, but a valuable difference regarding safety of the functioning point of view is that the language and its tools require a set of choices to ensure that every variable is defined by the active branch or an explicit fault, at every cycle.

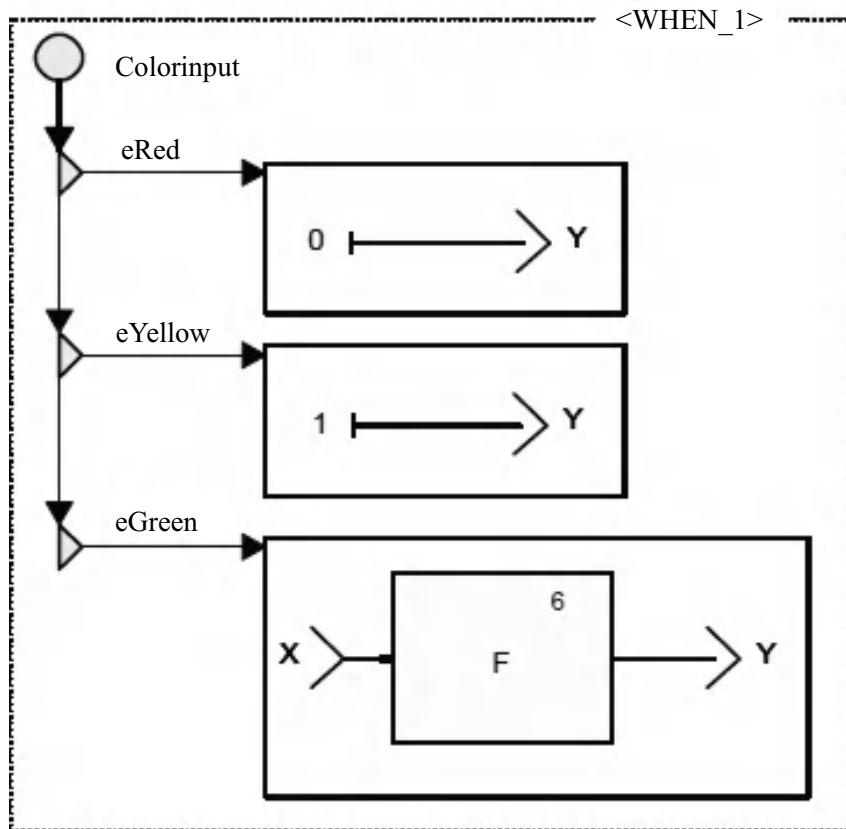


Figure 6.11. Imperative calculation diagram

For example, the set of choices for an enumerated selector must cover the value field of this enumeration; there is thus no risk of having an indeterminate value.

NOTE 6.1.– These structures, much like controllers, are formally based on clocks.

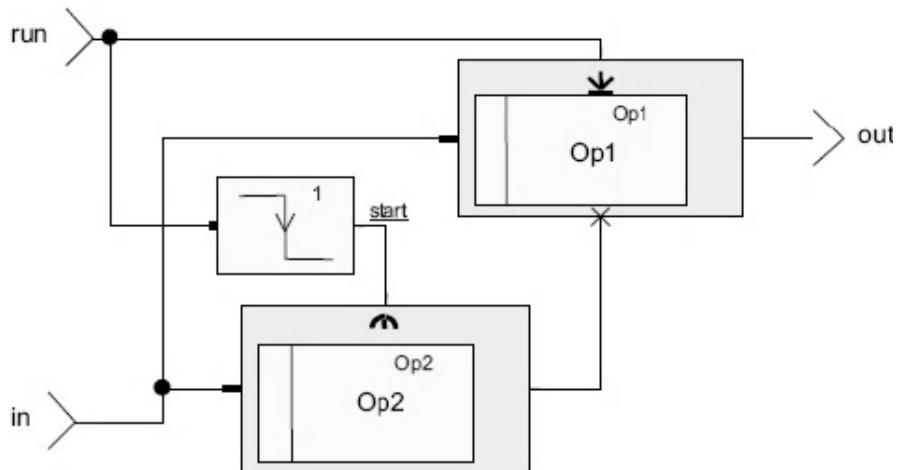


Figure 6.12. Conditional activation of operators

6.4.9. Rigor and functional safety

The SCADE language relies on the formal synchronous semantic definition of the Lustre language. While being close to the concepts of the discrete control engineering models, SCADE possesses characteristics essential for the development of embedded safety-critical systems:

- the behavior is totally deterministic (that is to say, a given input sequence will always produce the same output sequence). As will be seen later in this chapter, this is notably the case for state machines;
- the language is strongly typed: all data is typed and coherence of types is verified;
- initialization completion is formally defined and verified;
- all data must be produced once and only once in a cycle, or more exactly, at moments where its clock enables it to be consumed. There is thus never any data that is undetermined or overwritten by successive calculations within a cycle;
- there are no possible side effects between operators. There is no global variable and all interactions between operator instances necessarily involve explicit connections defined by the modeler (these are visible in the form of model graphics);
- there is no possibility of direct access to the physical memory from the language. In practice, this means that a person (or team) dedicated to the design of a

software application cannot, by definition, alter the memory, since the KCG code generator rigorously checks these rules;

- SCADE expressions and operators can be composed as mathematical functions. Operator properties are preserved by composition, which concerns simple operators (like a rising edge detector) or complex operators (like control law engagement logic). The results of the costly work of component design and verification does not risk being suddenly compromised on the occasion of an integration with other components;

- iterative schemes (see next section) restrain access to tables to safe forms.

The definition of the language comprises several hundred semantic rules, which are systematically verified by the SCADE tools, notably as a pre-requisite to code generation.

The benefit of the SCADE approach is to clearly define the software specification, expressing the dependency between the computational aspects and application logic perfectly.

SCADE users have thus been able to observe that the modeling activity tends to bring the presence of ambiguities in the highest-level requirements to light quite systematically.

It is generally true that error detection at the modeling stage is greatly preferable to a situation where these errors are discovered much later in the project, for example during system tests on the target.

Typically, a case where a piece of data is used while its value has not been initialized is detected at SCADE modeling stage.

6.5. Conclusion: extensions of languages for controllers and iterative processing

6.5.1. *Objectives*

The traditional pure “data flow” type SCADE language has enabled development of embedded systems as complex and safety-critical as the flight commands of the Airbus A380 (4,000 SCADE diagrams). Nonetheless, for some other applications, users have expressed the need for extensions to:

- state machines: to express modal logics;
- iterative processing: to apply calculations to tables or more generally, to iterate any processing whatsoever.

6.5.2. Control flow

NOTE 6.2.– In this document, the terms “state machine” and “automaton” are synonymous.

6.5.2.1. Outline

As for all “automata” type notations, state machines are based on some “states” (bubbles or rectangles) and some “transitions” (arrows between states), which are fired when their “guard” (*trigger*) satisfies certain conditions. Figure 6.13 shows an example of a SCADE model, which assembles functional schematic and state machines. This model graphically specifies the behavior of automobile cruise control in SCADE 6.

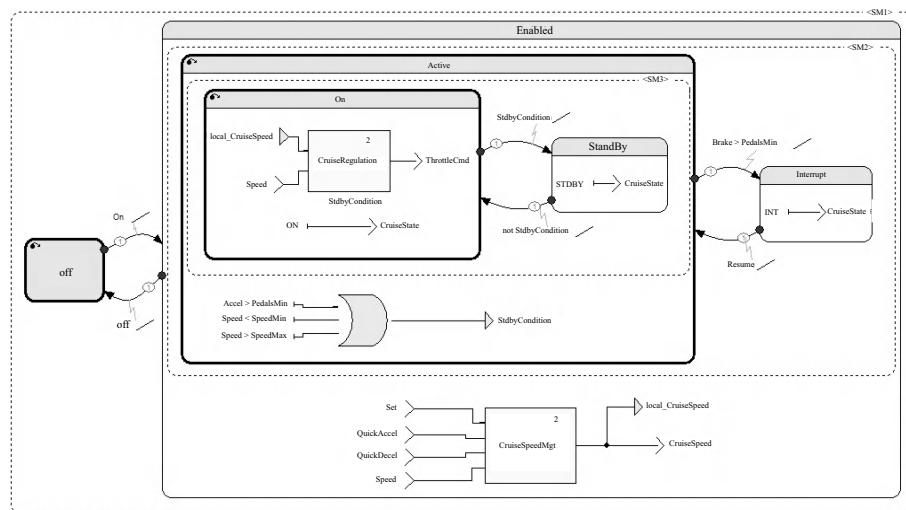


Figure 6.13. The highest-level SCADE model of cruise control

At the highest level, it involves a state machine constituted of two states, namely the “Off” state where the regulator is inactive and the “Enabled” state where the regulator is activated. Down inside the “Enabled” state, yet another state machine is to be found, again with two states, namely the “Active” state where the regulator is active and the “Interrupt” state where the regulator is interrupted, the driver having applied the brake pedal (Brake > PedalsMin), as long as the driver does not press the “Resume” button of the regulator.

Inside the “Active” state, we have another state machine with an “On” state, where the regulator efficiently regulates vehicle speed, its primary function, and a

“StandBy” state where speed regulation is temporarily suspended, as long as the driver takes the speed controls in pressing on the accelerator pedal. Inside the “On” state, a functional schematic, the “CruiseRegulation” operator models the control law defined by Figure 6.14.

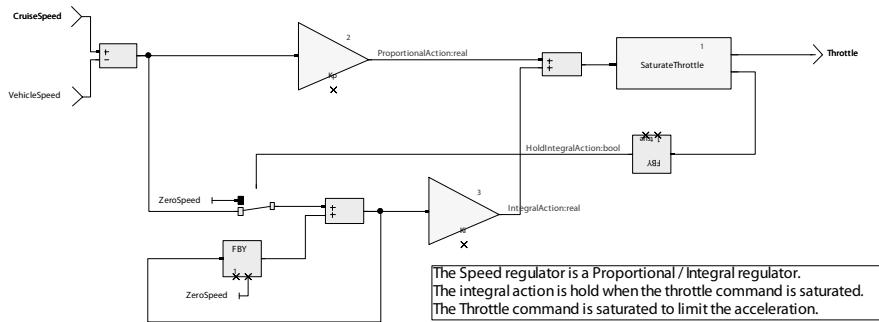


Figure 6.14. Example of control law

6.5.2.2. Characteristics of SCADE state machines

There are currently several dozen variants of “state machine” type notation, which can all seem alike at first sight. There are however some important differences from the point of view of power of expression and rigor. SCADe language state machines were designed to ensure all the rigor necessary for the development of embedded safety-critical applications.

SCADE state machines rely on a formal semantic, based on that of the data flow kernel of Lustre [COL 05]. This common basis enables state machines to be limitlessly interleaved in traditional data flow type processing and vice versa, and in this way perfectly coherent and rigorous, in contrast with solutions consisting of empirically assembling codes generated from environments separately designed from the outset.

SCADE state machines have a synchronous semantic, with single transition firing:

- at each cycle of a controller, the guards of the active state’s outgoing transitions are evaluated;
- the transitions possess priorities explicitly fixed by the user and independent of the position of states and transitions on the diagrams;
- these evaluations have no side effects;
- a transition at the most is fired (otherwise, the active state remains). This guarantees an execution time limited by construction.

This strongly differs from the semantic of notations based on the “microstep” concept where at each cycle transitions are fired from state to state right up to remaining in a state, no emerging transition of which can be fired. Their execution time is not guaranteed *limited* and their analysis is complex.

SCADE state machines support some advanced functionalities in a formal and determinist way:

- state hierarchies;
- strong or weak pre-emption (output of a state) from one or several levels;
- history (reactivation of a controller in the state where it was at the moment of a pre-emption);
- signal exchange;
- logical parallelism processing;
- synchronization between parallel processes.

6.5.3. Iterative processing

6.5.3.1. Approach adopted for iterative processing

It is frequent to have to apply some iterative processing such as:

- vectorial and matrix calculations, for example, for advanced filters such as the Kalman filter;
- homogeneous processes on sets of objects, such as sensors, satellite constellations, electric *switches* or even valves.

The immediate easy solution would consist of introducing “for” or “while” type imperative control structures into the SCADE language and accessing table elements by dynamic index. The inconveniences of this type of solution would be the same as those encountered in traditional programming, notably risks of error in memory accesses.

Furthermore, this type of low-level approach would not be homogeneous with the rest of the language and would compromise the functional approach, as much conceptually as for code generation.

The retained approach is thus based on higher-order functional operators, described in the next few sections, and not on imperative structures. This approach enables preservation of the fundamental properties of the language described in section 6.4.9.

Also, they are not limited to table calculations. It provides a general scheme enabling regular multi-instantiation of any operators whatsoever, from a simple addition right up to complex operators such as state machines.

6.5.3.2. Parallel iteration

Figure 6.16 shows the example of application of the SCADE language “map” meta-operator to execute the sum of two input vectors, element-by-element, in place of the “flat” form of Figure 6.15.

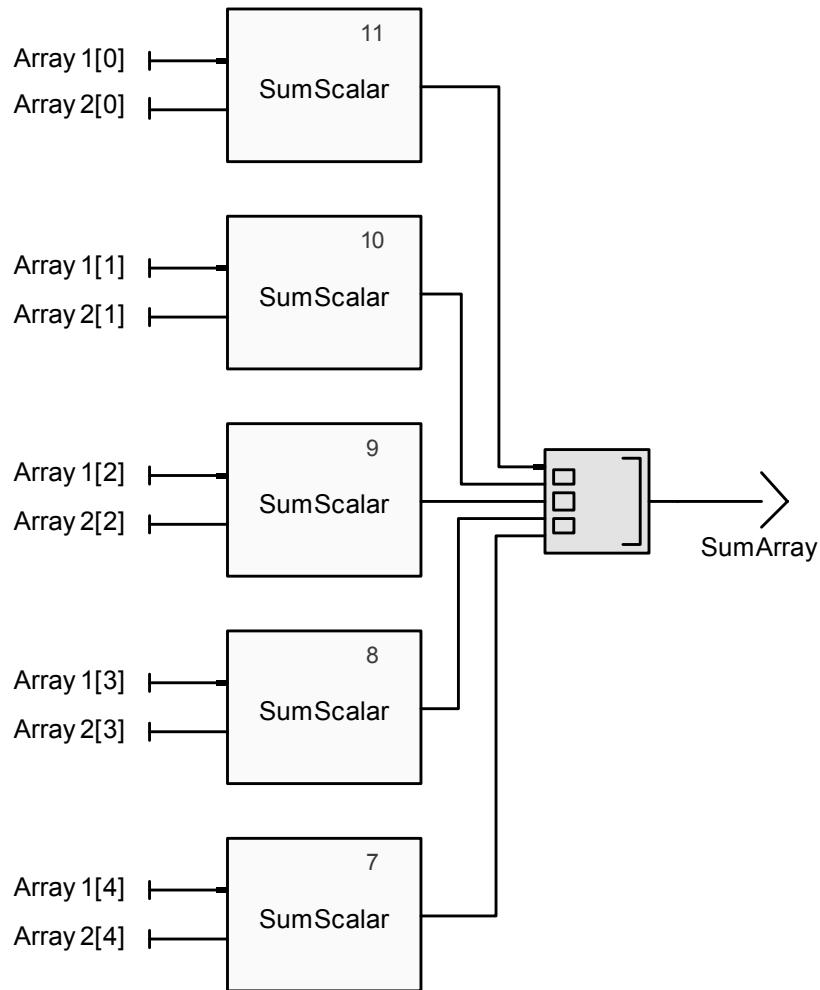


Figure 6.15. Flat expression of parallel operations

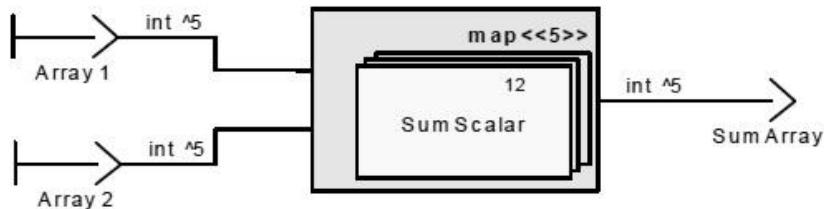


Figure 6.16. Parallel instantiations with “map”

The modeler does not manipulate the table index (this is implicit) and thus cannot corrupt the memory.

6.5.3.3. Iterator variants

There are variants of these operators above:

- the “fold” construction expresses operator instance serialization;
- the “mapfold” construction combines “map” and “fold”;
- the iteration can provide the value of the current index (for example, mapi);
- the iteration can terminate itself if a calculated condition is false.

6.5.3.4. Iterator composition

As indicated above, the SCADE language enables composition of operations in the functional sense, and this applies equally to iterators.

The scalar product can thus be expressed as the functional composition of sum and product operations that will be very simply depicted as serialization of functional boxes.

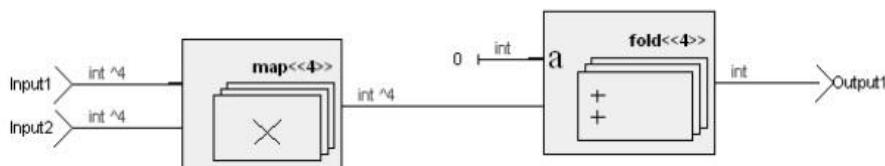


Figure 6.17. Model of the scalar product

Although there are two iterators on the original diagram, analysis of the structures enables the code generator to produce a factorized loop in the optimized generation modes.

```

kcg_int ScalarProduct6(
    array_1 *Input1 /* ScalarProduct6::Input1 */,
    array_1 *Input2 /* ScalarProduct6::Input2 */)
{
    /* ScalarProduct6::Output1 */
    kcg_int Output1;

    kcg_int i;

    Output1 = 0;
    for (i = 0; i < 4; i++)
    {
        Output1 = Output1 + (*Input1)[i] * (*Input2)[i];
    }
    return Output1;
}

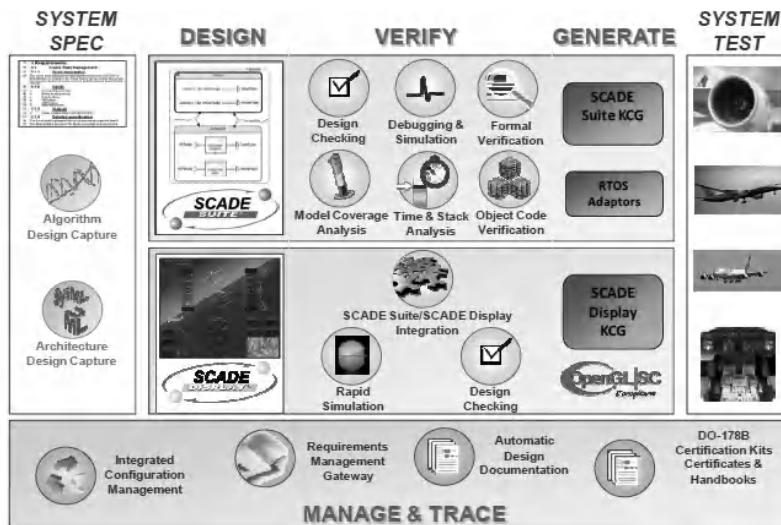
```

Figure 6.18. Code generated for the scalar product

6.6. The SCADE system

6.6.1. Outline of the SCADE workbench

The SCADE workbench offers an integrated set of tools based on the SCADE modeling language, covering all conception, coding and verification activities of the software application.

**Figure 6.19.** The SCADE workbench

The workbench (see Figure 6.19) integrates these functionalities for two parts of applications:

- the functional part, covered by the “SCADE Suite features” family of tools;
- the graphic part (interactive or not), covered by the “SCADE Display” family of tools.

In this section, we limit ourselves to the “SCADE Suite features” section, as much for the notation previously described as for the tools.

The heart of the system is the SCADE editor, enabling creation and modification of structured and ergonomic-style models.

Upstream, gateways enable links with system-level specifications modeled in SysML⁵ or SIMULINK⁶, or more traditionally, in the form of textual documents or requirements-based management such as DOORS.

Simulation, model coverage measurement, and the formal proof engine offer powerful and complete means of model verification before moving on to coding. It has been widely demonstrated for many years, that error detection in the phases upstream of the do-nothing cycle costs a lot less than late detection.

The KCG qualified code generator automatically generates a code conforming to the model, thus saving not only the effort of coding itself, but a large number of the modular tests.

This system is integrable with each user’s configuration management environment.

6.6.2. Model verification

6.6.2.1. Objective of the verification

The objective of SCADE model verification is to show that the software application developed with SCADE conforms to the higher-level requirements, laid down during the system definition stage and that the model does not contain unintended functions, which could be introduced during the modeling phase.

6.6.2.2. Model review

The first type of verification is model review, which verifies that each element of a SCADE model is in direct relation with one or several requirements input in the model design phase.

5 SysML for *Systems Modeling Language*, see [OMG 10] and: www.sysml.org.

6 Simulink is a trademark of The Mathworks Inc.

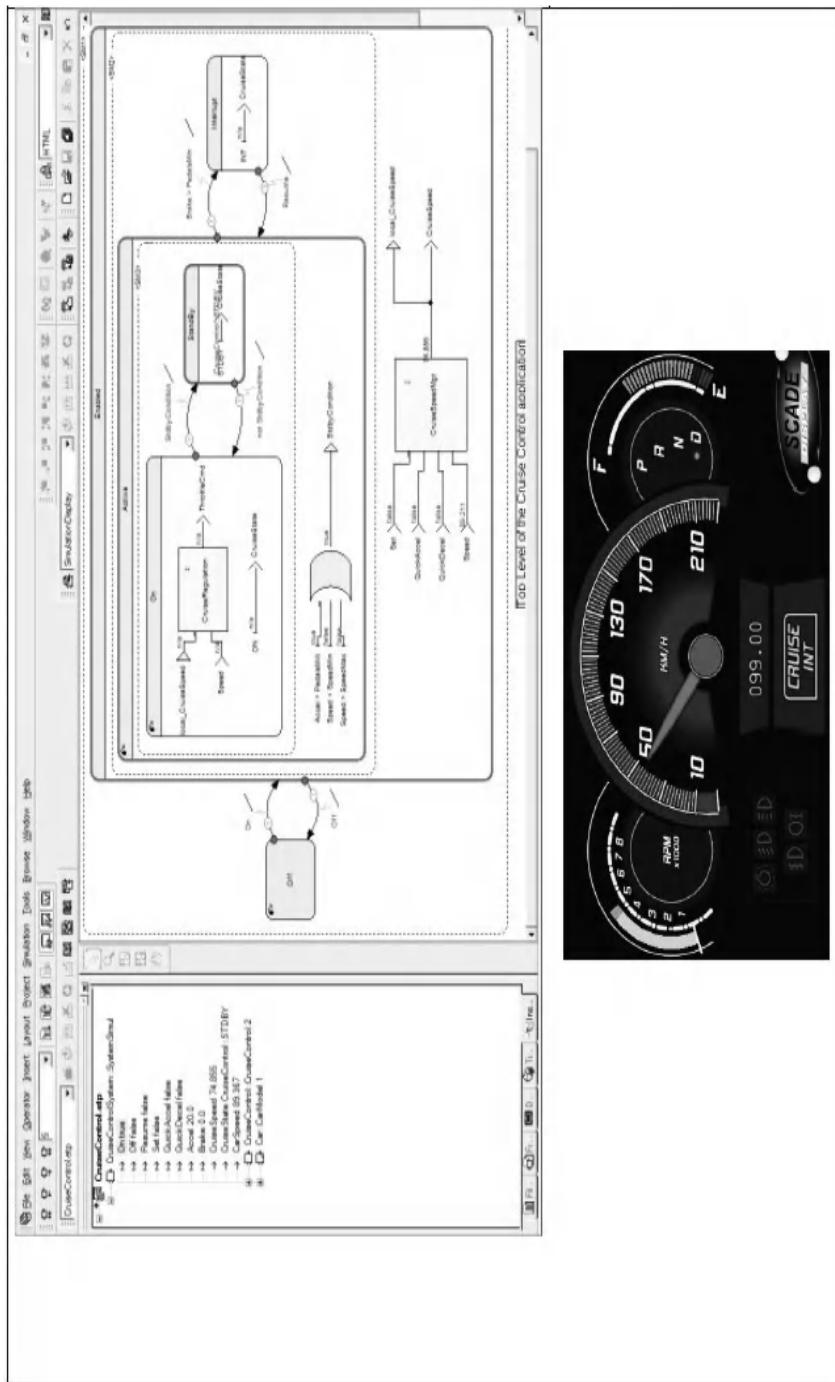


Figure 6.20. Cruise control simulation

To facilitate this stage of verification, the SCADE development environment provides the “SCADE Suite Requirements Management Gateway™” (RM Gateway) tool for managing traceability of the requirement.

With this tool, it is possible to define organization of the project documents and traceability information, which will enable connection of the higher-level requirements with elements of the SCADE model. The RM Gateway tool also enables calculation of the percentage of requirements covered by the model and execution of impact analysis of a modification of these requirements.

6.6.2.3. The simulator

For the rest of the verification stages, the most important property of a SCADE model is that it is executable. As a result, functional verification of the model can continue through simulation executed with the help of the “SCADE Suite Simulator™” tool, which enables debugging of the model to be executed. An essential benefit of this simulation is that its results are identical to results obtained later, during execution of the code generated from the model on the target.

The additional tool “SCADE Suite Rapid Prototyper™” enables the results of execution of the model to be displayed more intuitively and facilitates interaction with the simulation. Figure 6.20 shows a cruise control simulation session where input and output of the model is controlled using Rapid Prototyper. The benefit of this verification stage is that it allows functional errors to be discovered during the initial stages of the lifecycle yet again, thus reducing the project costs.

6.6.2.4. Coverage measurement of the model

Structural coverage of software allows measuring how thoroughly software has been tested. It enables identification of the parts of this software, which have never been tested and indicates the measures necessary to correct this situation.

Structural coverage of a SCADE model by a series of tests can be measured by the “SCADE Suite Model Test Coverage™” (MTC) tool. This type of measurement, which is required by norms such as RTC 11-2, can highlight the insufficiency of testing (some requirements would not have been tested), contradictions in the requirements or the parts of the model which are inaccessible, and which are therefore useless.

Figures 6.21 and 6.22 enable comparison of the case of a traditional development flow (Figure 6.21) where the activity of structural coverage measurement is executed at source code level, with a case where the activity of measurement is executed at model level with MTC (Figure 6.22).

In the case of traditional flow, in executing the instrumented code on the target, it is necessary to await the end of the project to discover that there are coverage faults, which bring errors to light.

This activity is costly because it is executed late in the project, where it brings to light errors originating from every level (requirements, design and coding) or the different parties involved often speak very different languages.

In the case of structural coverage measurement at model level, the activity is much more efficient. In effect, it enables errors to be discovered as early as possible, when models begin to be available. It directly involves parties that speak the same language, SCADE, and which are in an extremely close and efficient loop of interaction.

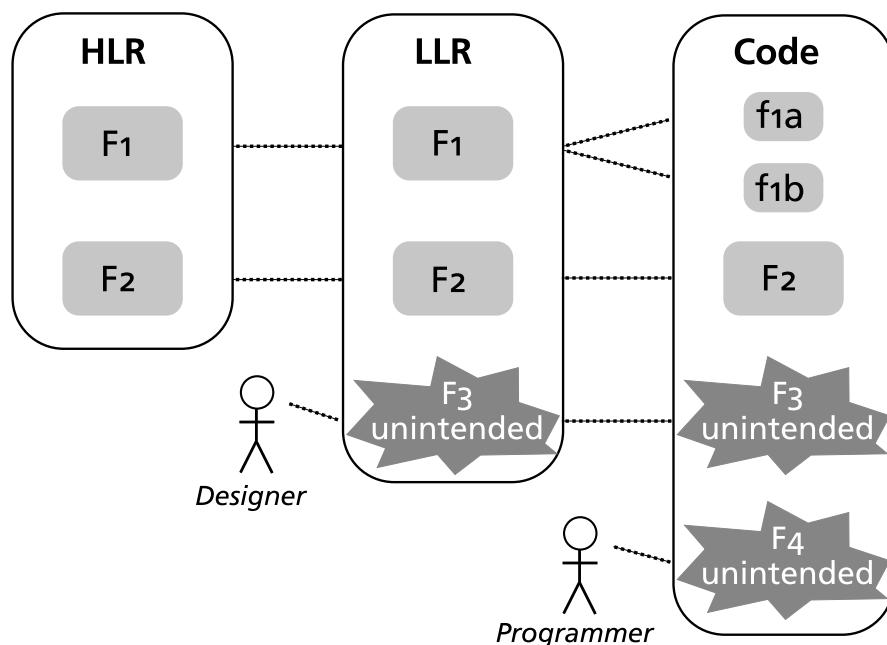


Figure 6.21. Structural coverage in a traditional flow

If this approach is complemented by certified code generation (as described in the following section), it will then be useless to repeat structural coverage measurement at code level since the certified code generator does not itself introduce any unintended functions.

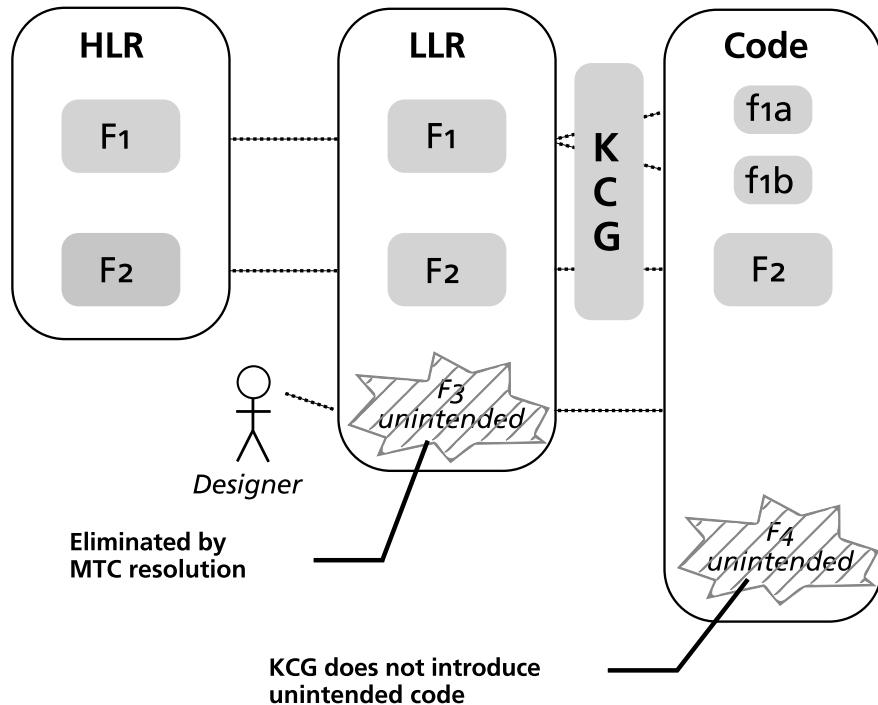


Figure 6.22. Structural coverage at model level

6.6.2.5. The formal verifier

In addition to these verification activities that rely on the SCADE model tests, and thanks to the “SCADE Suite Design Verifier™” (DV) tool, integrating Prover Technology™’s Prover Plug-In™, it is equally possible to implement *model checking* type formal verification. The testing activity enables verification that a SCADE design is correct in relation to the higher-level requirements. However, with testing, it is never possible to be 100% sure that the model is correct because it is in general impossible to test all possible execution scenarios.

Formal verification of computer systems provides a mathematical framework enabling reasoning on system behavior in a rigorous and certain manner. On the basis of a current SCADE model, the DV user describes a set of safety properties to formally verify, they themselves being described in the form of a SCADE model.

Next, the Design Verifier will execute an exhaustive mathematical analysis of the set of possible behaviors of the system. For each of the safety properties, two cases are possible: either the property is valid for the application model, whatever

the input scenario may be; or the property is not valid for some input scenarios. In this last case, Design Verifier provides an input scenario that leads to an erroneous execution and it is then possible, with the aid of the simulator, to understand the origin of the problem and to correct it by modifying the model.

Let us consider the example of the landing gear of an airplane. A safety requirement will naturally be expressed in the following manner: “For the set of possible behaviors of the landing gear controller, no landing gear retraction command will be issued in the touchdown or taxiing phases”. It involves ensuring that the landing gear will only be retracted in conditions ensuring safety of the airplane (No Unsafe Retract).

Figure 6.23 shows how to connect the SCADE model, which manages the landing gear (Doors Control) with an observer, which will have the role of verifying the property expressed above (Retract Safety Observer).

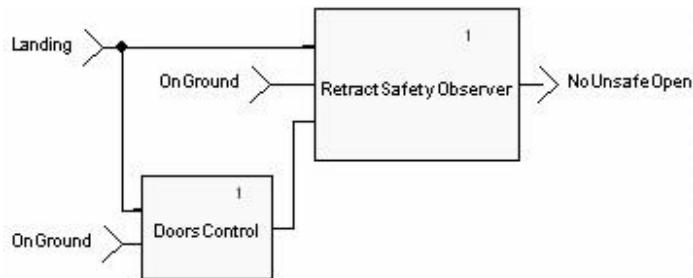


Figure 6.23. The formal verification model

Figure 6.24 shows the logic description of the safety property to verify in SCADE.

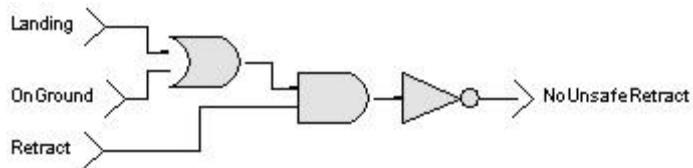


Figure 6.24. The property to verify

6.6.3. Performance prediction

Lastly, the SCADE product integrates a tool called “SCADE Suite Timing and Stack VerifierTM” (TSV). For a given hardware target, this tool gives an estimate of maximum execution time of the C code, which will be generated for the SCADE

model and its maximum stack space consumption (see section 6.6.4 for the code generator).

This tool integrates static analysis technology developed by Absint with the aiT tool. It relies both on the SCADE model of the application and on a model of the target processor, which will enable performance to be evaluated.

So as to execute these estimates, it will be necessary to generate the C source code with KCG and compile this code in binary code form using the cross compiler for the target, as indicated in Figure 6.25.

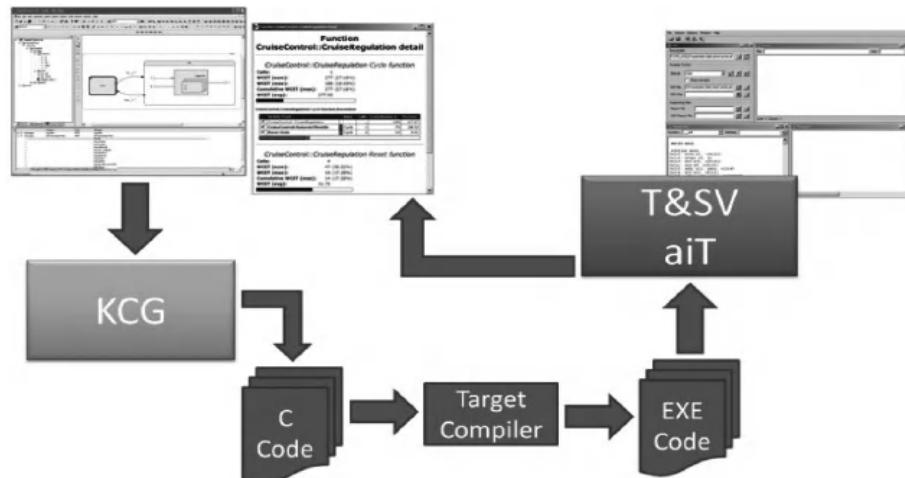


Figure 6.25. The flow of performance prediction tools

This analysis produces an HTML report in SCADE Suite enabling information calculated by the tool to be shown at model level, thanks to traceability between the model and the generated code.

This report details the distribution of execution time per operator in the SCADE model, as indicated in Figure 6.26.

6.6.4. The qualified code generator

In a classic software development flow, the specification and software design phases are followed by a coding phase. Afterward, a phase of verification of the code is necessary to ensure that the source code corresponds to its specification.

Function CruiseControl::CruiseControl detail (session TSV)

CruiseControl::CruiseControl Cycle function					
Calls:			Kind	Contribution	%
CWCEI (sum):	1169	(100.00%)			
WCET (max):	656	(56.12%)			
CWCET (max):	1169	(100.00%)			
CWCET (avg):	1169.00				

CruiseControl::cruiseControl Cycle function descendant					
SCADE Path #	Calls	Kind	Contribution	%	
CruiseControl::CruiseSpeedMgt	1	RESET		16	1.37
CruiseControl::CruiseSpeedMgt	1	CYCLE		162	13.86
CruiseControl::CruiseRegulation	3	RESET		48	4.11
CruiseControl::CruiseRegulation	1	CYCLE		287	24.55
CruiseControl::cruiseControl	0	CYCLE		656	56.12

CruiseControl::CruiseControl Reset function					
Calls:			Kind	Contribution	%
CWCEI (sum):	102	(100.00%)			
WCET (max):	76	(74.51%)			
CWCET (max):	102	(100.00%)			
CWCET (avg):	102.00				

CruiseControl::CruiseControl Reset function descendant					
SCADE Path #	Calls	Kind	Contribution	%	
CruiseControl::CruiseSpeedMgt	1	RESET		10	9.80
CruiseControl::CruiseRegulation	1	RESET		16	15.69
CruiseControl::cruiseControl	0	RESET		76	74.51

Figure 6.26. The performance analysis report

Here also, code reviews, examination of traceability of the code with the specifications, definition and execution of tests of the code and finally, structural coverage measurements of the code by testing will have to be relied upon to verify that it is functionally correct in relation to its specification. These activities are naturally very costly.

With SCADE, this coding phase is completely automated by use of the certified code generator “SCADE Suite KCG™”. The C source code generated by KCG possesses properties essential for critical applications from a safety point of view [FOR 09].

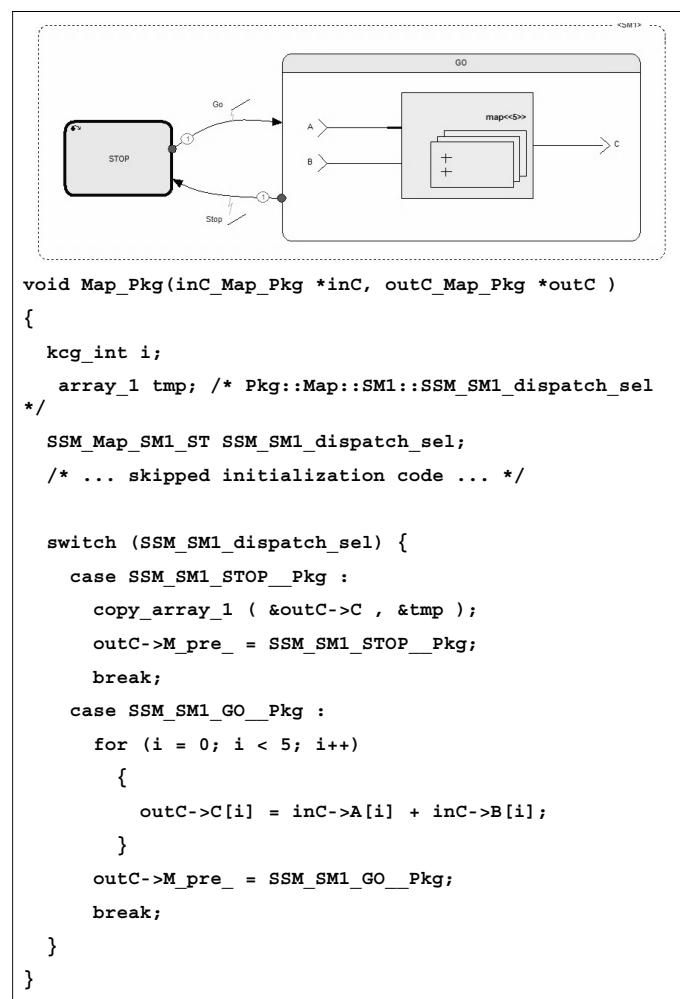


Figure 6.27. An example of code generated by KCG

The source code is traceable in relation to the model input, and it is readable. The code conforms to the ISO-C standard and is independent of the target platform. The determinism of the behavior of this code is guaranteed. It does not include dynamic memory allocation (an operation which could compromise good functioning of the application and which is, as a general rule, not permitted for this type of application). The code does not include arithmetic operations on the pointers.

Access to a table is not available outside of the bounds, etc. Figure 6.27 gives an example of C code generated from a state machine and a calculation in tables.

Certification of the KCG code generator is obtained on the basis of development and verification of this tool by Esterel Technologies in using a process of the same level as that which applies to the code that it will generate. For example, in order to achieve qualification of KCG at level A of the DO 178C standard, which is applied to the most critical avionics systems, it was necessary to develop KCG with the same requirements that apply at level A.

This allows KCG users to generate code for such an application, without having to verify that the code is correct in relation to the model, which constitutes its specification. Being able to bypass these low-level verification activities represents substantial savings regarding the total cost of a project.

6.7. Application of SCADE in the aeronautical industry

6.7.1. History: Aérospatiale and Thales Avionique

Aérospatiale developed the first electric flight control (*Fly By Wire*) for civil aircraft at the time of Concorde. Then, it involved analog computers.

The first generation of numerical flight command computers appeared at the beginning of the 1980s, with the A310, then the A320 in 1988.

From the Concorde years, engineers had got into the habit of describing control laws in the form of “box-and-arrow”-type diagrams, popular among control engineers in most companies. Aérospatiale, however, introduced the SAO (*spécification assistée par ordinateur* – computer-aided specification) approach to avoid the usual gap between control engineers and software engineer, which consisted of sharing these diagrams within the company and automatically coding from them.

In a first stage, the company developed its own internal toolchain, from the editing to the generating of code (GAC), via system simulation (OCASIME). Then with Merlin Gérin and Verilog, it devised the development of SCADE⁷ to:

- have a second code generator available to ensure dissimilarity between primary and secondary flight commands;
 - externalize tools not specific to the company (OCASIME for example remains an internal tool);
 - facilitate deployment of a homogeneous language and environment with its subcontractors.

6.7.2. Control/command type applications

The original domain of SCADE, which remains dominant, is that of control/command applications such as:

- flight control;
 - electrical load management;
 - engine control *Full Authority Digital Engine Control* (FADEC);
 - braking systems.

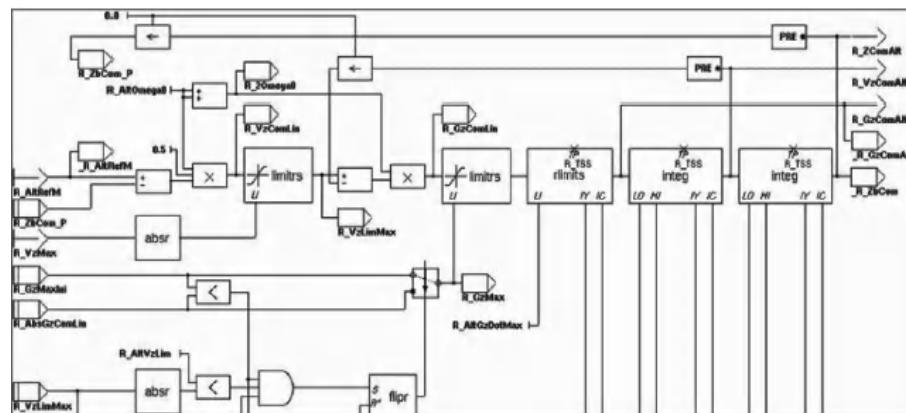


Figure 6.28. Fragment of a helicopter autopilot model

Thus, the flight control system of the Airbus A380 is modeled with close to 4,000 SCADE diagrams.

7 See section 6.3.5.

Aircraft/engine	Manufacturer	Equipment manufacturer	Equipment	Level
A340	Airbus	Airbus	<i>Flight Controls</i>	A
	Airbus	Intertechnique	<i>Electrical Load Management</i>	B
A350	Airbus	Airbus	<i>Flight Controls</i>	A
	Airbus	Intertechnique	<i>Electrical Load Management</i>	A
	Airbus	Intertechnique	<i>Electrical Distribution Management Unit</i>	B
	Airbus	Intertechnique	<i>Protection Device Monitoring & Management Function</i>	C
	Airbus	Intertechnique	<i>WindShield Heat Control</i>	A
	Airbus	Intertechnique	<i>Anti-ice Control</i>	A
	Airbus	Intertechnique	<i>Engine Interface Function</i>	A
	Airbus	Meggitt	<i>Fire Protection</i>	B
	Airbus	Thales	<i>Cockpit Display</i>	A
	Airbus	Thales	<i>Head-Up Displays</i>	A
A380	Airbus	Parker HSD	<i>Hylift Hydraulics</i>	B
	Airbus	Thales	<i>Air Data & Inertial Reference Unit</i>	A
	Airbus	Intertechnique	<i>Integrated Modular Avionics</i>	C
	Airbus	Thales	<i>Head-up Display</i>	
	Airbus	Airbus	<i>Flight Controls</i>	A
	Airbus	Airbus	<i>Flight Warning</i>	A
	Airbus	Airbus	<i>Data Concentrator</i>	A
A400M	Airbus	Hispano Suiza	<i>Thrust Reverser</i>	A
	Airbus	Intertechnique	<i>Electrical Load Management</i>	A
	Airbus	Intertechnique	<i>Fuel Quantity Data Concentrator - Monitoring Software</i>	A
	Airbus	Intertechnique	<i>IMA - Fuel Command Function</i>	A/B/C
	Airbus	Intertechnique	<i>IMA - Electrical System Functions</i>	C
	Airbus	Intertechnique	<i>Oxygen System Control Unit</i>	B
	Airbus	Airbus	<i>Flight Controls</i>	A
	Airbus	Airbus	<i>Flight Warning</i>	A
	Airbus	Airbus	<i>Loadmaster Logic</i>	B
	Airbus	Airbus	<i>Loadmaster Display</i>	B
	Airbus	Intertechnique	<i>Fuel Control</i>	A

Table 6.1. Use of SCADE in aeronautics (source: Esterel Technologies)

Aircraft/engine	Manufacturer	Equipment manufacturer	Equipment	Level
	Airbus	Intertechnique	<i>Anti Icing Control</i>	A
	Airbus	Saab Avitronics	<i>High-Lift Controls</i>	A
	Airbus	Sagem	<i>Navigation Unit</i>	A
	Airbus	Thales	<i>Cockpit Display</i>	A
	Airbus	Thales	<i>Cockpit Display</i>	A
ARJ21	COMAC	Meggitt	<i>Braking</i>	A
ATR-42	ATR	Thales	<i>Cockpit Display</i>	A
AW 149	Agusta Westland	Liebherr	<i>Flight Controls</i>	A
B 787	Boeing	GE Aviation	<i>Landing Gear</i>	A
Leap 56	COMAC	GE Aviation	<i>FADEC</i>	A
C series	Bombardier	Liebherr	<i>Air Management</i>	A
	Bombardier	Liebherr	<i>Landing gear</i>	A
	Bombardier	Meggitt	<i>Electrical Brake Controls</i>	A
C27J	Alenia	Intertechnique	<i>In-flight Refueling Controls</i>	B
CH-53GA	Sikorsky	Eurocopter	<i>Flight Controls</i>	A
CRJ 1000	Bombardier	Liebherr	<i>Rudder Controls</i>	A
EC 135	Eurocopter	Eurocopter	<i>Automatic Pilot</i>	A
EC 145	Eurocopter	Eurocopter	<i>Automatic Pilot</i>	A
EC 225	Eurocopter	Eurocopter	<i>Automatic Pilot</i>	A
EC 225	Eurocopter	Eurocopter	<i>Cockpit Display</i>	A
Phenom 100	Embraer	Meggitt	<i>Braking System</i>	A
Falcon 7X	Dassault Aviation	Dassault Aviation	<i>Flight Controls</i>	A
	Dassault Aviation	Meggitt	<i>Braking</i>	A
Falcon SMS	Dassault Aviation	Dassault Aviation	<i>Flight Controls</i>	A
G250	Gulfstream	Intertechnique	<i>Fuel Management</i>	B
G500	Gulfstream	Meggitt	<i>Braking</i>	A
G650	Gulfstream	Intertechnique	<i>Fuel Quantity Signal Conditioner</i>	B
Learjet LJ85	Bombardier	Liebherr	<i>Cabin Pressure</i>	A
	Bombardier	Liebherr	<i>Flight Control</i>	A
	Bombardier	Meggitt	<i>Fire Protection</i>	B

Table 6.1. (continued) Use of SCADE in aeronautics (source: Esterel Technologies)

Aircraft/engine	Manufacturer	Equipment manufacturer	Equipment	Level
	Bombardier	Messier Bugatti	<i>Braking</i>	A
	Bombardier	Messier Bugatti	<i>Landing Gear</i>	A
M88-2	Dassault Aviation	Hispano Suiza	FADEC	A
MRJ	Mitsubishi	Intertechnique	<i>Fuel Management</i>	B
MS21	UAC	Intertechnique	<i>Fuel Management</i>	B
Neuron UCAV	Dassault Aviation	Dassault Aviation	Flight Controls	A
PW210	Sikorsky	Pratt & Whitney	FADEC	A
PW535B	Cessna	Pratt & Whitney	FADEC	A
PW535E	Embraer	Pratt & Whitney	FADEC	A
PW610	Eclipse	Pratt & Whitney	FADEC	A
PW615F	Cessna	Pratt & Whitney	FADEC	A
PW617F	Embraer	Pratt & Whitney	FADEC	A
SuperJet 100	Sukhoi	Intertechnique	<i>Fuel Management</i>	B
	Sukhoi	Liebherr	<i>Flight Controls</i>	A
	Sukhoi	Liebherr	<i>Environmental control</i>	B
	Sukhoi	Liebherr	<i>Flight Control</i>	A
	Sukhoi	Messier Dowty	<i>Landing Gear</i>	A
Sam-146	Sukhoi	Hispano Suiza	FADEC	A
SMS	Dassault Aviation	Intertechnique	<i>Fuel Management</i>	B
SU-35	Sukhoi	Avionika	<i>Flight Controls</i>	A
Watchkeeper	Elbit/Thales	Elbit/Thales	<i>Flight Controls</i>	A
US Army Helicopter	US Army	Rolls Royce/AEC	FADEC	A

Table 6.1. (continued) Use of SCADE in aeronautics (source: Esterel Technologies)

6.7.3. Monitoring/alarm type applications

SAO then SCADE are the favored tools for monitoring and surveillance functions. In effect, this most frequently involves functions composed essentially of comparisons, logic gates, and hysteresis structured by states and transitions. SCADE is, for example, used for the alarm systems of the different Airbus craft or the fire alarms of the Learjet LJ600.

6.7.4. Navigation systems

Navigation systems present a particular challenge, as much for a manual development as for SCADE:

- they must merge data from different sources: inertial and baro-anemometric data, GPS, radar;
- they must get the best possible parts of this data contingent upon its quality (it is always susceptible to being degraded, not to mention unavailable);
- they execute high-frequency matrix calculations (for example, the Kalman filter).

Their development with versions preceding SCADE 6 had been impossible, as iterators were unavailable for this type of application. The navigation software of the A400M (GADIRS) was developed with SCADE 6. Its complexity is in the order of 100,000 lines of code.

6.8 Application of SCADE in the rail industry

Rail applications are most often highly critical. SCADE is used for applications as varied as:

- signaling and control of switches;
- Automatic Train Protection (ATP);
- *Communication-Based Train Control* (CBTC⁸ see [IEE 04]);
- *Automatic Train Operation* (ATO);
- automatic management of doors.

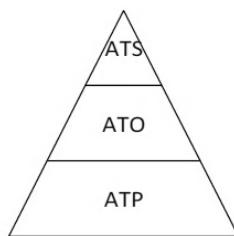


Figure 6.29. Hierarchical breakdown of a rail system

⁸ Chapter 1 of [BOU 11a] presents an example of CBTC.

Figure 6.29 presents the classic hierarchical breakdown of a rail system. ATS is responsible for traffic supervision (traffic control, power management, communication, etc.), ATO is responsible for managing operations (manual driving, automatic driving, etc.) and ATP is responsible for managing overall safety.

6.8.1. First applications

The first major application of SCADE in the rail industry was the re-engineering of the Hong Kong metro supervision system by CS Transport (currently Ansaldo STS) in 1999. This application had been deployed at 16 metro stations. It comprised 80,000 lines of code handling 2,705 inputs/outputs. It was developed in 12 man-months, which corresponds to a productivity of 300 lines per day in the place of the usual 20 for this type of application. Users had indicated that one of the unforeseen benefits was the detection of causality problems, thanks to the semantic checks of SCADE. In effect, this re-engineering work consisted in implementing the old relay logic into software, yet this type of system possesses a specific behavior due to physical inertia and propagation time. Naive translation into program statements may lead to unintended behavior.

6.8.2. Applications developed for the RATP and other French metros

Project	Equipment onboard	Ground equipment	Entered service
OCTYS L3	Atelier B (Siemens)	SCADE–Proof Toolkit (Ansaldo STS) ⁹	12/2009
SAET L1	Atelier B (Siemens)	Atelier B (Siemens)	2011
OURAGAN L13	SCADE (Thales RSS)	SCADE (Thales RSS)	2011
OCTYS L5	SCADE (Areva-TA)	Atelier B (Siemens)	2011
PMI L1, L2	N.A.	SCADE–Proof Toolkit (Thales) ¹⁰	2009 – 2010

Table 6.2. Tools used for the RATP CBTC (source RATP–SCADE User Group 2009)

⁹ To find out more on SCADE development in the context of the OCTYS L3 project, see Chapter 5 of the volume *Formal methods in the rail industry*.

¹⁰ To find out more on the use of proof tools in the context of the PMI of the OCTYS L1, see Chapter 4 of *Formal methods in the rail industry*.

The RATP has a long tradition of using formal methods, notably through the B-method [ABR 96]. SCADE is used more and more by RATP equipment manufacturers, owing to its ease of use and the productivity of development teams (see Table 6.2).

Renewal of the Lyon PA is in the process of development for a part with SCADE 5 and SCADE 6.

6.8.3. Generic PAI-NG applications

The PAI-NG¹¹ project (*poste d'aiguillage informatisé de nouvelle génération* – new generation computerized signal box) is occurring in the context of an ambitious French rail network renewal project. Its deployment will take about 15 years.

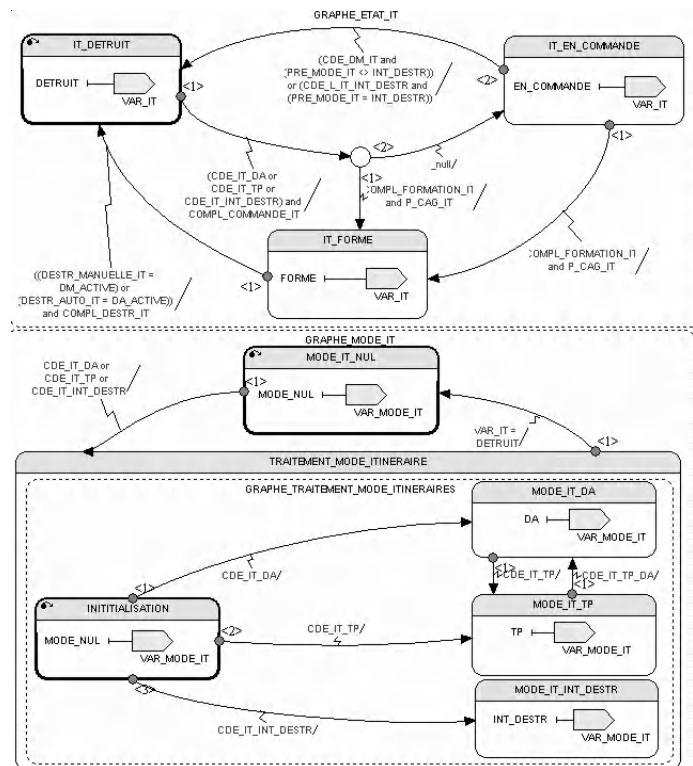


Figure 6.30. Example of a signaling principle in SCADE
(Ansaldo STS - SCADE User Group 2006)

11 For more information on the PAI-NG developed by THALES, see Chapter 5 of [BOU 11b].

The strategy of the SNCF consists of defining a generic specification for equipment, and having this equipment made by various equipment manufacturers. This generic specification is based on a set of “signaling principles” described in the form of state machines and interconnections between these machines.

Next, it is destined to be instantiated in various configurations according to the controlled zones. Some of them, notably stations, can be very complex (for example, 4,800 inputs).

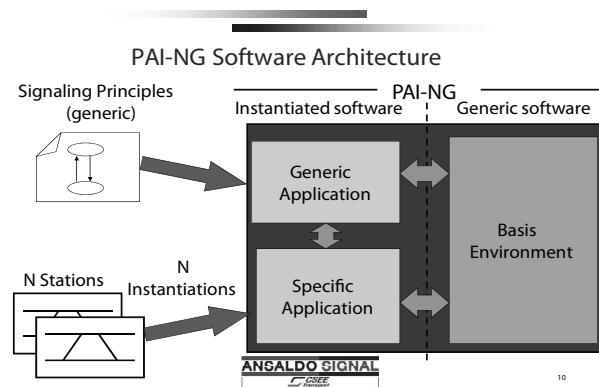


Figure 6.31. Principle of the PAI-NG system (Ansaldo STS – SCADE User Group 2006)

SCADE was adopted for PAI-NG production by Alstom Transport and Ansaldo STS. The Ansaldo STS application (Figure 6.31) was certified in 2010 and that of Alstom in January 2011. Each of the users had developed their own approach for implementing PAI-NG with SCADE, but with some common points:

- development of SCADE state machines corresponding to the signaling principles of the SNCF (see Figure 6.30);
- tool generating SCADE models integrating state machines and inputs/outputs, based on a model of the network topology;
- integrated simulation;
- lastly, generation of the final code with KCG and integration of the code with the manual code.

6.8.4. Example of automated door control

PSD System (a Korean engineering group) used SCADE to develop the command software for the entry doors of lines 2 and 4 of the São Paulo metro. This

SIL 3 application (Safety Integrity Level), had to be developed in 12 months. The project was finished within the deadline. The experience feedback provided by POSCON during the SCADE User Group 2009 was as follows:

- the functional approach enables an easy link with the structure of specifications;
- thorough upstream verification with the simulator and coverage measurement of the model eliminates most of the errors upstream of coding.

6.9. Application of SCADE in the nuclear and other industries

6.9.1. *Applications in the nuclear industry*

6.9.1.1. *Types of application*

SCADE has been used for *instrumentation and control applications* (Figure 6.33) as well as for *surveillance applications* (Figure 6.34).

6.9.1.2. *History*

During the 1980s, the Grenoble-based Merlin Gérin company had the responsibility of developing the SPIN (*système de protection intégré numérique* – integrated numerical protection system) computer. The level of criticality and complexity of this type of software prompted Merlin Gérin to innovate.

NOTE 6.3.– Merlin Gérin became Schneider Electric and is currently part of Rolls Royce Civil Nuclear.

Use of the Lustre language was introduced at Merlin Gérin in 1985, through collaborations between this company and l’Institut de mathématiques appliquées de Grenoble (IMAG) (Grenoble Institute of Applied Mathematics), birthplace of Lustre, as well as through recruitment of young engineers/researchers originally from this laboratory.

Merlin Gérin thus created the first editing and industrial code generation studio system based on Lustre in 1986, named SAGA.

Later, in 1991, Merlin Gérin adopted the approach of industrializing this type of equipment with Aérospatiale and the tools editor Verilog, as described in section 6.3.5.

SCADE was then used in 1996 for the development of instrumentation/control of the Kozloduy power station (Bulgaria), for Bugey and Fessenheim in 1998, then for all the following power stations.

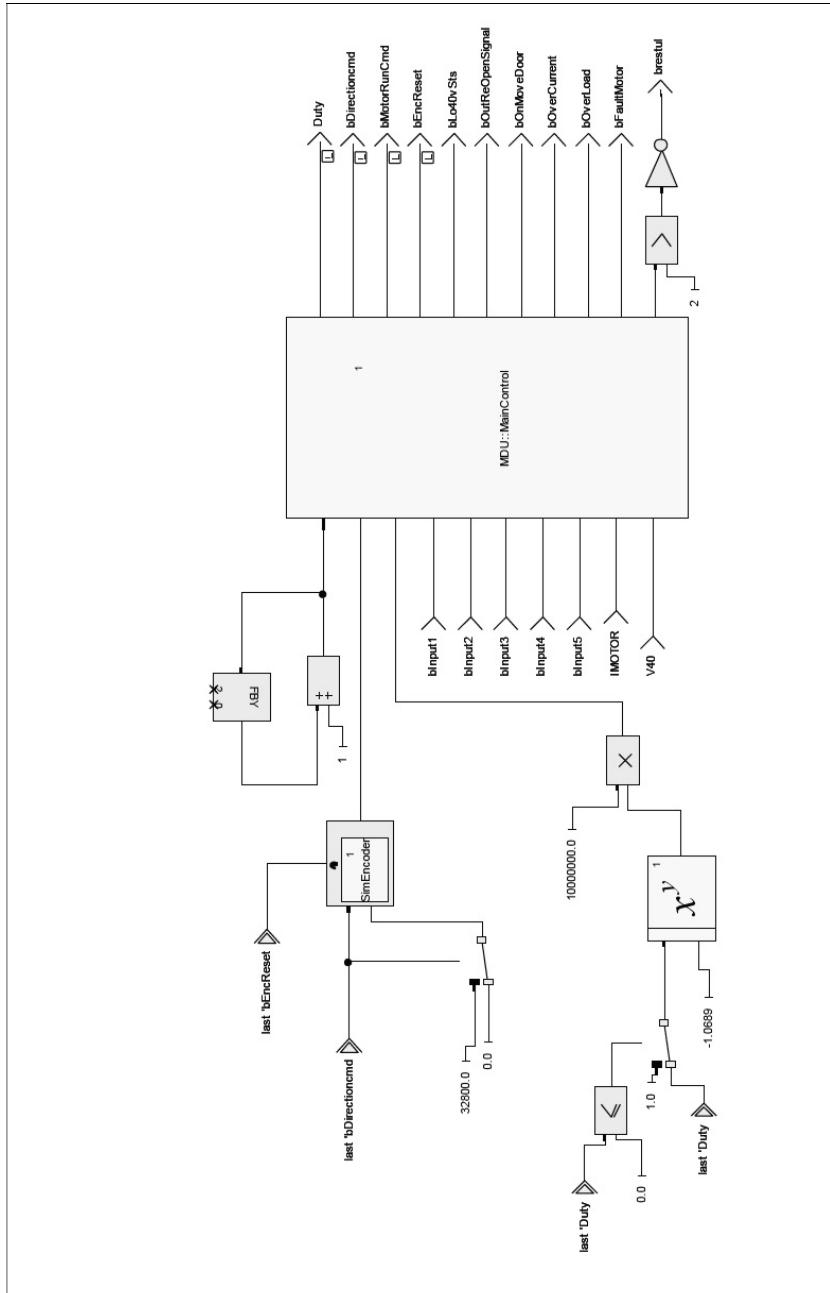


Figure 6.32. Fragment of the SCADE model of São Paulo metro door control

For the last generations (SPIN N4), the code generated (200,000 lines of code) represents 95% of the software.

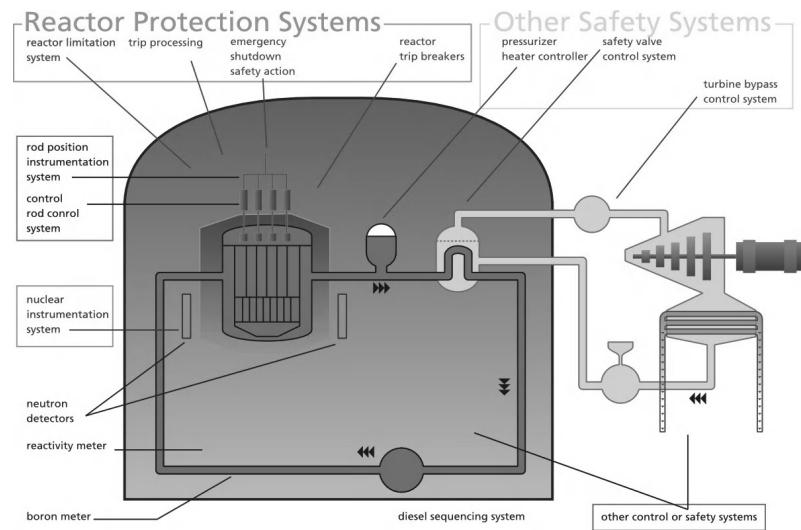


Figure 6.33. Instrumentation and control applications

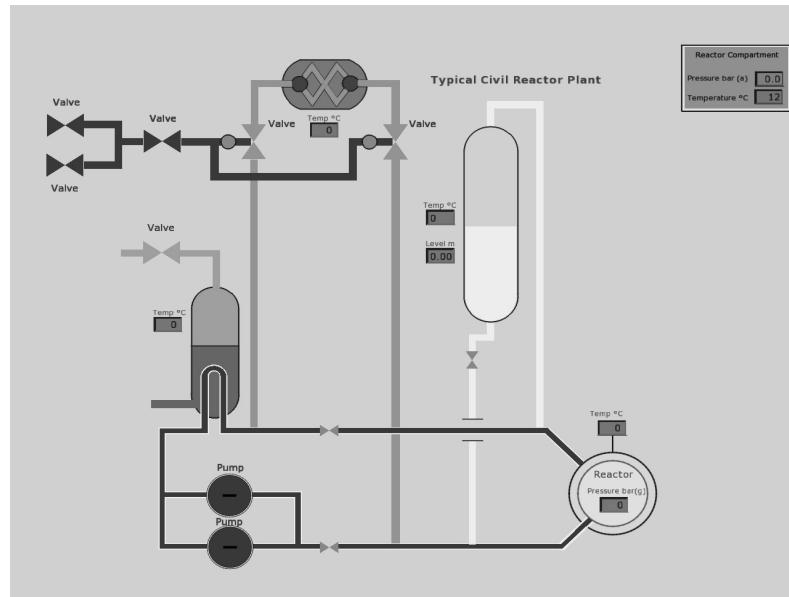


Figure 6.34. Surveillance applications

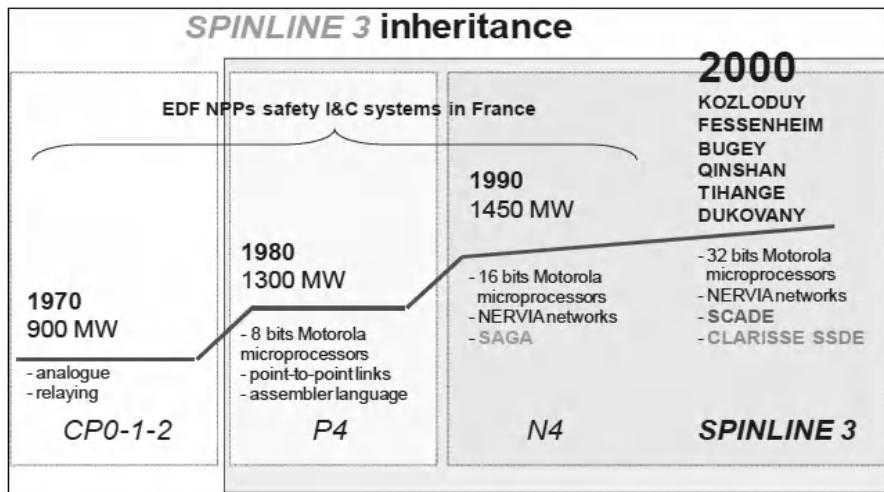


Figure 6.35. SPINLINE history

6.9.2. Deployment of SCADE in the civil nuclear industry

SCADE was adopted by other companies besides Schneider Electric, as Table 6.3 shows.

Company	Country
AREVA NP	France
BARC	India
IGCAR	India
Rolls-Royce Civil Nuclear	France, UK
Schneider Electric	France
KAERI	Korea
KOPEC	Korea
NPIC	China

Table 6.3. Companies using SCADE in the civil nuclear industry

There are currently 37 reactors in production, of which one part of the control/command systems has been developed thanks to SCADE (Table 6.4).

Place	Power station type	Country
BELLEVILLE 1, 2	Production	France
BUGEY	Production	France
CATTENOM 1, 2, 3, 4	Production	France
CHOOZ B1, B2	Production	France
CIVAUXT 1, 2	Production	France
FLAMANVILLE 1, 2	Production	France
FESSENHEIM 1, 2	Production	France
GOLFECH 1, 2	Production	France
NOGENT 1, 2	Production	France
OSIRIS	Experimentation	France
PALUEL 1, 2, 3, 4	Production	France
PENLY 1, 2	Production	France
ST ALBAN 1, 2	Production	France
METSAMOR	Production	Armenia
TIHANGE 1	Production	Belgium
QINSHAN II 1, 2	Production	China
DUKOVANY 1, 2, 3	Production	Czech Republic
IGNALINA 1, 2	Production	Lithuania

Table 6.4. Nuclear power stations including software developed with SCADE

6.10. Conclusion

In this chapter, we have briefly presented the SCADE formal notation and the associated development environment. Its applications in the railway and civil nuclear industries has been outlined.

SCADE has been increasingly adopted for the development of safety critical software by aeronautic equipment manufacturers, for rail systems and more recently for construction machines. This confirms the validity of the combination of intuition and rigor of the creators of Lustre, SAO and Saga.

6.11. Bibliography

- [ABR 96] ABRIAL J.R., *The B-Book – Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.
- [ANS 83] ANSI, Norme ANSI/MIL-STD-1815A-1983, Langage de programmation Ada, 1983.
- [BAU 10] BAUFRETON P., BLANQUART J.-P., BOULANGER J.-L., DELSENY H., DERRIEN J.-C., GASSINO J., LADIER G., LEDINOT E., LEEMAN M., QUERE P., RICQUE B., “Multi-domain comparison of safety standards”, *ERTS2 Conference*, 2010.
- [BEN 03] BENVENISTE A., CASPI P., EDWARDS S.A., HALBWACHS N., LE GUERNIC P., DE SIMONE R., “The synchronous languages 12 years later”, *Proceedings of the IEEE*, vol. 91, no. 1, January 2003.
- [BER 00] BERRY G., “The foundations of esterel”, *Proofs, Languages and Interaction, Essays in Honour of Robin Milner*, MIT Press, Cambridge, MA, 2000.
- [BOU 11a] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques industrielles*, Hermès Lavoisier, Paris, 2011.
- [BOU 11b] BOULANGER J.-L. (ed.), *Utilisations industrielles des techniques formelles – interprétation abstraite*, Hermès Lavoisier, Paris, 2011.
- [CEN 01] CENELEC, EN 50128, Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, May 2001.
- [COL 05] COLAÇO J.-L., PAGANO B., POUZET M., “A conservative extension of synchronous dataflow with state machines”, *EMSOFT*, 2005.
- [FOR 09] FORNARI X., “Understanding how SCADE suite KCG generates safe C code”, *Esterel Technologies*, www.estrel-technologies.com, 2009.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous dataflow programming language lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305-1320, September 1991.
- [IEC 98] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, IEC, 1998.
- [IEC 06] IEC 60880, Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions, IEC, 2006 (2nd edition).

- [IEE 04] IEEE, 1474.1, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements, 2004.
- [ISO 09] ISO 26262, Road vehicles – Functional safety, ISO, 2009.
- [LEG 91] LE GUERNIC P., GAUTIER T., LE BORGNE M., LE MAIRE C., “Programming real-time applications with SIGNAL”, *Proceeding IEEE*, vol. 79, p. 1321-1336, September 1991.
- [OMG 10] OMG Systems Modeling Language (OMG SysML), v. 1.2, June 2010.
- [POL 05] POLCHI J.-F., “Développement système. Un exemple avec l’outil SCADE: les commandes de vol Airbus”, *AFIS/Journées outils de l’ingénierie système*, Toulouse, 2005.
- [RTC 11-02] DO-330, Mode-Based Development and Verification Supplement to DO-178C and DO-278A, RTCA Inc, 2011.
- [RTC 11-1] DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc, 2011.
- [UIT 94] UIT Z-100: CCITT – Specification and description language: SDL, technical report, UIT, Geneva, 1994.

Chapter 7

GATeL: A V&V Platform for SCADE Models

7.1. Introduction

The use of a formal specification in a development process is of great interest even if lack of time or resources limits its application. Indeed, a formal specification provides a concise and precise document for testing purposes: it gives a solid basis for oracle decisions, and, when executable, it allows the automation of test data generation and submission. We present a model-based testing technique from SCADE models supported by the tool GATeL [MAR 91, BLA 03].

SCADE is a specification and programming language for the description of synchronous data-flow computations. It is used for reactive control/command systems, mainly for aeronautics control and electrical power production applications. This language allows mixed graphical and textual descriptions into the corresponding SCADE SUITE¹. Historically, the language was first promoted through its academic version called Lustre [HAL 91]. Note also that some SIMULINK models can be imported in SCADE through a dedicated gateway.

While this language benefits from powerful verification tools (using model-checking techniques: DV², Abstract Interpretation as in NBac [GAU 03], or K-Induction as in KIND [HAG 08]), there is still a demand for adequate testing techniques. Several methods and tools have been proposed for the testing of synchronous reactive systems. They include those based on a Lustre model

Chapter written by Bruno MARRE, Benjamin BIANC, Patricia MOUY and Christophe JUNKE.

1 <http://www.estrel-technologies.com>.

2 <http://www.prover.com>.

such as Lutess [BOU 99] and Lurette [RAY 98]. Lutess and Lurette both consider the testing of reactive programs in a black-box framework. Lustre is not imposed for the system specification but is used for the specification of the environment in which the program is embedded. Input of the program under test at each cycle must then conform to this specification. The computed output is then checked against an oracle defined by invariant (safety) properties.

GATEL is a complete testing platform supporting test sequence generation from SCADE models. It can be called either directly from SCADE SUITE or through its own GUI. Test sequence generation is directed to specific test objectives: a test objective can be a safety property or a declarative characterization of some interesting states of the system under test, expressed with a `reach` directive specific to GATEL. The characterization of the states to be reached is a Boolean property expressed in SCADE, so it is possible (with the help of temporal operators) to test any property that can be expressed as an observation of the past. The model itself and the test objective are translated into a constraint system using an interpretation of the language constructs as Boolean, numerical interval constraints and guarded constraints (for the handling of control and temporal operators). Test sequence generation is then automated using constraint logic programming techniques. The use of a specialized constraint solver for the generation of test data allows us to precisely fit it to our needs.

In most model-based approaches, a particular testing strategy is defined according to the underlying model. The strategy usually relies on a predefined coverage criterion chosen according to the structure of the model. For instance, if the model belongs to the finite state machine family, the criterion can be to cover all states, or all transitions, or both. Concerning state-based models, it can be coverage of boundary values of state components, or coverage of operator sub-cases (e.g. branches of if-then-else statements). Each criterion addresses a particular class of faults. However, for the certification of highly critical systems, certification agencies must apply the diversification paradigm when they have to validate such systems. In the context of software testing, this means that they must be able to apply their own test selection strategies. For these reasons, we preferred in GATEL to provide the user with the basic mechanisms allowing the definition of customized selection strategies for the application under test.

After a brief presentation of SCADE and the resolution procedure used in GATEL, we will present these basic selection mechanisms in the rest of the chapter. The first relies on the notion of test objective, and we will take a microwave oven controller as a simple example. We will then explore the technique of interactive domain splitting by examining sub-cases of the initial constraint system. This technique is based either on predefined operator sub-cases or user-defined integration scenarios.

7.2. SCADE language

SCADE and its academic initiator Lustre belong to the synchronous data-flow language family. SCADE provides both a textual and graphical notation, the latter being similar to those used in hardware design (block diagrams, operators net, etc.). The underlying model of a SCADE program is that of a time-driven automaton. It describes cyclic behavior between two consecutive ticks of a global clock. At each tick, it gets all its input data and computes it so as to define the corresponding output. Each input (or output) data-flow is the sequence built from the successive data received (or emitted) during the temporal run of the program. The latest version of the SCADE language, Scade6, embodies a well-known feature of I&C programs: their decomposition into different modes. Modes are represented at a high level using state-machines.

Programs and sub-program blocks are called *nodes*. Each computed variable (output or local) is defined by a single equation in each mode. At the node level, these equations are combined so as to form a comprehensive view of the variable.

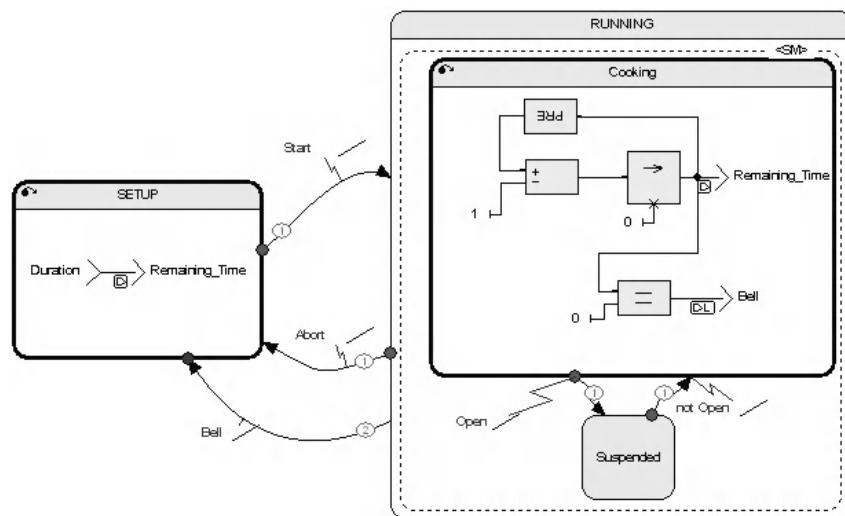


Figure 7.1. A simple microwave oven controller

Let us consider the simplified model of a microwave oven controller above. The reaction is described with four inputs and three outputs. The Boolean input data flows Start and Abort correspond respectively to the start and abort buttons of the oven. They are true when the user pushes the corresponding button. The input Open is set by a sensor on the oven door; it is true while the door remains open. The input Duration is the cooking duration in seconds programmed by the user. The output

parameter `Remaining_time` is the remaining time until the end of cooking, which is displayed on the screen of the oven according to the `Cooking` variable. The Boolean output `Bell` triggers the ringing of a bell when cooking is finished. Let us describe the definition of the local variable `Remaining_time`. It is set to the input `Duration` in mode `Setup`, and decreased by 1 (assumed to be the cycle duration) in the mode `Cooking`. Finally, in the mode `Suspended`, it is maintained to its previous global value (as opposed to the previous value when it was in the same mode).

Invariant properties can be stated with the `assert` directive, which operates on a Boolean expression that must be satisfied at each cycle.

```
assert not (Start and Abort) ;
assert (Duration > 0) and (Duration <= 3600) ;
assert implies(Start, not(Open) and (true -> not(pre(Open))));
```

The last one states the user behavior for `Start`: it can only be pressed when the door has been closed for the last two cycles.

An interesting objective to reach would be a completed cooking session, which is an occurrence of `Bell`. This is stated by the following sentence added in the description of the node:

```
(*! reach Bell !*)
```

7.3. GATeL prerequisites

A testing method usually follows three classical steps: test case *selection* and input data *generation, submission, oracle*. GATeL provides mechanisms allowing a tester to define his own selection strategy. It completely automates the generation of input sequences for each test case derived from the selection strategy. GATeL also provides the information needed to construct an oracle. Our tool systematically computes from the SCADE model, input, output and truth values of the test objective at each cycle. This evaluated output constitutes the expected ones, which should be compared to actual output of the program under test, and thus represents a partial oracle. The mechanisms proposed to assist the definition of selection strategies use some of the control features of the resolution procedure involved in test sequence generation. Before going further, we first describe the principles of our resolution procedure.

7.3.1. GATEL kernel

Given a Lustre model and a reachability objective, GATEL automatically generates backward a sequence of input leading to this objective. From a model gathering the model with an identified root node and a reachability objective, GATEL first compiles these elements into a static logic program $\mathcal{P} = (\mathcal{V}ars, \mathcal{E}, \mathcal{R})$. $\mathcal{V}ars$ is a set of Lustre identifiers relevant to the given objective. $\mathcal{E}(Id)$ is a term corresponding to the defining equation of output or local identifier $Id \in \mathcal{V}ars$. \mathcal{R} is a Boolean term denoting the reachability objective. These terms are naturally built over an abstract syntax reflecting all Lustre operators and logical variables in place of Lustre identifiers. Each logical variable has a domain according to the type of its corresponding Lustre identifier:

- **bool** is interpreted in the $[f, t]$ domain;
- **int** is interpreted in standard interval integer arithmetic;
- **real** is interpreted either in floating point interval arithmetic or in real interval arithmetic;
- **enum** types are interpreted in their corresponding finite domain.

This program \mathcal{P} is used by the main component of GATEL: its resolution engine is shown in the following algorithm. The goal of this engine is to produce a matrix \mathcal{M} whose rows are identifiers from $\mathcal{V}ars$ and columns are cycles. For each known cycle C , $\mathcal{M}(Id, C)$ stores the logical variable associated with Id whose domain is initialized according to its type. New cycles are created on demand by the resolution engine, extending \mathcal{M} accordingly. The algorithm follows a standard DPLL scheme (following SMT-based solvers), alternating between deterministic and non-deterministic phases. The deterministic one is managed by a set of filtering rules, aiming at generating values in \mathcal{M} according to the definition of Lustre operators.

Algorithm 7.1. GATEL resolution algorithm

Require: Logic program \mathcal{P}
Ensure: Test sequence matrix \mathcal{M}

1. $\Gamma \leftarrow \mathcal{M} = \emptyset, \mathcal{S} = \emptyset, C_M(\text{base}) = 0, S_M(\text{base}) = -$
2. $\Gamma \vdash \{\mathcal{R} = t, 0\}$
3. **while** $\mathcal{S} \neq \emptyset$ **do**
4. $V, C = choose_var(\mathcal{S})$
5. $v = choose_val(dom(V))$
6. $\Gamma \vdash \{\text{var}(V) = v, C\}$
7. **end while**

The resolution algorithm 1 initializes the environment with an empty matrix \mathcal{M} and residual store \mathcal{S} . The highest known cycle C_M of the system is set to zero and its

status S_M (initial or non-initial) is free (noted $_$). It then starts with a first deterministic phase launched with the reachability objective \mathcal{R} set to true. The current cycle for these two equations is zero, which represents the final cycle of the sequence and the first column of matrix \mathcal{M} . Cycles are numbered backward from the last cycle to an initial one. The deterministic phase goes on until a fixed point is reached over the set of filtering rules. Note that this filtering may fail when detecting an unsatisfiability.

The algorithm then proceeds as a standard DPLL algorithm: a variable is chosen among the variables occurring in some residual of \mathcal{S} , a value is chosen in the allowed domain of this variable, and the corresponding equation is filtered until another fixpoint is reached. Backtracking due to a failure of the filtering is managed by the non-deterministic *choose_val* procedure, iterating over possible values, then through the **while** to return to a higher level of choice. To get a terminating algorithm, global bounds on the length of the sequences (noted Max_C) and on the numerical domains are positioned by the user. When the algorithm stops with no solution (failure for each valuation of \mathcal{M}), this only implies that there is no solution within the user bounds.

Examples of the filtering rule include INIT-T: it states that if the current cycle is the highest known one and its status is *initial*, then the filtering procedure applies on its first argument. Rule INIT-F is complementary: when the current cycle is smaller than the highest known cycle C_{Max} , this means that C is non-initial. Therefore the filtering procedure applies on the second argument.

$$\frac{\begin{array}{c} Status = initial \quad C_{Max} = C \quad \Gamma \vdash (X \rightarrow Y = R, C) \\ \hline \end{array}}{\Gamma \vdash (X = R, C)} \text{ INIT-T}$$

$$\frac{C_{Max} > C \quad \Gamma \vdash (X \rightarrow Y = R, C)}{\Gamma \vdash (Y = R, C)} \text{ INIT-F}$$

More details on GATEL filtering rules are given in [BLA 03].

7.3.2. Example

Loading the example model in GATEL leads to this partial sequence:

#Cycle	Start	Abort	Open	Dur	Rem_time	Cooking	Bell
?
1	_	false	false	1..3600	1	-	-
0	false	false	false	1..3600	0	true	true

Indeed, due to the test objective, the values of some flows are known at the last cycle (cycle 0) and at a previous cycle. These values and the existence of at least two cycles

are direct consequences of the testing models deduced by the initial deterministic propagation step. The generation of a sequence for this test objective can give, for instance, the completed cooking session of 4 cycles duration:

#Cycle	Start	Abort	Open	Dur	Rem_time	Cooking	Bell
4	true	false	false	4	4	true	false
3	false	false	false	4	3	true	false
2	false	false	false	4	2	true	false
1	false	false	false	4	1	true	false
0	false	false	false	4	0	false	true

Due to random instantiations during labeling steps and constraint propagation, another run would lead to sequences of different length with different values. Thus, we cannot guarantee minimality of the generated test sequences. However, we can exhibit instances of behavior that can further be differentiated (see section 7.4).

7.4. Assistance in the design of test selection strategies

At this point, we have just shown how to select one test case using a test objective. This shows how to use GATEL in full automatic mode. However, keeping only one solution of the constraints system amounts to considering that each solution has an equal interest. In other words, we assume that any test sequence reaching the objective has the same power to reveal faults. Such *uniformity hypotheses* are common but often too strong. Test cases are smaller domains on which these hypotheses get more realistic.

There are two different ways to describe test cases in GATEL. The first one relies on an interactive unfolding of SCADE operators of the current constraint system. In this case, test cases are defined by a structural decomposition of the initial constraint system. This method allows a fine-grained coverage of SCADE expressions, and thus is best used within unit testing. The second one uses predefined functional scenarios attached to variables. A scenario can be seen as a high-level splitting method, since the user identifies the parts of the domain to explore, only exhibiting particular instances of behavior among all the possible ones.

7.4.1. Unfolding of SCADE operators

We propose a splitting technique of the current constraints system that makes it possible to recursively split sub-domains according to predefined sub-cases of SCADE operators.

For example, for a constraint “ $S_i = \text{if } \text{Cond}_i \text{ then ExpThen else ExpElse}$ ” where Cond_i is a variable at the cycle i , by unfolding of `if then else`

we can derive two sub-domains. The first sub-domain includes all test sequences such that Cond_i is true, while the second sub-domain includes all test sequences such that Cond_i is false. These two sub-domains are characterized by the constraints systems obtained after propagation of Cond_i valuations. They can be split again by unfolding an operator occurring in their constraints.

At each unfolding step, GATEL shows the operators that can be unfolded. Thus, the user can interactively and dynamically tune the type and number of sub-domains. Furthermore, domain splitting can be applied to the constraints system defining an output chosen by the user. In this case, unfolding builds the path predicates of the selected output at an arbitrary cycle. In this way a *structural* coverage of the expressions involved in the computation of the selected output can be obtained.

Here is an excerpt of the list of operators that can be unfolded by GATEL (and their sub-cases):

- $A_i = \text{not}(\text{Exp})$: 2 cases corresponding to A_i valuations;
- $A_i = \text{Exp}_1 \text{ and } \text{Exp}_2$: 3 possibilities tunable by user:
 - sequential and (default): 2 cases corresponding to A_i valuations,
 - lazy and: 3 cases,
 - 1. $A_i \leftarrow \text{true} \wedge \text{true} = \text{Exp}_1 \wedge \text{true} = \text{Exp}_2$,
 - 2. $A_i \leftarrow \text{false} \wedge \text{false} = \text{Exp}_1$,
 - 3. $A_i \leftarrow \text{false} \wedge \text{false} = \text{Exp}_2$;
 - normal and: 4 cases corresponding to its truth table,
- $A_i = \text{Exp}_1 ==< \text{Exp}_2$: 2 cases (A_i valuations) or 3 cases ($=, <$ or $>$), tunable by user:
 - $A_i = \text{Exp}_1 -> \text{Exp}_2$:
 - 1) the status of cycle i is `initial`,
 - 2) the status of cycle i is `non_initial`.

If a complete coverage is sought, a systematic unfolding could be undertaken. This would lead to an “all paths” coverage criterion. However, due to the presence of temporal operators, an infinite unfolding (in fact, bounded by a global parameter) would be possible. It is because of this specificity of SCADE that we believe interactive operation is more appropriate.

Suppose one wants to observe two different types of sequence for the test objective given in the previous section: those where the door remains closed during cooking, and others where the door has been opened during cooking, both without abort since the beginning of cooking. This corresponds to sequences that passes through the Suspended mode of the model or not. We will try to characterize some test sequences for both cases by successive unfolding of SCADE operators.

After loading the former model, a first way to proceed consists of a selection of unfoldable operators from the SCADE definition of “active” variables (whose definition has already been propagated). Several operators can be selected at each step, each operator can be unfolded at several cycles and in several cases. Since a passage through the Suspended mode requires more than 2 cycles, the first selection step is to choose the \rightarrow operator at cycle 1. Selecting this operator, we get the two sub-domains corresponding to the possible status of cycle 1. If cycle 1 is initial, we get a partial sequence of two cycles where the door was not opened. Otherwise (if cycle 1 is not initial) another previous cycle is created. In this second case, the user can then select the transition from Suspended to Cooking at cycle 1 through a merge operator. This leads to two new sub-cases, the first one where the preceding state was Cooking, the second one where the preceding state was Suspended. We then stop domain splitting and ask for the generation of one test sequence for each of these three sub-domains.

- 1) The first test sequence reaches the objective when cycle 1 is set to be the initial cycle:

#Cycle	Start	Abort	Open	Dur	Rem_time	Cooking	Bell
1	true	false	false	1	1	true	false
0	false	false	false	1	0	false	true

- 2) The second test sequence reaches the objective when this Start is a re-start (Remaining_time is maintained). Thus, Open must occur at cycle 2 to create this situation. This sequence is then finished to complete the test objective.

#Cycle	Start	Abort	Open	Dur	Rem_time	Cooking	Bell
4	true	false	false	1	1	true	false
3	false	false	true	1	1	false	false
2	false	false	false	1	1	false	false
1	true	false	false	1	1	true	false
0	false	false	false	1	0	false	true

- 3) The last sequence corresponds to normal cooking with any duration time.

#Cycle	Start	Abort	Open	Dur	Rem_time	Cooking	Bell
2	true	false	false	2	2	true	false
1	false	false	false	2	1	true	false
0	false	false	false	2	0	false	true

7.4.2. Functional scenarios

Unfolding is a simple answer to the need for splitting the domain of a test objective. However, for methodological reasons, this process may not be applicable on complex examples. When the decomposition sought implies several variables at several cycles, the choice of the right operator to unfold may become harder. This is also the case even with simpler decompositions, when many operators are unfoldable. Moreover, as we saw in the above example, the splitting process may create auxiliary test cases, which complicate the examination of the generated test sequences. For

instance, only two cases in four were really needed (2 and 3), while the other two are superfluous.

For these reasons, we also propose a `split` directive in GATEL to attach predefined functional scenarios to one variable. Each scenario represents one expected case of the decomposition and is defined as any Boolean SCADE observation of the past. This directive follows the syntax:

```
(*! split var with [case_1,...,case_n] !*)
```

When activated, this directive splits the global system into n cases containing respectively each Boolean expression $case_i$ constrained to be true. Notice that to be activated, this directive needs the definition of its attached variable to have been introduced into the constraint system (either due to direct constraint propagation or due to an interactive unfolding).

Let us consider again the decomposition of the previous section. To force this decomposition with a `split` directive, two scenarios are defined: one where the door remains closed during cooking, and another one where the door has been opened during cooking, both without abort since the beginning of cooking. With these scenarios, the test objective is simplified as follows:

```
(*! reach Bell !*)
(*! split Bell with [
    Bell and never_since_last(Open,Start),
    Bell and current_when_bool(
        (Duration > Remaining_time) and
        (Remaining_time > 0),
        Start)] !*)
```

The `current_when_bool` operator states that when the `Start` variable is true, then `Remaining_time` is less than initial duration and strictly positive, which corresponds to a restart configuration.

When the corresponding SCADE model is loaded, the `split` directive is directly unfoldable. When selected, two test cases are created according to each predefined scenario. The generation of test sequences then gives for example two sequences similar to sequences 3 and 4 above.

This directive may also be seen as an integration testing technique. When declared in an embedded node, it defines the integration strategy of related output very early in the development process, by stating functional scenarios for them. In such situations, unfolding techniques are complementary since the activation of `split` directives requires the attached variable to be present in the current constraint system.

7.5. Performances

GATEL performances allow complex SCADE models to be treated. It is especially efficient for models containing timers, state-machines control, or floating point computations.

For instance, it is commonplace to find large confirmation time in nuclear plant protection units. In three case studies, GATEL was able to generate test sequences for test objectives involving thousands of cycles before the alarm was raised. The first objective given by Schneider-Electric was to enter the alarm mode of the neutronic counter of a nuclear plant. This mode requires you to exhibit a divergence of this counter for exactly 1,000 cycles. It takes GATEL 3s to generate such a sequence. The second one was given by IRSN (French nuclear safety agency) in the BE-SECS European project. One of the test objectives is reached in 80 cycles, each containing 247 procedures call, and involving 10,000 constraints altogether. The generation of test sequences takes 4s. The third one was given by Rolls-Royce CN, and involved safety properties of a TDF component. One of these properties was to reach an alarm after 30,000 cycles and 100,000 constraints. GATEL took 1 hour to generate such a sequence.

Concerning state-machine models, our tool was challenged on the description of the MSU (Monitoring and Safing Unit) of the automatic vehicle (ATV) from Astrium. The MSU controls the nominal navigation toward the International Space Station, and defines degraded modes in which it is supposed to avoid a collision with the station when the trajectory is not safe anymore. The Scade6 model contains a hierarchical state-machine describing this procedure. The highest level of the state-machine has five states to control the triggering of this procedure (*StandBy*, *Ready*, *Waiting*, *InProgress*, *Inhib*). The corresponding transitions are managed by global conditions (sensor status, configuration, position, etc.). Once entering the state *InProgress*, the procedure evolves intrinsically, then finishes in state *Inhib*. The procedure itself consists of successive temporizations, each one corresponding to the time needed to conduct specific operations on actuators. The test objective was thus to generate a sequence covering a complete collision avoidance procedure. GATEL was able to generate 50 such cycles in sequence, involving 1800 I/O.

Finally, we conducted several experiments involving floating-point computations. As mentioned in section 7.3, our tool allows several semantics for the *real* data-type: either an approximate but conservative real one, or an exact floating-point one. The discrepancies between these two often exhibit dangerous modelization choices when it comes to precision. Moreover, floating-point computations are often a weak point of usual model-checking tools.

7.6. Conclusion

We describe various test selection techniques from SCADE models using the tool GATEL. The SCADE language is declarative and describes synchronous data-flow computations. Our test generation tool interprets the language constructs as boolean, numerical, and temporal constraints. Test sequence generation is automated using constraint logic programming techniques. GATEL provides various mechanisms to allow testers to define their own selection strategies. They are illustrated on an simple example.

GATEL provides several basic mechanisms to define selection strategies: generation of a sequence leading to a test objective, interactive domain splitting from predefined operators sub-cases, or user-defined subdomains. As discussed in the introduction, no particular strategy has been defined. Our choice was to allow the users to fine-tune their own selection strategies according to the testing context. Classical strategies for unit or integration testing can nevertheless be implemented through unfolding of operators and split directives. Branch testing or bounded path testing can be achieved by a systematic unfolding of Boolean operations. A systematic unfolding of comparisons and Boolean decisions is a way to provide some boundary testing. Concerning integration testing, sub-cases defined by split directives give a functional decomposition, which abstracts the behavior of the integrated components.

On the other hand, when the tester suspects some special types of behavior to be badly implemented, it is important to provide him or her with the means to focus on the relevant parts of the model. The fine-tuning of predefined operator case analysis combined with interactive unfolding and split directives make it possible to discard many irrelevant details during test case selection. We are aware that the use of GATEL requires some understanding of the propagation mechanism, particularly during interactive domain splitting. This difficulty has been alleviated by the many ways in which information from GATEL's constraint store is made available to the user (analysis and pretty-printing of current constraints, choice tree presentation, etc.), and by navigation facilities between the various representations.

In GATEL, a test case is characterized by a partial instantiation of relevant data flows and a constraint system. Given a test sequence provided by an external source, it is possible to check whether such a test case is covered or not by this sequence. This can be achieved through sequence overlapping and constraint resolution. Generalized to several test cases and sequences, this can be used to qualify these sequences for acceptance testing or certification purposes (as done by IRSN). In case of a lack of coverage in the resulting matrix, GATEL can be called to check whether a sequence can be found or not. This GATEL fonctionnality of test campaign evaluation is also available for the MTC tool in SCADE SUITE, but for DC or MC/DC criteria.

7.7. Bibliography

- [BLA 03] BLANC B., JUNKE C., MARRE B., LE GALL P., ANDRIEU O., “Handling state-machines specifications with GATEL MBT 10”, *ENTCS*, vol. 264, no. 3, Springer-Verlag, January 2003.
- [BOU 99] DU BOUSQUET L., OUABDESELAM F., RICHIER J.-L., ZUANON N., “Lutess: a specification-driven testing environment for synchronous software”, *21st International Conference on Software Engineering*, ACM, May 1999.
- [HAG 08] HAGEN G., TINELLI C., “Scaling up the formal verification of lustre programs with SMT-based techniques”, *Proceeding of 8th International Conference on Formal Methods in Computer-Aided Design*, Portland, Oregon, IEEE, 2008.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous data flow programming language lustre”, *Proceeding of the IEEE*, vol. 79, no. 9, p. 1305-1320, September 1991.
- [MAR 91] MARRE B., ARNOULD A., “Test sequences generation from lustre descriptions: GATEL”, *ASE '00, Fifteen IEEE International Conference on Automated Software Engineering*, p. 229-237, IEEE Computer Society Press, 1991.
- [RAY 98] RAYMOND P., WEBER D., NICOLLIN X., HALBWACHS N., “Automatic testing of reactive systems”, *19th IEEE Real-Time Systems Symposium*, IEEE, 1998.
- [GAU 03] GAUCHER F., JAHIER E., JEANNET B., MARANINCHI F., “Automatic state reaching for debugging reactive programs”, *5th International Workshop on Automated and Algorithmic Debugging*, September 2003.

Chapter 8

ControlBuild, a Development Framework for Control Engineering

8.1. Introduction

For some years now, progress in transport has increased continually and shows no sign of stopping any time soon. In effect, more and more towns want a tramway or metro of their own. To meet these demands, train builders have had to rival each other in ingenuity so as to provide as many new trains and innovative functionalities as possible within deadline and under budget.

Indeed, electronics have become widely present in transport systems (see Figure 8.1), as between 50 to 200 computers/ECUs (*Electronic Control Units*) can be counted in one train. Increasing functionalities, service quality, safety constraints and complexity of hardware architectures require serious reorganization of the development process as updating tools supports the design and validation of control systems.

ControlBuild is an environment that enables a virtual train to be created (real reference for the project), validation of all customer requirements by simulation, RFQ (specifications consultation) generation for equipment manufacturers, code generation for the train control and monitoring system (generally written TCMS), support of the activities of integration and qualification of all electronic equipment with HiL (*Hardware in the Loop*) tests. Currently, ControlBuild is implemented on a large-scale by train builders, some having imposed this technology on new trains across the world since 2005.

Chapter written by Franck CORBIER.

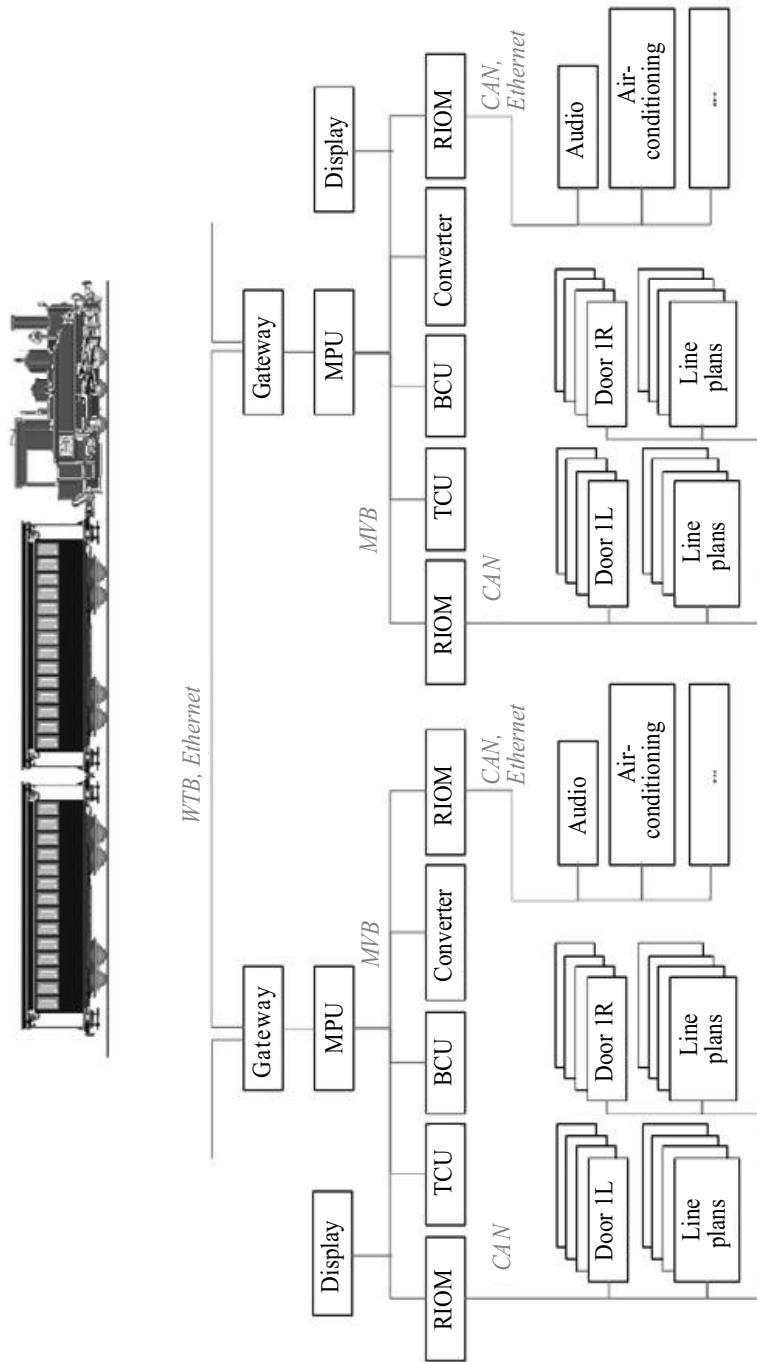


Figure 8.1. Increased use of electronics in control and monitoring systems

Transport operators must on the one hand take up the permanent challenge of moving more and more travelers (needing to reduce the gap between trains while also guaranteeing unfailing safety) and, on the other hand, uphold European standards such as ERTMS¹ (*European Rail Traffic Management System*²) [EUA 06], and ensure the happy coexistence of different system generations and interpretation of standards.

For these two scenarios, integrating new control technologies in a rail infrastructure imposes long and costly qualification testing, with the added constraint of not disturbing commercial traffic. ControlBuild enables transport operators to validate requirements relating to train movement control (autopilot). The objective is to ensure that required specifications are unfailing (nothing slipping through the net) and that suppliers and subcontractors will immediately understand the functional needs and safety constraints expressed.

As well as for trains, ControlBuild and its libraries enable a virtual model of the line to be created, in which trains of different generations travel on rails and switches abiding by (or not - malfunctioning simulation) current signaling and the prototype of future automatic signaling.

The direct path from virtual model to “Hardware in the Loop” testing enables platform-based qualification of delivered equipment, in making it believe that the real system is present and creating authorized or forbidden operating states. Final system qualification on the real site is divided by three and the transport operator can move more rapidly to commercial use.

The following sections present:

- introduction of integration in such a design and validation environment, in the development process of a control system;
- the properties of languages used;
- the plans implemented to ensure the required safety level;
- some demonstrations applied to rolling stock and signaling.

8.2. Development of the control system

Transport system builders need execution methods and design/test tools to deal with the growth in the amount of electric equipment and networks in trains, stronger

1 To find out more, see www.ertms.com.

2 Chapter 11 of this book presents an example of formalization of the ERTMS system.

demands in terms of service quality and more severe certification constraints. The development process must enable them to:

- validate the system requirements specification as early as possible in the project;
- validate the safety requirements relating to equipment and passengers;
- analyze the service quality expected by the customer;
- qualify builders' electronic control units and electro-mechanical equipment by progressive integration;
- manage changes easily (modification of requirements, design choices, hardware architecture, etc.);
- reproduce failures observed in operation;
- cover maintenance activities in operational condition.

The development process must conform to usage rules and rail transport industry standards (see Figure 8.2) such as those defined in the CENELEC³ EN 50126 [CEN 00] standard, EN 50128 [CEN 01], and EN 50129 [CEN 03].

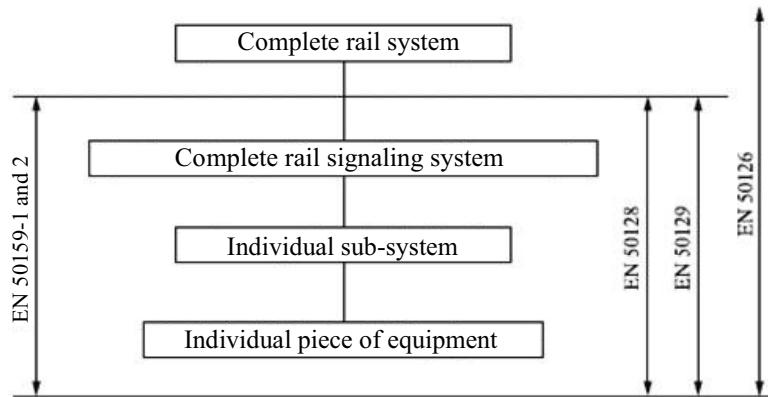


Figure 8.2. Standards applicable to rail systems

8.2.1. ERTMS

Guided by software development referentials (ISO 9001:2008 [ISO 08] and CENELEC EN 50128 [CEN 01]), the development process of embedded computing

³ See: www.cenelec.eu/Cenelec/Homepage.htm.

(or electronics) in a train is based on a cycle (said to be “V” shaped, as Figure 8.3 shows) defining a succession of stages, each producing one or more deliverables. Perceived quality (via documents) sees a net increase but, on the contrary, the final product suffers as deadlines and budgets are not always met.

Although increased documentation contributes to perceived quality, the final product is frequently affected by time and budget overruns.

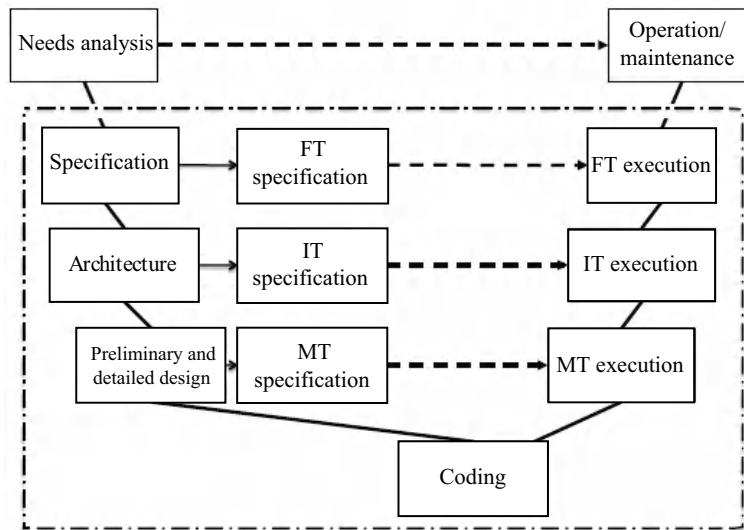


Figure 8.3. *V-cycle including test specifications*

This method of working no longer favors re-use of functions or components already produced in a previous project, which limits possible gains in designing and testing.

Despite this excess of work, serious difficulties appear during the equipment integration phases. Each third party has done its job (design and coding) well, in line with a comprehension (or interpretation) of the input documents.

8.2.2. Development process equipment

Requirements linked to the design and supply of embedded and networked computer systems are increasingly demanding in terms of service and safety quality and necessitate:

- controlling costs despite increasing complexity induced by electronics;
- offsetting, by better use of the technology, questions of organization compelling fewer people to do better and faster;
- dealing with more projects to withstand the growing demands of towns for means of transport.

As in other industries (avionics and automobiles), train builders make important decisions regarding changes, not merely to the current development process, but also the manner in which it is supported and equipped. “Paper”-based approaches are difficult to execute and distance engineers from the final objective.

“Model” and “testing”-based approaches enable communication between different project actors and decisions to be made more rapidly and constructively. The project documents, which are mandatory for certification, are produced predominantly from these models. This enables acceleration of the development and integration cycle and, ultimately, control of the delivery date and quality of the product.

Technologies supported by ControlBuild have been implemented and deployed by many architects, designers and integrators at every stage of the development cycle (see Figure 8.4) of the train and embedded electronic equipment, according to this example of the process:

– creation of a system model:

- functional modeling of system requirements without taking into account constraints, which are induced by the hardware architecture and technological choices (electronics, conventional controls, etc.),
- validation of these requirements by closed loop simulation (in an environment including sensors, actuators and physical laws),
- integration of functions and incremental validation;

– porting to the hardware architecture:

- definition or import of hardware architecture,
- function allocation in the tasks, components and execution targets,
- signal allocation in the input/output interfaces and communication networks;

– automatic generation:

- of code for control and *monitoring* computers (MPU and ECU),
- of schematics for the electrical CAO tools,

- of specifications and interface documents for equipment manufacturers,
- of code for real-time simulation targets,
- automatic development of testing, integration and system qualification benches.

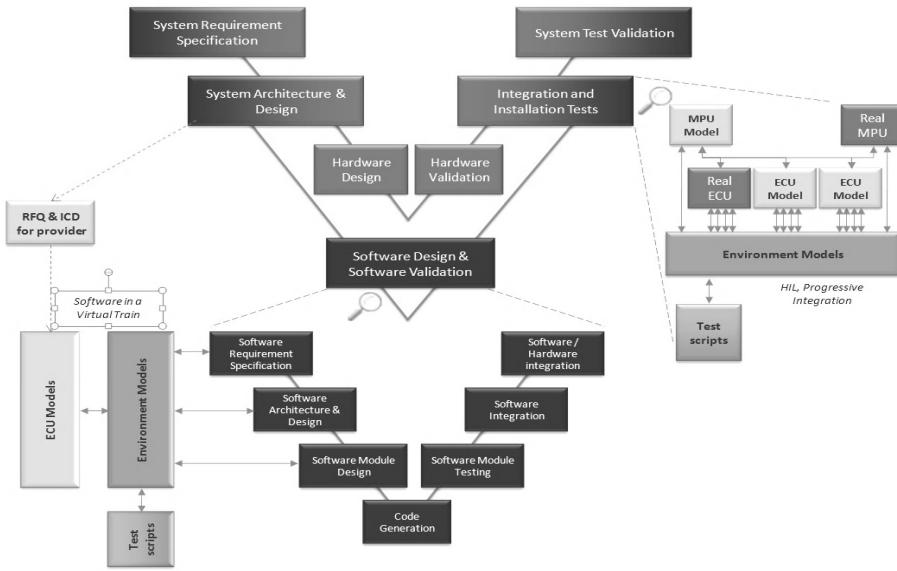


Figure 8.4. Support to the EN50128 development process

8.2.3. A component-based approach

Use of components at all levels of the model architecture enables design and validation micro-cycles between the customer and all the players on the project.

8.2.3.1. Components to do what?

All trains are different (look, number of carriages, door type, power, hardware architecture, etc.).

At a more acute level, it is noticeable that there are many shared functions (the functions “are all the doors closed and locked” or “authorize the opening of doors if speed is less than the threshold”, for example).

Modeling of these “elementary” functions enables know-how to be capitalized on in component libraries, reduction in development costs, and delay in putting the equipment and transport system on the market.

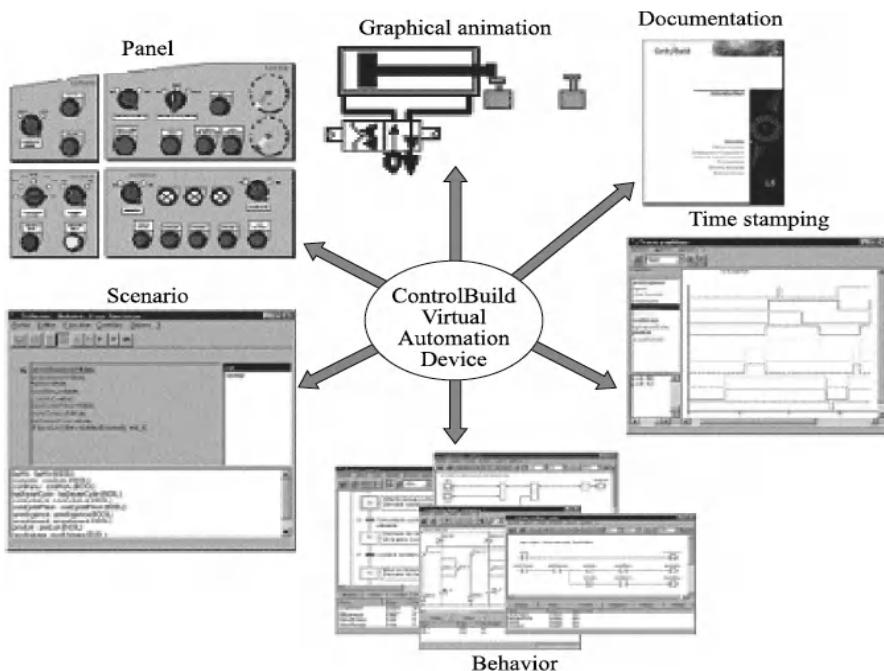


Figure 8.5. Different views of a component

8.2.3.2. What is a component?

The principal quality of a ControlBuild component is to be re-usable in several projects. A component is consequently independent of the hardware platform; it is not allocated to a piece of electronic equipment or to an OS.

A component represents the behavior model of the function “select the coach in service” of the door subsystem or of the entire train (two trains can be instantiated to build the model of a multiple unit).

A component is described as an object with facets (see Figure 8.5) presenting different viewpoints: customer, end-user, designer, quality/safety engineer, supplier, etc.

8.2.4. Development methodology

ControlBuild distinguishes itself from other development tools by its ability to support a large part of the development cycle according to a process integrated from the specifications, right up to the integration and system validation phases.

8.2.4.1. Functional specification

The functional description of a train is achieved in accordance with a hierarchical approach of breakdown of principal functions into sub-functions and so on, up to obtaining components simple enough to design.

8.2.4.1.1. Definition of the main functions of the train

A train control project is constituted of a principal set of main functions (doors, power distribution, air conditioning, lighting, fire detection, traction, braking, driving and operation assistance, etc.).

The main functions are integrated in a train-level component, which enables high-level definition of flows exchanged between these main functions. Each main function is then assigned to a specific expert team.

8.2.4.1.2. Creation of the hierarchical structure of a main function

The design methodology is based on a *Top/Down* approach (see Figure 8.6). The objective of the first level of breakdown is to identify the specific part of the functional subject.

At every level, experts will detail the elementary sub-functions and, for each one, the interfaces with other functions and requirements that they must cover as well as a share of documentation elements (generalities, usage, functional description, anomaly detection, operative modes, availability, safety, etc.).

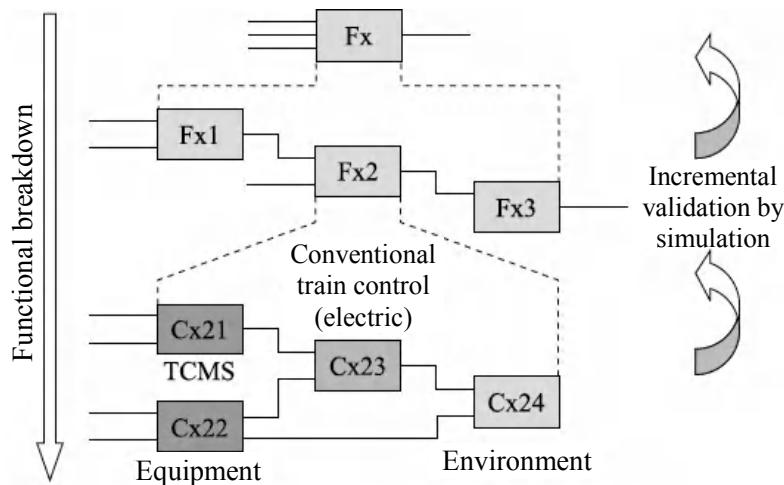


Figure 8.6. Functional breakdown

8.2.4.1.3. Re-use of know-how

At each level of subsystem composition, the team of engineers must identify if the functionality to design already exists or not in the know-how libraries. If so, they must establish a link with the re-usable component so as not to have to re-design it. Re-use at this level of system study offers important benefits as much from the “time saving”, as well as from the “quality” viewpoint.

There are several tools on the market that implement this concept of re-use, but the proposed re-use applies only to the code!

ControlBuild is unique in its ability to enable knowledge management and to re-use all component facets: interfaces (interoperability), model, graphics views, textual descriptions, integration and validation test scenarios, test result logs, etc.

8.2.4.2. Design specification

Each function can be described using standardized formalisms. Whatever its fine details, each function will be associated with an executable model: the model becomes an intrinsic part of the specifications.

At any stage of system composition, simulation of the model enables unambiguous and interactive discussions between the various actors of the development chain, customers included. Dependent on use cases and players (designer or end-user), two animation possibilities are available, based on either language or graphic animation.

The first, titled dynamic execution of the simulation necessitates the use of the language most suited to the objective of functional description sought:

- IEC 61131-3 ([IEC 03] – part 3): languages applying to the modeling of sequences, logic and interlocking;

- *electrical schematic*: when command laws are implemented hardware-style (and for power distribution and safety control);

- *physical models*: to describe sensors and actuators as well as laws of physical behavior.

The second titled “graphic animation” is used to present the dynamic result of execution of a function to a customer or end-user who is not necessarily a technical modeling expert, but who controls the function at the level of their requirements and specifications. In this execution context, it is necessary to make abstract views of all or part of the system available to them using the possibilities of depiction and graphic animation (see Figure 8.7).

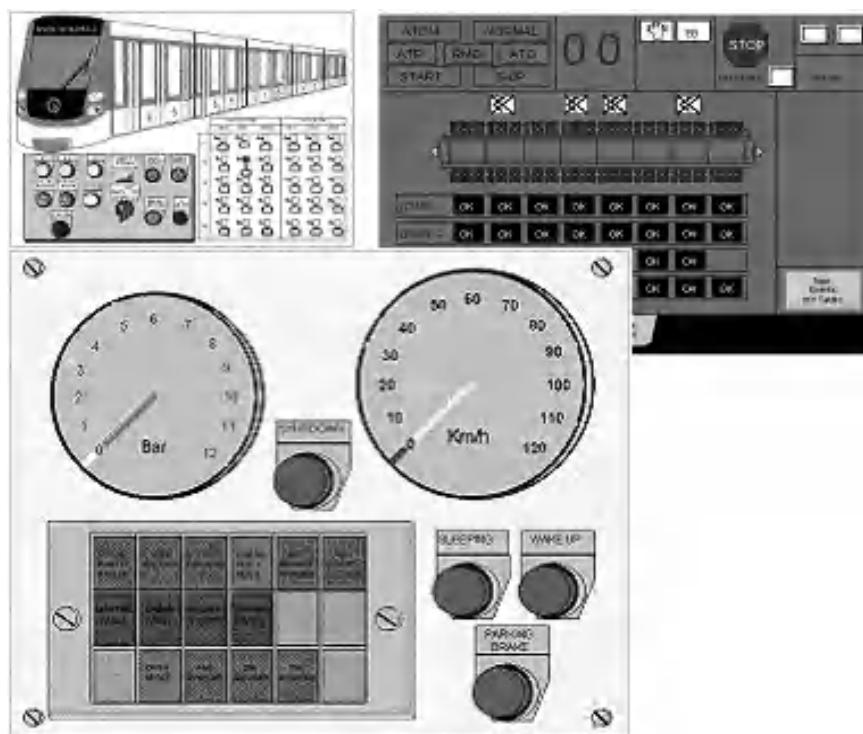


Figure 8.7. Graphic animation views

With this second animation-dedicated view, ControlBuild offers powerful mechanisms enabling conformity of the system to customer requirements to be verified early in the development chain.

Other facets can be associated with components: specification and design elements, qualification manuscripts, formal test results, maintenance notices, integration and regulatory directives, etc.

8.2.4.3. Integration of software in hardware

At this stage of the development cycle, all functionalities and requirements are taken into account and validated by the engineers, customers, end-users and suppliers.

Now, it is time to change the objectives and begin the function allocation of the train model to the control hardware architecture equipment. The process proposed is as follows:

– *description of the hardware*: hardware architecture must in the first place be described in terms of physical execution resources (targets or virtual components) and tasks;

– *allocation of software to hardware*: each function of the virtual train model must be placed in a task of a hardware architecture component, as Figure 8.8 shows;

– *interface configuration*: an assistant available in ControlBuild enables different interface classes to be identified;

– exchanges between two tasks inside a CPU are automatically included and addressed by the ControlBuild code generator;

– physical input/output between the CPUs and physical equipment must be manually addressed in field networks or local and transported E/S cards;

– *code generation*: each real target must then be declared and configured (for example, which operating system, network address, time of cycle of each task, etc.). At this moment, the code of each target (see Figure 8.9) can be generated and linked with the dedicated environment (make, libraries, etc.);

– *production requirements generation*: some targets must be developed by equipment manufacturers or subcontractors. ControlBuild then automatically generates the specification requirements document, real specifications which, besides the functionalities, also define the subsystem interfaces to be supplied.

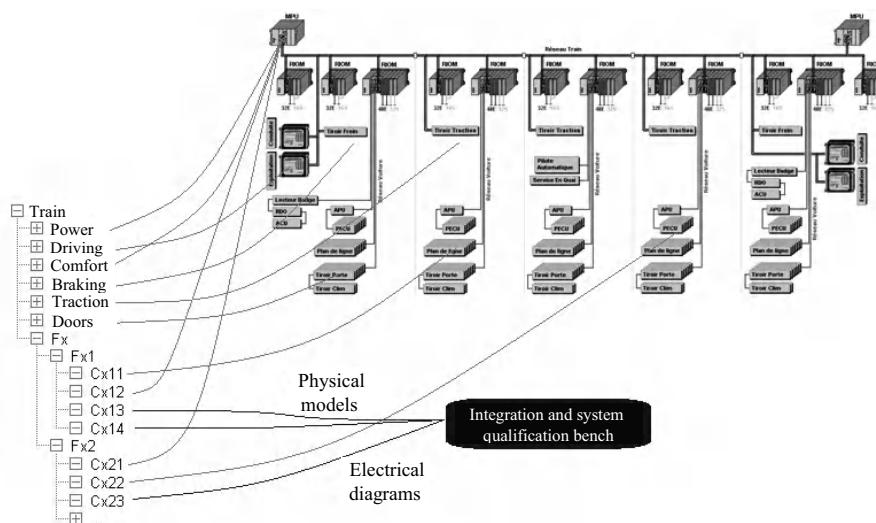


Figure 8.8. Installation of functions in the hardware

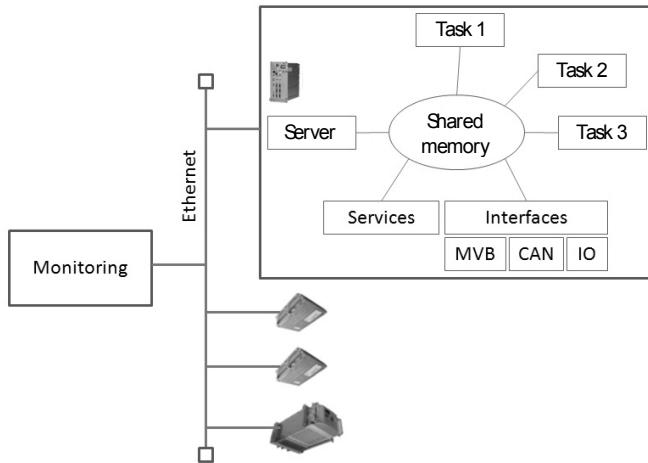


Figure 8.9. Multi-task/multi target code generation

8.2.4.4. Progressive integration

Integration of all train functions is progressive:

- individual integration and validation of each electronic control unit or piece of physical equipment;
- integration and validation of targets and equipment contributing to development of a function of the train;
- integration and validation of all or part of the train ECUs.

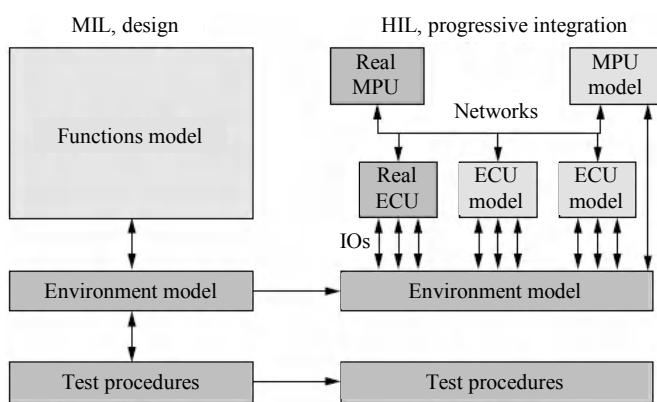


Figure 8.10. Re-use of the environment model and testing procedures

Consequently, the progressive integration platform makes use alternately of physical elements and of their model, contingent upon their availability at a given moment, as Figures 8.10 and 8.11 show.

This approach also provides important advantages throughout the life of the train; that is to say during 30 to 40 years, for maintenance in operational condition: non-regression tests imposed by progressive obsolescence of electronic units, management of specification advances, or change to the requirements.



Figure 8.11. Progressive integration

8.3. Formalisms used

The objective of ControlBuild is to enable modeling and verification of the different aspects of a system (control, physical environment, electrical schematic, man/machine interfaces, etc.) and finally, to validate it all in simulation. The formalisms implemented in ControlBuild must enable description of different viewpoints, taking into account the knowledge and habits of engineers:

- system breakdown;
- control modeling;
- languages IEC6131-3 for embedded electronics;
- electrical schematics for power distribution and conventional train control;
- environment modeling (hydraulic and pneumatic systems, movement/state of equipment, running of trains, etc.).

8.3.1. Assembly editor

ControlBuild only manipulates components. At the lowest level, components are of course described in dedicated languages as we will see later. However, we need to make the language commonplace as fast as possible and only manipulate “boxes” and “arrows”.

ControlBuild thus uses an editor that enables these elementary components to be assembled so that they co-operate together. This editor then creates new components termed *complex* or *composite*, which will be instantiated in their turn in a higher-level component. Indeed, the model of a train also corresponds to a component that can be instantiated with others to form a higher component called *Multiple Unit* and integrated in the model of a railway line.

The assembly editor enables a hierarchical composition tree of the application to be created (see Figure 8.12). It also defines the order in which components are executed.

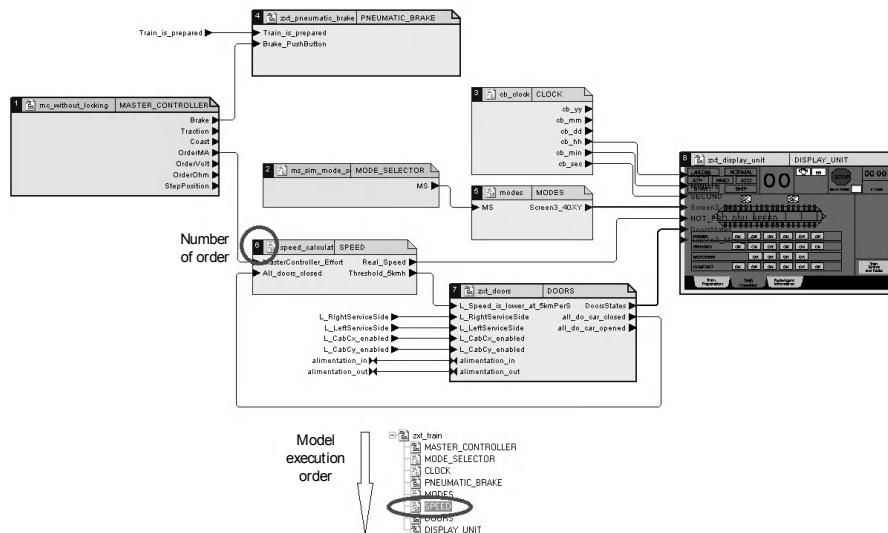


Figure 8.12. Assembly of components

The ControlBuild assembly editor uses an algorithm, which enables the component execution order to be automatically calculated dependent upon precedence relationships established by the connections between components and position in the diagram:

– *connection*: if a component produces data for a second component, it will be executed first, to immediately provide the new value of the data to the following component;

– *position*: if two components are level in the order (to execute at the same time), that which is more to the left will be executed first.

This automatic calculation of the execution order is useful during iterations of model prototyping. It is advisable to fix the scheduling from when a version in configuration is fixed.

Propagation of variable values is achieved in the following way:

- if an output variable producer is executed between variable consumers, all consumers will simultaneously have input of the same value of the variable produced;
- if a consumer is executed before the producer of one of its inputs, it will use the value calculated in the previous cycle.

The assembly editor integrates constraints on connections:

- an output variable produced by a component can be connected to from one to N consumers;
- a variable consumed by one component can only be connected to one producer.

This last constraint guarantees that data is calculated only in one single component. As global variables are forbidden in ControlBuild, it is very easy to show that the model is upright, since its behavior depends only on its interfaces.

8.3.2. IEC1131-3 languages for embedded control [IEC 03]

Part 3 of the IEC61131 standard specifies the syntax (textual or graphic) and semantic of a suite of programming languages for industrial controllers. This suite includes two textual languages, IL (*Instruction List*) and ST (*Structured Text*), and three graphic languages, SFC (*Sequential Function Chart*), LD (*Ladder Diagram*), and FBD (*Function Block Diagram*).

ControlBuild was designed to model functions to validate each requirement through simulation. It was important to suggest some high-level languages in ControlBuild, which allow for observation in simulation (therefore predominantly graphic) and review by verifiers and evaluators (by code re-reading).

Indeed, the IL language (succession of instructions containing an operator and according to the operation type, one or several operands separated by commas) has

not been implemented as a modeling language because it is too low-level (assembler, machine code).

8.3.2.1. Sequential function chart [SFC]

SFC is a graphic language which is derived from the IEC848 standard relating to Grafset (see Figure 8.13).

SFC is a language that describes a sequence of actions to execute. The sequence graph is composed of steps and the move from one step to another is conditioned by a transition.

The actions associated with each step are described using the Structured Text (ST) syntax. The receptivity conditioning a transition is also described using the ST syntax (just the logical/Boolean expression to test; allocation to the transition is automatic).

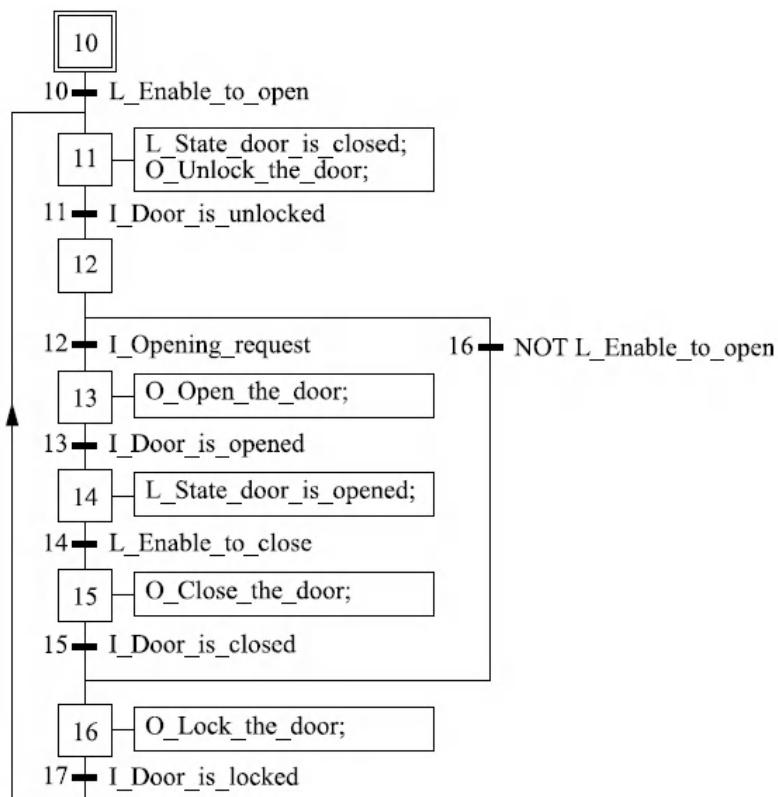


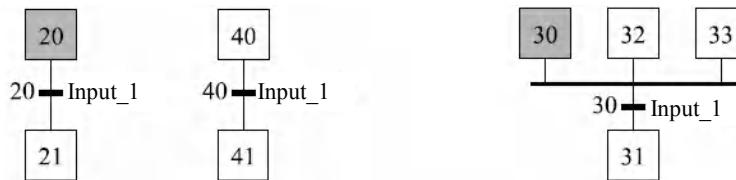
Figure 8.13. SFC language

8.3.2.1.1. Execution

At component startup, the state of the initial step is activated by the system. At each step of the execution, the component reads and fixes its input, identifies transitions, which are surmountable, executes status change to obtain the new active state, processes actions continued in the active step, and finally produces the output (see Figure 8.14).

The notion of surmountable transition is defined by two types of information:

- the fact that the previous step is active in the case of a simple transition or that all the previous steps are active in the case of a convergence in ET;
- the TRUE state of the Boolean expression (receptivity) associated with the transition.



If the variable *Input_1* is TRUE, then:

- transition 20 is surmountable;
- transition 40 is not surmountable because stage 40 is not active;
- transition 30 is not surmountable because stages 32 and 33 are not active.

Figure 8.14. Surmountable state of transitions

Search without stability

Some builders implement *search algorithms* said to be *without stability*. These algorithms enable all successive transitions to be surmounted, which would be TRUE in one single cycle, on the basis of a rule signifying that all simultaneously surmountable transitions are simultaneously passed.

An execution cycle is in fact composed of as many micro cycles as there are surmountable transitions.

Interpretations then diverge contingent upon the execution or not of actions contained in activated steps then deactivated steps in stabilization micro cycles (processing of actions on output only, on local variables, on storage only, on allocations etc.).

The choice made by ControlBuild is to propose a runtime, which corresponds to what a specifier, operator or certifier would understand reading Grafset or SFC, whatever the functioning mode of the target interpretation engine may be.

If some steps must be “jumped” in certain conditions, this will be written in the graph (with more transition links), similarly for the execution (or not) of actions.

This search runtime *without stability* (see Figure 8.15) is the most widespread and, notably, is implemented on the majority of programmable controllers.

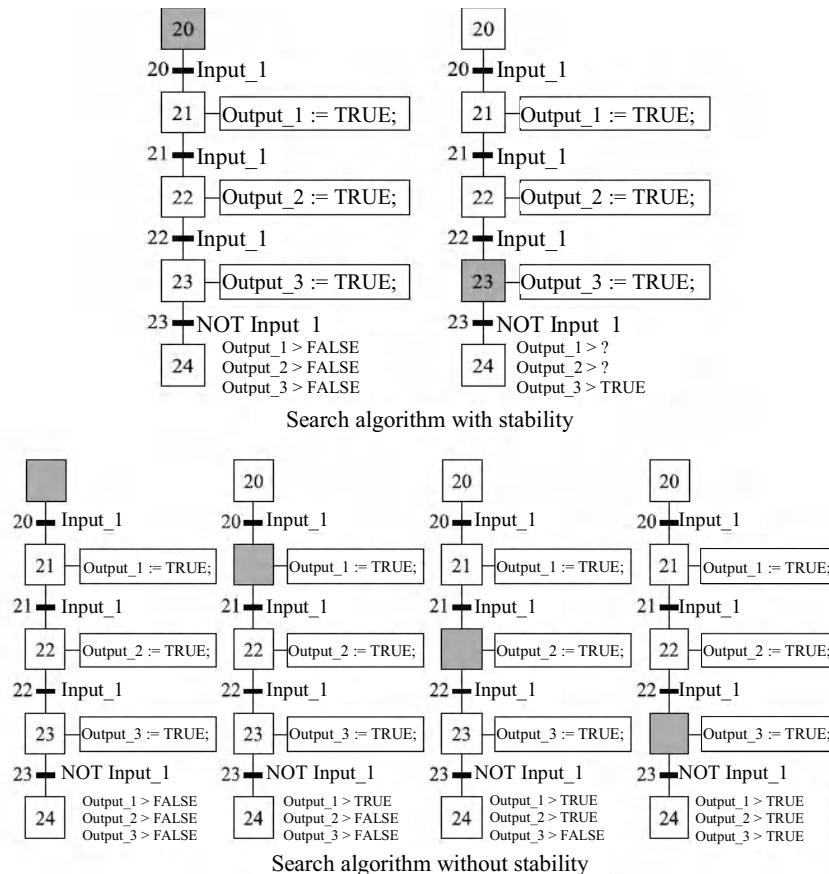


Figure 8.15. Execution algorithm

Interpreted parallelism

Divergence processing in OR should also be taken into consideration.

As for search without stability, the notion of interpreted parallelism (see Figure 8.16) can lead to execution anomalies contingent upon the coding tools used.

To ensure an identical behavior is obtained whatever the runtime, it is essential to force the state to complete in describing exclusive conditions on the branch of divergences in OU (see Figure 8.17).

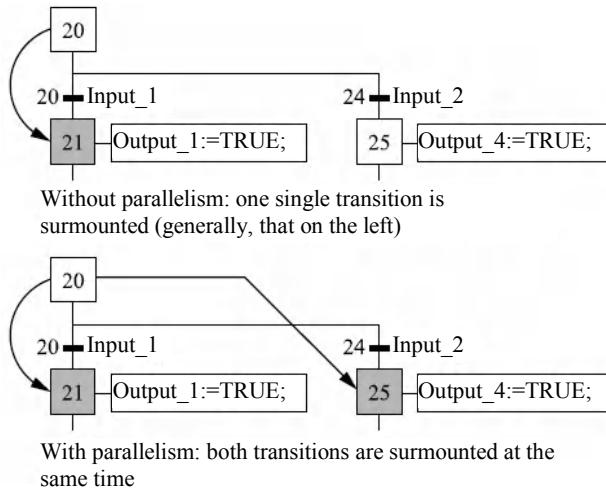


Figure 8.16. With or without parallelism

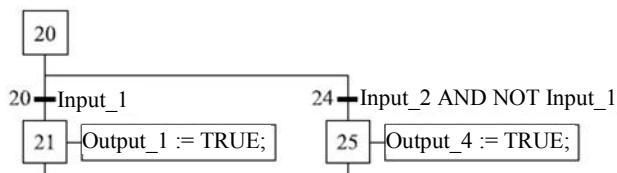


Figure 8.17. Modification of interpretation readability

8.3.2.1.2. Restriction

Other descriptions are to be avoided as they are subject to interpretation. ControlBuild warns the developer of these sensitive descriptions but does not prevent the saving, execution, or re-use of the model.

Actions in the initial step

We advise not describing actions in an initial step. In effect, according to the runtime, these actions will be processed (or not) if the transition immediately downstream is true to the first execution cycle of the model.

Several initial steps

A system or a model cannot have more than one initial step at a time; ControlBuild will deliver this warning when saving the model. However, this does not prevent description of several graphs, each related by an initial step.

8.3.2.2. Function block diagram

The FBD language (see the example of Figure 8.18) enables us to describe logical expressions using graphical representation.

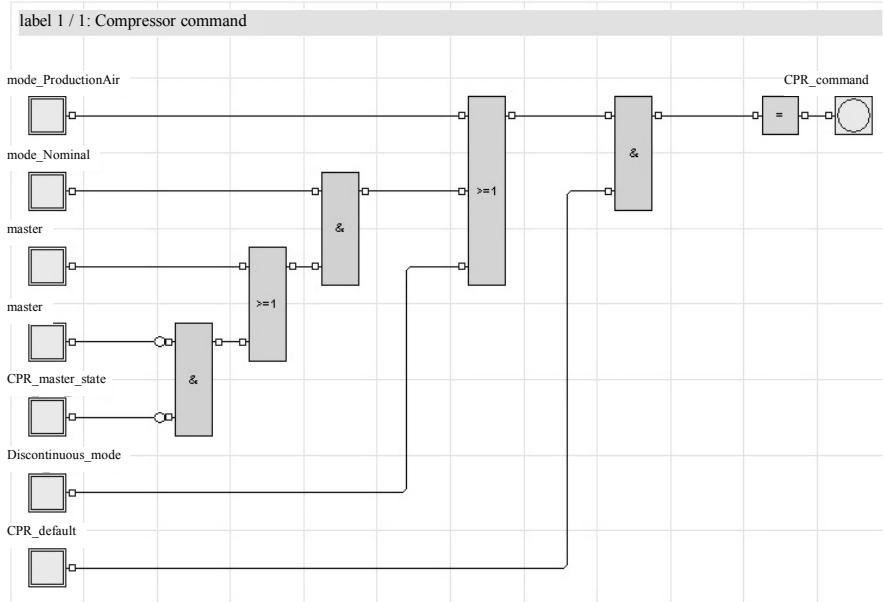


Figure 8.18. Function block diagram/flow-chart

This language is used largely to specify the control functions of a train or automated system. In effect, most of these functions manipulate Boolean variables and execute logical calculations (actuator command, authorizations, functional demands, activation or deactivation condition, interlock and protection, etc.). This specification language is comprehensible to model designers as well as to the end-users; there is no need to follow programming courses to be capable of reading a logic schematic widely called a *flow-chart*. This expression described in a component is analyzed from top to bottom and in increasing description page order (*labels*).

8.3.2.3. Ladder diagram

The *Ladder Diagram* (Figure 8.19) is essentially used by electrical and control engineers in manufacturing industries for control of production and machinery. The execution properties of a ladder diagram are quite similar to a function block diagram.

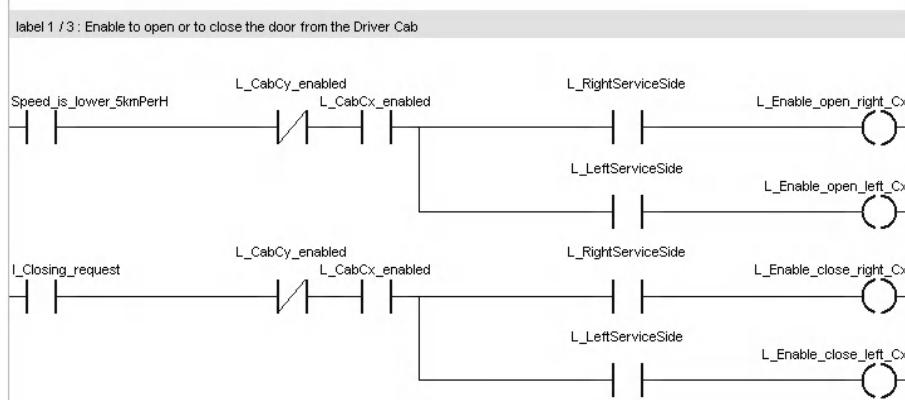


Figure 8.19. Ladder diagram/relay logic schematic

```

1 pressureDrop := 0.0;
2 IF (d_compensatedLeal) THEN
3     pressureDrop := ((leakOfPressure / v_d_compensatedLeal) * (CLOCK_PERIOD / 1000000.0));
4 END_IF;
5 IF (d_noCompensatedLeal) THEN
6     pressureDrop := ((leakOfPressure / v_d_noCompensatedLeal) * (CLOCK_PERIOD / 1000000.0));
7 END_IF;
8
9 differenceCharge := pressureOnLeftSide + pressureOnRightSide + chargeDeCompreseur - pressureDrop;
10
11 pressureOnThePipe := pressureOnThePipe + differenceCharge - brakingPressure;
12
13 (* Limit the pressure : safety valve *) 
14 (* -----
15 IF (pressureOnThePipe > pressionMax) THEN
16     pressureOnThePipe := maxPressure;
17 END_IF;
18
19 (* Limit the pressure : zero pressure *) 
20 (* -----
21 IF (pressureOnThePipe < 0.0) THEN
22     pressureOnThePipe := 0.0;
23 END_IF;
24
25 displayedPressure := pressureOnThePipe;
26

```

Figure 8.20. ST language

8.3.2.4. Structured text

Structured text (or text/literal structured) is based on a set of instructions (see Figure 8.20), which can be executed in having defined some conditions

(IF..THEN..ELSE) or in the form of loops (WHILE..DO). ST is used for implementing models of behavior such as control algorithms, pressure regulation, temperature management, and establishment of braking and acceleration curves.

Even if it is possible to develop complex models, ST is not as powerful as the C and C++ languages.

This is not a problem though; on the contrary, ST syntax is reduced (no pointers, no address, no compressed writing) and its grammar and priority execution rules are perfectly defined.

It is easy to understand what the model does by reading the written instructions in structured text, and it is exactly what the certification authorities expect to validate.

8.3.3. Electrical schematics for conventional control

Train control was historically carried out (and still is currently) in hardware with relay logic (see Figures 8.21 and 8.22).

This technology is difficult to validate onsite by train builders (type tests are complicated) and has a serious impact on the project when corrective modifications have to be made.

In this context, rolling stock builders use ControlBuild to describe the hierarchy of the application and to model basic schematics thanks to its electrical diagram editor.



Figure 8.21. A train

By simulation, train builders have been able to validate the functional behavior of control laws and power distribution, as described in the electrical schematic and their interaction with simplified models of each piece of electronic equipment (door bearing plates, pantographs, traction/braking, etc.) and of the physical environment of the train.

During type testing (tests on real rolling stock), anomalies detected can no longer refer to design errors, but only faults with the physical wiring. The time devoted to tests on the real train is then halved.

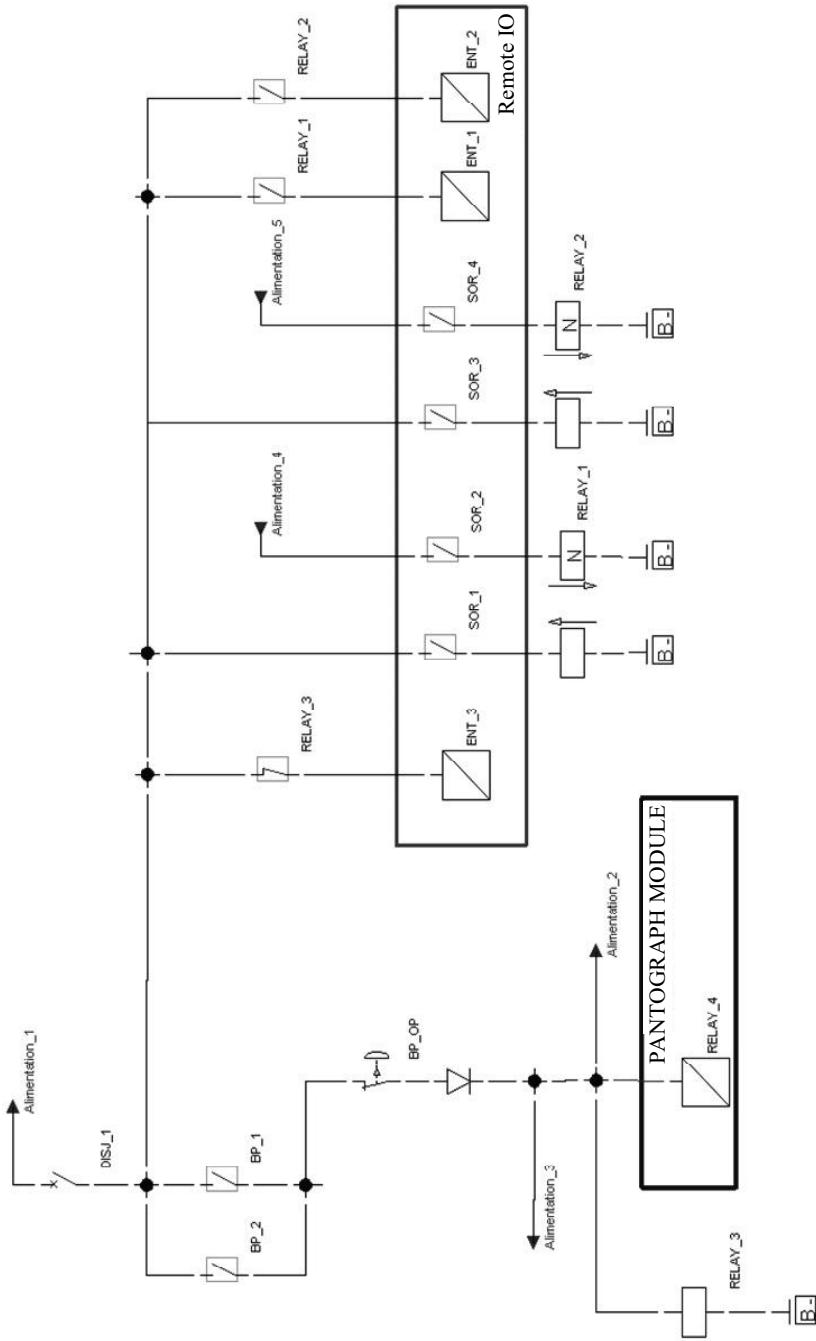


Figure 8.22. Conventional train control in an electrical diagram

8.3.4. Electromechanical and physical environment

The value added by function modeling (and at the end of the complete train) resides in the capacity to simulate closed-loop control (in connection with an environment model).

Depending upon the need for representativeness, the environment model can be *simple* or *complex*:

– simple: the behavior of the model must be compatible with that which is measured by the *control model*. ControlBuild offers libraries, which enable rapid construction of environment models in line with reality;

– complex: the objective from the outset is to size the system by applying *physical laws*. In this case, ControlBuild enables integration of complex models and sharing and instantiation of these models as standard ControlBuild models.

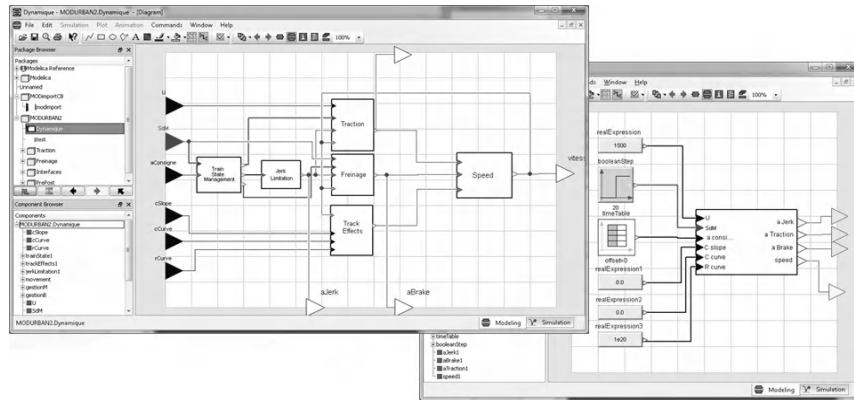


Figure 8.23. Traction/brake model in dymola

External models (see Figure 8.23) can be integrated in ControlBuild. They come predominantly from SIMULINK (using RTW) or DYMOLA (Modéllica language).

With FMI technology, new models (electromechanical, pneumatic, hydraulic, or other physical) can be integrated in the global ControlBuild model of the simulated system. ControlBuild is then an integrated platform for HiL & System Qualification.

8.4. Safety arrangements

ControlBuild enables validation of each software module by open loop simulation (using user interfaces) or closed loop simulation (connection to a reactive environment model). These elementary components can be integrated together to

build new components (component assembly models or composite models), which will also be validated by simulation and instantiated in their turn, right up to forming the complete system.

Ref.	Level	Requirement	Controlbuild compliance
A4		<i>Software design and implementations</i>	
A4.1	R	<i>Formal methods comprising for example CCS, CSP, HOL, LOTOS, temporal logic, VDM, Z and B</i>	
A4.2	HR	<i>Semi-formal method</i>	<i>CB models and tools are based on semi-formal methods (Grafset, Ladder)</i>
A4.3	HR	<i>Structured methods comprising for example JSD, MASCOT, SADT, SDL, SSADM and YOURDON</i>	<i>The software decomposition of CB is close to SADT approach</i>
A4.4	M	<i>Modular approach</i>	<i>The notion of module is central to CB, this notion is supported by components</i>
A4.5	HR	<i>Design and coding standards</i>	<i>A document describes all the coding standards used CB, the rules are applied on the automatic generated code and also on the manual code (libraries)</i>
A4.6	HR	<i>Analyzable programs</i>	<i>Static analyser functionalities are included in CB</i>
A4.7	HR	<i>Strongly typed programming languages</i>	<i>CB uses the type definitions of the IEC 1131 standard and includes type checking functionalities</i>
A4.8	HR	<i>Structured programming</i>	<i>ST is a structured language (like Pascal)</i>
A4.9	HR	<i>Programming language</i>	<i>ST is a general programming language</i>
A4.11	HR	<i>Validated translator</i>	<i>The code generator is validated with an exhaustive test suite</i>
A4.12	HR	<i>Translator well-validated through a large number of use case</i>	<i>More than 1000 test cases are applied during the validation</i>
A4.13	R	<i>Library of secured/verified modules and components</i>	<i>The IEC 1131 library will be validated with an independent test suite</i>
A4.14	HR	<i>Functional/Black-box testing</i>	<i>CB includes mechanisms for describing the environment and run this environment connected to the application</i>

Figure 8.24. Extract from the compliance matrix

This incremental validation approach enables time to be saved on software design and integration testing activities. To cover the requirements relating to safety level 2 (see Figure 8.24) and software quality (the much-mentioned SSIL⁴), ControlBuild offers a number of devices:

⁴ There are five SSIL (Software Safety Integrity Levels) from 0 to 4; see the CENELEC EN 50128 [CEN 01] standard.

- quality metrics;
- assertions;
- test procedures;
- automatic testing;
- code coverage;
- SSIL2 code generation;
- documentation generation;
- traceability of requirements.

8.4.1. Metrics

Metrics (see for example Figure 8.25) are indicators that enable the quality level of a component or application to be obtained. In each of its editors, ControlBuild executes a statistical analysis based on metrics calculations (measurements) and their evaluations with the goal of bringing to light potential non-conformities relative to design and programming standards.

Metric	Value	Warning limit	Error limit
Edited documentations ratio	0 %	—	—
Number of inputs	5	---	---
Number of inputs BOOL	5	---	---
Number of outputs	1	---	---
Number of outputs BOOL	1	---	---
Number of parameters	1	---	---
Number of parameters BOOL	1	---	---
Commented variables ratio	0 %	---	—
Number of numerical inputs without boundary values	0	---	—
Number of numerical outputs without boundary values	0	---	—
Number of unused variables	0	0	0
Number of unproduced variables	0	0	0
Number of labels	1	—	—
Number of components in Ladder or FBD	14	—	—
Maximum number of components by label	14	---	---
Number of connections in Ladder or FBD	13	---	---
FBD Complexity	13	---	---

Figure 8.25. Examples of metrics for an FBD

Some metrics are defined by:

- two thresholds, one of warning and the other of error (depth of the application call graph, complexity of Grafset, number of IF loops, input number, output number, number of instances, etc.);

- a condition (to authorize (or not) unlimited numerical input, unconnected variables, uncalculated variables, etc.).

Metrics can be extracted and integrated in the *project quality* documentation.

8.4.2. Assertions

The developer will design one or several models to satisfy each requirement. ControlBuild also enables the test engineer to describe properties or contracts, which represent what the model must verify:

- INTEGER and REAL type variables must have a *minimum limit* and a *maximum limit*;
- preconditions (input assertion) must protect the model from its environment (that it perceives from its input);
- post-conditions (output assertion) must verify that the model produces output vectors compatible with the requirements to be covered and the safety rules (not giving contradictory orders, for example).

These properties are automatically checked during the execution of a component in simulation or when it is embedded in a target. If one of the properties is violated, the simulation is immediately frozen and ControlBuild presents the error and the offending model to the user.

In embedded mode, an error code is produced on the system for a safety decision (stopping execution, going into degraded mode, generating an alarm, etc.).

8.4.3. Automatic test procedure execution

Different levels of components can be manually tested thanks to consoles and mimics editors offered by ControlBuild as well as those that have been developed by the developer or tester.

The manual test procedure is always long and cumbersome, carrying the risks of manipulation errors.

ControlBuild enables test procedures to be modeled and automatically executed on models or embedded targets. In effect, test procedures (called *scenarios*) are modeled from the design phase, validated on ControlBuild components, and automatically applied on the hardware architecture during integration testing of the equipment.

8.4.4. Functional tests

A specific test module enables a maximum of test vectors to be applied during a determined period (100,000 tests in 10 seconds). The objective is to put the model under stress so as to violate a property to check (limit, pre/postcondition assertion).

In this case, the simulation process is stopped and ControlBuild presents the scenario that has enabled the fault to be brought to light: the tester can then manually replay the test, provide a diagnostic, ask for modification of the model, and verify that the assertion is no longer violated.

NOTE 8.2.– Even if all the tests are executed in the allocated time, we cannot say that there is no risk (probabilistic vector generation). We can say, though, that undesired cases are generally detected from the first test vectors.

8.4.5. Code coverage

Whether tests are carried out manually, by a test engineer, automatically thanks to scenarios, or by test vector generation, ControlBuild provides code coverage indications like:

- percentage of numerical variables, of which the value has been modified during the test and name of the non-covered variables;
- percentage of Boolean variables, of which the state has changed twice (false>true + true>false) and name of the non-covered variables;
- lines of code/branches, which have been executed (rapid visualization of dead code or code not covered by the previous test).

This functionality also enables verification that the defined test scenarios are sufficiently complete to cover the instructions implemented in the model.

8.4.6. SSIL2 code generation

It is extremely important to ensure that the properties verified by the model during the different activities that make up the development cycle are well transcribed in the code, which will be embedded in the targets. Achieved through close collaboration with train builders according to a safety process, ControlBuild integrates an SSIL2 code generator.

This generator is delivered with its SSIL2 qualification kit to aid certification of the system control hardware architecture.

The code to embed in a target is composed of three parts:

- *the application*, which is automatically translated from the models. This code is independent of the hardware platform and operating system. In fact, whether in simulation or execution on a real-time target, this code is generated once only, immediately upon saving the component;
- the IEC 61131-3 ([IEC 03] – part 3) or user libraries, which are dependent on the platform and which must be analyzed and verified with classic code analysis tools;
- the operating system *lower layers* (memory management, access to the IOs, network management, clock management, safety device management: *watchdog*, redundancy, etc.) that must also be analyzed and verified according to an SSIL2 process.

Ultimately, the objective is to eliminate activities relating to demonstration of the SSIL of the generated code and save time in deployment and certification of the software in the targets.

8.4.7. Management of the project documentation

The various stages of development of a train control and driving system according to the CENELEC EN 50128 [CEN 01] standard produce quantity vast number of documents. This is an enormous burden for the designers, who are preoccupied with writing and updating these documents rather than with actual development and testing activities.

With ControlBuild, teams of engineers spend more time on design and a lot less on writing documents. During the analysis, modeling and incremental validation phases, each component is completed by textual information (specification, operative modes, safety of functioning, requirements covered, etc.).

This data enables the content of different documents to be produced:

- software specification (SRS);
- software modules design specification (SMDS);
- software module test procedures (SMTP).

The documentation extractors are configurable to adapt ControlBuild to the quality standards of the provider or end-user content and *look*.

As all the information is integrated in the global model of the project, we have the assurance that all the documents will always be coherent and updated automatically as soon as even the smallest modification is made to the project.

8.4.8. Traceability of requirements

Like all testing and design tools, ControlBuild is part of a global development process in which tracing of the customer requirements is fundamental to ensure that they are covered at every level:

- traceability of design models with the functional requirements of the customer (Office, Adobe, Doors, Requirement Central, etc.);
- traceability of test models with test requirements;
- traceability of test results with the test models and testing campaigns.

ControlBuild gives an interface with Reqifify which enables verification of complete coverage of the project from the customer requirements. Reqifify supports the quality development process of the train maker and provides the traceability documents expected by the certification authority engineers who must qualify the system.

Even more than traceability, a tool like Reqifify enables gains to be made during the life of the system from changing a requirement or if a piece of equipment becomes obsolescent and has to be replaced (see Figure 8.26).

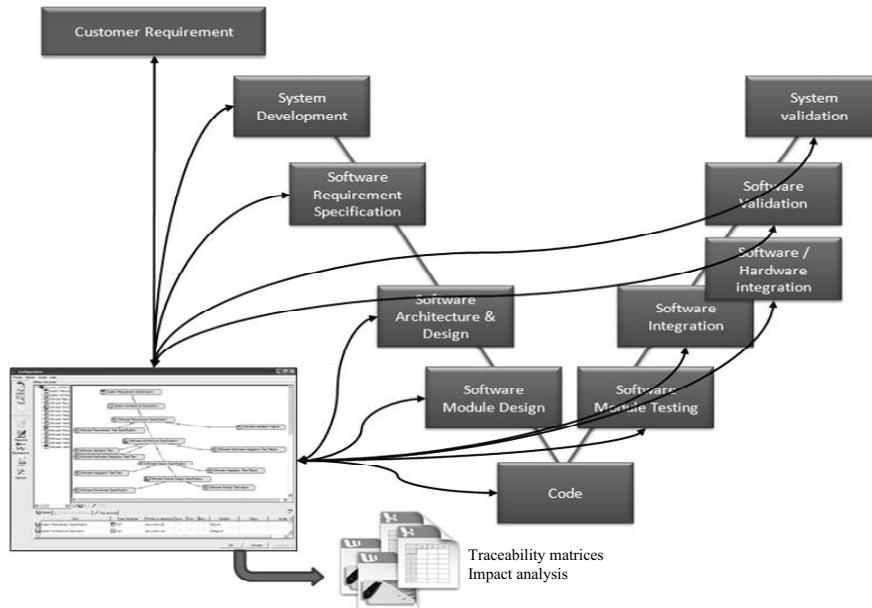


Figure 8.26. Traceability of requirements and impact analysis

It is then possible to produce an impact analysis showing the functionalities to repeat, tests to replay, and documents to update.

8.5. Examples of railway use cases

ControlBuild is used all through the development cycle of a control system. The use cases are nonetheless different, dependent upon the industries or disciplines (train operator, train builder, equipment manufacturer, RAMS engineer, designer of the train, driver training system, etc.):

- validation of system requirements specifications;
- development of the TCMS (*Train Control and Monitoring System*);
- integration and system qualification benches.

8.5.1. Specification validation

ControlBuild is implemented in signaling projects from the specification phase of safety and non-safety functions, which constitute the movement control of trains. Users are either equipment manufacturers who will prototype the requirements and simulate them very early in the scenario in a railway line model, or transport operators (from System and RAMS teams) who want to validate that the functional requirements present no risk of interpretation before making the call for bids.

ControlBuild uses new libraries enabling modeling of a whole railway line:

- track plans (rails, switches, signals, platform, etc.);
- electric signaling (conventional control in *hardware*);
- traction logic (power distribution and catenaries);
- operator command tables (local management of routes);
- central control room.

Automated signaling functions are modeled next, then integrated in the line model and validated in closed loop.

The quality and precision level required for the line model is very high. The validation teams go as far as using testing procedures, which are applied to the real track equipment to qualify the signaling model. These investments are important but will largely pay for themselves during the phase of equipment integration in the control system and during reception tests onsite.

EXAMPLE 8.1.— The RATP modeling system. Updating the RER-A train movement control automation. The virtual system is composed of a model (see Figure 8.27) of all the stations of the Eastern part of the line (from Nation to the Torcy terminus as well as a part of the Nogent Eastern branch). This project corresponds to one of the largest applications created with ControlBuild: 10,000 components, 1,500,000 variables, 30km of simulated lines (tracks, signaling, catenaries), configurable US-MU (simple unit, multiple unit) trains, traction laws, virtual driver, etc.

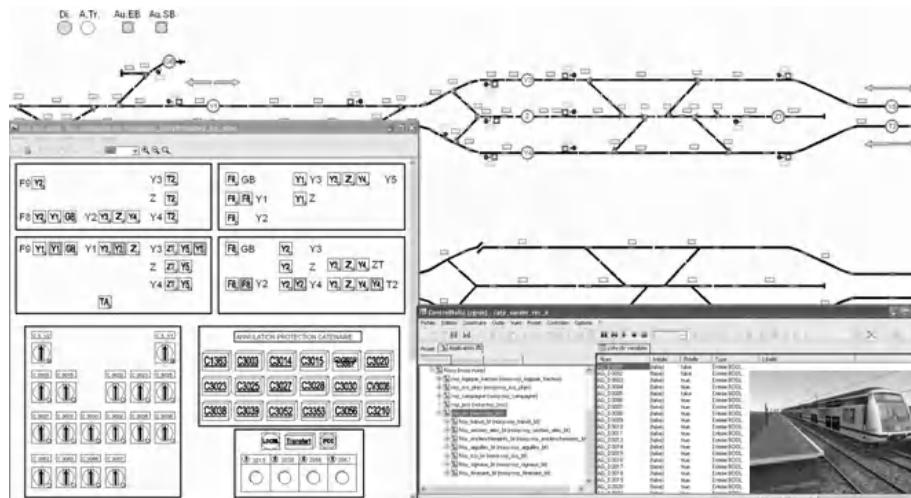


Figure 8.27. Validation of the signaling systems requirements specification

8.5.2. TCMS development

A train is composed of electronic equipment connected in networks and controlled or coordinated by the TCMS (*Train Control and Monitoring System*). This electronic equipment pilots the train sensor-actuators via networked remote input and output (reduction in cost of copper and wiring).

The electrical schematic depicts the functions of power distribution, the train safety lines, and a more or less important part of the control (contingent upon the builders). The complexity of a train system combined with the need to reduce production delays leads builders to adopt techniques enabling validation of their design as early as possible and to save time on the on-site testing stages (in the train). Modeling the functionalities of a complete train (simulation at a system level) with ControlBuild helps the engineering team to improve productivity and increase quality and security according to the CENELEC EN 50128 [CEN 01] standard. The objectives are as follows:

- use of open hardware targets;
- design of a virtual reference (an executable model) before any implementations;
- automation of function allocation on the hardware resources of the distributed architecture;
- automatic generation of the code and communication for each target;
- risk reduction thanks to progressive integration with a HiL (*Hardware in the Loop*) platform;
- automatic generation of different project files (analysis, design, testing, validation, etc.).

Train builders who currently implement ControlBuild at different levels in their projects are Alstom Transport, Siemens Mobility and Bombardier Transportation.

8.5.3. Progressive integration bench – HiL

Engineering based on models has many benefits during the system design phases. ControlBuild implements features, which enable the train design model and all its test cases to be re-used in the integration and system qualification benches.

ControlBuild can then control and monitor simulators as well as configuration of hardware integrated in the bench. In fact, the hardware and software architectures are “automatically” reconfigurable contingent upon the availability (or not) of real equipment connected on the test bench. The term used is “progressive integration”.

8.5.3.1. Embedded systems in the train

Equipment manufacturers supply electrical components (ECUs), which have been validated one by one on dedicated benches. The difficulties of train validation appear after integration of this equipment in the system during “physical” tests of the rolling stock (see Figure 8.27).

This validation is a long and costly stage, which becomes difficult due to the complexity of the architecture and accessibility of the equipment to be tested.

The laying flat of embedded electronic architecture on “the table” (*according to* test engineers in the automobile industry) and its connection to a simulation environment enables many tests to be carried out in the train and engages fewer people (automation of tests, accessibility of the equipment under testing).



Figure 8.28. Example of a test environment

ControlBuild test benches provide:

- signal data capture and simulation (Digital IO, Analog IO, PWM) and network management modules (MVB, CAN);
- simulation models of the train equipment (pneumatic electromechanical, electrical relay, etc.);
- simulation models of computers of the embedded electronic architecture in the train.

Computer simulation models will be activated or deactivated dependent upon the availability (or not) of physical targets. In fact, it is not necessary to connect 40 door controllers on the bench; a single real one is sufficient and the remaining 39 can be simulated by ControlBuild.

EXAMPLE 8.2.– The Alstom Transport TCMS validation bench. Constructed to validate the TCMS (*Train Control and Monitoring System*), these benches are destined to stimulate the MPU (*Main Processor Unit*) interfaces and networks using some models to make them believe that the other electrical equipment and the train sensor-actuators are present. Some guest equipment is also connected and functionally tested together. In addition to the data capture and real-time response capabilities, these benches implement ControlBuild model distribution technologies on architecture with networked simulators, model transformation (automatic building of the test bench software architecture), automatic establishment of inter-target communications and reconfiguration dependent upon present or absent targets.

8.5.3.2. Signaling system

Onsite validation of the automation of a line necessitates about 1,000 nights and involves manufacturers (who have carried out autopilots), signaling and safety specialists, train drivers (with their trains) and operators to create routes. Some transport operators have explored train movement control systems validation and qualification tracks for years, on test benches. However, their objective is not to eliminate onsite tests, but to drastically reduce the length of these tests and thus put new products on the market faster.

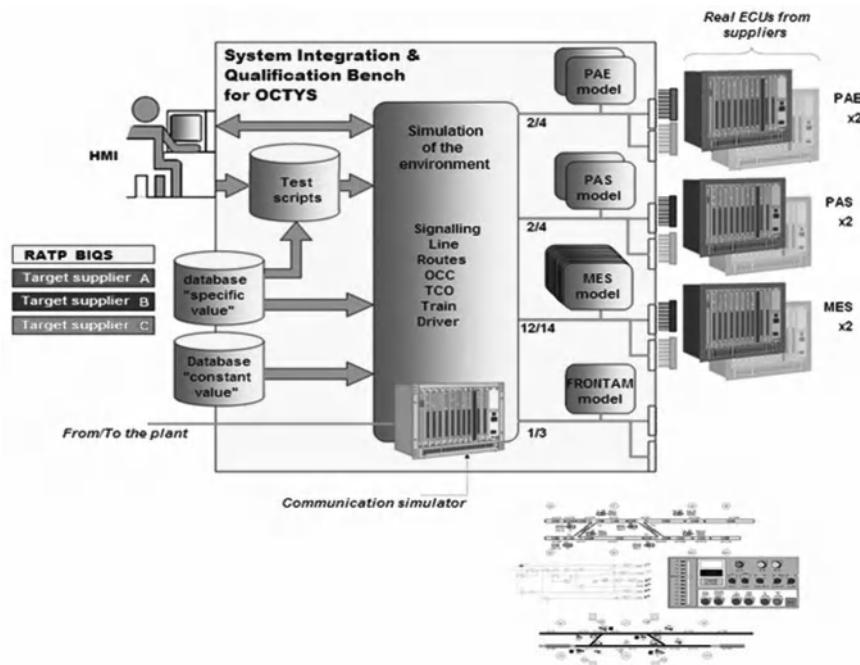


Figure 8.29. Integration and system qualification bench

The RATP developed an *integration and system qualification bench*, which was first used at the test base at the Porte de Charenton (see Figure 8.29). This test line serves to validate integration and interoperability of CBTC⁵ equipment carried out by an association of manufacturers.

⁵ Communication-Based Train Control: a train and metro operating, driving, and safety system. CBTC is a system composed of embedded equipment onboard trains and fixed equipment communicating between them (in general by radio). CBTC was the object of a standard [IEE 04].

The signaling, “physical” line (with rails, switches, platforms, signposts and signals), running of trains, local command posts, a part of the central control room as well as some safety functions, constitute the complete model of the test station.

Tests to carry out on the real equipment have been validated in the complete simulation model before applying them in connection with the real equipment.

Then, the manufacturer’s equipment has been successively connected to the qualification bench and their respective simulation models have been deactivated.

8.6. Conclusion

The quantitative value of the ControlBuild approach can be measured in terms of gains in productivity and the quality improvement strategy of the development process provides several benefits:

- executable specifications enable design choices to be validated more easily in making internal experts (supplier) communicate with their external interlocutors (final customer). The number of iterations is divided by two;
- the possibility of evaluating different hardware architectures for one project;
- use of an SSIL2 qualified code generator enables the effort of certification to be reduced to each new version of the system;
- generation of simulation platform and integration tests in a minimum of time compared to classic techniques. Effectively, the test bench is no longer considered as a separate project but is carried out in continuation of the system design;
- functional validation on an HiL test bench enables all operating scenarios to be reproduced and a system to be delivered in time;
- as the model becomes the reference for the train, ControlBuild facilitates system maintenance and advancement operations all through its life (maintenance in operational condition, obsolescence management, accident reproduction, evaluation of changes).

8.7. Bibliography

[CEN 00] CENELEC, NF EN 50126. Applications ferroviaires – spécification et démonstration de la fiabilité, de la disponibilité, de la maintenabilité et de la sécurité (FMDS), January 2000.

[CEN 01] CENELEC, NF EN 50128. Applications ferroviaires – système de signalisation, de télécommunication et de traitement – Logiciel pour système de commande et de protection ferroviaire, July 2001.

[CEN 03] CENELEC, NF EN 50129. Applications ferroviaires – systèmes de signalisation, de télécommunications et de traitement systèmes électroniques de sécurité pour la signalisation. European standard, 2003.

[EUA 06] EUROPEAN UNION, “The technical specification for interoperability relating to the control-command and signalling subsystem of the trans-European conventional railsystem”, *Official Journal of the European Union*, 2006.

[IEC 03] IEC 61131, Programmable controllers, 2003.

[IEEE 04] IEEE 1474.1, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements, 2004.

[ISO 08] ISO 9001:2008. Systèmes de management de la qualité – Exigence, December 2008.

Chapter 9

Conclusion

9.1. Introduction

The objective of this chapter is to summarize what has been presented in this book.

DEFINITION 9.1.— *Certifiable application.* A certifiable software application is a software application, which has been built in such a way that it can be certified.

The development of a certifiable software application (see definition 9.1) is governed by standards associated with each domain (aeronautics [ARI 92], automotive [ISO 09], railway [CEN 00, CEN 01], nuclear [IEC 06], and generic [IEC 98]).

These standards define the notion of “safety level to be achieved”: (DAL in aeronautics, SIL for the generic standard, SSIL for rail, and ASIL for automotive). The safety level can be linked to an objective to attain (aeronautics) or to a means to implement (automotive, rail, generic).

The applicable standards recommend the implementation of a quality assurance (section 1.4.2), which is based on the implementation of a “V cycle” type development process (see section 1.4.1 and more specifically Figure 9.3), which is based on verification and validation tests¹ (UT, IT, FT).

Chapter written by Jean-Louis BOULANGER.

1 UT for Unit Tests, IT for Integration Tests and FT for Functional Tests.

The implementation of test activities poses several problems such as:

- the cost and difficulties of test activities;
- the late detection of faults;
- the difficulty in carrying out all the tests in the allocated time.

This is why it is necessary to implement other practices (see Chapter 1), which must enable us to detect the faults of software application as early and as widely as possible, at the same time offering similar guarantees for the types of faults detected.

9.2. Problems

Formal techniques (simulation, *model-checking*, abstract interpretation, proof, etc.) are not new. Indeed, the first chapters that cover the subject date from the 1970s (for examples, see [HOA 69, COU 77, DIJ 76]). But the implementation of formal methods dates back to the 1980s [SPI 89, JON 90, HAL 91], with industrial uses starting in the 1990s [BEH 93, ARA 97].

In [BOW 95, ARA 97], we find the first feedback from industrialists concerning formal techniques and, in particular, feedback on the B method [ABR 96, BEH 93, BEH 96, BOU 06]; the language LUSTRE [HAL 91, ARA 97]; SAO+ predecessor² of SCADE [BEN 03, DOR 08]. Other works, like [MON 00, HAD 06], provide an overview of the formal methods in a more scientific approach.

Within the framework of this book, we presented various industrial uses of formal methods. These uses cover various fields (railway, automotive, aeronautics, nuclear power, etc.).

In the first book, *Static Analysis of Software – the abstract interpretation* [BOU 12b] we present some formal tools based on the static analysis and abstract interpretation of code. In this book, several environments and formal methods are presented: the B method in Chapters 2 and 3, SCADE in Chapters 6 and 7; Mathlab/simulink in Chapters 4 and 5; and ControlBuild in Chapter 8.

The goal of this chapter is to synthesize what was presented and to provide an update on the changes taking place. Recall that the objective of this three-book series is to make an inventory of the use of *formal techniques* (simulation, *model-checking* [BAI 08], proof, abstract interpretation, etc.) and *formal methods* (for example, the B method [ABR 96], LUSTRE [HAL 91, BEN 03, HAL 05], SCADE [HAL 05,

² It should be noted that initially SCADE was a development environment based on the language LUSTRE and that since Version 6, SCADE became a language of its own (the code generator for Version 6 works well to provide input for a SCADE model but not a LUSTRE code).

DOR 08], AltaRica, SPARK Ada [BAR 03], etc.) by industrialists. We will therefore not discuss the scientific challenges and possible uses (in the next 5, 10 or 15 years).

9.3. Summary

The various chapters in this book gave us the opportunity to present examples in the industrial environment use formal methods. Even if, to date, there are several environments and a significant number of applications, we should note that there was a change. We passed from a *timid* implementation of formal techniques to an *industrial* implementation as will be explained later.

But this change has not occurred without posing some difficulties. The development of a software application impacting safety is a rather difficult endeavor that is oriented *quality control*³.

Indeed, the only effective technique for creating safety software is to avoid introducing defects and/or to detect the maximum amount of defects. To do this, quality must be controlled. Quality control passes by the definition of process and considerable document production.

Introducing formal methods into the process makes us pass from a *quality control* approach to a *model-centered* approach that can give the impression that documentation becomes useless and that produces the level of safety of the software application depending on the levels of safety of the tools implemented.

9.3.1. Model verification

It should be noted that formal techniques were initially used by industrialists to verify that specification, structure, design principles and/or code respect properties.

As shown in Figure 9.1, the verification process consists of:

- identifying the properties: an analysis of the elements as input (specification, structure, principles of realization, code, etc.) aimed at identifying the properties (see definition 9.2). The properties can be classified into two families: *properties of safety* (see definition 9.3) and *properties of liveness* (see definition 9.4);
- modeling: after choosing a technology (technique + tools), it is then necessary to carry out a model M that can be interpreted by the tools and that will make it possible to implement the selected verification technique (*model-checking*, proof, simulation, etc.);
- analyzing: the third phase consists of carrying out the verification itself.

³ Quality control, in the sense of applying a process that conforms to a standard concerning quality control, for example the ISO 9001:2008 [ISO 08].

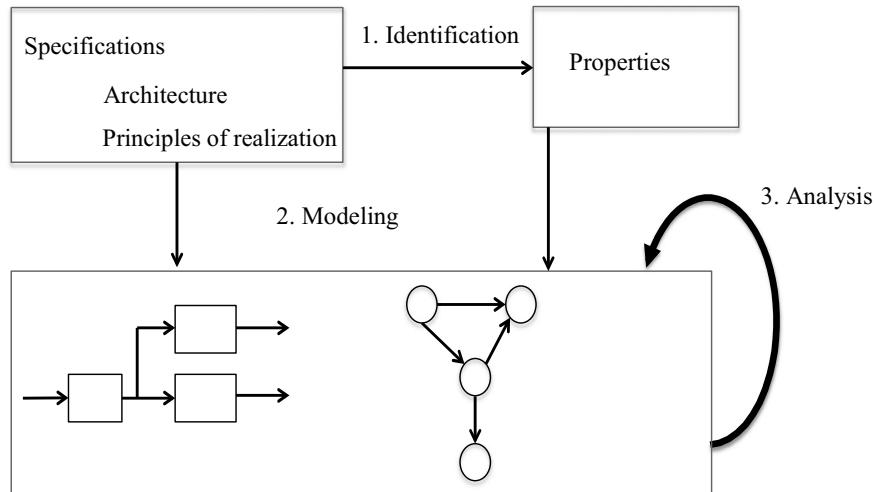


Figure 9.1. Identification, modeling and analysis

For example, Chapter 2 shows how the approach in Figure 9.1 was implemented by RATP⁴ to validate the specifications of the safety functions of SAET-METEOR⁵.

9.3.2. Properties and requirements

The concept of verification is related to the concept of property.

DEFINITION 9.2.– PROPERTY. A property is a characteristic that a (hardware or software) component must verify.

The properties are conventionally separated into two groups: properties of safety (see definition 9.3) and properties of liveness (see definition 9.4);

DEFINITION 9.3.– PROPERTY OF SAFETY. A property of safety asserts that something bad never occurs during execution.

As an example of the property of safety, we can cite the absence of blocking, mutual exclusion, etc. and in our case, the absence of a collision between trains.

4 For more information, see www.ratp.fr.

5 The system [MAT 98] that has equipped Line 14 of the Paris metro since October 1998 is called SAET-METEOR (French: Système d'automatisation de l'exploitation des trains-METro Est Ouest rapide; English: Automation system of train operations).

DEFINITION 9.4.– PROPERTY OF LIVELINESS. A property of *liveness* expresses that something good inevitably occurs during execution.

The termination of an application, the absence of famine (constant progression of applications) and the guarantee of service are many examples of properties of liveliness. It is possible to show that any property P can be expressed as the conjunction of a property of liveliness V and of a property of safety S.

As we presented it, the concept of property is related to identification *a posteriori* of need that the application being analyzed must respect. But within the context of the approaches currently implemented, the properties are identified after the phases of analyses of the schedule of conditions, which is called a *requirement*.

The difficulty then resides in the definition of the concept of a requirement. There are several works that attempt to identify what a requirement is and how to account for it. [HUL 05] presents one of the most complete summaries. We revisit the following definition, which is the result of work led by industrialists.

DEFINITION 9.5.– REQUIREMENT⁶. A requirement is a statement that translates a need and/or constraints (techniques, costs, times, etc.). This statement is written in a language that can be natural, mathematical, etc.

To get a clear identification, a requirement is a labeled element (the label allows a single identification) that characterizes an element of the system to be carried out. For each requirement, it is necessary to know the source.

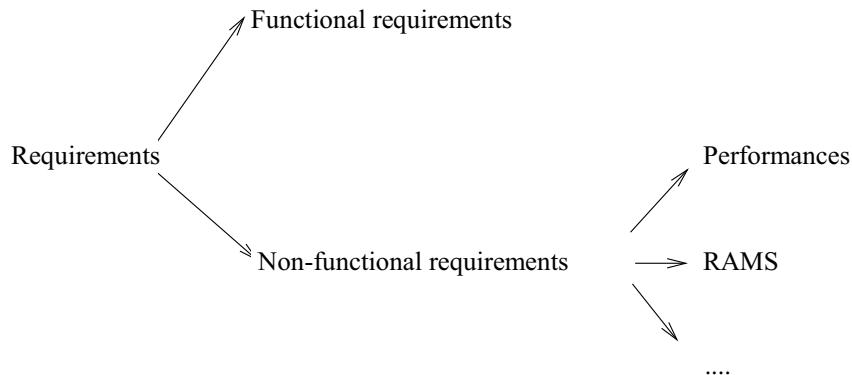


Figure 9.2. Different types of requirements

⁶ AFIS is the French Association of System Engineering. One of its groups of work is specifically dedicated to managing the requirements. For more information: www.afis.fr.

The requirements (definition 9.5) will be analyzed and must be taken into account during each stage of the cycle of the system execution. In the context of the phases of hardware and/or software design, a requirement could be translated in the form of properties (definition 9.2) carried on the components (software or hardware).

Remember (Figure 9.2) that there are two families of requirements, the functional ones that are related to the behavior of the system, and the non-functional ones (also called extra-functional requirements) that are related to quantifiable objectives (performance, level of safety, reliability, availability, etc.).

9.3.3. Implementation of formal methods

Two types of approaches were presented in different chapters:

- the first consists of starting from a specification to create a formal model (see Figure 9.3) and to carry out verifications on the model;
- the second consists of carrying out formal analyses on a code carried out traditionally (in C, Ada, or C++ for example) starting from a specification (see Figure 9.4).

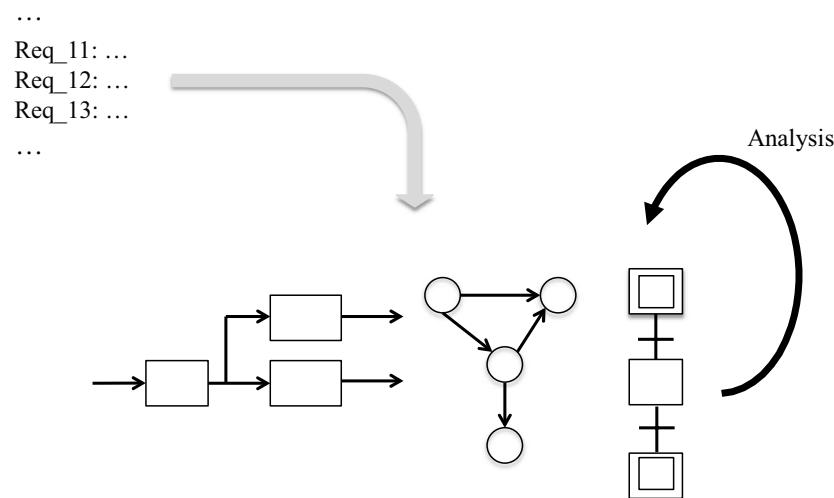


Figure 9.3. Realization of a model

Within the framework of [BOU 12b], we presented several examples of the implementation of a static analyzer of the *abstract interpretation* family (see

[COU 77, COU 00, BOU 11c] Chapter 3). We therefore presented examples of using FramaC, Polyspace, Astrée and CodePeer.

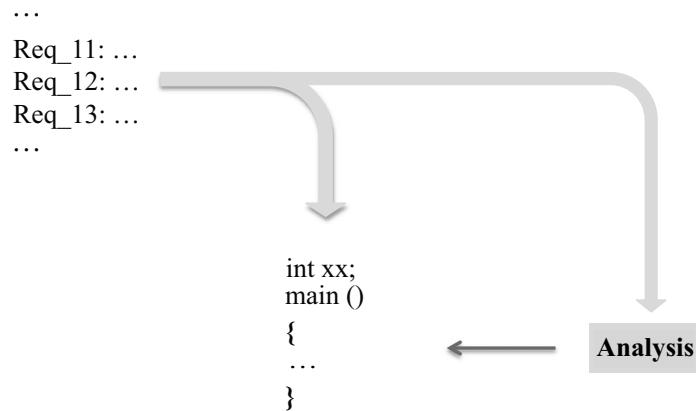


Figure 9.4. Formal analysis of a code

Consideration of formal methods goes through an evolution in the execution process that must account for the modeling phase, as shown in Figure 9.5.

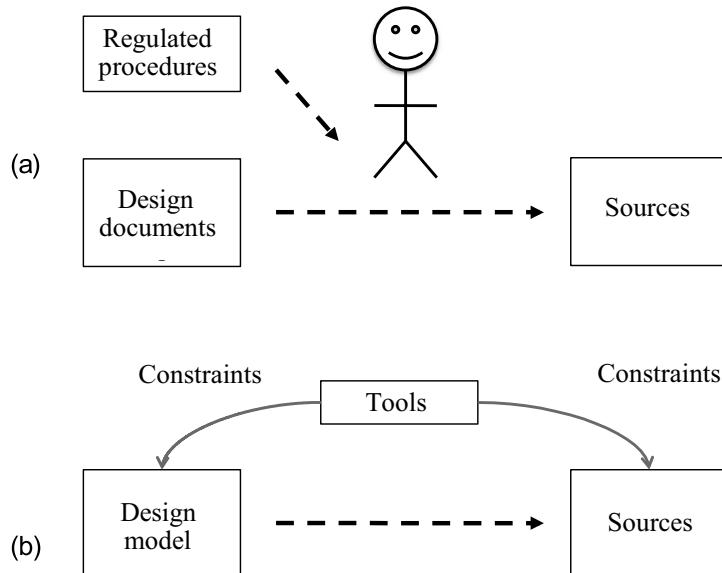


Figure 9.5. Formal analysis of a code

Conventionally (Figure 9.5a), starting from the design documents (specification, structure, design, etc.) and the trade procedures, it is necessary to produce the code. Passing by a modeling phase (Figure 9.5b) makes the concept of a tool and the need for qualifying the tools according to their impact on the software application more important.

9.4. Implementing formal methods

9.4.1. Conventional process

The conception of a software application is broken up into stages (specification, design, coding, tests, etc.). This is called its *lifecycle*.

The lifecycle is necessary to describe the dependences and the sequences between activities. This lifecycle makes it possible to identify the phases, and for each phase we can identify the input, output, activities to be executed, implied people, and the means of the implemented verification.

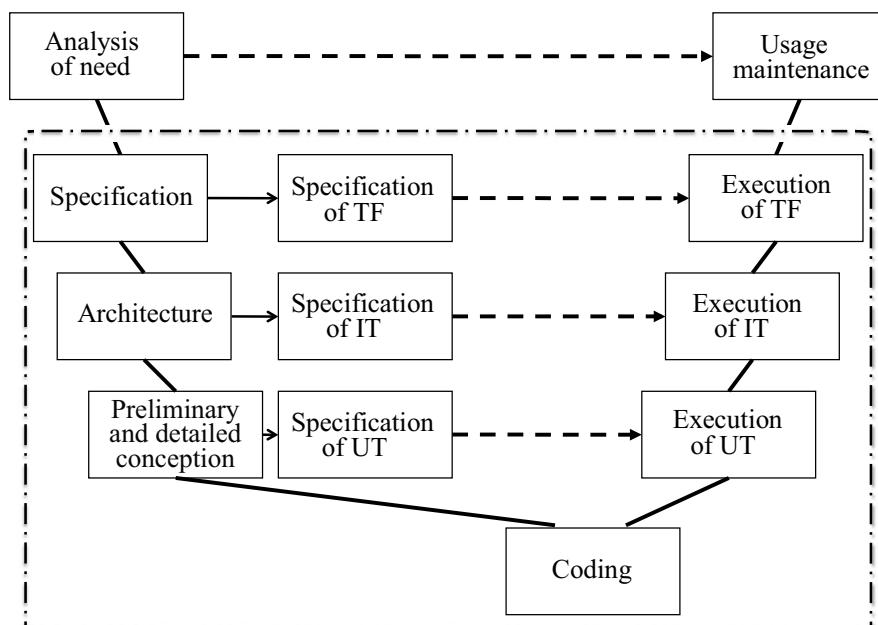


Figure 9.6. V-cycle including the specification of tests⁷

⁷ UT for Unit Testing, IT for Integration Testing, and FT for Functional Testing.

Figure 9.6 shows a conventional V-cycle. The V-cycle is thus broken up into two phases, the descending and the ascending phase⁸. There are various test phases: unitary tests (focused on the lower-level components); integration tests (focused on the software and/or hardware interfaces); functional tests (sometimes called *validation tests*), which seek to show that the product conforms to its specification. The activities of the ascending phase (execution of the UT/IT and TF) are prepared during the descending phase.

The use/maintenance phase relates to the operational life and control of eventual evolutions. Maintaining the software application remains the most delicate activity following an evolution; it is necessary to maintain a level of safety while controlling the cost of the evolution and minimizing the impact on a running system.

9.4.2. Process accounting for formal methods

The realization process of a software application generally passes through various phases requiring the execution of the software application. As shown in Figure 9.7, the concept of execution is essential to show that the software application functions correctly.

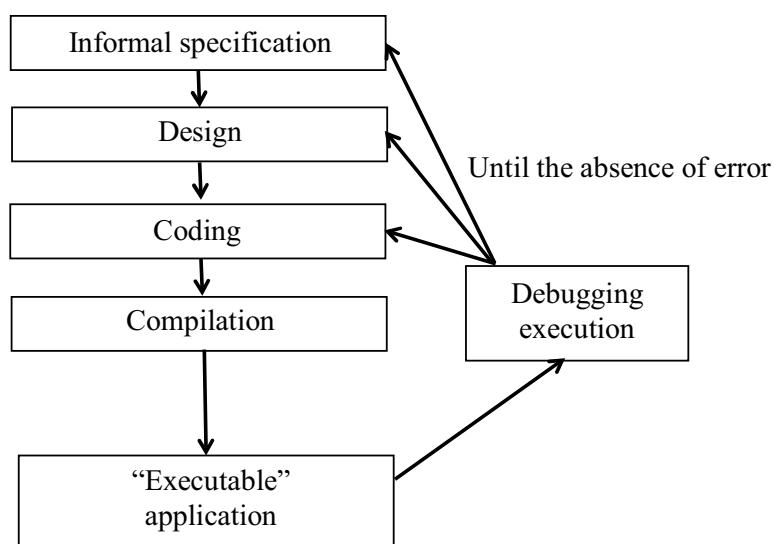


Figure 9.7. Conventional process

⁸ There is a horizontal correspondence (dotted arrow) between the activities of specification and design, and the test activities.

Within the context of a development based on the concept of modeling (Figure 9.8), the process is not based on the execution of the application but on the realization process to show the correct operation of the application. The software application is then considered *correct* following the process that was implemented. The correction is thus intrinsic to the realization of the product.

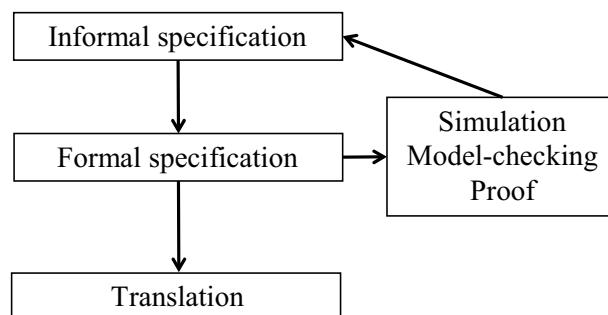


Figure 9.8. Formal process after specification

Figure 9.8 presents an example where the realization of a model is made after the specification and, in this case, the model, is a complement an abstract specification that can contain the requirements. But the model can also be introduced on the design level (Figure 9.9).

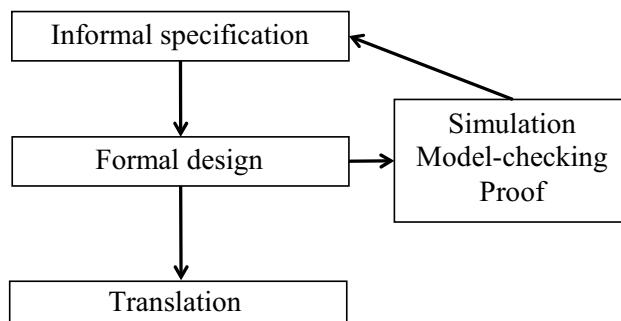


Figure 9.9. Formal process beginning from the design level

In the context of Figure 9.9, the specification is indicated as being informal, but it is possible to have a first model (formal or at least structured), that aims to structure the requirements and to help to identify the inconsistencies and the incompleteness. This model can then, during the design phase, be associated with a formal design model.

In the context of the development of the SAET-METEOR (Chapter 2), we showed how the B method [ABR 96] was implemented to carry out the design of the safety software. The process of the SAET-METEOR is similar to that of Figure 9.9.

In the field of railway applications, the specifications of the software are generally associated with a structured model based on SADT⁹ [LIS 90] and/or SART.

Within the context of the software development of a new CBTC¹⁰, an industrialist in the railway industry is carrying out a software application where the specification is modeled with SART and where the design combines SCADE and the B method.

9.4.3. Problems

The introduction of techniques and formal methods within the realization of the software applications reveals new processes that are not in line with the cycle in V recommended by the standards. The cycle of realization of formal methods moves the verification activities toward the descending phase and concentrates the set of information in the model (see Figure 9.10).

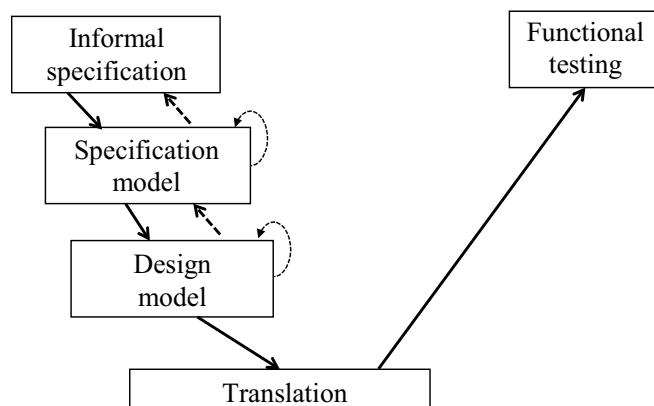


Figure 9.10. Process accounting for the models

⁹ The acronym SADT means *Structured Analysis and Design Technic*. This method was developed by the company Softech in the United States. SADT is a method of analysis by successive levels of descriptive approach of a unit.

¹⁰ *Communication-Based Train Control*: this concerns an operating system, control, and safety of trains and metros. The CBTC is a system made up of embedded equipment on trains and fixed equipment communicating between each other (usually by radio). The CBTC is the subject of a standard [IEE 04].

The concentration of information in the model means that this does not concern an *oriented model* approach, but instead a *model-centered* approach, which are two very different entities. Indeed, in a model-centered approach, the model becomes what supports all the activities, it even becomes what contains the activities (in Figure 9.10, the arrows with dotted lines correspond to verification activities).

The standards of quality control (ISO 9001:2008 [ISO 08], for example) recommend identifying the processes, the activities, and the documents to be produced. The trade standard, like the standard CENELEC EN 50128 [CEN 01], identifies a small list of documents to be produced.

Within the context of a *model-centered* process, UT and IT can be replaced by simulations and/or a combination of proof obligations. But these verification elements are integral parts of the model just like type elements: metrics calculation, verification of the model's coherence, verification of compliance with coding rules, etc.

This type of *model-centered* approach is interesting, but it eliminates many documentary elements, which are then known as *unnecessary* due to the requested producibles. So we then have a *non-conformity* potential for the oriented standards *objective* such as the CEI/IEC 61508 [IEC 98], the ISO 26262 [ISO 09], CENELEC EN 50128 [CEN 01, CEN 11], etc.

The *model-centered* approach has greatly contributed to the convergence of trades as shown in [BOU 06, BOU 99]. Indeed, the various teams (software and hardware, design, V&V and safety) can share information concerning their activities, which makes it possible to better deal with the difficulties.

In general, the team in charge of safety that uses its own functional models and methods, can have a vision somewhat different from the design team; these divergences can have a strong impact on costs and deadlines. Using a common model means that the recommendations and safety requirements can be used as soon as possible, and we can link these elements to parts of the model.

Three difficulties that will appear with the model-centered approach:

- if a tool is obsolescent and its documentation absent, its design cannot be remade;
- the tools will become the central point of the process, meaning that it will then be necessary to show that the tools are usable for the safety level applicable to the software application. This qualification will have to apply as much to the code generating aspect as to the verification tools;
- the complexity of the techniques implemented will require a high level of proficiency from the people in charge of the activities (for example, see the tools for abstract interpretation [BOU 11b, BOU 12b]).

9.5. Realization of a software application

Note the difference between the *creation* of a software application and the *development* of a software application. The realization of a software application contains the development activities, but also the activities of verification, validation, production, installation, and maintenance of the software application (see Figure 9.11).

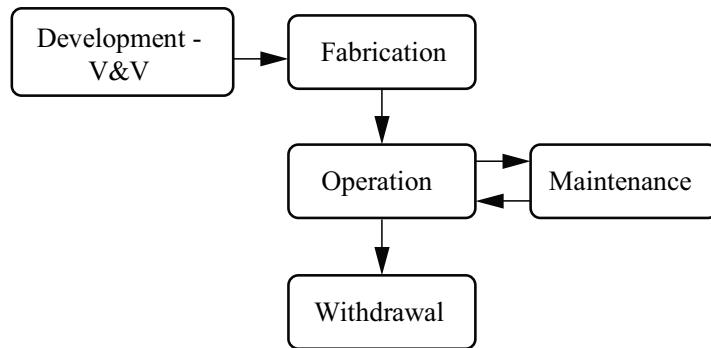


Figure 9.11. Realization of a software application

The verification and validation (V&V) activities are important and will be more or less developed according to the required level of safety.

Activities of the production of the final application and installation are crucial and require the use of specific processes. It is then necessary to formalize (list of all the tools¹¹, description of the stages and their sequences, identification of the options used, etc.) and to describe the process of application generation and the activities to be implemented to install a new version.

The withdrawal of a software application is mentioned, but it does not pose any concern unlike withdrawing a complex system like a nuclear thermal power station or a railway installation.

Maintaining the software application remains the most delicate activity. Indeed, following an evolution, a level of safety must be maintained while controlling the cost of the evolution and minimizing the impact on the system in use.

Maintaining a software application is faced with a difficulty: lifespan. Indeed, for the railway industry, the lifespan is 40 to 50 years, for aeronautics, it is 40 years, for

¹¹ It is sometimes surprising to discover that a tool was developed manually (script, file batch, etc.) to carry out a specific action without there being a minimum documentation.

nuclear power, 50 years, for automobiles, 15 years. For these lifespans, it is necessary to take measures to guarantee the service and software application maintenance. These measurements must take into account the type of machine used for the development, the tools implemented, the documentation to be produced to carry out maintenance and recovery as needed.

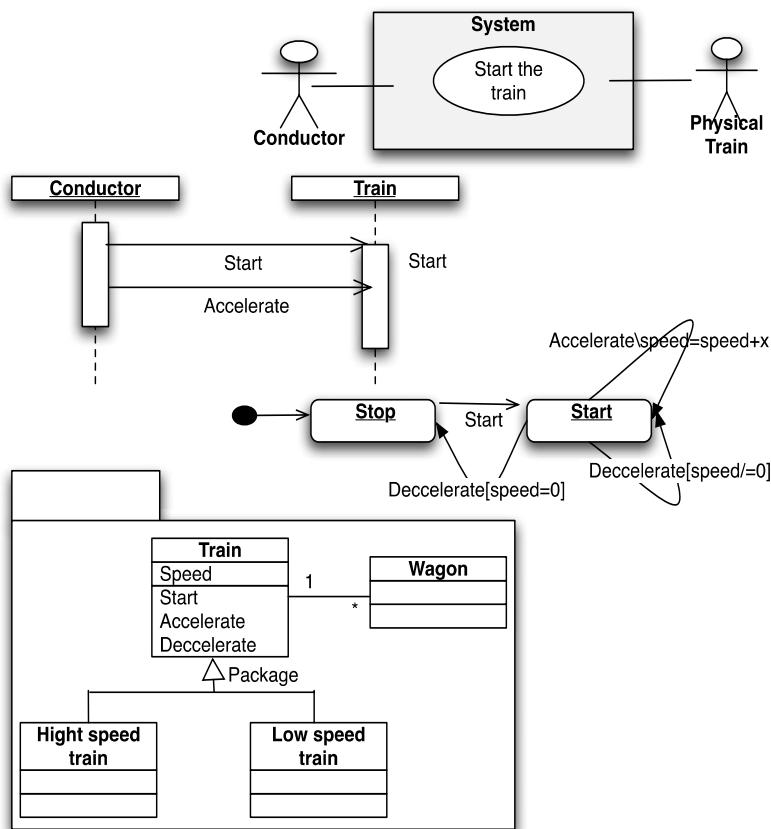


Figure 9.12. Example of an UML model

The SACEM (French: *système d'aide à la conduite, à l'exploitation et à la maintenance*; English: assisted driving, control, and maintenance system) is a system see [BOU 11a] that was brought into use in 1986 and is regularly updated. The choice of a language like Ada [ISO 95] or C [ISO 99] makes it possible to make safe machine, operating system, and tool obsolescence problems. But will there be formal development environments such as Atelier B and/or SCADE? Will it be possible to bring the model of one tool to the other? Will it be possible to redevelop tools?

These questions are even more current with models using the UML notation (see the example in Figure 9.12). Indeed, even if the UML notation¹² [OMG 06a, OMG 06b, OMG 07] is standardized (in its graphics), the associated semantics are not standardized (or at least partially).

Within the context of [BOU 05, BOU 07b, BOU 08, IDA 07b], we proposed some methods of reflection concerning the use of the UML notation to carry out critical software applications.

In recent years, several works have proposed a formalization of UML notation through a transformation toward formal languages, such as [IDA 09, IDA 07a, IDA 06, OKA 07, BOU 07a, BOU 09, RAS 08]; also note work like [MOT 05], which made it possible to propose complementary rules to verify UML models.

9.6. Conclusion

This book is related to several other books by the same authors that cover different aspects of formal techniques: static analysis of code, formal methods, and tools.

The formal techniques and methods are now successfully being used for projects of various sizes in the industry. The associated tools have reached a maturity so that we can account for the complexity of such applications. The complexity of industrial applications very often has an impact on the processing time¹³.

Taking into account the techniques and formal methods has an impact on the implemented process, and it is therefore necessary to build a new referential that conforms to the quality standards being enforced. The construction of this referential must take into account the fact that the model can replace documents that were initially to be produced but that it is necessary to account for in the lifespan of the system and the objectives of associated maintenance. Indeed, if the tool is no longer maintained and formalism is proprietary, it is difficult or even impossible to update the application without the model.

Another difficulty of the *model-centered* approach is bringing the level of safety of the software application to that of the tools implemented.

Though it is difficult to show that a compiler C is SSIL4, it is even more difficult to show that a prover is SSIL4. Within the context of compilers, it was possible to

12 For more, see the OMG site www.omg.org.

13 Within the context of the SAET-METEOR, it took longer than a week in 1998 to analyze 100,000 lines of Ada code with the Polyspace tool (see Chapter 8 for more information), whereas maintaining that would take one or two hours.

set up strategies based on redundancy and diversity. As for specific tools such as provers, *model-checking* tools, and/or tools for abstract interpretation, it is difficult to have two tools of similar effectiveness within the same field¹⁴. Therefore, it is necessary to set up tool qualification files.

The different standards (DO 178, ISO 26262, CEI/IEC 61508, CENELEC EN 50128) show the concept of a *qualification file*. The qualification of a tool depends on its impact on the final product.

Concerning formal techniques and methods, the question of tool qualification is of primary importance because the complexity of the implemented technologies (prover, *model-checking*, etc.), the confidentiality aspect (licensed algorithm, etc.), the innovation aspect (new technology, few users, etc.), and the maturity aspect (product resulting from research, carried out under a “free” license, etc.) do not allow confidence to be built easily.

9.7. Bibliography

- [ABR 96] ABRIAL J.R., *The B-Book - Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.
- [ARA 97] ARAGO, “Applications des méthodes formelles au logiciel”, *Observatoire français des techniques avancées* (OFTA), vol. 20, Masson, Paris, June 1997.
- [ARI 92] ARINC, Software considerations in airborne systems and equipment certification, DO 178B and EUROCAE, no. ED12, edition B, 1992.
- [BAI 08] BAIER C., KATOEN J.-P., *Principles of Model Checking*, The MIT Press, Cambridge, May, 2008.
- [BAR 03] BARNES J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison Wesley, Boston, 2003.
- [BEH 93] BEHM P., “Application d’une méthode formelle aux logiciels sécuritaires ferroviaires”, *Atelier Logiciel Temps Réel, 6^e Journées internationales du Génie Logiciel*, 1993.
- [BEH 96] BEHM P., “Développement formel des logiciels sécuritaires de METEOR”, DANS H. HABRIAS (ed.) *Proceedings of 1st Conference on the B Method, Putting into Practice Methods and Tools for Information System Design*, p. 3-10, IRIN Institut de recherche en informatique, Nantes, November 1996.
- [BEN 03] BENVENISTE A., CASPI P., EDWARDS S.A., HALBWACHS N., LE GUERNIC P., DE SIMONE R., “The synchronous languages 12 years later”, *Proceedings of the IEEE*, vol. 91, no. 1, January 2003.

¹⁴ Recall that for several of the tools currently in use, the algorithms are not public and are even under *copyright*.

- [BOU 99] BOULANGER J.-L., DELEBARRE V., NATKIN S., "METEOR : validation de spécification par modèle formel", *Revue RTS*, no. 63, p. 47-62, April-June 1999.
- [BOU 05] BOULANGER J.-L., BERKANI K., "UML et la certification d'application", *ICSSEA 2005*, CNAM, Paris, 1-2 December 2005.
- [BOU 06] BOULANGER J.-L., Expression et validation des propriétés de sécurité logique et physique pour les systèmes informatiques critiques, thesis, University of Technology Compiègn, 2006.
- [BOU 07a] BOULANGER J.-L., BON P., "BRAIL: d'UML à la méthode B pour modéliser un passage à niveau", *Revue RTS*, no. 95, p. 147-172, April-June 2007.
- [BOU 07b] BOULANGER J.-L., "UML et les applications critiques", *Proceedings of Qualita '07*, p. 739-745, Tanger, Morocco, 2007.
- [BOU 08] BOULANGER J.-L., "RT3-TUCS: how to build a certifiable and safety critical railway application", *17th International Conference on Software Engineering and Data Engineering, SEDE-2008*, p. 182-187, Los Angeles, 30 June-2 July 2008.
- [BOU 09] BOULANGER J.-L., IDANI A., PHILIPPE L., "Linking paradigms in safety critical systems", *revue ICSA*, September 2009.
- [BOU 11a] BOULANGER J.-L. (ed.), *Sécurisation des architectures informatiques industrielles*, Hermès-Lavoisier, Paris, 2011.
- [BOU 11b] BOULANGER J.-L. (ed.), *Utilisation industrielles des techniques formelles – interprétation abstraite*, Hermès-Lavoisier, Paris, 2011.
- [BOU 11c] BOULANGER J.-L. (ed.), *Techniques industrielles de modélisation formelle pour le transport*, Hermès-Lavoisier, Paris, 2011.
- [BOU 12a] BOULANGER J.-L. (ed.), *Mise en œuvre de la méthode B*, Hermès-Lavoisier, Paris, forthcoming 2012.
- [BOU 12b] BOULANGER J.-L. (ed.), *Static Analysis of Software*, ISTE Ltd., London, John Wiley & Sons, New York 2012.
- [BOW 95] BOWEN J.P., HINCHEY M.G., *Applications of Formal Methods*, Prentice Hall, 1995.
- [CEN 00] CENELEC – EN 50126, Applications Ferroviaires, Spécification et démonstration de la fiabilité, de la disponibilité, de la maintenabilité et de la sécurité (FMDS), January 2000.
- [CEN 01] CENELEC – EN 50128, Railway applications, Communications, signaling and processing systems, Software for railway control and protection systems, May 2001.
- [CEN 11] CENELEC – EN 50128, Railway applications, Communications, signalling and processing systems, Software for railway control and protection systems, January 2011.
- [COU 77] COUSOT P., COUSOT R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*, Los Angeles, January 1977, p. 238-252, ACM Press, New York, 1977.

- [COU 00] COUSOT P., “Interprétation abstraite”, *TSI*, vol. 19, no. 1-2-3, www.di.ens.fr/~coussot/COUSOTpapers/TSI00.shtml, 2000.
- [DIJ 76] DIJKSTRA E.W., *A Discipline of Programming*, Prentice Hall, Upper Saddle River, 1976.
- [DOR 08] DORMOY F.-X., “Scade 6, a model based solution for safety critical software development”, *Embedded Real-Time Systems Conference*, 2008.
- [HAD 06] HADDAD S., KORDON F., PETRUCCI L. (eds), *Méthodes formelles pour les systèmes répartis et coopératifs*, Hermès-Lavoisier, Paris, 2006.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., “The synchronous dataflow programming language lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, p.1305-1320, September 1991.
- [HAL 05] HALBWACHS N., “A synchronous language at work: the story of lustre, MEMOCODE ’05”, *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, 2005.
- [HOA 69] HOARE CAR, “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, p. 576-580-583, 1969.
- [HUL 05] HULL E., JACKSON K., DICK J., *Requirements Engineering*, Springer, New York, 2005.
- [IDA 06] IDANI A., B/UML: mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B, PhD thesis, Joseph Fourier University, Grenoble, November 2006.
- [IDA 07a] IDANI A., BOULANGER J.-L., PHILIPPE L., “A generic process and its tool support towards combining UML and B for safety critical systems”, *CAINE 2007*, San Francisco, 7-9 November 2007.
- [IDA 07b] IDANI A., OKASA OSSAMI D-D., BOULANGER J.-L., “Commandments of UML for safety”, *2nd International Conference on Software Engineering Advances IEEE CS Press*, August 2007.
- [IDA 09] IDANI A., BOULANGER J.-L., PHILIPPE L., “Linking paradigms in safety critical systems”, *Revue ICSA*, September 2009.
- [IEC 98] IEC 61508, Sécurité fonctionnelle des systèmes électriques électroniques programmables relatifs à la sécurité, International standard, 1998.
- [IEE 04] IEEE 1474.1, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements, 2004.
- [ISO 95] ISO/IEC Information technology – Programming languages – Ada, ISO/IEC 8652:1995
- [ISO 99] ISO, ISO C standard 1999, Technical report, www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, 1999.

- [ISO 08] ISO 9001:2008, Systèmes de management de la qualité – Exigences, 2008.
- [ISO 09] ISO, ISO/CD-26262, Road vehicles – Functional safety – unpublished, 2009.
- [JON 90] JONES C.B., *Systematic Software Development using VDM*, Prentice Hall International, (2nd edition) 1990.
- [LIS 90] LISSANDRE M., *Maîtriser SADT*, Armand Colin, Paris, 1990.
- [MAT 98] MATRA/RATP, “Naissance d’un Métro. Sur la nouvelle ligne 14, les rames METEOR entrent en scène. PARIS découvre son premier métro automatique”, *La vie du Rail et des transports*, no. 1076, hors-série, October 1998.
- [MON 00] MONIN J.-F., *Introduction aux méthodes formelles*, Hermès, Paris, 2000.
- [MOT 05] MOTET G., “Vérification de cohérence des modèles UML 2.0”, *Première journée thématique Modélisation de Systèmes avec UML, SysML et B-Système*, Association française d’ingénierie système, Toulouse, 2005.
- [OKA 07] OKALAS O., MOTA J.-M., THIRY L., PERRONNE J.-M., BOULANGER J.-L., “A method to model guidelines for developing railway safety-critical systems with UML”, *ICSOFT ’07, International Conference on Software and Data Technologies*, Barcelona, 2007.
- [OMG 06a] OMG, Unified Modeling Language: Superstructure, version 2.1, OMG document ptc/06-01-02, January 2006.
- [OMG 06b] OMG, Unified Modeling Language: Infrastructure, version 2.0, OMG document formal/05-07-05, March 2006.
- [OMG 07] OMG, Unified Modeling Language (uml), version 2.1.1, 2007.
- [RAS 08] RASSE A., BOULANGER J.-L., MARIANO G., THIRY L., PERRONNE J.-M., “Approche orientée modèles pour une conception UML certifiée des systèmes logiciels critiques”, *CIFA Conférence internationale Francophone d’Automatique*, Bucharest, November 2008.
- [SPI 89] SPIVEY J.M., *The Z Notation - A Reference Manual*, Prentice Hall International, 1989.

Glossary

AADL	Architecture Analysis and Design Language
ACATS	Ada Conformity Assessment Test Suite
ACU	Alarm Control Unit
AFIS	French: <i>Association française d'ingénierie système</i> ; English: French Association of Systems Engineering
AMN	Abstract Machine Notation
ANSI	American National Standards Institute
APU	Auxiliary Power Unit
ASA	Automata and Structured Analysis
ASIL	Automotive SIL
ATO	Automatic Train Operation
ATP	Automatic Train Protection
ATS	Automatic Train Supervision
BDD	Binary Decision Diagram
CAN	Controller Area Network
CAS	Computer-Assisted Specification
CBTC	Communication-Based Train Control
CbyC	Correct by Construction
CCP	Centralized Control Point

CdCF	French: <i>cahier des charges fonctionnel</i> ; English: functional requirements specification
CENELEC ¹	European Committee for Electrotechnical Standardization
CMMI	Capability Maturity Model Integration
CPU	Central Processor Unit
DAL	Design Assurance level
DoD	Department of Defense
DV	Design Verifier
E/E/PE	Electric/electronic/programmable electronic
EAL	Evaluation assurance level
ECU	Electronic Control Units
ERTMS	European Rail Traffic Management System
FADEC	Full Authority Digital Engine Control
FBD	Function Block Diagram
FC	Failure condition
FDA	Food and Drug Administration
FMEA	Failure Mode and Effects Criticality Analysis
FT	Functional Testing
GPS	GNAT Programming Studio
GSL	Generalized Substitution Language
GUI	Graphical User Interface
HIL	Hardware In the Loop
HR	Highly Recommended
IDE	Integrated Development Environment
IEC ²	International Electrotechnical Commission
iFACTS	interim Future Area Controls Tools Support

1 See www.cenelec.eu.

2 See www.iec.ch.

IL	Instruction List
IMAG	French: <i>Institut de mathématiques appliquées de Grenoble</i> ; English: Institute of Applied Mathematics of Grenoble
IPSN	French: <i>Institut de protection et de sûreté nucléaire</i> ; English: Institute of Nuclear Protection and Safety
IRSN	French: <i>Institut de radioprotection et de sûreté nucléaire</i> ; English: Radioprotection and Nuclear Safety Institute
ISO ³	International Organization for Standardization
ISSE	International Symposium on Software Reliability Engineering
IT	Integration Testing
KCG	Qualifiable Code Generator
KLOC	1000 LOC
LaBRI	French: <i>Laboratoire bordelais de recherche en informatique</i> ; English: Bordeaux Laboratory for Computer Research
LD	Ladder Diagram
LOC	Lines of Code
MaTeLo	Markov Test Logic
MBD	Model-Based Design
MBT	Model-Based Testing
METEOR	<i>METro Est Ouest Rapide</i> , train operation system used by the Paris metro
MISRA ⁴	Motor Industry Software Reliability Association
MMI	Man-Machine Interface
MPU	Main Processor Unit
MTC	Model Test Coverage
MTTF	Mean Time To Failure
MU	Multiple Unit

3 See www.iso.org/iso/home.htm.

4 See www.misra.org.uk.

NHMO	NATO HAWK Management Office
NR	Not Recommended
NSA	US National Security Agency
NSE	No Safety Effect
OCR	Optical Character Recognition
OFP	Operational Flight Plan
OMG ⁵	Object Management Group
OOTIA ⁶	Object-Oriented Technology in Aviation
OPRI	French: <i>Office de protection contre les rayonnements ionisants</i> ; English: Office of Ionizing Radiation Protection
OS	Operating System
PADS	French: <i>pilot automatique double sens</i> ; English: two-way autopilot
PAI-NG	French: <i>poste d'aiguillage informatisé de nouvelle génération</i> ; English: next-generation computerized signaling control
PO	Proof obligation
PWM	Pulse With Modulation
QTP	Quick Test Professional
R	Recommended
R&D	Research and Development
RAMS	Reliability, Availability, Maintainability and Safety
RAT	Ram Air Turbine
RATP ⁷	French: <i>Régie autonome des transports parisiens</i> ; English: Autonomous Operator of Parisian Transports
RER	French: <i>Réseau express régional</i> ; English: Regional Express Network

5 See www.omg.org.

6 See www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot.

7 See www.ratp.fr.

RFT	Rational Functional Tester
RM	Requirement Management
ROM	Read-Only Memory
SACEM	French: <i>Système d'aide à la conduite, à l'exploitation et à la maintenance</i> ; English: Assisted driving, control, and maintenance system
SADT	Structured Analysis and Design Technique
SAET	French: <i>Système d'automatisation de l'exploitation des trains</i> ; English: Automation system of train operations
SAO	French: <i>Spécification assistée par ordinateur</i> ; English: computer-aided design
SAS	Software Architecture & design
SCADE	Safety-Critical Application Development Environment
SFC	Sequential Function Chart
SHOLIS	Ship/Helicopter Operational Limits Instrumentation System
SIL	Safety Integrity Level
SIS	Safety Instrumented System
SMDS	Software Module Design Specification
SMTP	Software Module Test Plan
SOC	System On a Chip
SPIN	French: <i>système de protection intégré numérique</i> ; English: digital integrated protection system
SRS	Software Requirement Specification
SSIL	Software SIL
ST	Structured Text
SU	Single Unit
SUT	Software Unit Test
TCMS	Train Control Management System
TCO	French: <i>tableau de contrôle optique</i> ; English: visual control panel

350 Formal Methods

TFTA	Terrain Following Terrain
TIS	Tokeneer ID station
TOR	French: <i>Tout ou Rien</i> ; English: hit-or-miss
UML	Unified Modeling Language
UT	Unit testing
V&V	Verification and Validation
WCET	Worst-Case Execution Time
WP	Weakest Precondition

List of Authors

Benjamin BIANC
CEA LIST
France

Jean-Louis BOULANGER
CERTIFER
Anzin
France

Jean-Louis CAMUS
Esterel Technologies
Toulouse
France

Mirko CONRAD
The MathWorks GmbH
Munich
Germany

Franck CORBIER
Dassault Systèmes
Nancy
France

Véronique DELEBARRE
SafeRiver
Paris
France

Jean-Frédéric ETIENNE
SafeRiver
Paris
France

Christophe JUNKE
CEA LIST
France

Bruno MARRE
CEA LIST
France

Pieter J. MOSTERMAN
The MathWorks, Inc.
Natick MA
USA

Patricia MOUY
CEA LIST
France

Index

A

abstract, 24, 68, 71, 82,
87, 99, 128-133, 135-142,
146-149, 161, 185, 191,
273, 277, 296, 326, 334,
336, 340, 345
Airbus, 230, 231, 240, 257, 260
algorithm, 95, 98, 102, 112, 135,
164, 167, 191, 228, 277, 278,
301, 304, 305, 309, 340
ansaldo STS, 262-264
assertion, 76, 184, 187, 188, 199,
204, 313-315
ATO, 261, 262, 345
ATP, 58-61, 70, 75, 78, 91, 105, 114,
201, 261, 262, 345
ATS, 262, 345
availability, 69, 218, 295, 300,
320, 321, 330, 348

B

B method, 64, 67-70, 74, 78,
95, 112, 114, 127-129, 133,
135-137, 141, 148-155, 191,
326, 335

C

CBTC, 151, 194, 261, 262, 322, 335,
345
CENELEC, 115, 116, 142, 152-154,
219-221, 290, 312, 316, 319,
336, 340, 346
certification, 177, 227, 256, 274,
284, 290, 292, 309, 315-317,
323
certified, 227, 250, 255, 264, 325
code, 63, 65-68, 74, 76-78, 85, 95,
100-104, 109, 128-130, 142,
146, 149-151, 154, 155, 159-
166, 169-178, 184-188, 192,
193, 208, 228, 232, 236, 240,
243, 245, 247, 249, 250, 252,
253, 255-257, 261, 262, 264-
267, 287, 292, 293, 296, 298,
299, 302, 303, 313-316, 320,
323, 326, 327, 330-332, 336,
339, 347
combination, 99, 102, 115, 117, 159,
173, 175, 176, 197, 205, 270,
336
compilation, 130, 148, 174,
176, 235

compiler, 66, 74, 107, 109, 150, 154, 155, 253, 339
 component, 64, 65, 68, 78, 79, 82, 128, 133, 137, 141, 144, 145, 149-160, 175, 176, 187, 221, 226, 232, 240, 277, 283, 291-298, 301, 302, 304, 307, 311-314, 316, 319, 320, 328
 confidentiality, 340
 conformity, 137, 149, 187, 191, 220, 221, 297, 345
 critical, 62, 67, 69, 79, 97, 114, 127, 128, 213, 219, 220, 225, 226, 228, 230, 239, 240, 242, 255, 256, 261, 270, 339, 349
 cycle, 59, 62, 67, 72, 73, 75, 84-86, 93, 96, 98, 105, 106, 135, 136, 142, 148, 149, 166, 172, 173, 186, 193, 199, 202, 206-209, 217, 218, 221, 232, 236-239, 242, 243, 247, 249, 274-281, 291-294, 297, 298, 302, 304, 306, 315, 318, 325, 330, 332-335
 cycle in V, 149, 291, 325, 333, 335

E

error, 62-65, 67, 74, 77, 100, 102, 108, 109, 111, 118, 119, 143, 146, 166, 167, 184, 187, 188, 197, 199, 201, 203-206, 209, 210, 213, 218, 227-229, 236, 238, 240, 243, 247, 249, 250, 252, 254, 255, 265, 274, 309, 313-315, 320, 321
 ERTMS, 289, 290, 292, 346
 evaluation, 92, 96, 98, 102, 242, 246, 284, 284, 313, 319, 323, 331, 346
 event, 90, 97, 104, 105, 109, 112, 117, 118, 127, 160, 161, 187,

188, 190, 201, 203, 204, 206, 211, 216, 231, 233
 exploration, 104, 127, 185, 199, 224
 by model, 218, 224

F

fault, 58, 87, 201, 204, 211, 216, 226, 238, 240, 250, 255, 309, 315, 321, 326
 formal, 55, 63-65, 67-70, 72, 74, 78, 79, 80, 82-84, 87, 93, 107, 112-116, 121, 124-128, 135-139, 140-146, 149, 150, 152, 154, 155, 160-162, 174-178, 183, 191-198, 204, 208, 218-220, 223-226, 230-235, 239, 242, 243, 245-247, 251, 252, 257, 258, 263, 269, 271, 273, 278, 297, 300, 326-328, 330-335, 337-342
 formalization, 80, 84, 186, 187, 189, 191, 200-206, 208, 210, 212, 214, 218, 223, 289, 290, 339, 341

G

graph, 92, 97-100, 103, 105-107, 147, 148, 150, 151, 160, 184-186, 196, 199, 205-213, 217, 218, 222, 223, 303, 305, 307, 309, 312, 313, 315, 320

I

IEC, 177, 180, 181, 227, 296, 299, 300, 302-304, 306, 307, 322, 325, 336, 338, 340, 342, 346, 347
 interpretation, 61, 62, 161, 162, 174, 177, 185, 273, 274, 289, 290, 291, 293, 304, 306, 309, 311,

318, 326, 327, 336, 339, 340,
342

abstract interpretation, 161, 162
185, 273

ISO, 67, 69, 84, 90, 129, 130, 177,
180, 181, 227, 256, 263, 290,
292, 325, 327, 336-338, 340-
342, 347

M

machine, 6, 40, 41, 48, 70, 71,
74, 80, 82, 84-87, 89-97,
99-102, 107, 112, 116, 121,
122, 127-132, 135, 137-142,
144, 147, 148, 150, 151, 168,
230, 232, 236, 237, 239-247,
256, 264, 270, 272-274, 279,
300, 303, 304, 307, 338, 340,
341, 345, 347, 348

MMI, 347, 348

model, 19, 25-27, 29-33, 37-41, 45-
48, 52-56, 63, 65, 67-74, 76,
78-115, 121, 122, 125, 127-130,
135-139, 141, 142, 145, 147,
149-155, 159-181, 183-196,
198-228, 232-235, 237, 239,
241, 242, 244, 247, 249, 250,
252-260, 263-265, 273-278,
280-284, 289, 290, 292-302,
303, 305-307, 309, 311, 312,
314-323, 326-328, 330, 331,
334-342, 346-348

module, 3, 5, 6, 24, 28, 56, 68, 71,
80, 85, 86, 89-92, 95-97, 99,
129, 147, 151, 175, 178, 214,
221, 219, 227, 234, 235, 311,
315, 316, 318, 321, 323, 349

O

OCTYS, 262

P

PMI, 262

Proof, 28, 31, 32, 39-41, 47, 48,
62-64, 68, 69, 74, 78, 80, 98,
107, 112, 114, 116, 127, 128,
137-146, 148, 149, 151, 155,
183-187, 189, 190, 192-195,
198-201, 203, 205, 208-220,
231, 247, 262, 326, 327, 336,
348

protection, 36, 58, 107, 201, 261,
265, 283, 307, 345, 347-349

Prover, 17, 48, 142, 143, 145, 146,
155, 192, 193, 195, 196, 198,
199, 218, 251, 339, 340

R

railway, 6, 55, 56, 59, 65, 67, 72,
73, 78, 84, 98, 103, 107, 116,
117, 121, 127, 150, 151, 159,
162, 187, 201, 205-208, 225,
269, 301, 318, 325, 326, 335,
337

RATP, 40, 44, 47, 55, 56, 62, 69, 78-
80, 82, 102, 103, 108-110, 112,
114, 115, 153, 262, 263, 319,
322, 328, 348

refinement, 2, 40, 41, 68, 70, 71, 127-
129, 133-135, 138-140, 142,
144, 146-148, 155, 189, 191

reliability, 5, 10, 18-21, 24, 35, 40,
58, 67, 114, 127, 173, 227, 330,
347, 348

requirement, 18-21, 24, 40, 48, 62,
63, 76, 78, 82, 83, 104, 106,
107, 112, 116, 187, 209, 219,
225-227, 231, 240, 247, 249-
252, 256, 287, 289-292, 295-
298, 298, 300, 302, 312-314,
316-319, 329, 330, 346, 349

S

SACEM, 40, 62-65, 68, 80, 81, 128, 151, 192, 338, 349
 Safety, 4, 5, 7-10, 12-18, 21, 27, 30, 31, 33-36, 39, 40, 44-48, 55, 56, 58, 59, 61-67, 70, 74, 76, 77, 79, 81-84, 87, 92, 95-98, 103, 107, 109, 112-114, 116, 120, 127, 128, 142, 150, 152, 170, 177, 178, 184, 187, 188, 194, 202-204, 218, 219, 225-228, 230, 238-240, 242, 251, 252, 255, 262, 265, 270, 274, 283, 287-291, 294-296, 312-316, 318, 319, 322-325, 327-330, 333, 335-337, 339, 347-349
 SAGA, 42, 230, 265, 270
 SAO, 42, 43, 230, 256, 260, 270, 326, 349
 SCADE, 23, 32, 39, 42-48, 151, 183, 186, 187, 193, 219, 220, 225, 228, 230-235, 239-247, 249-253, 255-265, 268-270, 273-276, 279-284, 326, 335, 338, 349
 security, 226, 319, 348
 SIL, 7, 34, 36, 202, 219, 220, 265, 325, 345, 349
 SNCF, 264
 SSIL, 7, 10, 12, 13, 16, 21, 36, 45, 107, 115, 116, 142, 152-154, 202, 312-316, 323, 325, 339, 349
 standard, 6, 7, 18, 22, 25, 27, 28, 31-39, 48, 112, 116, 142, 152-154, 163, 177, 198, 202, 219-221, 227, 256, 277, 278, 289, 290, 302, 303, 311, 312, 316, 319, 322, 325, 327, 335, 336, 339, 340
 structure, 31, 38, 62, 64-66, 74, 78, 80, 84-88, 90, 93-95, 109-111,

133, 138, 140, 148, 150, 154, 172, 185, 186, 190, 193, 202, 204, 205, 208, 216, 218, 221, 228, 232-238, 243, 245, 247, 260, 265, 274, 295, 302, 303, 308, 309, 327, 332, 334, 345, 349

system, 1, 5, 15, 16, 19, 21, 25-27, 29-31, 33-38, 40, 44, 45, 55-59, 62-68, 71-75, 78-81, 84, 92, 96-99, 102, 103, 105, 107, 109, 112-115, 120, 121, 127, 128, 146, 151, 152, 160-164, 166, 170, 178, 183-193, 196-198, 201-203, 208, 211, 218, 225, 226, 228, 230, 232, 239, 240, 246, 247, 251, 256, 257, 260-265, 269, 270, 274, 277, 279, 280, 282, 284, 287, 289-294, 296-298, 300, 304, 307, 311, 312, 314-323, 328-330, 333, 335, 337-339, 345-349

T

test, 3-5, 8, 16, 17, 32, 33, 47, 74, 78, 79, 94, 97, 100-107, 113, 142, 149, 150, 155, 161, 163, 175, 176, 184, 193, 195, 212, 217, 221, 240, 247, 249, 251, 255, 273, 274, 276-284, 287, 289, 291, 296-297, 303, 313-317, 320-323, 325, 326, 332, 333, 345, 347-349
 testing, 4-6, 30, 32, 48, 69, 103, 115, 148, 162-164, 174-177, 184, 218, 221, 231, 249, 251, 255, 273, 274, 276, 279, 282, 284, 289, 291-293, 299, 309, 312-314, 316-320, 332, 346, 347, 350

THALES, 44, 201, 230, 256
 tools, 9, 11, 12, 16-18, 33, 35,
 38, 39, 41, 44, 46, 48, 66, 74, 75,
 78, 84, 107, 109, 111, 136, 141,
 142, 150, 155, 163, 174, 175,
 186, 188-193, 225, 226, 231, 238,
 240, 246, 247, 253, 257, 260,
 262, 265, 273, 283, 287, 289,
 292, 294, 296, 305, 316, 317,
 326, 327, 332, 336-340, 346
 train, 24, 36, 41, 56-59, 61, 62, 75,
 83, 84, 86, 87, 90, 93, 100, 105,
 107, 117, 119-121, 128, 191,
 194, 200-203, 205-211, 221,
 261, 287, 289, 291, 292, 294,
 295, 297-301, 307, 309, 311,
 315-323, 328, 335, 349

V

validation, 4, 16, 27-29, 31, 32, 48,
 55, 62, 69, 74-76, 78, 79, 82, 91,

94, 97, 100-103, 105-110, 112-
 115, 127, 137, 141, 147, 150,
 159, 162, 163, 173-178, 189,
 194, 202, 287, 289, 292-294,
 296, 299, 311, 312, 316, 318-
 323, 325, 333, 337, 350
 verification, 5, 6, 12, 16, 19, 27-32,
 38, 39, 47, 48, 62, 68, 75, 76,
 78-82, 87, 89, 96-98, 100, 101,
 103, 104, 106-109, 111-114,
 116, 137, 139, 144-146, 149,
 150, 155, 159, 162, 163, 173-
 178, 183-188, 190-195, 198,
 200, 203, 204, 213, 220, 221,
 227, 240, 246, 247, 249-253,
 256, 265, 273, 300, 315, 317,
 325, 327, 328, 330, 332, 335-
 337, 350

W

WCET, 350