

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Sureté de Logiciel et Calcul à Haute Performance

Présentée et soutenue par :

M. ARNAUD DIEUMEGARD

le vendredi 30 janvier 2015

Titre :

FORMAL GUARANTEES FOR SAFETY CRITICAL CODE GENERATION: THE CASE OF HIGHLY VARIABLE LANGUAGES

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. YAMINE AIT AMEUR

M. MARC PANTEL

Rapporteurs :

M. CLAUDE MARCHE, UNIVERSITE PARIS 11

M. RICHARD PAIGE, UNIVERSITY OF YORK

Membre(s) du jury :

M. PHILIPPE LAHIRE, UNIVERSITE DE NICE SOPHIA ANTIPOLIS, Président

M. MARC PANTEL, INP TOULOUSE, Membre

Mme VIRGINIE WIELS, ONERA TOULOUSE, Membre

M. PAUL GIBSON, TELECOM SUD PARIS, Membre

M. PHILIPPE CUENOT, SOCIETE CONTINENTAL AUTOMOTIVE FRANCE SA, Membre

Abstract

Control and command softwares play a key role in safety-critical embedded systems used for human related activities such as transportation, healthcare or energy. Their impact on safety makes the assessment of their correctness the central point in their development activities. Such systems verification activities are usually conducted according to normative certification guidelines providing objectives to be reached in order to ensure development process reliability and thus prevent flaws. Verification activities usually relies on tests and proof reading of the software but recent versions of certification guidelines are taking into account the deployment of new development paradigms such as model-based development, and formal methods; or the use of tools in assistance of the development processes.

Automatic code generators are used in most safety-critical embedded systems development in order to avoid human related software production errors and to ensure the respect of development quality standards. As these tools are supposed to replace humans in the software code production activities, errors in these tools may result in embedded software flaws. It is thus in turn mandatory to ensure the same level of correctness for the tool itself than for the expected produced code. Tools verification shall be done according to qualification guidelines. We advocate in our work the use of model-based development and formal methods for the development of these tools in order to reach a higher quality level.

Critical control and command software are mostly designed using graphical dataflow languages. These languages are used to express complex systems relying on atomic operations embedded in blocks that are gathered in block libraries. Blocks may be sophisticated pieces of software with highly variable structure and semantics. This variability is dependent on the values of the block parameters and of the block's context of use.

In our work, we focus on the formal specification and verification of such block based languages. We experimented various techniques in order to ensure a formal, sound, verifiable and usable specification for blocks. We developed a domain specific formal model-based language specifically tailored for the specification of structure and semantics of blocks. This specification language is inspired from software product line concepts in order to ensure a correct and scalable management of the blocks variability. We have applied this specification and verification approach on chosen block examples from common industrial use cases and we have validated it on tool prototypes.

Blocks are the core elements of the input language of automatic code generators used for control and command systems development. We show how our blocks formal specification can be translated as code annotations in order to ease and automate the generated code verification. Code annotations are verified using specialised static code analysis tools. Relying on synchronous observers to express high level requirements at the input model level, we show how formal block specification can also be used for the translation of high level requirements as verifiable code annotations discharged using the same specialised tooling. We finally target the assistance of code generation tools qualification activities by arguing on the ability to automatically generate qualification data such as requirements, tests or simulation results for the verification and development of automatic code generators from the formal block specification.

Resumé

Les fonctions de commande et de contrôle sont parmi les plus importantes des systèmes embarqués critiques utilisés dans des activités telles les transports, la santé ou la gestion de l'énergie. Leur impact potentiel sur la sûreté de fonctionnement fait de la vérification de leur correction l'un des points les plus critiques de leur développement. Cette vérification est usuellement effectuée en accord avec les normes de certification décrivant un ensemble d'objectifs à atteindre afin d'assurer un haut niveau de qualité du système et donc de prévenir l'apparition de défauts. Cette vérification du logiciel est traditionnellement basée sur de nombreux tests et des activitiés de relectures de code, toutefois les versions les plus récentes des standards de certification permettent l'utilisation de nouvelles approches de développement telles que l'ingénierie dirigée par les modèles et les méthodes formelles ainsi que l'utilisation d'outil pour assister les processus de développement.

Les outils de génération automatique de code sont exploités dans la plupart des processus de développement de systèmes embarqués critiques afin d'éviter des erreurs de programmation liées à l'humain et pour assurer le respect des règles de production de code. Ces outils ayant pour vocation de remplacer les humains pour la production de code, des erreurs dans leur conception peuvent causer l'apparition d'erreurs dans le code généré. Il est donc nécessaire de vérifier que le niveau de qualité de l'outil est le même que celui du code produit en s'assurant que les objectifs spécifiées dans les normes de qualification sont couverts. Nos travaux visent à exploiter l'ingénierie dirigée par les modèles et les méthodes formelles pour développer ces outils et ainsi atteindre un niveau de qualité plus élevé que les approches traditionnelles.

Les fonctions critiques de commande et de contrôle sont en grande partie conçues à l'aide de langages graphiques à flot de données. Ces langages sont utilisés pour modéliser des systèmes complexes à l'aide de blocs élémentaires groupés dans des librairies de blocs. Un bloc peut être un objet logiciel sophistiqué exposant une haute variabilité tant structurelle que sémantique. Cette variabilité est à la fois liée aux valeurs des paramètres du bloc ainsi qu'à son contexte d'utilisation.

Dans notre travail, nous concentrons notre attention en premier lieu sur la spécification formelle de ces blocs ainsi que sur la vérification de ces spécifications. Nous avons évalué plusieurs approches et techniques dans le but d'assurer une spécification formelle, structurellement cohérente, vérifiable et réutilisable des blocs. Nous avons finalement conçu un langage basé sur l'ingénierie dirigées par les modèles dédié à cette tâche. Ce langage s'inspire des approches des lignes de produit logiciel afin d'assurer une gestion de la variabilité des blocs à la fois correcte et assurant un passage à l'échelle. Nous avons appliqué cette approche et la vérification associée sur quelques exemples choisis de blocs issus d'applications industrielles et l'avons validé sur des prototypes logiciels que nous avons développé.

Les blocs sont les principaux éléments des langages d'entrée utilisés pour la génération automatique de logiciels de commande et de contrôle. Nous montrons comment les spécifications formelles de blocs peuvent être transformées en des annotations de code afin de simplifier et d'automatiser la vérification du code généré. Les annotations de code sont vérifiées par la suite à l'aide d'outils spécialisés d'analyse statique de code. En utilisant des observateur synchrones pour exprimer des exigences de haut niveau sur les modèles en entrée du générateur, nous montrons comment la spécification formelle de blocs peut être utilisée pour la génération d'annotations de code et par la suite pour la vérification automatique des exigences. Finalement, nous montrons dans quelle mesure les spécifications de blocs permettent de générer des données de qualification tel que des exigences, des tests ou des données de simulation utilisées pour la vérification et le développement de générateurs automatiques de code.

Table of Contents

Contents 9

List of Figures 12

List of Tables 13

1 Introduction 1
- 1.1 Embedded critical software . . . 1
- 1.2 Software verification and validation . . . 2
- 1.3 Model Driven Engineering and Formal Methods . . . 2
- 1.4 Automatic code generation . . . 3
- 1.5 The Simulink use case . . . 3
- 1.6 Research objectives . . . 3
- 1.7 Contributions . . . 4
- 1.8 Plan . . . 5

I Critical embedded systems design and implementation 7

2 Industrial context 9
- 2.1 Certification/Qualification . . . 9
  - 2.1.1 Terminology . . . 9
  - 2.1.2 *DO-178B*– Software Considerations in Airborne Systems and Equipment Certification . . . 10
  - 2.1.3 *DO-178C*– Software Considerations in Airborne Systems and Equipment Certification . . . 10
  - 2.1.4 *DO-330*– Software Tool Qualification Considerations Companion . . . 11
  - 2.1.5 *DO-331*– Model-Based Development and Verification Supplement . . . 12
  - 2.1.6 *DO-332*– Object-Oriented Technology and Related Techniques Supplement . . . 13
  - 2.1.7 *DO-333*– Formal Methods Supplement . . . 13
- 2.2 The need for domain specific languages . . . 14
  - 2.2.1 From DSML to embeddable software through automatic code generation . . . 14
- 2.3 Our local research contributions to the ACG development field . . . 15
  - 2.3.1 The GeneAuto ACG example . . . 15
  - 2.3.2 The Projet-P/Hi-MoCo approach . . . 17
  - 2.3.3 Other research contributions to the ACG development field . . . 18
- 2.4 Synthesis . . . 19

3 Languages formal specification - State of the art 21
- 3.1 Preliminary definitions . . . 21
  - 3.1.1 Language . . . 21
  - 3.1.2 Formal . . . 23
  - 3.1.3 Specification . . . 23

3.1.4 Variable . . . 23
3.1.5 Highly variable language . . . 23
3.2 Model Driven Engineering . . . 24
3.2.1 Definition . . . 24
3.2.2 Use . . . 24
3.2.3 State of the art frameworks and languages . . . 24
3.2.4 Software product line engineering . . . 26
3.3 Software formal verification . . . 28
3.3.1 Static analysis of source code . . . 28
3.3.2 Deductive verification . . . 30
3.3.3 The Why3 platform . . . 31
3.4 Domain analysis: languages variability . . . 32
3.4.1 Languages variability examples . . . 33
3.4.2 Language variability analysis . . . 34
3.5 Synthesis . . . 36

4 Dataflow languages **37**
4.1 Dataflow languages . . . 37
4.1.1 Origins . . . 37
4.1.2 Dataflow languages for critical systems development . . . 38
4.1.3 Synchronous dataflow languages . . . 38
4.1.4 Realistic implementation of KPN . . . 38
4.1.5 Dataflow model execution . . . 38
4.2 Dataflow model structure . . . 40
4.2.1 Graphical dataflow model structure . . . 41
4.2.2 Dataflow languages . . . 43
4.3 Well founded dataflow model . . . 44
4.3.1 Causality errors . . . 44
4.3.2 Data type overflow . . . 45
4.4 Dataflow languages block semantics variability . . . 45
4.5 Challenges to tackle regarding specification . . . 46

II Highly variable languages formal specification **47**

5 Experiments with classic software engineering tools **49**
5.1 Block specification requirements . . . 49
5.2 Block examples . . . 50
5.2.1 MinMax block . . . 50
5.2.2 Delay block specification . . . 53
5.3 Mathematical notation for the specification of blocks . . . 54
5.4 UML for the specification of blocks . . . 56
5.4.1 UML block specification harness . . . 57
5.4.2 Injecting variability into UML . . . 61
5.4.3 Profiled UML + OCL specification for *MinMax* block . . . 63
5.4.4 Profiled UML + OCL specification for *Delay* block . . . 70
5.4.5 Choices made regarding UML modeling . . . 73
5.4.6 Limitations of the UML + profile + OCL specification approach . . . 73
5.5 SPLE for the specification of blocks . . . 73
5.5.1 SPLE specification approach . . . 74
5.5.2 SPLE specification of the *MinMax* block . . . 74

5.5.3 SPLE specification of the *Delay* block . . . 76
5.5.4 Limitation of the SPLE specification approach . . . 78
5.6 Two complementary approaches . . . 79
5.6.1 Methodology proposal . . . 79
5.6.2 From SPLE analysis to UML model . . . 79
5.6.3 Limitations of the methodology . . . 80
5.7 Synthesis . . . 81

6 A Domain Specific and Product Line experiment for language specification **83**
6.1 Domain analysis . . . 83
6.1.1 Domain of study . . . 83
6.1.2 Variability modeling . . . 84
6.2 The BlockLibrary DSML . . . 84
6.2.1 *Delay* block interfaces specification . . . 84
6.2.2 *Delay* block textual specification . . . 87
6.2.3 BlockLibrary metamodel abstract elements . . . 87
6.2.4 Annotations . . . 89
6.2.5 Data types specification . . . 91
6.2.6 Block structural features . . . 92
6.2.7 BlockLibrary metamodel variability structure . . . 94
6.2.8 BlockLibrary metamodel specification containers . . . 98
6.3 Relation to feature modeling . . . 99
6.3.1 Conversion of a `BlockType` to a feature model . . . 99
6.3.2 Automatic feature model analysis . . . 100
6.4 From block specification to configurations . . . 101
6.4.1 Preliminary operations definitions on BlockLibrary elements . . . 101
6.4.2 `Configuration` and `Signature` constructs . . . 102
6.4.3 Operations based on `Signature` constructs . . . 104
6.4.4 Extraction of `Signature` elements . . . 105
6.4.5 Extraction of `Configuration` elements . . . 105
6.5 Semantics modeling . . . 107
6.5.1 Block semantics phases contracts . . . 108
6.5.2 Block semantics contract encoding with dynamic behaviors . . . 109
6.6 Specification verification properties . . . 111
6.6.1 Well-formedness . . . 111
6.6.2 Variability coverage . . . 111

7 BlockLibrary specifications formal verification **113**
7.1 Verification prerequisites . . . 113
7.1.1 OCL specification . . . 114
7.1.2 BAL specification . . . 114
7.1.3 Why3 platform . . . 115
7.1.4 Transformation technology choice . . . 115
7.2 BlockLibrary specification example . . . 116
7.3 Why3 libraries . . . 116
7.3.1 Primitive data types theory . . . 116
7.3.2 BlockLibrary `StructuralFeature` theory . . . 119
7.4 OCL expressions transformation . . . 123
7.4.1 OCL standard library operations . . . 124
7.4.2 Collection operations . . . 125
7.4.3 Logical property assessment iteration operations . . . 126

7.4.4 Value extraction iteration operations . . . 127
7.5 The BAL expressions transformations . . . 130
7.6 BlockLibrary verification transformations . . . 131
7.6.1 Variability verification transformation . . . 131
7.6.2 Semantics verification transformation . . . 135
7.7 Variability verification through SMT solving . . . 135
7.7.1 Specification extract variability verification . . . 136
7.7.2 Entire specification verification . . . 137
7.7.3 Goals transformation as a mean to ease the verification . . . 137
7.8 Semantics verification through SMT solving . . . 140
7.8.1 Hoare triple verification . . . 140
7.8.2 Adding loop invariants for the verification . . . 141
7.8.3 Automatic generation of invariants . . . 142
7.9 Scalability . . . 142
7.10 Limitations . . . 143
7.10.1 Dataflow languages capabilities limitations . . . 143
7.10.2 OCL and BAL Expressiveness limitation . . . 143
7.10.3 BlockLibrary limitations . . . 143
7.11 Synthesis . . . 143

III Automatic code generation verification based on the block library specification 145

8 Verification of generated code **147**
8.1 Annotations for code verification . . . 147
8.1.1 `Configuration` matching of block . . . 147
8.1.2 Annotation generation . . . 148
8.1.3 Annotation verification . . . 152
8.1.4 Tool support . . . 152
8.2 Formal verification . . . 154
8.2.1 Synchronous observers . . . 155
8.2.2 Concrete application on the *Counter* system . . . 155
8.2.3 Logical expression extraction . . . 158
8.2.4 Main module generation . . . 160
8.2.5 From specific kind of properties to annotations . . . 161
8.2.6 A parallel work based on *SO* . . . 161
8.3 Gain wrt classical verification activities . . . 162
8.3.1 A complement to state of the art design verification . . . 162
8.3.2 Current limitations and perspectives . . . 162

9 Certification/Qualification data generation **163**
9.1 BlockLibrary for qualification . . . 163
9.1.1 *DO-331*: Model-based technology . . . 164
9.1.2 *DO-333*: Formal methods . . . 164
9.2 BlockLibrary use for the certification of an ACG tool . . . 165
9.2.1 Providing unambiguous expression of requirements and architecture . . . 165
9.2.2 Supporting the use of automatically generated code . . . 167
9.2.3 Supporting the use of analysis tools for verification of requirements and architecture. 167
9.2.4 Supporting the use of simulation for partial verification of requirements, architecture, and/or Executable Object Code. . . . 167
9.2.5 Supporting the use of automated test generation . . . 169

9.3 Additional required verifications . . . 169

10 Conclusion & Future work **171**
10.1 Research objectives fulfillment . . . 171
10.1.1 Research objective 1: Formal specification and verification of highly variable languages . . . 171
10.1.2 Research objective 2: Uses of highly variable language formal specification for automated generated code verification . . . 172
10.1.3 Research objective 3: Uses of highly variable language formal specification for ACG qualification . . . 172
10.2 Concrete productions . . . 173
10.3 Future research directions . . . 173
10.3.1 BlockLibrary-related activities . . . 173
10.3.2 Overall approach future works . . . 175

Appendices **177**

A Complete block specifications **179**
A.1 *Delay* block specifiation . . . 179
A.2 *MinMax* block specification . . . 189

B OCL grammar **193**

C BAL grammar **199**

D Why3 libraries **203**
D.1 Why3 data types theories . . . 203
D.1.1 Numeric data types definition theories . . . 203
D.1.2 Common functions definition theories . . . 208
D.1.3 String data types definition theories . . . 208
D.2 BlockLibrary `StructuralFeature` definition theory . . . 212
D.3 Generic functions definitions and general purpose lemmas . . . 214
D.4 OCL language operations definitions . . . 216
D.5 OCL iteration operations definitions . . . 227

E ACSL verification using Frama-C **233**

References **239**

List of Figures

1.1 Code generation verification strategy . . . 5

2.2 GeneAuto toolkit architecture . . . 16
2.4 Projet-P toolkit architecture . . . 19

3.1 Space probe system Feature Model . . . 26

4.2 MultiRate dataflow model, boolean clock flow and block activation . . . 39
4.5 Metamodel for dataflow models . . . 42
4.6 GeneAuto DataTypes metamodel . . . 43
4.7 Simulink model for a modulo 3 counter . . . 44
4.8 A causality error example in a Simulink model . . . 45
4.9 Simulink model with different configurations of the *Sum* block . . . 46

5.9 Common specification data types . . . 54
5.10 Common specification for block structural elements . . . 54
5.11 Common specification operations of block structural elements . . . 55
5.12 *Sum* allowed inputs specification . . . 55
5.14 Sum semantics specification . . . 56
5.15 Specification of the block structural elements value using UML + OCL . . . 57
5.21 Generic block specification using UML + OCL . . . 61
5.24 SMarty variability UML profile . . . 62
5.26 MinMax block specification using UML + variability profile + OCL . . . 63
5.33 Semantics variants definition for the *MinMax* block . . . 68
5.38 Delay block specification using the UML . . . 71
5.41 A feature model for the *MinMax* block structure and semantics . . . 74
5.42 A feature model for the *Delay* block structure and semantics . . . 76
5.44 Delay block specification using UML + OCL . . . 81

6.1 The *BlockLibrary* metamodel . . . 85
6.2 The *Delay* block specification hierarchy . . . 86
6.5 The BlockLibrary metamodel abstract metaclasses . . . 89
6.6 The *BlockLibrary* `Annotations` metaclass definition . . . 90
6.8 The BlockLibrary metamodel structural features definition elements . . . 93
6.11 The BlockLibrary metamodel variability structure . . . 95
6.16 FM extracted from the *Delay* block `BlockType` specification . . . 100
6.17 The `Configuration` metaclass . . . 103

7.1 MDE architecture of the transformation . . . 115
7.24 Select lemmas verification with Why3 through SMT solvers and proof assistants . . . 129
7.26 Overview of the BlockLibrary to Why3/WhyML transformation . . . 132
7.27 Block domain extraction of `StructuralFeature` data types . . . 132
7.28 Block domain extraction of `StructuralFeature` `INVARIANT` `Annotation` . . . 133

7.29 `Signature` domain extraction of `MODE_INVARIANT` `Annotation` 134
7.31 `Signature` function with contract extraction of `BlockMode` specification 136
7.36 Completeness goal transformation application methodology 139
7.37 Disjointness goal transformation application methodology 139

8.1 `Configuration`-specific generated metamodel example 148
8.2 Counter Simulink model 150
8.6 GeneAuto annotations extension metamodel 153
8.8 The *Observer* block, its content and its parameters view in the Simulink environment . . 155
8.9 The *Counter* model with its *counter_spec* synchronous observer 156
8.10 The *counter_spec* synchronous observer content 156
8.14 An abstract synchronous observer 159

9.1 BlockLibrary use for ACG verification and development 166
9.2 BlockLibrary use for test procedures generation and verification 168

10.1 Generic language variability specification metamodel 176

D.11 String theory lemmas verification with Why3 and SMT solvers 209
D.13 InPortGroup theory lemmas verification with Why3 and SMT solvers 213
D.22 CommonFunctions theory lemmas verification with Why3 and SMT solvers 215
D.24 *OclType* theory lemma verification with Why3 and SMT solvers 216
D.34 OCL operations theory lemmas verification with Why3 and SMT solvers 221
D.37 OCL operations theory lemmas verification with Why3 and SMT solvers (II) 225
D.47 OCL operations theory lemmas verification with Why3 and SMT solvers (III) 231

List of Tables

2.1 *DAL* and *TQL* relations . . . 11

5.1 Block structure specification requirements . . . 50
5.2 Input/Output ports structure specification requirements . . . 50
5.3 Parameters structure specification requirements . . . 50
5.4 Memories structure specification requirements . . . 51
5.5 Data type and dimensionality specification requirements . . . 51
5.6 Semantics specification requirements . . . 51
5.7 Specification verification requirements . . . 52
5.8 Related tooling requirements . . . 52
5.13 *Sum* allowed parameters specification . . . 55
5.34 Semantics variation point cross tree constraints . . . 68
5.43 Mapping from *Delay* block feature mode elements to UML classes . . . 80

6.3 Relation between Delay value, U dimension, IC dimension and M dimension . . . 86

7.3 Mapping between our type system and the Why3 types . . . 116
7.11 OCL collections characteristics . . . 123
7.13 OCL primitive numeric types operations mapping to Why theories functions . . . 124
7.14 OCL String operations mapping to Why theories functions . . . 124
7.15 OCL logical operators mapping to Why operators . . . 125
7.16 OCL arithmetic operators mapping to Why operators . . . 125
7.17 OCL relational operators mapping to Why operators . . . 126
7.18 OCL collection operations mapping to Why operators . . . 126
7.20 OCL logical property verification operations mapping to Why expressions . . . 127
7.21 OCL iteration operations mapping to Why expressions . . . 128
7.22 OCL iteration operations mapping to Why high order logic functions . . . 128
7.25 BAL expressions to WhyML translation rules . . . 131
7.34 Some blocks specification verification performances . . . 137

8.7 OCL to ACSL translation rules . . . 154
8.12 *counter_obs* blocks post-conditions . . . 157

Acknowledgments

Je souhaite en premier lieu remercier Philippe Cuenot pour m'avoir initié à la recherche, aux concepts de l'ingénierie dirigée par les modèles et pour avoir suffisamment cru en moi pour me présenter à ses collègues académiques qui m'ont permis de faire cette thèse.

Cette thèse n'aurais bien entendu jamais pu être possible sans le soutient et l'aide de Marc que je remercie tout particulièrement. Je souhaite aussi le remercier pour toutes ces discussions que nous avons pu avoir durant ces quelques années passées au laboratoire, discussions qui ont tendance la plupart du temps à digresser mais qui ont toujours été très instructives.

Ces années ont été d'autant plus agréables que j'ai été accueilli dans une équipe qui m'a beaucoup apporté et avec qui j'ai passe de très bons moments. Je remercie (et je l'espère sans omission): Yamine, Philippe, Philippe, Xavier, Xavier, Aurélie, Mamoun, Manuel, Célia, Florent, Guillaume.

Je souhaiterais remercier tout spécialement celles qui ont su supporter à la fois mes boulettes et omission, les organisations de déplacement et d'évènements quasiment toujours a la dernière minute et ce toujours avec le sourire (et quelques menaces quand même): les Sylvies.

Il est bien entendu un grand nombre d'amis qui m'ont soutenu durant ces années, qui ont été la pour m'aider, me faire changer d'air et me proposer des apéros au bords de l'eau. Pour tout cela, je dis un grand merci à Stiff, Dédé, Tito, Kaka, Matt, Camille, Fabichou, Jean-Marc, Ben, Juju, Sisi, Nico, Emilie, Benoît, Annaick, Guido, Minette, Daminou, Julie, Gibon, Morue, Nico, Mawish, Toto, Erell et ceux que j'oublie.

A mes parents qui m'ont toujours soutenu et poussé à aller plus loin, et ce malgré le baobab qui peut me pousser dans la main parfois, je souhaite dédier cette thèse.

Finalement, à celle qui partage ma vie, qui a supporté mes sautes d'humeur sans soucis, celle sans qui tout cela aurais été bien plus difficile, je souhaite dire infiniment merci et je t'aime ma duchesse.

1 Introduction

Industrial size experiments and real products have shown Model Driven Engineering and Formal Methods to be key assets in the development of complex safety critical systems. Both relies on domain specific modeling languages and associated verification and generation tools. Language engineering is once again an enabler to ease the development of these languages. This PhD targets highly variable languages engineering through a key case study in the development of safety critical systems: dataflow languages, also called block diagrams, like Simulink[1] or Scade[2].

We characterise the high variability of languages through the ability for languages to be composed of elements of variable structure or semantics. To handle this variability, we provide in this PhD a tooled formal specification methodology for our use case of block diagrams that is representative of such languages. We also develop on the uses of such formal specification for automatic code generation verification and for automatic code generators qualification in the context of embedded critical software. This work should pave the way for more generic methods for specifying high variability languages and deriving generation and verification tools from such specification.

1.1 Embedded critical software

Embedded critical software are pieces of software that take part in the global behavior of a complex system whose function is not limited to this software. For example, the control of actuators in transportation systems. The purpose of the system is to provide transportation to human, whereas the embedded software system goal is often to control or command hardware pieces like sensors and actuators that provide transportation. Embedded software systems are used in a wide variety of environment: from everyday devices like coffee machines to the most complex systems like space probes, planes or magnetic resonance imaging scanner.

According to the functionality of the controlled/commanded system, its quality must be adapted: if for some reasons, a coffee machine software enters a wrong state that causes a coffee cup to be over-filled, the consequences are not as catastrophic as if a plane door control system enters a state causing the doors to open during a flight. The level of quality of the software in this case is related to the need for safety and reliability regarding the appearance of system failures that may have a potential impact on human lives or economic losses.

Embedded safety-critical software systems are software systems that must not experience unhandled failures leading it to a state where its safe behavior is not ensured. To our knowledge, the need for these

[1] http://www.mathworks.com/products/simulink/
[2] http://www.esterel-technologies.com/products/scade-suite/

systems reliability is the highest.

Our technology-driven society is in need for an ever increasing improvement, automation and efficiency of systems and as a consequence of their embedded and control/command software. This leads the systems complexity to rise, leading to higher reliability concerns. Safety-critical software systems are also subject to such changes and are thus also affected by these issues. Their nature makes such evolution difficult to manage and the assurance of safety and reliability a difficult problem to assess.

1.2 Software verification and validation

Safety-critical systems industries have long been aware of the previous problems. Ensuring a certain level of quality in their developed software is mostly done by relying on highly documented development process with a very important emphasis put on verification and validation (V&V) activities and on the emphasis of traceability information. These terms: verification and validation are differently interpreted according to the application domain. In this document, verification activities aims at ensuring that the developed software/system is rightly done according to a certain set of rules and criteria (the software is rightly implemented) whereas validation activities aims at ensuring that the developed software/system is the one that was expected (the right software is implemented). Traceability information provides the links between the developed artifacts in the different steps of the development process. Traceability informations are among the most used artifact for highlighting the modifications operated on the software or by the software. It is thus of primary interest to ensure that traceability is ensured all along the development process of safety-critical systems and that traceability informations are provided by tools used during such development processes.

Ensuring the correctness of the whole development process and V&V activities is achieved according to regulations by fulfilling qualification or certification objectives. Qualification/Certification objectives are supposed to enforce a certain level of quality and confidence on the developed system by requiring the development process actors to provide data on the performed verification and validation activities. Qualification/Certification is domain and even product specific and the set of objectives to achieve is defined in domains specific documents among which are: *ECSS-Q80A* for space, *DO-178* for avionics, *CEI62[278|279|425]* for railway, *ISO26262* for automotive, *CEI60880* for nuclear energy management or *CEI60601-1-4* for medical applications. Other documents aims at applying on a wider variety of domain like *ISO61508* dealing with *Functional safety of electrical/electronic/programmable electronic safety-related systems*.

In our work, we are targeting aeronautic applications as it is known as being among the more stringent ones with regard to safety regulation. In this domain, establishment of qualification/certification is required by authorities like the Federal Aviation Administration (FAA[3]) in the USA or the European Aviation Safety Agency (*EASA*[4]) that have been granted the authority to allow or not any civil aircraft to operate on their respective geographic area.

1.3 Model Driven Engineering and Formal Methods

From the earliest application of mathematics to model the physical world to the modern days of computer science, Model Driven Engineering (MDE) has grown as an answer to systems design and development complexity. It allows through the use of models to abstract ourselves from the system complexity and through the use of model-based tools to manipulate models. From such tools, the MDE user will gain the ability to perform model analysis, extract informations, transform models to other models, source code or documentation. The wide adoption of MDE by researchers and industrials has been driven to a large extend by technology standardisation consortium like the Object Management Group (OMG) [5] whom led the way to the definition of standards, or the Eclipse community that has developed and provided as

[3] http://www.faa.gov/
[4] https://www.easa.europa.eu/
[5] http://www.omg.org

open source software many tools implementing these standards.

Formal Methods in computer science have been developed in order provide means to formally reason about the soundness of programs and electronic hardware. Formal reasoning settles on mathematical logic and has evolved to complex mathematical-based reasoning frameworks. A large nebula of logical field specific formal methods has since flourished allowing to analyse systems and software against various requirement domains like for example real-time, concurrency, or run-time.

While the formal nature of these methods was a restrain to its adoption in the computer science community, the advent of MDE provided the necessary abstraction and automation mechanism in order to help on their adoption and use. While still not being straightforward to apply on industrial applications, formal methods advantages are no more discussed and their adoption is well on the way as advocated by their integration in industrial standards like the aeronautic one.

1.4 Automatic code generation

As system complexity is ever rising, the writing of their software is an activity that suffers from the same complexity issue. The use of tools such as automatic code generators has been experienced and adopted in order to tackle this complexity issue. Indeed tools are more likely to avoid errors than humans if they are used for repetitive activities. This is particularly true for code production activities when the requirements are precise enough like models.

Automatic code generators (ACG) are tools whose function is to apply transformation (code production) rules on input language elements and producing formatted text (such as source code). It is therefore of particular interest to formally know what are the input language elements in order to specify and develop the ACG itself. Scade and Simulink are widely used in the modeling of safety critical systems in the transportation domains. ACG like KCG, RTW-EC or Target Link are then used to produce the software.

1.5 The Simulink use case

Embedded control and command software are designed by engineers whose background is mostly focused on automation analysis and mathematical modeling of systems. The leading formalism in this domain is the Simulink language which is built on the mathematical scientific computing platform Matlab[6].

Simulink models are made up of blocks linked through their ports by signals. While signals are only carrying data between blocks, the blocks are using the values obtained on their inputs and their internal state to compute their outputs. Blocks are gathered in block libraries containing their structural and semantics informations. Each block is configurable according to a set of parameters that allow selecting a specific structure and semantics. The context in which the block is used, defined by the data types and dimensions (if it is a scalar, a vector, a matrix) of its input ports will also impact the block semantics.

System design models written in Simulink are used as specifications for the development of software that is embedded in the end system. This development is done either by manual software writing leading to human related unavoidable errors requiring huge testing and review operations, or by automatic code generation which is more likely to cause less coding errors but whose soundness must be ensured as the tools can introduce errors.

In this PhD, we will focus on this second option and provide means to ensure such a soundness based on the formal specification of the Simulink language. Both configuration and context of use of the block implies a high level of variability of the block that must also be specified in order to fulfill this objective.

1.6 Research objectives

We have identified the following three research objectives whose fulfillment will be of principal interest in this PhD work:

[6] http://www.mathworks.com

- **Research objective 1: Formal specification and verification of highly variable languages**

  Highly variable languages, like block diagrams, while being widely used in the industry for the design of highly critical control and command software are most of the time under-specified and thus cannot be used as is during a full software development process based on MDE and formal methods. In order to limit the number of translation steps in the system development and thus their impact on the quality and verification of the system; it is required to formally specify the languages including their variability and to factorise existing common points between tools. Such formalisation must reduce the misinterpretations of the language and enhance the system development activities quality and as such is now required in avionics certification and qualification standards.

- **Research objective 2: Uses of highly variable language formal specification for automated generated code verification**

  Automatic code generation is used as a replacement for traditionally humanly carried out software writing activities. From a formally defined language, the verification of an automated code generation must be simplified as we may rely on the specification for a formalisation of the code generation requirements and may be able to formally reason on them. Two level of requirements are at stake here, low level requirements (LLR) that are close to the language specification and translation, and high level requirements (HLR) that may be provided as design language constructs and then translated as annotations on the code for verification.

- **Research objective 3: Uses of highly variable language formal specification for ACG qualification**

  Automatic code generator are sensitive tools whose output and correct operation must be verified. In the context of their use for safety-critical embedded systems development, their development and verification must be done according to the system domain certification/qualification standards. Such standards require the production of detailed data regarding the conducted development and verification activities. From the formal specification of the ACG input language and the formal verification of the generated code, such data production must be simplified, automatised and its reliability might be improved.

1.7 Contributions

In this PhD, our purpose is to provide a first approach to the formal specification and verification of high variability languages by providing a methodology and associated tools focused on the Simulink language specification. From such a specification, we will show how to assist the verification of code automatically generated from such languages and the certification and qualification activities related to the development of safety-critical embedded software. The overall architecture of our approach is provided in Figure 1.1. Our contributions are highlighted in this Figure and detailed in the following:

1. **Highly variable languages specification** In the block specification approach, we detail various methodologies and technologies for the specification of highly variable language. We then discuss their advantages and drawbacks and finally provide a customised approach that better achieves this objective thanks to a formal, tooled and domain dedicated specification language. This contribution is developed though the case of the Simulink dataflow language and the specification of its functional blocks.

2. **Specification verification transformation** From a structured highly variable language specification we provide a verification mechanism translating the specification to formal languages on which manual or automatic formal verification can be done. We also emphasis on the verification criteria used in order to ensure variability verification correctness. We rely on state of the art verification technologies having the advantage of being formal, well maintained and under active development: the Why3 toolset.

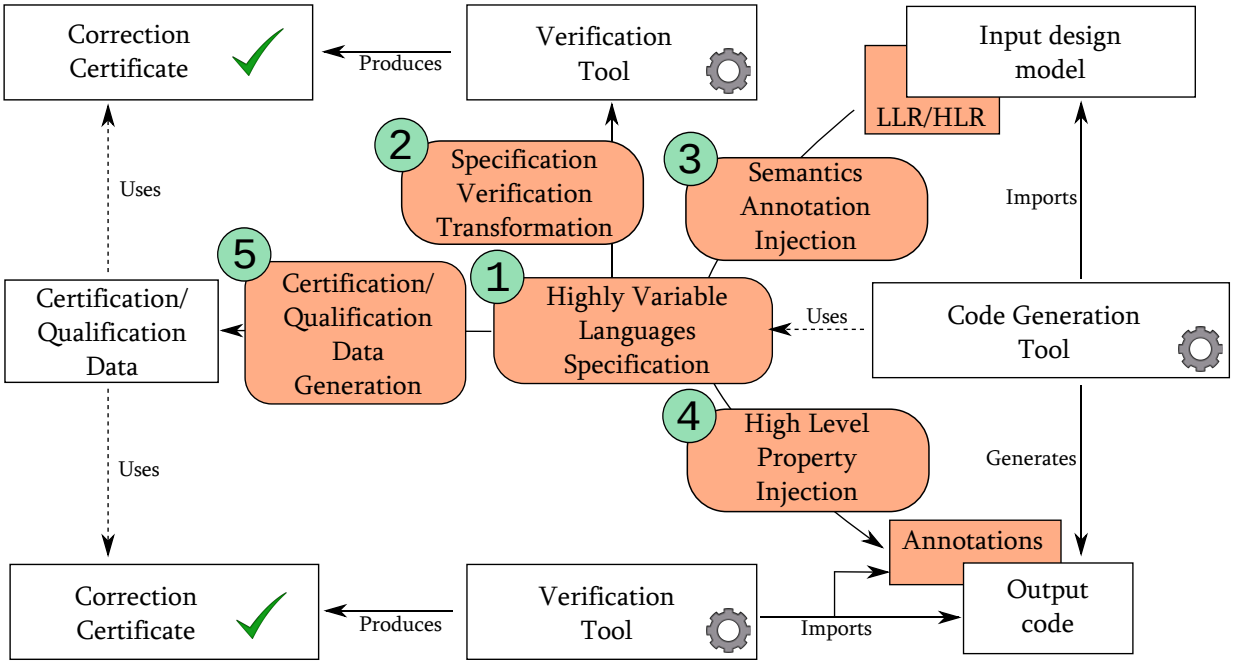


Figure 1.1: Code generation verification strategy

3. **Semantics annotation injection** We advocate our ability to automatically inject input languages related semantics informations as annotations on automatically generated code from formally specified highly variable languages. We then highlight on the formal verification of these semantics annotations on the generated code and on the guarantees it provides on the verification of automatic code generators outputs.
4. **High level properties injection** We argue on the possibility to rely on language specifications in order to translate high level properties (*HLR*) expressed using these languages. Such properties might then be manually or automatically verified with the help of state of the art verification tools.
5. **Certification/Qualification data generation** Highly variable languages automated code generators verification is strongly dependent on the ability to provide a specification for their input languages. We argue on the wide possibilities offered by a formally verified formal specification of such languages in order to automatically generate reliable data that can be used in certification/qualification activities.

1.8 Plan

We divided this PhD into 2 parts comprising 10 chapters and 5 appendix providing additional informations on some elements detailed in this PhD:

- Chapter 1: introduces the PhD work and states the challenges we tackle.
- Chapter 2: presents the industrial context in which our PhD work evolves.
- **Part 1: Highly variable languages formal specification**
  - Chapter 3: presents the state of the art in programming languages formal specification.
  - Chapter 4: focuses on dataflow languages specificities and presents the challenges for their specification.
  - Chapter 5: proposes dataflow languages symbols specification methodologies using model-based approaches.

  - Chapter 6: defines our BlockLibrary specification language.
  - Chapter 7: describes our BlockLibrary specifications language formal verification methodology and tooling.

- **Part 2: Automatic code generation verification based on the block library specification**

  - Chapter 8: gathers the block library specification use for automatic code generation verification.
  - Chapter 9: proposes block library specification uses for certification/qualification activities.
  - Chapter 10: concludes the principal part of the PhD and outlines future research directions.

- **Appendix**

  - Appendix A: contains complete examples of block specification.
  - Appendix B: depicts the grammar for our implementation of the OCL language.
  - Appendix C: provides the grammar for our custom action language.
  - Appendix D: contains the Why3 theories developed during this PhD.
  - Appendix E: contains a complete example of generated code verification with observers.

Part I

Critical embedded systems design and implementation

2
Industrial context

Establishing the certification of a developed system or the qualification of a tool for the development or verification of systems is checked by certification authorities. Their mission is to ensure the achievement of the required objectives according to the assurance level that needs to be reached. Development techniques and technologies used for system development are assessed by certification authorities in the domain of interest. These ones will only allow the use of reliable technologies and techniques for safety-critical embedded software systems development.

In this chapter we will go in details on the current status of industrial uses, limitations and constraints applied on embedded critical software systems development. We will first detail recent updates of qualification/certification methodologies in this context and their impact on system development. We will then emphasize on the use of domain specific languages (DSL) and their interest in this field. Finally, the use of Automatic Code Generators (ACG) for the development of embedded critical software will be detailed through two industrial-size ACG tools.

2.1 Certification/Qualification

The development quality of the safety-critical embedded software has been, for many years now, ensured by applying precise and constraining development processes on each activity of the system development: from requirements elicitation to concrete embedding of code through software design, code production and code verification. For each of these activities data are produced aiming at providing evidence of the means used for the activity achievement. These data are mandatory to provide parts of the informations required by certification standards.

In this work, we will focus on the application of *DO-178* certification document as it is used for the regulation of aeronautical software systems development activities that are, to our knowledge, the more stringent ones.

2.1.1 Terminology

In the following, we will use both qualification and certification words.

Certification stands for the process of ensuring that a product (system or equipment) is "*approved*" for use. It is not the software by itself that is certified. Certified system software components development must be defined in the Plan for Software Aspects of Certification (*PSAC*) containing the required informations about the developed software in order to determine "*whether an applicant is proposing software life cycle that is commensurate with the rigor required for the level of software begin developed*" [114].

Qualification is used in the context of tools when these ones are expected to be used for the "*elimination, reduction or automation*" of certification activities "*without its [the tool] output being verified*" [114].

2.1.2 *DO-178B*– Software Considerations in Airborne Systems and Equipment Certification

DO-178B is the reference certification document for most current airborne systems and equipment certification. It provides a set of objectives to be fulfilled in order to ensure the system development safety operations. *DO-178B* defines 5 level of systems to be certified, these levels reveal the criticality of the system with regards to on-board safety. These levels are referred to as Design Assurance Levels (*DAL*) and are classified from A to E, A being the most critical one that applies to safety-critical systems like fly-by-wire, landing gears or doors opening systems; and E being the less critical one that applies to non safety related systems like entertainment systems for example. Fulfillment of each objective is dependent of the *DAL* that must be reached by the certification applicant. *DAL* levels have first been introduced and standardized in the *Aerospace Recommended Practice Guidelines For Development Of Civil Aircraft and Systems* (*ARP4754*).

In addition to the *DAL*, *DO-178B* defines the degree of independence between activities of the development (if needed) that must be applied in order for the objective to be achieved. Objective fulfilment with independence means that requirements, development and verification activities must be produced by different mean and persons. The independence may also be related to the developer conducting the mean itself.

2.1.3 *DO-178C*– Software Considerations in Airborne Systems and Equipment Certification

The work done for the release of *DO-178C* is a strong revision effort of the previous *DO-178B* version to take into account the clarifications provided throughout the years in the Frequently Asked Questions (*FAQ*) and the Certification Authority Software Team (*CAST*) papers released after the *DO-178B* and to handle new technologies like object oriented programming, model driven engineering and formal methods.

DO-330– Tool qualification according do *DO-178C*

A major shift between *DO-178B* and *DO-178C* is on the use of tools for safety-critical systems development. Throughout the years, efforts have been done on the development of domain-specific and objective-centered tools used as an assistance in safety-critical systems developments. They are developed with the goal of obtaining certification credits for their use. This leads to the natural question of the qualification of these tools as they are likely to impact the final software quality.

DO-178C defines three tool qualification criterion used to categorize the tools according to their purpose and potential impact. These ones are:

- **Criteria 1:** A tool whose output is part of the airborne software and thus could insert an error.
- **Criteria 2:** A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
  1. Verification process(es) other than that automated by the tool, or
  2. Development process(es) that could have an impact on the airborne software.
- **Criteria 3:** A tool that, within the scope of its intended use, could fail to detect an error.

Criteria 1 tools are tools that are producing part of the embedded software. Criteria 2 tools aims at automating the verification of embedded software whose result is used in order to leverage other verification processes. An example of criteria 2 tool, would be a tool allowing to automate some verifications on the source code of the developed embedded system and based on this verification and the confidence that is put on the tool to avoid to insert runtime verification mechanisms. Criteria 3 tools are simple verification

tools used only for verification purpose whose results are not used for the leveraging of other certification activities.

The previous release from 1992, *DO-178B*, defined two kind of tools: development tool (criteria 1) and verification tools (criteria 3). In *DO-178C* we see that a third kind of tool has been added: Criteria 2 tools. These tools are of specific interest as their results might be used to leverage other verification or development activities for which certification should be provided. This impact on other activities leads them to the necessity to provide greater confidence for criteria 2 tools than for criteria 3.

According to the chosen criteria, tools qualification activities will differ. These activities are classified using Tool Qualification Level (*TQL*) defined in *DO-178C* and matches to a pair of tool criteria and software Design Assurance Level (*DAL*). According to the chosen *TQL*, a specific set of objectives have to be fulfilled to ensure the tool qualification. "*For a tool that can introduce an error in the outputs of a tool, the applicable* TQL *is the same as the tool being developed. For a tool that cannot introduce an error in the output of the tool, but may fail to detect an error in the tool life cycle data, the applicable* TQL *is* TQL-5" [114]. The lower the *TQL* is, the stronger is the set of objectives to match. *TQL* ranges from *TQL*-1 to *TQL*-5. The relations between *TQL* and embedded software *DAL* is provided in Table 2.1.

DAL	Criteria		
	1	2	3
A	*TQL* 1	*TQL* 4	*TQL* 5
B	*TQL* 2	*TQL* 4	*TQL* 5
C	*TQL* 3	*TQL* 5	*TQL* 5
D	*TQL* 4	*TQL* 5	*TQL* 5

Table 2.1: *DAL* and *TQL* relations

From the definition of the criterion and the *TQL*, one can refer to a specific document focused on the certification of tools. This 'companion' document to *DO-178C* is the *DO-330* [115] (Software Tool Qualification Considerations Companion). This document is, in fact, a rewriting of *DO-178* focusing on tools used for the development of safety critical systems.

Technology specific approaches in *DO-178C*

Since *DO-178B* release, software design, development and verification techniques have evolved leading to the emergence and adoption of new technologies and approaches by industrial users. With this adoption came the necessity to ensure the safety of the use of these techniques in safety-critical software development. Considerations regarding the use of these specific technologies during the software development have been detailed in external 'supplement' documents: *DO-331* [6] on model-based development and verification; *DO-332* [7] on object-oriented technologies and related techniques; and *DO-333* [8] on formal methods. The choice to rely on external documents to deal with the use of these approaches instead of integrating it directly to *DO-178C* was more reasonable as it eases the transition between *DO-178B* and *DO-178C* whilst allowing the use of alternative techniques for the certification of systems.

Technology-specific 'supplement' documents provides the modification to objectives, activities, explanatory text, and software life cycle data that can be applied to *DO-178C* when technology-specific development and verification is used. Those documents explain in which context and for which activities linked to certification, technology-specific approaches can be used instead of classical approaches.

2.1.4 *DO-330*– Software Tool Qualification Considerations Companion

In the *DO-330* document, guidelines regarding the certification data that must be produced for tools qualification are provided. Data are to be produced according to the level of criticality of the activity the tool is meant to replace. Indeed, tools ensuring coding standard respect or aiming at limiting the use of tests does not have the same potential impact on the final produced software.

The *DO-330* document is meant to be domain-independent and thus must be applicable on tools used in any domain where qualification is needed and then not necessarily on *DO-178* related systems. Indeed, other domains like space or automotive seems to agree that *DO-330* is also sensible in their context. Tools are used in their operational environment and thus are qualified according to this environment. Thus, as soon as a qualified tool environments changes, qualification must be re-considered.

Tool qualification relies on, among other data, providing the Tool Requirements (*TR*) and the Tool Operation Requirements (*TOR*).

- *TR* "*describe all the tool functionality*", this encompasses among others: the description of the tool functions and features (modes of operation); documentation (user instructions, installation instructions, error messages, ...); and informations about failure modes, abnormal operations, inconsistent inputs response.
- *TOR* "*define the tool's functionality and interface from a software life cycle process perspective*" [115]. One of the main aspects for a criteria 2 or criteria 3 tool qualification is to provide evidences on the respect of the *TOR* by the tool implementation.

A tool must be validated according to the tool operational verification and validation process. Both have the purpose of ensuring that the tool complies with its user requirements.

Here we roughly provided an overview of the tool qualification according to *DO-330*. Tool qualification requires many additional steps that are not mentioned here as it is not our purpose to fully detail the qualification process. Further details on the use and benefits of the *DO-330* document can be found in Frederic Pothon's *DO-330* focused document [126].

2.1.5 *DO-331*– Model-Based Development and Verification Supplement

The use of models has become a typical approach for the specification and development of complex systems. These ones provide the benefits of abstraction and formalisation. A model is defined in *DO-331* document as "*an abstract representation of a set of software aspects of a system that is used to support the software development process or the software verification process*".

This 'supplement' document provides informations on the use of models in the development process of safety-critical systems. According to *DO-331* document, a model must have the following characteristics:

a The model is completely described using an explicitly identified modeling notation.

b The modeling notation has a precise grammar (also called "syntax") and meaning (also called "semantics"). The modeling notation may be graphical and/or textual.

c The model contains software requirements and/or software architecture definition.

d The model is of a form and type that are used to direct analysis or behavioral evaluation as supported by the software development process or the software verification process.

If a model complies with the previous characteristics, one may benefit from its use for: "*Providing unambiguous expression of requirements and architecture; Supporting the use of automated code generation; Supporting the use of automated test generation; Supporting the use of analysis tools for verification of requirements and architecture; Supporting the use of simulation for partial verification of requirements, architecture, and/or Executable Object Code.*"[6]

Models can be well defined and formalised, this makes them interesting artifacts for the specification, development and verification of safety-critical embedded systems. *DO-331* can also be used for the development of tools used in the development of such systems and thus benefit to the qualification activities.

For example, a model containing system requirements can be refined until it is expressed as executable code or as test cases to be checked on the system code; model can be fed to ACG tools in order to automatically generate code or configuration files; it can also be used as a configuration file of the automatic code

generator itself; if a model can be simulated, its simulation results can be used as oracles for the verification of test cases execution on the final object code; metamodels can be used to model languages in the *TOR* and *TR* documents. This last example, is one of the purpose of the experiments conducted in this PhD.

2.1.6 *DO-332*– Object-Oriented Technology and Related Techniques Supplement

Object-oriented technologies are nowadays widely used in software development. This use for the development of safety-critical systems has increased in the last years. As for any other programming paradigm used in this context, safety and integrity of development techniques need to be ensured.

DO-332 document "*provides guidance for the production of software using object-oriented technologies and related techniques for system and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements*".

Object-oriented techniques and technologies introduces a set of features and potential issues that does not exists in traditional embedded systems development approaches. *DO-332* defines object-oriented: basic concepts – classes and object, types and type safety, Liskov substitution principle, hierarchical encapsulation, polymorphism, function passing and closures and method dispatch – ; and key concepts – inheritance (and related subtyping), parametric polymorphism, overloading (ad hoc polymorphism), type conversion, exception specification and handling, dynamic memory management, object pooling, activation frame, manual and automatic heap management and virtualization techniques. This supplement provides supporting informations and advices on the use of the previously cited concepts such as: ensure type conversion uses are safe (for example downcasting my cause problems), ensure that overloading is not confusing. Regarding traceability the supplement provides traceability objectives that are adapted to object-oriented architecture such as if subtyping is used, traceability has to be done to the sub-classes in order to, for example, strengthen the traceability by ensuring that traced element are not overloaded.

2.1.7 *DO-333*– Formal Methods Supplement

DO-333 deals with the use of a specific family of formal methods in the context of safety-critical software systems: software-related formal methods. These methods are defined in this document as "*mathematically based techniques for the specification, development, and verification of software aspects of digital systems*"

The definition of this formal methods 'supplement' document has been supported by industrial actors in the field of safety-critical systems like the *Airbus Group* or *Rockwell Collins*. These actors have worked during the last decades with these approaches brought by computer science research. Improvements in the usability of formal verification techniques and methods has motivated their use and is providing "*the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design*".

DO-333 document identifies applicability of formal methods in order to replace traditional (*DO-178C*) certification activities or objectives. Related activities are identified as "*modeling and analysis*". According to *DO-333* document, it is the combination of formal modeling and formal analysis that is producing a formal method. A formal model is defined as a model that "*should have an unambiguous, mathematically defined syntax and semantics*", a formal analysis is defines as such "*if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true.*"

Several concrete applications of formal methods in the certification of safety-critical software systems have been conducted. The *Airbus Group* applications as presented by Souyris in 2009 [140], Bedin França in 2011 [70] or Moy in 2013 [111] or the *Rockwell Collins* ones by Cofer in 2014 [43] relates some of the success stories in this field. An extensive discussion on advances regarding software certification is provided in a 2013 *Dagstuhl* report [44].

DO-333 can also be used for the development of tools used in development of safety critical systems. Assisted or automated proof can be used to assess consistency and completeness of requirements in *TOR* and *TR*, to ensure correctness of tools with regards to *TOR* and *TR*. This is one of the purpose of the experiments conducted in the PhD.

2.2 The need for domain specific languages

Software development relies on programming languages. Embedded software systems are nowadays mostly developed using general purpose programming languages like C or Ada. These are selected for their expressiveness and efficiency (regarding both memory and computing power use). These properties are also considered as a problem when developing applications for which a high level of confidence is required as their expressiveness (the variety and number of available code constructs) and efficiency (obtained by hiding complex memory management operations) makes the verification of their correct operation more complex.

While embedded software systems became more complex, the necessity for their specification and accurate development arose. Tackling this problem is often done by relying on DSL whose primary purpose is to provide an higher level more focused view of the developed systems and automated code generators that bridge the gap with programming languages. High level development of software systems allows to focus on the functional part of the software development while abstracting from programming languages complexity. These more abstract description are usually called models even if they can be quite similar to programs. DSL are then named DSML. We will rely on this wording there after. We detail the DSML technological viewpoint through the description of Model Driven Engineering (MDE) in Section 3.2.

2.2.1 From DSML to embeddable software through automatic code generation

While a DSML may provide an abstract view of the software system, they cannot be usually directly used as embedded software. It is thus required to translate them to common programming languages used in embedded software. This phase is related to the compilation used for translating classical low level programming language to executable machine code. Compilation of high level conception language to embeddable programming languages is done using automatic code generation tools.

ACG advantages and drawbacks

Since a few decades, ACG tools are widely used in industrial applications. Their interest for software development is multiple:

- Code production efficiency: Using ACG makes it possible to produce code from a model by relying on the use of the tool. A modification of the input model can automatically be impacted on the generated code. During the development process, code is tested at various level and bugs or flaws are detected. If they are reported as being present at the design level, these errors can be corrected and then the ACG allows to update them instantly. The efficiency of adding new features is also improved as the time needed for their development is lowered.
- Code quality improvement: An ACG is producing code according to code production rules. These rules are developed in order to respect standard coding rules making the generated code cleaner and respectful regarding standards. The automated production of code also normalises the generated code structure and content and thus make it easier to verify.
- Code traceability: Code generation rules allows to match sections of generated code with DSML elements. Each generated element must thus be traceable to a DSML construct. As a model is functionality related, its elements are more easily linked to the software requirements. Through the traceability mechanism, generated code constructs traceability to requirements is by consequence eased. This is a huge advantage as such a traceability is required by certification/qualification standards.

While ACG are helpful tools for the development of embedded systems, they are still developed by human beings and thus their reliability must be demonstrated. In it indeed impossible to rely on the code produced by using an ACG if the ACG itself is not reliable. ACG are complex pieces of software and thus they need to be reliable, usable and useful. If not, the generated code must be verified.

ACG verification

Compilers verification can be done using multiple approaches [55]. The same ones can be used for the verification of ACG as their purpose, use and structure are very close:

- Traditional approach: Compiler verification is done in most cases by testing and proofreading. While having the obvious disadvantage of not being exhaustive, these techniques are widely used and are considered as being acceptable to some extend in industrial applications. Such traditional verification is done by first proofreading each translation rule and then by verifying each one with multiple test cases with a high coverage like MCDC. From these activities, one can get a light assurance of the ACG correct operation.

- Proved development: At the opposite side of the testing/review use for the verification of an ACG is the proved development approach providing the proof of correction of the ACG. By relying on proven development the verification is done once and for all as there is no need for verification while using the tool. One of the earliest example of proved development was provided by Milner et al. [109]. The current state of the art is the work done by Leroy et al. [26, 100, 101] around the CompCert project. These works provides a strong foundation on the feasibility of the approach and its potential applications [70, 140]. Despite the ground-breaking nature of the CompCert work, the approach remains of difficult access for an industrial diffusion because of its technological and theoretical complexity. The AbsInt *SME*, well known for developing the aiT Worth Case Execution Time (WCET) analyser and industrializing the Astrée static analyser targets to industrialise soon CompCert.

- Translation validation: During the SACRES and SafeAir I and II research projects, Pnueli proposed not to rely on the verification of the compiler itself but on the systematic verification of the generated elements. This approach referred to as *Translation Validation* (TV) [124], has the advantage of not verifying the various steps composing the ACG that is complex but has the drawback of necessitating the verification at each use of the ACG. TV have shown its applicability [95, 159] but suffers from the fact that it is very complex to automatically verify the ACG without the knowledge of the translation (compilation, code generation) semantics. It is easier to achieve with the knowledge of the translation semantics and of a mean to analyse the generated code to prove its semantics matching to the translated input semantics. CompCert relies on TV for the hardest parts like graph coloring for register allocations[129].

2.3 Our local research contributions to the ACG development field

In the past decades, new approaches regarding the definition and verification of ACG have been investigated in the ACADIE team where I conducted my PhD. These investigations have been done in cooperative projects like TOPCASED, ES_PASS, GeneAuto, OPEES, QuarteFT, Hi-MoCo and Projet-P, openETCS, CESAR or Spacify. Most of these are related to the integration of MDE and/or formal methods in the development and/or verification of the ACG in collaboration with industrial partners and qualification experts. We will detail here some research projects the ACADIE team has been participating on.

2.3.1 The GeneAuto ACG example

GeneAuto[1] is an open code generator project for transforming a set of high-level graphical modelling languages to selected common textual programming languages (see [28, 76, 148, 149] that describe the evolution of the toolset in the last 6 years). It currently supports subsets of Simulink, Stateflow and Scicos as input and C and Ada language as output. It is intended to be used and certified for critical

[1] http://www.geneauto.org/

embedded systems. In that purpose, its design follows a clear modular MDE approach allowing to independently verify different transformation phases and build easily variants of the toolset depending on the end user requirements.

GeneAuto architecture

Figure 2.2 provides an overview of the ACG architecture. After the initial importing step transformations are carried out as a sequence of refinements of intermediate models. Altogether there are about 50-60 transformation passes in the tool depending on the configuration. Some of them are rather small and simple structure preserving transformations (such as the Preprocessing), but others are rather complex or change significantly the model structure (for example the Code Model Generation). For practical purposes, collections of transformations are combined into independent executables called elementary tools. An elementary tool reads its input model from a file and writes the output model to another file. The ACG uses two intermediate representations during the whole transformation process: the *GASystemModel* that is a generic representation of the possible input model formats and the *GACodeModel* that is a generic representation of the possible output code formats. The *GASystemModel* contains design informations, the *System* term was not a judicious choice leading to potential misleading regarding the content of the model. This error was corrected in the Projet-P project (see the following subsection), in the following, we will use the term *DesignModel* instead of *SystemModel*.

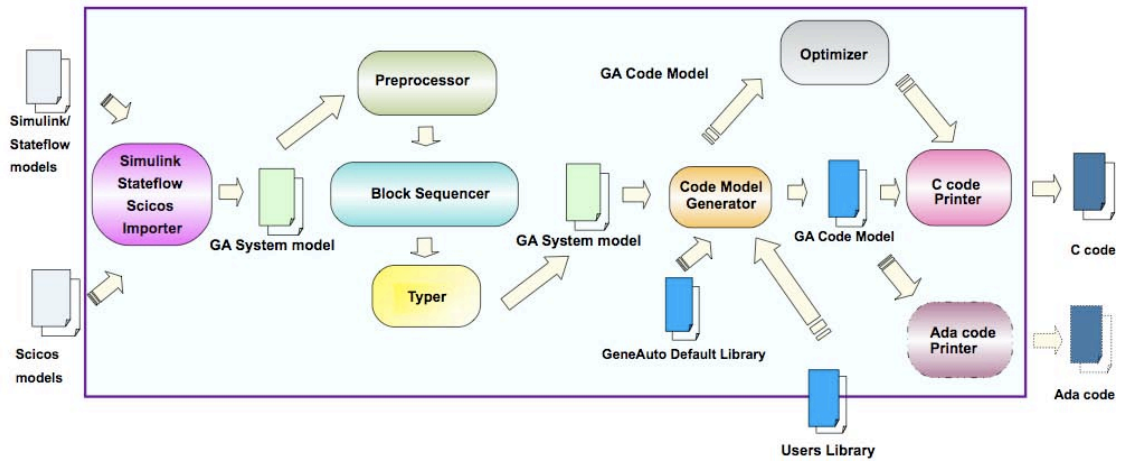


Figure 2.2: GeneAuto toolkit architecture

One of the main phases in the tool process is the translation from the *GADesignModel* to the *GA Code Model*. This translation is done according to block-specific modules of the ACG named *code backends*. They provide for each possible configuration of a block the corresponding code model elements that must be generated. In the GeneAuto ACG, blocks configurations that are handled by the tool are provided in an XML file referred to as the block library. Listing 2.3 is an extract of this block library. Its content provides informations about the blocks handled by the ACG and for each block some information about its parameters. These informations are used by the ACG to define the allowed parameter names and if the parameter values must be evaluated or they are literal values (the *evaluateEML* attribute). It also contains information about the *typer* and *code backends* that correspond to modules of the ACG containing the required verification (*typer*) or code model element generation (*code backend*) operations.

```
<BlockType name="UnitDelay" type="SequentialBlock" preprocessorPriority="0">
  <parameterTypes type="gaxml:collection">
    <ParameterType name="InitialValue" evaluateEML="true" fullNormalization="true">
      <inputLanguageMappings type="gaxml:collection">
        <InputLanguageMapping language="Simulink" version="7.3" name="X0"/>
      </inputLanguageMappings>
    </ParameterType>
```

```
  </parameterTypes>
  <libraryHandler type="gaxml:object">
    <typer type="gaxml:object">
      <UnitDelayTyper />
    </typer>
    <backend type="gaxml:object">
      <UnitDelayBackend />
    </backend>
  </libraryHandler>
</BlockType>
```

Listing 2.3: GeneAuto block library exptract for UnitDelay block

GeneAuto generated code

Code generated using GeneAuto aims at being embedded and thus must comply with common coding rules applied in industry. The generated code must also be at least as efficient as a code obtained without the tool with equivalent safety guaranties as requested by certification objectives.

Experimentations done during the GeneAuto project [12] have shown that the ACG fulfils its requirements as a development tool: the code correctness has been verified; generated code size is lesser than other previously used ACG; traceability between generated code and model has been assessed as sufficient for an industrial use; code execution has been evaluated as having similar performances as reference code either generated with industrial dedicated tools or handwritten with the usual coding process.

GeneAuto added value

The specification for most of the elementary tools in GeneAuto has been written in the English language and simple UML diagrams, with a notable exception of the Block Sequencer tool that has been specified and implemented using the Coq proof assistant [77]. This experiment has been pushed up to the certification activities with a very positive result [78].

The GeneAuto ACG is not anymore under active development but its support is still active. The tool is currently in use by several industrial end users and is freely available[2] and open source. On a research point of view, the tool is now mostly used as a sandbox for experimentation on automatic code generation and relations with formal verification. We will present some of these experiments in this document.

2.3.2 The Projet-P/Hi-MoCo approach

Results gathered throughout the GeneAuto project provided good informations on the feasibility of the development of a qualifiable code generator by relying on an MDE-based approach. It was one of the purpose of the Hi-MoCo and Projet-P projects to carry on these results and use them on a newer ACG development. The Projet-P is dedicated to:

a) help industrial partners in the deployment of MDE for the development of real time embedded-critical systems

b) contribute to interoperability initiatives and cooperations like CESAR[3], INTERESTED[4] and OPEES[5] for the French and European strengthening of their tooling ecosystem. Interoperability must be achieved through the pivot *P* formalism as an intermediate format for code generation and properties verification. This formalism is derived currently from the system and code Models.

c) position Projet-P *SME*'s to become a prominent international actors of the automatic multi-model code generation stage

[2] http://www.geneauto.org
[3] http://www.cesarproject.eu/
[4] http://cordis.europa.eu/project/rcn/85281_en.html
[5] http://www.opees.org/

By contributing to the creation of an open code generation toolset for the embedded world modeling languages, Projet-P aims at the creation of an open software product line with the same impact factor as GCC, the "*GNU Compiler Collection*"[6] has for programming languages, with an additional qualification kit.

Projet-P ACG architecture

The Projet-P ACG architecture is derived from on the GeneAuto ACG one. The *P* pivot formalism is a revised version of the GeneAuto design and code model formalisms.

Contrary to GeneAuto, Projet-P ACG was planned to be able to import real-time informations through a subset of MARTE[7] models and architecture informations through a subset of AADL[8] models along with the GeneAuto original inputs system design models.

In our work we choose to focus on automatic code generation from design models. We thus concentrated our efforts on the direct evolutions of the GeneAuto ACG. In both ACG-s, the design models are imported and parsed to get an internal representation of the imported model (conform to the design part of the ACG pivot format).

An overview of the Projet-P ACG internal architecture regarding the design model code generation is provided in Figure 2.4. While in GeneAuto block sequencing and typing of the input model is done in the ACG tools, in the Projet-P ACG, these informations are directly extracted from the design model platform. Typing is still checked internally in order to ensure the use of only the restricted allowed blocks configurations. Relying on informations extracted directly from the design tool first allows to avoid discrepancies between the input model expected sequencing and typing, and the one computed by the ACG and second to simplify the verification work related to the sequencing and a part of the typing activities.

As in the GeneAuto ACG, the code model version of the *P* formalism is produced according to the *P* formalism design model and informations about the models elements are gathered in a block library. Such block library (depicted in Figure 2.4 as the *BL Struct* and *BL Sem* elements) are block-specific modules written according to the blocks specification. As in the GeneAuto ACG, the block library contains informations about the block interfaces and parameters (the *BL XMI* elements in Figure 2.4). These informations are coded as Ada code modules used directly in the ACG. An extract of the block library is provided in Listing 2.5, for two simple blocks: *MinMax* and *Delay* that will be used and detailed in this document.

Projet-P added value

The Projet-P ACG is currently under active development and ready to conduct qualification when it will be used for a real industrial project. A first version of a derived product called QGen has been released by *AdaCore*[9], one of the leading partner of the project. It thus may become the first open source ACG for embedded critical systems development to achieve such a level of assurance and reliability. This open source approach is supposed to provide long term support and high quality of both the tool and the generated code.

2.3.3 Other research contributions to the ACG development field

Many other research contributions exists regarding automatic code generation. We limit ourselves to the ones dealing with the same level of quality as previously presented: safety-critical applications. Some provides insights on the issues related to the approach [116, 142, 155]; others focuses on specific source formalisms for the automatic code generation such as RSML [155], Event-B [108], Simulink [18] or AADL [97]; and finally many focuses on the assurance of correctness [59, 72, 125, 132, 151].

[6] http://gcc.gnu/org
[7] http://www.omgmarte.org/
[8] http://www.aadl.info
[9] http://www.adacore.com/qgen_demo

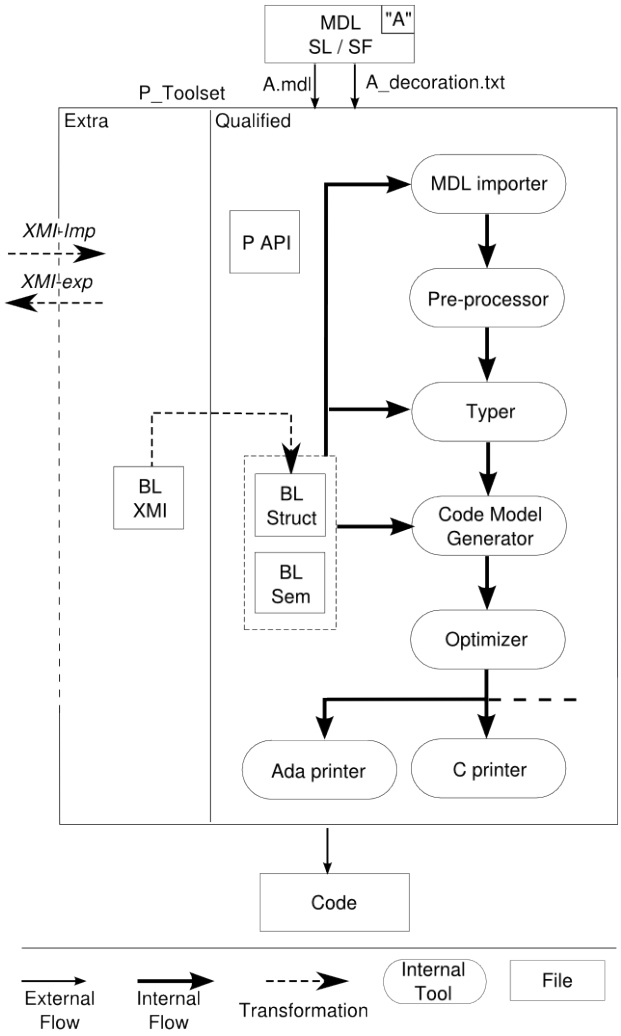


Figure 2.4: PROJET-P toolkit architecture

2.4 Synthesis

We have provided here a synthesis on the use of DSML for the design and development of embedded safety-critical software activities. Safety is ensured by the respect of certification standards. Recent versions of these standards now handle the use of new technologies and techniques such as models, object oriented technologies or formal methods for the assistance in these systems development.

Tools are increasingly used in the development of such software systems as they have shown their ability to be used as an assistance and as a safer mean to achieve systems development. This is at the condition that their development is as constrained as the system's development they are supposed to assist. In this context, ACG are amongst the most used tools and the tools whose qualification is the most challenging.

In the following, we will focus on the specification languages with a special emphasis on highly variable languages. We will define these elements and we will provide state of the art informations about both definition and verification of languages through languages factories. Languages variability will finally be analysed.

```
pragma Style_Checks (Off);
separate (HiMoCo.Block_Library_Manager)
procedure Load_Simulink_Lib (Self : in out Manager) is
  MinMax : constant Supported_Block := Get_Block_Type ("MinMax");
  MinMax_Map : Block_Parameter_Maps.Map;
  UnitDelay : constant Supported_Block := Get_Block_Type ("UnitDelay");
  UnitDelay_Map : Block_Parameter_Maps.Map;
begin
  if MinMax /= Unknown then
    Self.Simulink_Data (MinMax) := Block_Data'(Meta =>
                                 new String'("ElementaryBlock"));
    MinMax_Map.Insert (new String'("Function"), (new String'("Function"),
                                                 False, False, new String'("")));
    Self.Simulink_Map (MinMax) := MinMax_Map;
  end if;
  if UnitDelay /= Unknown then
    Self.Simulink_Data (UnitDelay) := Block_Data'(Meta =>
                                                  new String'("ElementaryBlock"));
    UnitDelay_Map.Insert (new String'("X0"), (new String'("InitialValue"),
                                              True, True, new String'("")));
    UnitDelay_Map.Insert (new String'("InitialCondition"), (new String'("InitialValue"),
                          True, True, new String'("")));
    Self.Simulink_Map (UnitDelay) := UnitDelay_Map;
  end if;
end Load_Simulink_Lib;
```

Listing 2.5: PROJET-P/HI-MOCO block library module as an Ada source code file

3

Languages formal specification - State of the art

The specification of languages has always been a core activity in software engineering. Whereas languages paradigms are not numerous, the number of programming and modeling languages created since the 70's span of thousands and is ever increasing[1]. This increase is now mostly due to the advent of domain specific languages having the advantage of conciseness with opposition to more general ones having the advantage of expressiveness. In both cases, language definition must be done methodologically and purposefully to ensure the adoption and usefulness of the language being developed.

In this section, we will first go through some definitions related to the formalisation of languages, their variability and their use for specification purposes. We will detail elements regarding language factories and model based approaches for the specification of languages and their verification through formal methods. Variability of programming languages will then be discussed. We will finally focus on a more domain specific family of languages which are the key use case in this PhD: dataflow languages expressed as block diagrams, their specificities and expose the challenges regarding their formal specification.

3.1 Preliminary definitions

3.1.1 Language

General definition

The term *Language* is defined in the Oxford dictionary as

> The method of human communication, either spoken or written, consisting of the use of words in a structured and conventional way.

This definition, focused on the human languages, has a computer-oriented variant definition (from the same source):

> A system of symbols and rules for writing programs or algorithms.

We highlight two important points in this definition: *a*) the notions of symbols and rules, referring to how the language is correctly expressed and structured; and *b*) the notions of programs and algorithms, referring to the language use and meaning.

Program and algorithm are most of the time written in order to represent a sequence of actions to be executed in a specific order and applying to a specific set of elements. The purpose and structure of the available action and element is language dependent. The ability to be executed is inherent to programs. In the following, if not specified otherwise, the term *language* will refer to executable languages.

[1] http://en.wikipedia.org/wiki/List_of_programming_languages_by_type

Technical definition

State of the art on language definition [144], split the definition of languages in four parts: concrete syntax, abstract syntax, static semantics and dynamic semantics:

- **Concrete syntax**: The human readable version of the language. This can either be a textual, graphical, audio or any medium representation allowing for the transcription of the language. This is related to the notions of symbols and rules restricting symbols composition. These rules ensure the concrete syntax structural correction. Concrete languages are most of the time used by humans to interact with computers. They are also used sometimes to store structured informations or exchange informations between computers.
- **Abstract syntax**: The machine usable version of the language. Abstract Syntax Trees (AST) is a traditional formalism ensuring its representation. AST are derived from textual parsing (or any mean of concrete syntax analysis like looking, hearing, ...) languages expressed using a concrete syntax. AST conforms to a specification provided in a tree-shaped data structure where each node is a specific structure. Structures holding abstract syntax are not necessarily tree structures, more general ones can be used like graphs. Metamodels are an example of graphs structures allowing for the storage of structurally more complex abstract syntaxes of languages. Abstract syntaxes are representations of the data carried by artifacts in the concrete syntax.
- **Static semantics**: A set of rules providing restrictions on syntactically correct programs in order to define semantically correct programs. Most of the time these constraints are not easy to (or even impossible to) enforce using only the concrete or abstract syntaxes of the language. Static semantics is impacting both the structure of the language and its potential uses. For example, static semantics verification allows to check the correct typing of the language constructs, or to verify whether or not all used artifacts are defined.
- **Dynamic semantics**: Dynamic semantics is meant to describe the interpretation of the language instances as an action (or event) driven state transition system and how to execute (attach a meaning) to syntactically correct language constructs. It is usually done by either expressing this meaning according to previously defined languages as for denotational or translational semantics definition; by writing ad-hoc interpreters as for operational semantics definition; or by providing pre/post-conditions as for axiomatic semantics.

In the following we will refer to these parts as the building blocks of a language. For the definition of a language, it is mandatory to first define its purpose to determine which building blocks need to be implemented. If the language aims at being usable by human beings then it is highly recommended to define concrete syntaxes. Indeed, manipulation of language instances in order to extract informations or transform the language are more conveniently done through an abstract syntax. Any language aiming at being used correctly (according to a set of rules like natural language grammar or spelling) should have a static semantics providing a set of correctness rules. Finally, if the language constructs aims at being executed, a dynamic semantics should be provided. Dynamic semantics description process is usually based on one of the following formal semantics approaches defined in [157]:

- **Denotational semantics**: It is concerned with giving a mathematical *models* to languages. The meaning for languages constructs is defined abstractly as elements of some suitable mathematical structure (i.e. translational semantics that translates the language to another language that has a more mathematical nature like "λ-calculus, fix point theory, ...).
- **Operational semantics**: The meaning for languages constructs is defined in terms of the steps of computation they can take during the their execution.
- **Axiomatic semantics**: The meaning for language constructs is defined indirectly via the axioms and rules of some logic of the described language constructs behavior.

These approaches have the same purposes. If several ones are used to give the semantics of the same language, their consistency must be ensured.

3.1.2 Formal

Formal is defined in the Oxford dictionary as

> Having a conventionally recognized form, structure, or set of rules.

This definition is not totally accurate in our context as what is considered as conventional in this definition should be defined according to methodology and proven characteristics. Any other definition extracted from the Merriam-Webster dictionary adds the missing methodological aspect:

> Characterized by punctilious respect for form : methodical.

3.1.3 Specification

Specification is defined in the Oxford dictionary as

> An act of identifying something precisely or of stating a precise requirement

The important information in this definition is the notion of precision. A specification must express all the mandatory informations related to the specified element allowing to identify the correct conditions for the use of the specified element. Formal specification would ultimately be the safest way to achieve such precise description.

A specification is often provided as a set of requirements expressed using natural languages or formal languages. Formal languages could be data structuring languages or formally defined expression languages. Lamsweerde [96] provides a definition for a formal specification:

> A specification is *formal* if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rule for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory). The latter component provides the basis for automated analysis of the specification.

According to our previous definitions of a language, the "grammatical well-formedness" referred here is related to concrete and abstract syntaxes as long as static semantics. Interpretation of sentences is related to the semantics provided for the specification. Rules for inferring informations from the specification are additional means to extract informations based on the semantics provided for the specification. These rules are more related to language transformations or language use according to its axiomatic semantics that will be detailed further in this document.

3.1.4 Variable

Variable is defined in the Oxford dictionary as

> Not consistent or having a fixed pattern; liable to change

This definition, while being general, has the interest of providing the notions of *not fixed pattern* that is highly related to computer science terminology. A *variable* element is characterized as being able to change according to some criteria. As an extension of the variable nature of an element, we will also refer to the *variability* of an element as the causes and effects of its allowed *variations*.

3.1.5 Highly variable language

A *highly variable language* is a language for which building blocks have a structure and/or a semantics that can vary depending on their configuration and/or context of use. A language variability is qualified as high if the number of its structural and semantics variants is important.

3.2 Model Driven Engineering

Model Driven Engineering (MDE) settles on the established fact that systems are of increasing complexity. Modeling have been found as a solution to complex system design and development by providing layers of abstraction for model analysis and tooling definition for model manipulation.

3.2.1 Definition

The central concept behind MDE is the one of *model*: " *An object 'A' is a model of an object 'B' for an observer 'C', if the observer can use 'A' to answer questions that interest him about 'B'* "[105]. The key purpose of models is the abstraction of objects/concepts/systems in order to get informations and answers regarding its properties and characteristics. Models hold a structured abstraction of the modeled elements.

3.2.2 Use

MDE aims at providing means and tools for the manipulation of models. Such tools will allow to analyse models in order to assess properties about them, extract informations for specific uses like documentation or testing, transform models into other models or even software/source code to make bridges between abstraction domains.

The needs for models and their specification led to the requirement to formalise the models themselves to secure the modeling activities. Formalisation of models have been proposed and expressed as a model being able to define itself: a metamodel. There is no commonly accepted definition for it. In our understanding of metamodel we would define it as : "*a precise definition of the constructs and rules needed for creating models*" [2]. Metamodeling is close to the concept of metamathematics [90, 156] with which it is possible to define mathematics using mathematics (i.e. logic, general algebra, ...).

3.2.3 State of the art frameworks and languages

Standardisation

Technology standardisation consortium like the Object Management Group (OMG) [3] led the way to the definition of modeling standards helping on the adoption of the MDE approach. They provide some of the most well known and used modeling standards like:

- The Unified Modeling Language (UML) providing "systems architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems" [120].
- The Meta Object Facility (MOF) providing "means for metamodel definition [... and] core capabilities for model management in general" [117]
- The Object Constraint Language (OCL) [118] is a constraint language that can be used to express constraints on models conforming to one of the two previous specifications.
- The Query View Transformation (QVT) [119] providing the specification for transformations languages that can apply on models conforming to MOF.

Standardisation of languages and approaches is important and valuable for industrial users as it provides additional confidence. Whereas standardisation of a language is not its formalisation, it is a first big step towards its safe definition. It is notable that in the case of the OCL the standard OMG document does provides a formalisation of the language based on class diagram and OCL itself that covers the entire set of the language constructs.

[2] Johannes Ernst, www.metamodel.com
[3] http://www.omg.org

Implementations

An initiative build around the Eclipse platform has seen the definition of the Ecore metamodeling language almost conforming to the Essential MOF (EMOF) standard – a subset of MOF. From this MOF implementation a large ecosystem of MDE tools dedicated to the manipulation of models and metamodels have been developed mostly relying on the Java language. We will describe here some of the tools we have used in our work:

- **Eclipse Modeling Framework (EMF)**: Provides means to create and manipulate metamodels, metamodels conforming models, and generate automatically API for model manipulation.
- **OCL tools**: Provides an implementation of the OMG OCL standard that can be used on metamodels and models expressed in Ecore.
- **XText framework**: A grammarware framework allowing to define concrete textual syntaxes and textual editors based on metamodels. It establishes a duality between language grammars and metamodels (allowing to transform one into the other). The framework is used in order to automatically generate lexers, parsers and serializers for textual languages. Other grammarware framework also exists such as EMFText[4].
- **Sirius framework**: A framework allowing to define concrete graphical syntaxes (and the related editors) based on metamodels. It is the dual of the XText framework but for graphical syntaxes.
- **ATLAS Transformation Language (ATL)**: A partial prototype implementation of the OMG QVT standard. It targets the definition of model to model transformations by defining transformation rules between metamodels elements. Transformations can then be applied on the metamodels conforming models.
- **Acceleo model to text facility**: A metamodel based framework for the definition of template-based model to text transformations conforming to the model to text (M2T) standard.

As all these tools rely on the same platform and programming language (Java), it is possible to build interactions between them. This is used for example as a way to develop model manipulation frameworks with multiple views of the model and extract scope-specific informations. We relied on this toolset for all the experiments conducted in this PhD.

Many other platforms have been defined for the same purpose such as the ones defined in the Epsilon project [5]. In this project a family of languages is built around a core language: the Epsilon Object Language (EOL). This language is *"a standalone generic model management language"* [92] but can also be used as an *"infrastructure on which task-specific languages can be built"* [92]. The EOL is built on OCL complemented with imperative constructs, and model manipulation features (creation, querying and modification). The family of languages built on top of EOL comprises a code generation oriented language: the Epsilon Generation Language (EGL); a model transformation language: the Epsilon Transformation Language (ETL); a model validation language: the Epsilon Validation Language (EVL); and a model comparison language: the Epsilon Comparison language (ECL).

Additional platform for the definition of languages and the manipulation of model can be cited such as GME[6], MetaEdit+[7], MPS[8], Software Factories[9] or Monticore[10] [94]. A comparative study of MDE tools have been done by Achilleos et Al [10]. Many work have been done on the design of DSML. Interesting ones, including their references, are provided [69, 85, 141].

[4] http://www.emftext.org/index.php/EMFText
[5] http://eclipse.org/epsilonMod/
[6] http://www.isis.vanderbilt.edu/projects/gme/
[7] http://www.metacase.com/products.html
[8] https://www.jetbrains.com/mps/
[9] http://www.softwarefactories.com
[10] http://www.monticore.org/

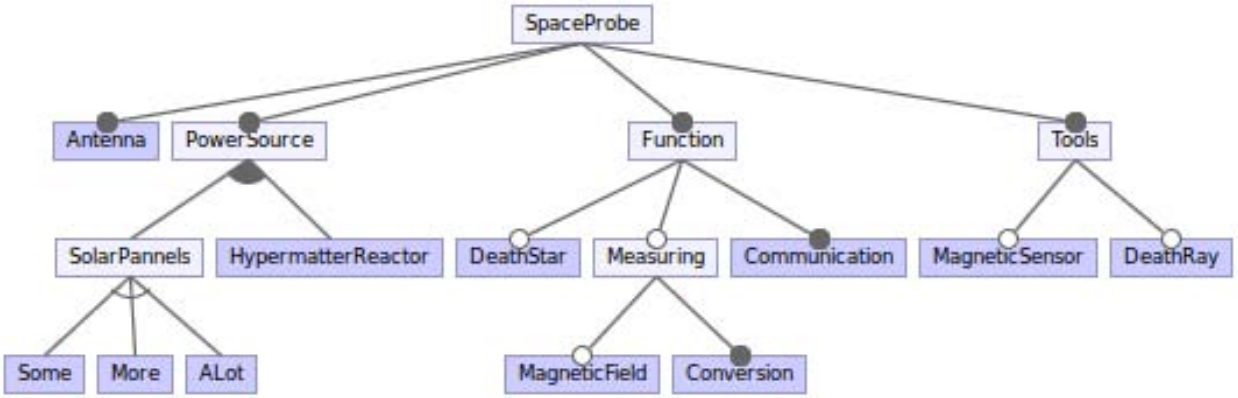


Figure 3.1: Space probe system Feature Model

Languages engineering started with grammarware approaches such as the Cornell Program Synthesizer [143] and more evolved language manipulation environment generators like the Centaur[11] [29] system.

Frameworks for languages development and transformation like Stratego[12] [102] and ASF+SDF[13] [153] provides language design capabilities. Those have respectively evolved to metaprogramming languages with automatic language edition environment generation capabilities: the Spoofax[14] [86] and the Rascal[15] [91].

3.2.4 Software product line engineering

Both the software system complexity and the demand for specialized versions of systems have always been increasing. Fulfilment of these two orthogonal requirements in the same deliverable product is difficult and have mostly been tackled by Software Product Line Engineering (SPLE) based approaches. SPLE is a model-based approach aiming at "the development of products from existing assets rather than the development of separated products one by one from scratch" [19]. It relies on the analysis of the domain under study resulting in the definition of the features of the system (referred previously as the assets). Compositions of features leads to products representing a configuration of the system under study and representing one variant (or a member of the variability domain of the model). Many methodologies have been defined in order to conduct this domain analysis [39, 145].

We will provide here informations on the SPLE approach based on the highly simplified and hypothetical example of a space probe system.

Feature composition through relations

Features represent atomic components of the studied domain. These features are organised in a Feature Model (FM). The simplified FM for the space probe system is provided in Figure 3.1. We define two kind of features: abstract and concrete features, abstract features are placeholders for the specification of other features whereas concrete features are supposed to represent system features and functionalities. In our space probe figure, abstract features are represented using light boxes, concrete ones are represented using light blue boxes.

System features are organized according to a hierarchy built by relations like:

- Relations between a parent and its child features defining a tree hierarchy of features. A feature can only have one parent feature. There are two kind of relationships: those defined between a child

[11] http://www-sop.inria.fr/croap/centaur/centaur.html
[12] http://strategoxt.org/Stratego/WebHome
[13] http://www.meta-environment.org/
[14] http://metaborg.org/spoofax/
[15] http://www.rascal-mpl.org/

feature and its parent feature among which are the *mandatory* and *optional* features and the ones applying on groups of child features of one parent feature among which are the *alternative* and *or* relationships. We detail them in the following:

  - Mandatory: A mandatory child features is present in all the products containing the parent feature. In Figure 3.1 this is represented using a black filled circle. In the space probe example, mandatory relations specifies that a *SpaceProbe* system must contain the *Antenna*, *PowerSource*, *Function* and *Tools* features.
  - Optional: An optional child features may be present in the products containing the parent feature. Optional references are represented using a white filled circle. In the space probe example, optional relations specifies that the *Tool* feature may be composed of a *MagneticSensor* or of a *DeathRay* or both, or none of them.
  - Or: Any number of the child features can be present in the products containing the parent feature. Or relations are represented on our space probe example as a filled arc between the referenced sub-features. The space probe system *PowerSource* is specified to be composed of the *solarPanels* or the *HypermatterReactor*, or both features according to an *or* relation.
  - Alternative: Only one of the child features can be present in the products containing the parent feature. In our example, alternative relations are represented using an empty arc between the referenced sub-features. The *SolarPanels* feature is divided in three alternative features: *Some*, *More*, *ALot*. If one of these features is selected for a product then the other two will be automatically deselected.

- Cross-tree relations between features defining either dependency or exclusion relations. They allow the definition of relationships between features having no direct hierarchical relations. In our space probe model, cross-tree relations will provide additional informations and constraints between features. For our space probe, we want to ensure that if the probe is made to measure magnetic fields then the magnetic sensor must be present in the probe specification. We thus add the (3.1) cross-tree constraint. Finally the *DeathStar* function of the probe needs a death ray tool that is usually powered by an hypermatter reactor or a lot of solar panels. We enforce this with the (3.2) cross-tree constraints.

$$MagneticField \Rightarrow MagneticSensor \tag{3.1}$$

$$\begin{aligned} &DeathStar \Leftrightarrow DeathRay \\ &DeathRay \Rightarrow HypermatterReactor \vee ALot \end{aligned} \tag{3.2}$$

FM and variability analysis

The FM structure is strongly focused on the analysis of the variability of the system. The previous relations between features allow to express this variability.

FM can be analysed in order, for example, to detect discrepancies: a cross tree constraint being incoherent with the relations implied by the hierarchy of features; and perform extraction of informations like the set of all products logically expressible in a FM. An extensive review of existing analyses and treatments that can be applied to FM is provided in [20], feature models edition languages and tools are detailed in [9].

For example, in our space probe model, we can extract 60 distinct products based on the relations expressed on the FM and no incoherence has been found.

FM extensions

As the modeling of the features variability of a system is of strong interest, extensions of FM have been studied in order to augment their expressiveness among which are: cardinalities [130] defined as an extension of relations to limit the number of features instances allowed by a relation; or extended features [84] allowing to define attributes on features in an extended FM.

FM as development artifact

Transformations may be applied to FM in order to extract a FM products by attaching to each feature a software artifacts like model elements or source code. If each feature specification is done by relying on models, the system specification is further carried on using models and then code generation based on models can be used for the development of the system. If feature specification is done by relying on source code, it can be directly used for system development (source code production). Generative programming approaches proposed by Czarnecki et al [50, 52, 53] focus on the latter approach.

3.3 Software formal verification

Software verification strongly depends on the nature of the language the software is based on and on the kind of properties to be verified on it. In embedded critical systems most of the source code is written using the C language. This work thus focuses on verification techniques for this language.

Classical verification of software relies mostly on tests that target the analysis of the program execution result according to some input data and expected results. Testing is widely used in industrial applications but suffers from a major weakness: its non exhaustiveness. Proof-reading is done by humans and thus is error-prone. Source code verification can also be done without relying on its real execution, for example by analysing its semantics. Such approaches are referred to as static analysis techniques.

In this section, we will first introduce some of the common approaches used for the verification of software, we will then introduce some formal methods used in this context, we will show how these methods can be used for the verification of software and finally we will introduce the Why3, and Frama-C toolset that are used in our work. We do not meant to provide a complete literature review about formal methods, interested readers can refer to [40] for a more complete overview.

3.3.1 Static analysis of source code

Static analysis of source code aims at the extraction of informations from a source code without executing it really (i.e. it is usually conducted using a kind of symbolic execution). The extracted informations are then used for the assessment of potential failures that may appear during the code execution. Various static analysis methods can be applied on source code among which are model checking, abstract interpretation and deductive verification (i.e. code interpretation through the use of Floyd-Hoare logic).

Model checking

Model checking was introduced by Clarke et al [41] and Queille et al [128]. Model checking aims at the exhaustive verification of all possible behaviors of a model according to some specified properties. If the property is proven incorrect, it usually then returns an execution trace violating the property. Model checking suffers from its exhaustiveness. It can only handle models whose state space is finite and is subject to combinatorial explosion (i.e. even simple models can lead to prohibitively big state space). It must thus be adapted in order to be applied on models with too wide or unbounded state spaces. Such adaptation can be done using various extensions to model checking like bounded model checking or predicate abstraction.

Model checking applied to the verification of source code has a limited use but bounded model checking has been used in tools like CBMC[16] for the verification of "*array bounds, pointer safety, exceptions and user-specific assertions*". Some approaches like the one described by Schlich in 2009 [134] also provides some

[16] http://www.cprover.org/cbmc/

intelligence on how model checking can be used for the checking of properties on assembly code.

Abstract interpretation

Abstract interpretation (AI) introduced and formalised by Cousot and Cousot in 1977 [49] is used in many fields related to software verification. According to verification specific goals, AI allows to statically analyse software source code and provide goal specific results. These results provide an abstraction of the source code that can be used either directly as a formal property or to assess the satisfaction of a formal property.

AI has been successfully used for the verification of many families of software issues like absence of run-time errors, floating-point computation errors, value bounding (interval analysis), worst case execution time, and many others. AI has successfully been applied on real industrial applications [21] and led to the implementation of many industrially successful tools like aiT and Astrée by AbsInt[17] for the analysis of runtime errors, the Fluctuat static analyser for floating-point computation errors or Polyspace for C,C++ and Ada runtime code analysis.

Floyd-Hoare logic and Dijkstra WP

Floyd-Hoare logic was proposed by C.A.R. Hoare [75] after Floyd works on flowcharts [68]. Floyd-Hoare logic "*provide a logical basis for proofs of the properties of a program*" [75]. This logic is centered on the use of the Hoare triple structure (3.3) that models the content of memory during the execution of a program.

$$\{\varphi\} \;\; P \;\; \{\psi\} \tag{3.3}$$

A Hoare triple holds the axiomatic semantics definition (φ and ψ) of a program (P). In a Hoare triple, φ is the pre-condition, P is the program and ψ is the post-condition, such that if φ is verified prior to the execution of P and if P is proven to terminate then after the execution of P, ψ will be verified too. If P is not proven to terminate, then only a partial correctness is proven. Termination is usually proven by relying on well founded ordering like variants functions from the state of the memory to natural numbers that are proven to be strictly decreasing with the loop iterations.

Hoare defined a set of deduction rules providing an interpretation of Hoare triples in the context of programming languages. These rules specify the semantics of programming languages by defining the required pre and post-conditions for each of their constructs.

Dijkstra Weakest Pre-condition (WP) [62] calculus is widely used and have proven its usefulness in many concrete applications. WP calculus is applied in order to compute the pre-condition implied by a Hoare triple providing the program and post-condition parts. Indeed, according to the annotated language deduction rules, it is possible, if we have the knowledge of the post-condition of a piece of code, to compute its corresponding pre-condition. Such computed pre-condition is called the weakest pre-condition as any other pre-condition implying it will be a pre-condition of the whole Hoare triple. By backward applying WP calculus on sequences of code instructions it is possible to use it to extract pre-conditions on wider code constructs.

Loop constructs are specific constructs on which the WP is not easily done. Indeed, the exact WP of a loop is usually an infinite formula as it is the solution to a fixpoint problem. However, theories predict that as soon as the logic used for the expression of the annotations is sufficiently expressive then there exists an equivalent finite formula. For loop constructs it is thus needed to provide an invariant that implies the post conditions in addition to a variant allowing to prove the termination of the loop. If only the invariant is provided and the proof a success then we speak of partial correctness, if we can add a variant then it becomes a total correctness proof as we also prove the termination of the loop.

Floyd-Hoare logic and Dijkstra WP has since been used for applications such as the definition of programming languages semantics, their analysis or for the verification of programs. They are at the core of the deductive verification approach as it provides a formalism for the definition of language constructs

[17] http://www.absint.com

semantics. It is also used in modeling and analysis languages like the B language for the definition of the language semantics [33].

3.3.2 Deductive verification

Deductive verification aims at the generation of proof obligations from the analysis of a software system and its specification. The generated proof obligations are then discharged using theorem proving approaches with manual tools like proof assistants, or automatic ones like boolean satisfiability (SAT) problem solvers or Satisfiability Modulo Theories (SMT) solvers.

Proof obligations generation

Proof obligations are generated by relying on previously presented Hoare triples and extensions of Dijkstra WP calculus.

The software specification must be expressed as a functional specification for which a formal semantics exists. Such specification is most of the time provided as annotations on the code, written using a specification language like the ANSI C Specification Language (ACSL) [1] used to express annotations on C code. Similar annotation languages exists for other languages like Spark for Ada, Spec# for C# and F# or JML for Java.

From the generated proof obligations and the formalisation of their definition domain, it is thus possible to apply automated or assisted theorem proving techniques to assess the correctness of the proof obligations.

Proof assistant

In order to assess the correctness of a generated proof obligation, one can apply common mathematical axioms and theorems in order to prove (discharge) the proof obligations. Proof assistant are formal tools allowing to formalise the logical mathematical proof process through the use of a computer. By applying known axioms and previously proven theorems, one can use proof assistants to prove other theorems or discharge previously generated proof obligations.

Some well known proof assistants are Coq [48], PVS [121] or Isabelle [123]. These tools have already been used in the verification of both theoretical and concrete applications and are well known to be formal and expressive enough to tackle concrete complex problem. A significant result is the one obtain by Leroy et Al. with the development of a C compiler by relying on Coq: CompCert. Despite these tools capabilities, their use is difficult and is for know reserved to highly trained scientists in the case of real industrial systems.

SAT/SMT solvers

SAT solvers are meant to find a solution to boolean problems expressed as formula in Conjunctive Normal Form (CNF– formula is a conjunction of clauses which are disjunctions of literals). Implementations of the Davis-Putnam-Logemann-Lovelan (DPLL) procedure [56, 57] allows for the decision of the satisfiability of such formulas. If no solution can be found the problem is then considered as UNSAT.

Evolutions of the original DPLL procedures [136, 161] led to greats improvements in the SAT approaches efficiency [110]. By relying on these implementations SAT solvers manage to tackle a wide range of problems among which are fault diagnosis [137], planing in artificial intelligence [87], or cryptography [139].

While SAT solvers does allow to solve a wide range of problems, its applications are limited to boolean problems. Whereas dealing with non boolean values can be done using a SAT approach by translating it to propositional logic, it is not always appropriate to do it because of the complexity of the generated formulae.

SMT approaches were developed in order to deal with formula expressed using more expressive logics than the propositional one. These logics are defined by extending propositional logic with theories

providing data types and operations defined through axioms. Works on proof principles for these theories [15, 25] and adaptation of DPLL [146] (and their derivatives) procedures have been done in order to check formula for satisfiability according to theories content and not only according to propositional logic.

SMT solvers have a wide range of both theoretical and practical applications. Applications are encouraged by the accessibility of SMT-based methods and their efficiency. It is considered by many as a consequent breakthrough in the field of formal methods: "*The biggest advance in formal methods in last 25 years*"[18], "*Most successful academic community related to logics and verification [...] built in the last decade*"[19].

Some of the most advanced SMT solvers are CVC4[20], Yices[21], Alt-Ergo[22] or Z3[23]. A detailed list of SMT solvers can be found in the smt-lib web-site[24].

SMT solving and proof assistants

SMT solving and proof assistant approaches use for discharging proof obligations are very close. They both rely on the same logical formalisation: theories. The difference lies in the use of these theories. While SMT solvers provides efficient automated strategies relying on theories content (theorems, lemmas and axioms), proof assistants use a mostly manual approach. Proof assistants are also providing tactics (this is the term in Coq) for the application of, to some extent, automated (scripted) reasoning on proof objectives.

SMT solvers are formal tools but their results are not necessarily proven, this is why an additional confidence is needed. In order to achieve this, a cooperation between both can be done by relying on proof assistants to verify the proof script generated by the SMT solver.

3.3.3 The Why3 platform

The Why3 platform has been introduced by Bobot et al. in [27]. According to its authors, "*Why3 is [...] an environment for logical specification that targets a multitude of automated and interactive theorem provers. It provides a rich syntax based on first-order language and a highly configurable toolkit to convert specification into proof obligations*". The full documentation for this platform is available on its website [3].

Origins

Prior to the release of the Why3 platform, multiple tools were developed in the LRI team focusing on the verification of programs using multiple solutions such as SMT solving and proof assistants. First version of the platform was developed in the early 2000's[25].

The Caduceus platform [65] was focusing on the verification of C program source code via the use of multiple provers. The approach evolved during the years and was then applied to different languages such as Java with the Krakatoa platform [66] or the B method [58, 106].

The Caduceus platform evolved to the Frama-C[26] toolset that is the up-to-date version of the C verification tool. The Frama-C toolset is an extensible tool for the analysis of C code. It includes for example a WP calculus and some static analysis plugins for C code. Frama-C relies on the Why3 platform in order to tackle the verification using SMT solvers.

Why3 platform description

The Why3 platform supports two languages, Why and WhyML. The first is a language for the expression of logical specifications: theories. Theories are populated with types definitions, axioms, lemmas, predi-

[18] John Rushby, FMIS 2011
[19] FMSD special issue on SMT, 2012
[20] http://cs.nyu.edu/acsys/cvc4/
[21] http://yices.csl.sri.com/
[22] http://ergo.lri.fr/
[23] http://z3.codeplex.com/
[24] http://smt-lib.org/
[25] http://why.lri.fr
[26] http://frama-c.com

cates and verification goals. The WhyML is an extension of the Why language allowing to express program specification. Using a restricted ML-like language, one can write program in modules by relying on previously defined theories as logical specification foundations. This language extends ML-like language with annotations capabilities.

From these language instances, the platform provides a proof obligation generation mechanism targeting automatic and interactive theorem provers formats:

- the standard SMT-LIB format [16]. This is an input format for many automatic SMT solvers such as Alt-Ergo, CVC4, Z3 and many others.
- specific formats used by proof assistants like Coq, PVS or Isabelle.

Regarding WhyML programs, the tool allows for the extraction of verification conditions expressed using the Why language. Bridges to formal tools can then be used for the verification of WhyML programs through the discharging of the generated verification conditions.

Why3 platform uses and successes

The Why3 platform is used in many context for the verification of programs. The Why3 platform development team provides on their website a library of verified programs[27] showing the expressiveness power of the language and the verification capacities of the tool. As an example, we provide in Listing 3.2 the *division* program computing the Euclidean division taken from the library of verified programs. On this program pre (*requires* clause) and post (*ensures* clause) conditions are provided and the overall Hoare triple is proven correct automatically by relying on SMT solvers. It is mandatory to provide both an *invariant* clause in order to ensure the verification of the Hoare triple and a *variant* clause to ensure that the while loop finishes. This program is proven correct in .02 seconds with the Alt-Ergo SMT solver.

```
module Division
  use import int.Int
  use import ref.Refint

  let division (a b: int) : int
    requires { 0 <= a && 0 < b }
    ensures  { exists r: int. result * b + r = a && 0 <= r < b }
  =
    let q = ref 0 in
    let r = ref a in
    while !r >= b do
      invariant { !q * b + !r = a && 0 <= !r }
      variant   { !r }
      incr q;
      r -= b
    done;
    !q
end
```

Listing 3.2: The Euclidean division algorithm

Recent publications [42, 104] show the applicability of the Why3 platform in concrete, industrial applications and in reseach and industrial cooperation projects like the BWare[28] project on the discharging of B proof obligations or the Spark[29] toolset for the verification of Ada programs.

3.4 Domain analysis: languages variability

We have seen a number of formal analysis techniques used for the verification of languages. In this PhD, we focus on the analysis of languages variability. We will provide in the following a preliminary domain

[27] http://toccata.lri.fr/gallery/why3.en.html
[28] http://bware.lri.fr/index.php/BWare_project
[29] http://www.spark-2014.org/

analysis on this subject. We will first provide examples and then explain what we mean by language variability.

3.4.1 Languages variability examples

Variability in textual concrete syntax

Textual programming languages may provide syntactic variability allowing to represent the same concept with different textual representations. The most common approach to this variability is the use of overloading (or ad-hoc polymorphism) used for operators definitions for example where the call to a method (for example the sum of two integers `a` and `b` done with the call to `sum(a,b)`) can be replaced by an infix notation like `a + b`. This mechanism allows for the evolution of the notation for elements without any changes in the semantics or abstract syntax of the language. The Ada, C++ or Haskell are example of languages implementing such overloading on operators under some constraints on the parameter types. Every language that is on top of the Java virtual machine (JVM) is also an example of textual concrete syntax variability as they all rely on the same object creation and polymorphic call. However, all languages are translated to assembly code, but the XText is a good example of framework for the definition of multiple concrete textual syntaxes for the same abstract syntax.

Variability in graphical concrete syntax

Graphical languages implementations are subject to variability regarding both structure and semantics. As previously shown, graphical variability can be the result of different graphical representations associated to the same concept – for example, representing the operation visibility by either *public* or + in the UML graphical notation of class diagrams. Such graphical variability is only representation related and therefore does not impact the semantics of the language. The representation of interfaces in UML is another example of graphical variability; an interface might be displayed as an annotated class (thus displaying informations on its content) or as a circle (and thus hiding its content).

The Sirius framework is the pendent of the XText framework for graphical concrete syntaxes. As such it allows for the definition of variant graphical representations of models but with the same semantics and the same metamodel (abstract syntax). This is widely used in the industry where each domain has its own notation for the same elements. This has the huge advantage of simplifying communication without sacrificing the meaning.

Program evaluation variability

Execution of a program is done according to evaluation rules. According to these, the result of the program computation might be different.

The aspect programming technologies [89] allows to dynamically change the semantics of a program and are thus an entry point for the definition of program execution semantics variations. This variation is a dynamic variation of the semantics and not a variation of the language semantics itself.

There are many evaluation strategies for programs, the first and more common one is often referred to as *eager* or *strict* evaluation where for example function arguments are evaluated before the function call or boolean conditions are completely evaluated before the branches; other approaches like *left to right*, *right to left* or *lazy* evaluation are starting the evaluation of binary expressions or function parameters before the availability of the results.

Chosen evaluation strategy is likely to influence the language user in its way to write its programs, for example, using *lazy* evaluation boolean expressions are not completely evaluated when their overall result can be known by relying on logical simplifications (short-circuit handling of boolean operations). Evaluation strategy choice may thus impact as, for example, a lazy evaluation may not execute some parts of the code and thus remove some side effects; or termination of the program may be impossible if a complete evaluation is done.

In real time systems, evaluation of the program result is usually done according to a modeling of the time. This one can be the real physical time (absolute time) or logical time depending of the machine on which the program is evaluated. Such a variation may produce different program execution and behavior if for example the program is supposed to interact with a system whose relation to time (physical/logical) is not the same.

Semantics variability of graphical languages is the subject of many discussions and the cause for misinterpretation or miscommunication. The UML variation points are an example of potential different meanings for the same model according to its interpretation. The purpose of the fUML standard was to ensure a common understanding of a subset of this too wide modeling language by defining restrictions on the number of allowed modeling constructs and by explicitly selecting semantics variation points that were not identified in the previous general UML specification. However, it only provides a sequential interpretation whereas the language contains concurrency related aspects.

Such programs execution/evaluation variations are grouped according Models of Computation (MoC).

3.4.2 Language variability analysis

This PhD work mainly focuses on the specification of highly variable languages and on the uses that can be made of such a specification for the development of tools in the safety critical embedded systems community. In this section, we will provide a detail on what part of the language can be variable and to what extent.

Defining the capabilities of a language regarding variability is not an easy task as it implies to define for every component and member of the language its ability to vary and the impact of the variation. Instead of defining the variability for every possible language, it is more accurate to define the variability according to variability criterion and the language building block they apply on.

In this preliminary study, we identified two criteria to qualify the variability of a language: **observability** and **granularity**.

Observable variations of a language are variations of the language representations (concrete syntax) and/or the language execution (its semantics in general) that the language user can observe either during the writing or the evaluation of the model. We chose to define language variability as the variation of one of the observable source (representation, execution) whereas variation of both observable sources leads to the definition of a different language. This proposal will require further studies and discussions in the language community to provide a definitive definition.

Variations may be applied on different levels of the language. This granularity of the variations impacts on the language definition. Indeed, if the variation applies on the whole language or if it applies only on some elements the result on the language definition will not be the same. If the variation of the language impacts on some constructs of the language without impacting the remaining elements we will speak of fine-grained variations of the language and thus of a variant of the language. On the contrary, if the variation impact largely the language as for example a switch of MoC, the variation is then defining a different language. We will speak of large-grained variations of the language and thus of a different language.

In the following we analyse the potential impact of each language building block variability on the other building blocks and on the overall language according to the previously defined variability criteria. This analysis is not meant to be exhaustive but rather relies on the definition of languages and on our experience on dealing with languages definition variability.

Languages building blocks variability

Concrete syntax variability allows to map different concrete syntaxes to language constructs with the same semantics. Example of such variable concrete syntaxes is the possibility to define both textual and graphical concrete syntaxes for a language. The well known embedded systems design language Lustre allows for such a variability as it provides both graphical syntax – through the Scade tool – and its classical textual syntax. Simple concrete syntax variations examples can be found in UML class diagrams where class attributes and operations visibility is set using values: *Public*, *Protected*, *Private* and *Package*. These

can also be set using respectively: +, #, - or ∼. In MDE, it is common to use grammarware tooling such as XTEXT or EMFTEXT, that allows the definition of a textual concrete syntax using a variant of BNF grammars that can be automatically transformed to metamodels; or SIRIUS[30] allowing to associate a graphical representation for a metamodel conforming model. Through the definition of a concrete syntax variability, the language capabilities and semantics remain exactly the same, indeed if only the concrete syntax varies then the represented language elements are still the same. On the contrary, if the language semantics also varies then the represented language elements are no more the same and in this case we do not speak of languages variability but of a different language.

Abstract syntax variability allows to have different abstract representations for meaning-equivalent language constructs. This includes the ability to remove or add some constructs in the language according to the user/implementer needs. In [79], the authors state:

> [Abstract syntax variability] refers to the capability of selecting the desired language constructs for a particular product as long as the dependencies are respected.

It is also possible to define different AST for the same language as data structure choices made during the AST creation are up to the AST specifier. In the context of MDE where AST are defined using metamodels, a different AST will mean a different metamodel and thus, in this context, a different AST is likely to define an observable variant of the language. But it is still possible from various metamodels to associate the same concrete syntax and the same semantics. Thus, abstract syntax variability by itself is not enough to define a different language, or even a variant of a language.

As abstract syntax variability may change the language capabilities, there may be an impact of such variability on the overall language semantics and the way this semantics is implemented. In the same way, variations of the abstract syntax may influence the concrete syntax especially if some language constructs are added or removed. Concrete syntax and semantics preservation while having a variable abstract syntax is not variability as it only points out implementation choices having no observable impact on the language itself.

Static semantics variability allow for different means to assess the correctness of a program. It would deal for example with the respect of the scope (dynamic, lexical, ...) of the identifiers defined in a language or on the typing of the elements of a language. It might be allowed (or not) to: a) redefine identifiers after their definition in inner blocks; or b) access their inner blocks. Typing of the language might allow sub-typing, overloading, coercion or the definition of polymorphic (often referred to as generic) language elements.

As static semantics modification only impacts the language elements interpretation and the allowed constructs, it may impact the execution of the elements of the language but will not impact on the language semantics itself. These modifications are thus defining variations of the language and not a new language.

In the MDE methodology, static semantics is partly specified on the metamodel level as OCL constraints.

Dynamic semantics variability allows for different executions of the same model. It would for example define the behavior of a switch construct regarding the execution of the remaining case statements (do we need a break statement in order to avoid the execution of the other statements or not). In the UML language, some semantics variability is identified as for example regarding the event dispatching and scheduling ([4], section 2.3) for state machine or activity diagrams. The semantics variations are defined at the language construct level and thus are defining variation points of the language.

Languages semantics is often defined from predefined MoC. MoC includes but are not limited to the: synchronous, asynchronous, concurrency or sequential paradigms. According to the choice of the MoC, the language semantics might be very different as investigated in the PTOLEMY project[31] [99], the MODHEL'X framework[32] [30, 47] and the recent GEMOC[33] initiative. In our opinion, MoC variations accom-

[30] https://projects.eclipse.org/projects/modeling.sirius
[31] http://ptolemy.eecs.berkeley.edu/
[32] http://wwwdi.supelec.fr/software/ModHelX/
[33] http://gemoc.org/

panied with a concrete syntax variation are defining different languages and not variations of the same language.

In MDE, dynamic semantics can be provided by generating code from the metamodel conforming models and thus be expressed relying on a previously defined language. Implementing such semantics variation point may have an impact on the static semantics and on the syntax definition.

Variability by extension

The Epsilon project[34] defines a family of languages[92]. Each language of this family EOL is embedded and the resulting language is task-specific. The EOL is a language with a well defined syntax and semantics. Each language of the Epsilon family, by embedding the EOL, is including its definition and builds around it a language extension.

This way of providing extended languages by relying on a core language provides the ability to simplify the extended languages definition as all the necessary mechanism for the manipulation of model is provided by EOL. The defined languages family can thus be considered as a set of variations on the use of the EOL.

Other frameworks and core languages have been defined in this purpose like XBase [64] that is a core language providing complex reusable programming language patterns aiming at being extended in the XText language framework for the definition of DSML.

3.5 Synthesis

We have defined here the variability of language according to two criterion: observability and granularity of the variability. We defined, in our preliminary generic study on languages variability, that languages variability should be studied from the point of view of the language building blocks variability and from the one of the observability and granularity of the variations. According to this, it is our belief that we can distinguish between variations of the same language and different languages definitions.

Extending a language from a core definition allows to build families of interoperable languages. Extensions of languages are defining new languages as they introduce new constructs, new syntax, and new semantics for the new elements.

In the following, we will focus first on the definition of dataflow languages in general as they are the main use case of this PhD. We will concentrate on the Simulink use case as it is the de facto standard for the design of safety critical embedded systems. This language has a fixed architecture semantics and syntax but its major components (the blocks) are highly variable as their semantics is varying according to their configuration. This language defines a fine-grained variability observable only while executing the language programs. In the following, we will study the variability of the blocks and the difficulties it involves regarding their specification.

[34] http://eclipse.org/epsilon/

4
Dataflow languages

Data Flow language are of strong interest for the design of embedded safety-critical systems. Their complexity is ever increasing and so is the related need for verification of the programs behavior. Some ACG process for embedded systems relies on dataflow languages and thus it is mandatory to have a full knowledge of their semantics.

In this chapter we will introduce dataflow languages as they will be the use case in this PhD thesis. We will start by providing historical informations about this language family; we will then define their structure and semantics; finally we will emphasis on the prominent difficulty that arise regarding their specification.

4.1 Dataflow languages

4.1.1 Origins

To our knowledge, dataflow languages first appeared with the BLODI language developed at the BELL laboratories by Kelly in 1961 [88]. This textual language "*corresponds closely to an engineer's block diagram of a circuit*". Its purpose was to "*lighten the programming burden in problems concerning the simulation of signal-processing devices*". BLODI-based programs are built "*from an alphabet of thirty types*" each one representing simple basic electronic circuits. Each "*type*" (or "*box*") has input and output ports, outputs are computed according to current and previous values of the inputs. Inputs of "*types*" are connected to other outputs. BLODI programs were specified using textual notation. An example of program extracted from [88] is provided in Listing 4.1. In this example, the "*Box UV is an amplifier with a gain of 5.28 which feeds box XY (first input) and the second input of box Z*". This language is considered by many as the ancestor of all dataflow languages.

```
UV AMP 5.28, XY, Z/2
```

Listing 4.1: A BLODI circuit example

Adams early works and Ph.D thesis [11], introduced a model of parallel computation that can be used to model dataflow languages execution. Latter works by Khan [82] or Dennis [60] formally defined dataflow languages. The first one studied the analysis capacity of dataflow programs whereas the second one formally defined the semantics of dataflow programs as described below. Relying on [82] and [60] works, Johnston [80] defined dataflow programs as:

> A dataflow program is represented by a directed graph. The nodes of the graph are primitive instructions such as arithmetic or comparison operations. Directed arcs between the nodes

represent the data dependencies between the instructions[93]. Conceptually, data are exchanged as tokens along the arcs [60] which behave like unbounded first-in, first-out (FIFO) queues[82].

4.1.2 Dataflow languages for critical systems development

Dataflow models as defined previously are called Kahn Process Networks (KPN). In these networks, nodes are executing operations and the arcs between them are behaving like unbounded FIFO queues holding data. The reading activity on a process inputs is blocking which means that as soon as a process starts reading on its input, it will wait until the required amount of data is available to go on with its execution while the writing is never blocking.

Dataflow execution is done either asynchronously or synchronously. In asynchronous dataflow programs, processes can be executed at any time in a similar way to Petri nets while in synchronous dataflow programs all the processes are executed at the same time according to a periodic (physical or logical) clock. Asynchronous processing of dataflow networks is used in parallel computing [80] whereas synchronous dataflow networks are compilable to sequential programs.

4.1.3 Synchronous dataflow languages

The use of synchronous logical time is an abstraction allowing to ease the management of time. A correct synchronous program will execute at each tick of its clock in a null time – the program is supposed to execute instantaneously: the node executions and the dataflows between nodes are instantaneous. The execution is deterministic as it removes problems related to concurrent behaviors. This property is very important while modeling embedded critical systems as it allows formal verification and reasoning on the program execution.

4.1.4 Realistic implementation of KPN

In KPN, unbounded FIFO queues in the arcs have nice properties in theory – every process can execute an infinite number of time as soon as it has enough input data; but it is impossible to implement it in real computers as the queues must be bounded. However it is undecidable to determine for any KPN if its execution can be bounded by some value thus it is impossible in the general case to choose a bound for the FIFO queues. It is therefore mandatory to define a behavior when – during a real KPN execution – the queue bound is reached: either the bound is increased at run-time which may use a lot of computing resources or a blocking write is used. The latter solution can lead to problems as it may introduce deadlocks in the program that are not present in an unbounded network. A solution for synchronous dataflow programs was to reduce the size or the FIFO queue to zero. With this additional constraint, every data that is produced on an output port must be immediately consumed on the input of an other node – this is allowed by the synchronous management of time. Work was conducted to handle sized queues in [103].

In the following we will refer to synchronous KPN with FIFO queues of size zero as dataflow programs (textual version) or dataflow models (graphical version). This is a huge language abuse that is common in the embedded systems community.

4.1.5 Dataflow model execution

During one execution of a dataflow model, each node is activated once and is producing its output data according to its input data. This execution of the whole model is called an execution cycle.

Dataflow model clocks

Execution cycles for synchronous dataflow model are periodic operations done according to a logical or physical clock. At each clock tick, the dataflow model nodes are activated and then executed.

It is possible to define for each node of the dataflow model a different clock but this clock must be an integer multiple or divisor of the main clock defined on the system. In concrete dataflow model definition, the clock is not always specified for each node and is computed at the same time as the sequencing of the model. Models with multiple clocks are specific and thus introduce some possible semantics variation points on the language execution.

A clock is defined for each block (by default it is the general clock of the system). When the clock associated to the block is active (level high) then the block is said to be enabled. If the block is enabled and it can be executed (its inputs are available), then the block is said to be activated.

Indeed, let us consider three nodes N_1, N_2,and N_3 that are activated according to their respective clocks C_1, C_2,and C_3; C_1 is twice as fast as C_2 that is twice as fast as C_3; the output of N_1 is used as an input of N_2; and the output of N_2 is used as an input of N_3. It is clear that the output of the model that is the output of node N_3 is provided only every 4 clock ticks of C_1 (every two clock ticks of C_2 or every clock tick of C_3). By the time that N_3 is computed, N_1 has been activated four times and N_2 two times. The order of node activation is constrained by the fact that N_3 must be computed after the two others at every tick of C_3 and N_2 must be computed after N_1 at every tick of C_2 but for all the other clock ticks of C_1 no order is constrained. A choice regarding these order must be done and specified in order to ensure the well-foundedness of the model. The example and ordering of block activations according to clocks is provided in Figure 4.2.

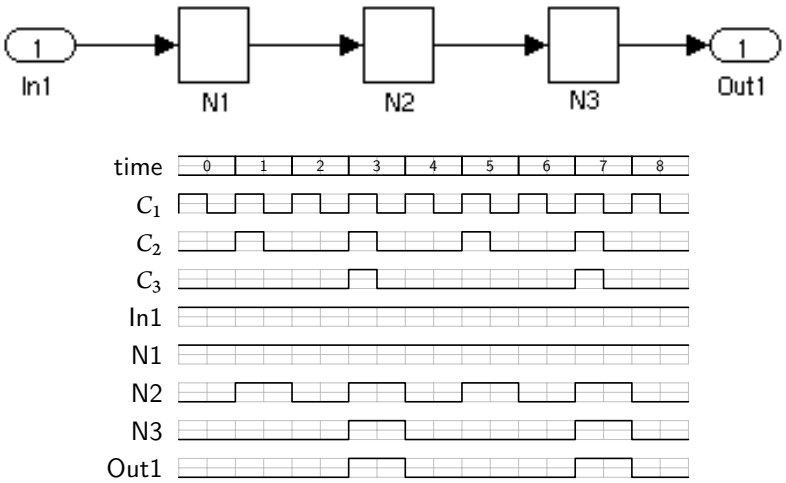


Figure 4.2: MultiRate dataflow model, boolean clock flow and block activation

If we use the example provided in Figure 4.2, the output is computed at time steps 3 and 7. Hence the final result of the model execution at time 3 can be decomposed as:
$N_1, N_1, N_2, N_1, N_1, N_2, N_3$ or $N_1, N_1, N_1, N_1, N_2, N_2, N_3$ which does not provides the same result.

Clock formalisation as been provided for the Lustre [71] dataflow language by Caspi et Al [34]. Some applications of these clock calculus have been provided for the Lustre [35] and Signal [112] languages.

Dataflow model execution semantics

In each execution cycle, the sequenced activated blocks are executed. The execution semantics of a model consist in three semantic phases: initialisation, computation and update.

A dataflow model needs to be initialised prior to its execution. This initialisation is thus done at the first execution cycle and before any other execution (compute, update) of any node. Initialisation of a dataflow model aims at providing the initialisation of the model memories.

As soon as this initialisation phase is done, the cyclic execution of the model can start: at each clock tick, activated nodes are executed: this is the computation phase. When every node has been executed,

```
L = list of all non sequenced nodes of the program
while (L not empty){
  oneSequenced = false
  foreach n in L {
    if (all the inputs of node n have been computed) {
      sequence n
      remove n from L
      oneSequenced = true
    }
  }
  if (not oneSequenced){
    signal a DEADLOCK ERROR
  }
}
```

Listing 4.3: Example of sequencing algorithm for dataflow models

the execution cycle enters the final execution phase where nodes containing memories see their memories values updated. Memories update is done without a specific order.

If a value of a block cannot be calculated for a clock tick, its value is set to $\perp$ (bottom) if its previous value was never set otherwise it stays at the same value.

Dataflow model sequencing

As data produced by a node execution are to be consumed immediately, it is thus mandatory to find the order in which the nodes must be executed prior to the execution of the network. This is referred to as the sequencing of the dataflow model. If sequencing of the dataflow model is not possible – for example if a model is not well founded (Section 4.3) – then the dataflow model will be considered as being incorrect.

There are multiple algorithms for the computation of this sequencing among which is the one presented in [98]. These are topological ordering algorithms. Sequencing a dataflow model is the process of finding an execution order for all its nodes. An example algorithm is provided in Listing 4.3 that is based on the *worklist* algorithm: a generic conceptual algorithm used for the application of an activity (the execution order assignment) on a set of elements until it has been successfully applied to all elements or the algorithm detects an impossibility to go on further with the activity application.

Izerrouken [77] provides a formal definition and implementation for such a variant of the *worklist* algorithm for the sequencing of the Simulink variant of dataflow models. The algorithm proof is done using the Coq theorem prover in the context of the GeneAuto project.

4.2 Dataflow model structure

Dataflow models are sets of equations that describe *elementary computations*. Each equation reads and writes sets of variables. A block of the dataflow model is an equation. It is therefore possible to extract *data dependencies* between equations. An equation can be computed as soon as the data that it depends on becomes available – i.e. as soon as they have been computed in other equations. The order of the equation writing is not important as their execution order is resolved statically. As an example, we provide a Lustre [71] program in Listing 4.4. This program is composed of one node: `Average` taking as input two integers `X` and `Y` (line 1). The computation result is an integer: `A` (line 2). A variable: `S` is declared as a local variable of the node (line 3), it can only be used inside this node. Finally, two equations are expressed providing the required activities for the computation of the average value of the two provided inputs (lines 5 and 6). In a Lustre program, a variable contains potentially infinite dataflows (data streams) that must satisfy the equations. This is denoting a fix-point semantics.

```
1  node Average(X, Y : int)
2  returns (A : int);
3  var S : int;
4  let
5    S = X + Y;
6    A = S / 2;
7  tel
```

Listing 4.4: A simple LUSTRE program

4.2.1 Graphical dataflow model structure

High level structure

A metamodel for dataflow models, derived from GeneAuto, is provided in Figure 4.5. In graphical dataflow models, we refer to blocks (metaclass `Block`) having inputs (metaclass `InputPort`) and outputs (metaclass `OutputPort`) ports, parameters (metaclass `Parameter`) and memories (metaclass `Memory`). Dataflow between blocks are modeled using signals (metaclass `Signal`) that links one output port to one input port.

Each element of a dataflow model is named – through the inheritance to the metaclass `NamedElement` – as it should be possible to refer to them. Blocks names should be unique within a `SystemBlock`.

Block element structure

When it is activated, each block performs an operation on its input ports according to its parameters, memories and special input ports trigger and enable. Memories grant a block with the ability to store one or more values that will be used during its following activations.

Blocks are categorized according to their capabilities, this is referred to as its category. Categories are: `COMBINATORIAL` if the block outputs only depend on the current values of its inputs and current values of its parameters; `SEQUENTIAL` if its outputs also depend on its input values from the past stored in its memories and thus the computation of the block inputs is independent of the current block inputs; `SOURCE` if the block has no input and thus reads data from outside of the system or from shared memories; or `SINK` if the block has no output and thus stores or sends data outside of the model or in shared memories. Block are either atomic (opaque) – computing a simple operation; or hierarchical – a composition of other blocks and signals (metaclass `SystemBlock`). Hierarchical blocks interest are used first to structure a model according to system engineering principles (virtual sub-systems), and second to tag a group of blocks and signals as being executed atomically with respect to the other sub-systems (non-virtual/atomic sub-systems).

Blocks ports

Port elements have an attribute named `kind`. It allows to characterise the port and define its use in the block specification. We emphasis four different kinds for a port: `DATA` ports carrying data. These are the most classical ports that receive the data from a signal and provides it to the block to compute its results and memories (for input ports) and receive the data computed by the block and transmit it to the signal (for an output port). `ENABLE` and `EDGE_ENABLE` ports are ports which according to the value they carry will activate or not the block and thus make it compute and update its output values and memories. In the `ENABLE` case, the block will be activated if the port value is interpreted as a boolean having a True value. In the `EDGE_ENABLE` case, the block activation will be conducted if a rising or a falling edge is detected on the input. This implies to compare the actual value of the input with its previous value. This port can be replaced by an enable port and a sub-system that detects edges. Only one of the `ENABLE` and `EDGE_ENABLE` input ports are allowed in one block instance. Finally `EVENT` ports carry events allowing to explicitly sequence the blocks when the event is produced instead of following the computed sequencing. These events are provided internally by the model or by other models. For example in a Simulink model,

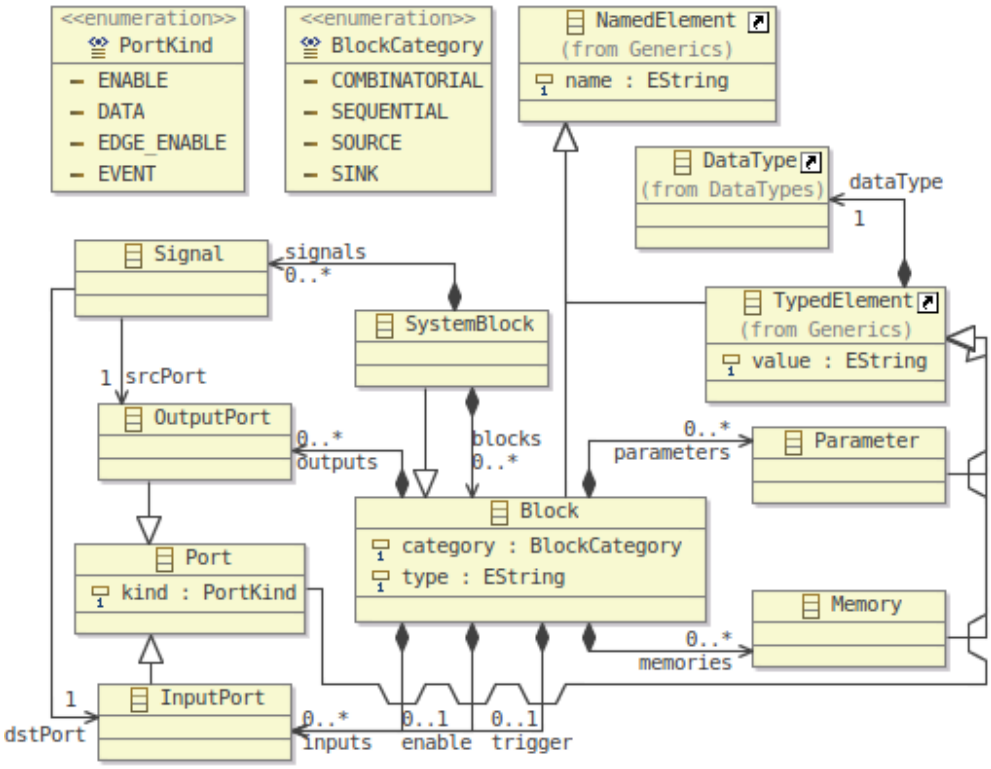


Figure 4.5: Metamodel for dataflow models

events might be sent by Stateflow models or by specific blocks like control flow logic blocks (if-else, for, switch, while, enable, ...) and can be used to provide different orders for blocks throughout the execution. The dual use of EVENT ports might be to drive a Stateflow model execution from a Simulink block execution.

Dataflow model values

Block parameters, memories and ports are carrying values, a value needs to have a data type. This is done through the inheritance of the metaclass `TypedElement`. Every `TypedElement` instance owns a `DataType` value and a string `value` attribute containing the literal value for the `TypedElement` that should be parsed according to the `TypedElement` data type. A possible hierarchy of data types derived from Simulink is provided in Figure 4.6.

This type system has been derived from the GeneAuto project results. It provides classical numeric types specification `TRealInteger`, `TComplexInteger`, `TRealDouble`. Standard programming languages data types such as `TString`, `TBoolean`, `TPointer`, `TEnum` are also provided. Finally, structured types like `TArray` are present.

The `TArray` data type has an attribute named `dimensions`, this vector of positive or null integers values contains the number and the size of the dimensions of the `TArray` (the number is the size of the vector). A `TArray` conforming element with a `dimensions` attribute equal to [4,5] will model a matrix value with 4 rows and 5 columns. It is allowed to set the size of a dimension to zero, in this case the dimension is set to be unbounded.

Some dataflow languages like Simulink allow to apply operations on values with different dimensions and datatypes. This is allowed only if the values are compatible which means that there is an allowed conversion between the values. Allowed dimension conversions are those transforming a scalar value into a vector or a matrix having all their values set to the value of the scalar. Allowed datatypes conversion are those transforming a value to a datatype whose definition includes the original value datatype definition

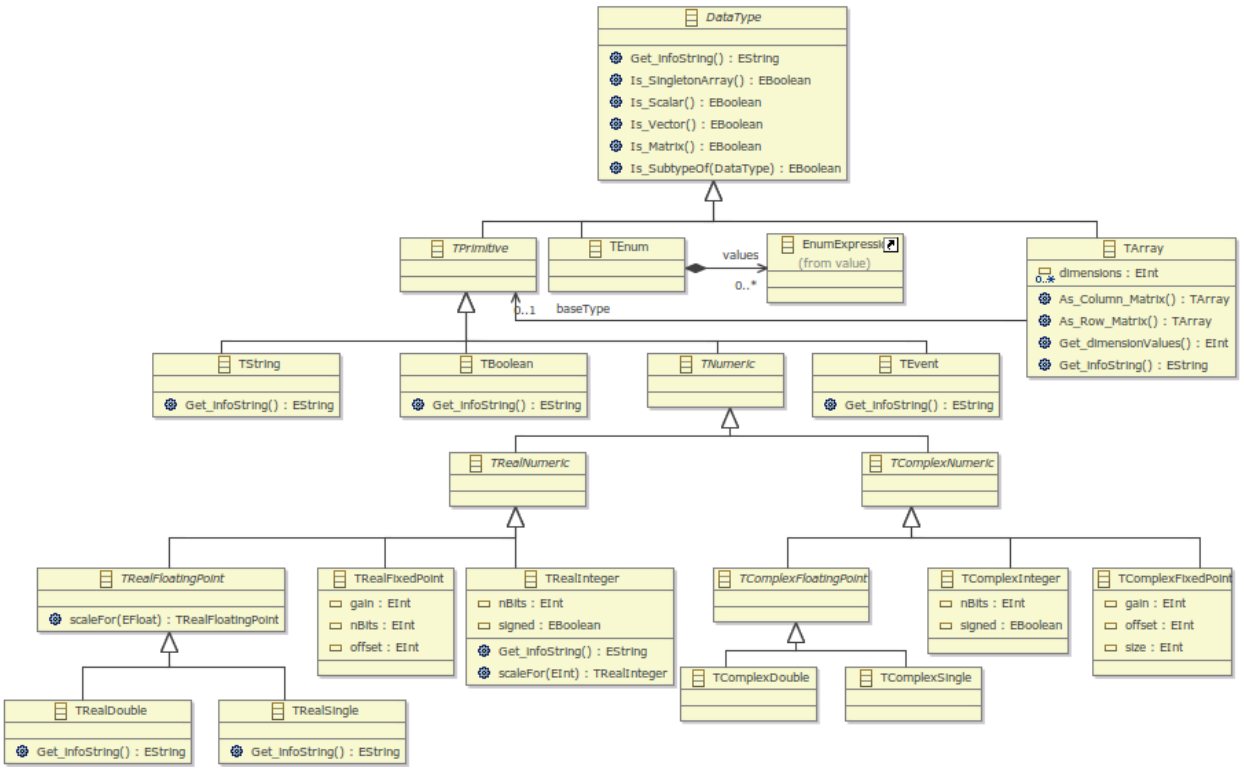


Figure 4.6: GeneAuto DataTypes metamodel

domain. The dimension conversion is often referred to as the expansion mechanism in SIMULINK and will be referred as such in this PhD.

4.2.2 DATAFLOW LANGUAGES

Among successful textual dataflow languages we can cite LUSTRE [71], LUCID-SYNCHRONE [36], SIGNAL [22] or PRELUDE [122]. These languages are mostly used for the development of real-time embedded control systems. They have been developed by research teams and formally specified. Their initial end users were in the academic world but their qualities made them very useful as semantics backend for the model based development of concrete industrial systems. Some of these languages have been given a graphical syntax and interface in order to simplify their adoption by industrial users. We can cite the SCADE[1] tool and language which is based on LUSTRE and POLYCHRONY/SME[2] or RT-BUILDER/SILDEX based on SIGNAL. Both of these graphical front-ends have been widely used in critical embedded system development.

SIMULINK[3] is another commercial tool largely adopted in the industry and SCICOS/XCOS[4] is a similar open source alternative in the SCILAB[5] community. These graphical languages have been developed for control and command engineers and their formal definition does not rely on the same dataflow synchronous language background. They have been widely used for control systems development in the last years. To our knowledge, the first one is the de-facto standard for such developments.

As an example, we provide in Figure 4.7 a SIMULINK model for a modulo-three counter. It is composed of one *Input* block – `reset`; one *Output* block – `active`; five LogicalOperator blocks – `LO` and two *Unit Delay* blocks – `UD`. Two variants of the same block are used: the `AND` LogicalOperator block variant (i.e. `LO1`) and its NOT variant (i.e. `LO`). Every signal value in this model has a boolean value evaluated at this precise execution time. The `LO` block is a logical not block. The `UD` block outputs the value of their input at the previous clock tick. This block can only output an undefined value at the first clock tick, this is handled

[1] http://www.esterel-technologies.com/products/scade-suite/
[2] http://www.irisa.fr/espresso/Polychrony/
[3] http://fr.mathworks.com/products/simulink
[4] http://www.scicos.org/
[5] http://www.scilab.org

by providing an initial value as a parameter of each UD block. In our model, both UD blocks have an initial value set to true. This system behavior is simple, it outputs a true value on the active output port every three clock ticks. If the `reset` *Input* port is set to true then the counter is set back to zero and it should then wait for 4 clock ticks (a modulo 3 counter), without the `reset` input port set to true, to produce an output value set to true. The formal semantics for this system is provided in (8.1) in which x_t is the value of x at the time step t.

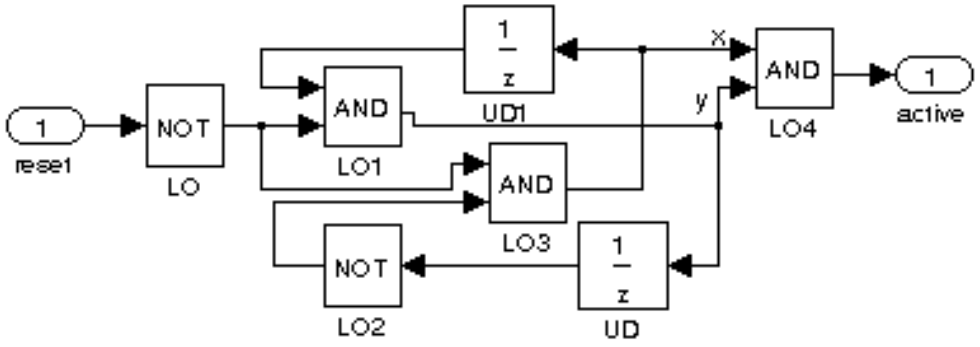


Figure 4.7: Simulink model for a modulo 3 counter

$$\begin{aligned} x_0 &= 0 \\ y_0 &= 0 \\ x_t &= \neg reset_t \wedge \neg y_{t-1} \\ y_t &= \neg reset_t \wedge x_{t-1} \\ active_t &= x_t \wedge y_t \end{aligned} \tag{4.1}$$

In the following we will refer to the graphical dataflow terminology (blocks, ports and signals) but principles and concepts similarly apply to textual dataflow programs. These kind of textual concrete syntaxes can be associated to the previous metamodel.

4.3 Well founded dataflow model

Dataflow models should be conceived according to structural and semantics well-foundedness rules. Some checks needs to be performed to ensure this correctness and thus ensure the possible execution of the model.

4.3.1 Causality errors

Computing the output value of a block first requires the sequencing of the block in the model. The block input values should then be available and should have been computed by the execution of the blocks where output ports are connected by signals to the input ports of the block. This backward analysis of block dependencies may be carried on until an already computed values are available or the required blocks have no inputs.

In specific cases, this backward analysis may conclude in a causal loop: reaching a block that has already been traversed. A simple example is provided in Figure 4.8 where the second input of the *Sum* block depends on the input of the *Product* block whose first input depends on the output of the *Sum* block. Such faulty models are caused by either a wrong conception of the modeled system or by a missing *SEQUENTIAL* block. Indeed, if there exists a loop in the graph where nodes are the blocks and arcs are the signals and this loop contains a *SEQUENTIAL* block then the loop is broken as the inputs are only needed in the

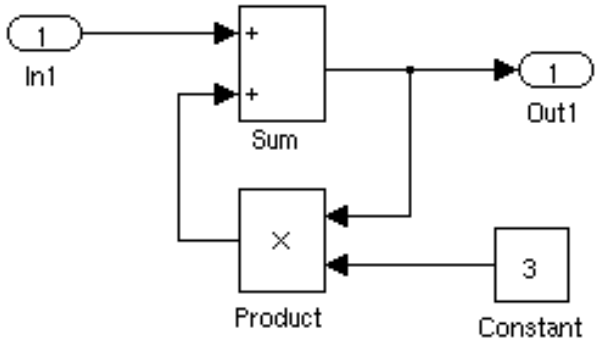


Figure 4.8: A causality error example in a SIMULINK model

update phase, not in the compute phase.

The backward causality loop detection approach provided here is dual to a forward one: if from a block it is possible by following its outputs to draw a path to its input ports that is not going through a *SEQUENTIAL* block, then a causal loop is present. If such a loop is present then the previously provided sequencing algorithm cannot finish as for some blocks the inputs will not be computable and at some point of the computation the error will be raised.

Using multiple clocks in a dataflow model may lead to causality errors. Such problems may also arise if for a block we do not know the value of its inputs as it depends on a block that has never been activated before the clock tick of the current block activation and computation. Enabled block are suffering from the same problem as, depending on the value on this specific input, they may not be computed.

Finally when a dataflow model depends on triggered elements, it is not possible in the general case to ensure the absence of causal loops as they are modeling explicit control flow depending on the computed values.

4.3.2 Data type overflow

Dataflow blocks are performing operations according to their inputs, parameters and memories. These operations can be logical, arithmetical or a combination of both. Arithmetics operations on values should be taken care of as they can lead to overflow problems and thus in some case cause a computation error or a false result. These kind of errors are most of the time related to the execution of the blocks thus to arithmetic operations because static assessment can be undecidable but may be verified by over approximation (using abstract interpretation) and thus cannot be checked at compile time. These are runtime errors.

It is also mandatory to ensure that the inputs of a block is of the allowed dimensions and for example a block might allow on an input port only scalar or vector or matrix values. This is the common typing constraints of the static semantics.

4.4 Dataflow languages block semantics variability

Beside their classical dataflow execution semantics, dataflow language core semantics mostly rely on the blocks themselves. The blocks are an important extension point of the core languages and determine the practical usefulness of the language. The blocks are gathered in block libraries. To reduce the number of blocks in the library and thus ease their maintenance, the semantics of blocks are often tunable by a number of static parameters. For example, these ones control the number and data types of inputs/outputs, their dimensions (scalar, vector, matrix) and the amount and type of memory the block relies on. This results in a quite complex variability of the blocks.

As an example, Figure 4.9 shows some configurations of the *Sum* block from the SIMULINK standard library with different parameters, types, dimensions and number of inputs/outputs. This block can compute the sum of inputs (the first *Sum* block: *Sum of inputs*), of all the elements of the single input (the

second *Sum* block: *Sum of input elements*) or of elements along a specified dimension of the single input (the third *Sum* block: *Subtract by the 2nd dimension*). Additional parameters allow to tune the signs at each input port, rounding and other computational details. The full specification of this block in the SIMULINK documentation is around twenty pages of natural language. We refer to this one as a semi-formal specification as it relies on natural language definitions for the meaning of the block components (parameters, inputs, outputs and memories) that are supplemented with structured informations like tables or graphs and on global language definitions provided in other documentation pages. Some elements of blocks semantics are provided using mathematical formulas. Semi-formal specifications provide mandatory informations for the use of the blocks but may suffer from a lack of details regarding corner cases that can only be provided with formal specifications, they may also be incomplete and even inconsistent as they are only proofread by fail prone human beings.

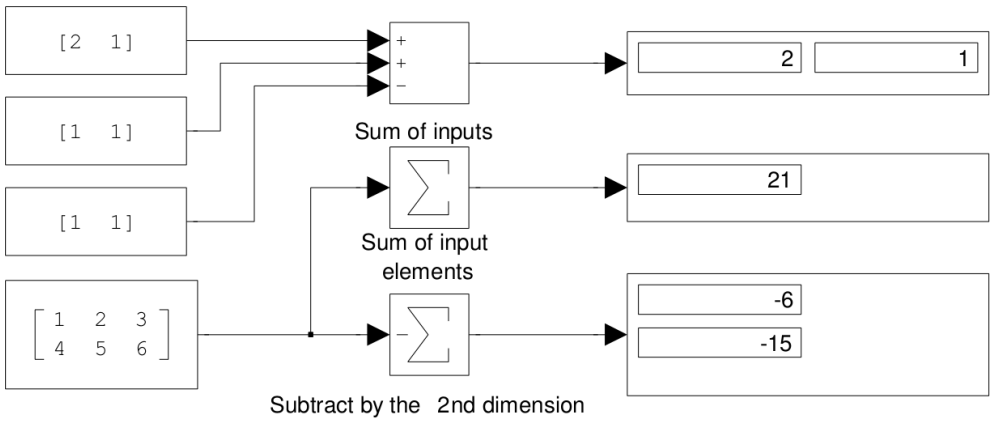


Figure 4.9: Simulink model with different configurations of the *Sum* block

4.5 CHALLENGES TO TACKLE REGARDING SPECIFICATION

Dataflow language have largely been studied [23, 37, 38, 45, 138] and especially synchronous languages. On the contrary, the semantics of blocks is most of the time left aside. These blocks can be quite complex and so is the writing of their specification. There is a lack in formal methodologies ensuring flawless specification of these languages elements. Both the formalism for the specification writing and the specification verification techniques should provide formal reasoning capabilities.

The block complexity is mostly related to their structure and semantics variability that needs to be handled in a formal and complete manner in order to simplify the formal specification of the language elements (blocks). In the following, we provide both methodology and tools in order to formalise block specifications and to verify the specification correction according to criteria that will also be provided.

Part II

Highly variable languages formal specification

5

Experiments with classic software engineering tools

The formal block specification activity aims at the production of flawless specification for blocks in dataflow languages. This specification can be expressed using any required means as soon as its understanding is free from interpretation (i.e. formal) and its assessment is possible according to predefined criteria.

In this chapter, we will go through the requirements that any block specification methodology should fulfill. We will detail the elements this method should allow to specify about the blocks structure and semantics; the detail and kind of informations it should provide; and the possibilities in term of automated verification and code or documentation generation it should allow.

We will then provide the specification for two blocks, a combinatorial one, the *MinMax* block and a sequential one, the *Delay* block. Three state of the art specification techniques will be used in order to provide the specification for these blocks: mathematical textual notations, UML diagrams extended with OCL constraints and a SPLE approach with feature models. Advantages and drawbacks of each method will be discussed. We will then draw a possible use of combined SPLE and UML, discuss its usability and finally motivate the use of a DSML for our purpose.

5.1 Block specification requirements

The design of block specifications must provide mandatory informations in order to ensure the usefulness of the specification. We detail in the following tables the requirements for the block specification tooling and approach. A requirement table row is made up of a requirement identifier, its description and additional informations such as justification or purpose, the requirement definition and additional non-formalised constraints.

Table 5.1 lists the requirements on the informations to provide for a block specification. Tables 5.2, 5.3 and 5.4 give structural specification requirements about block ports, parameters and memories. Table 5.5 lists requirements about data types and dimensions specification for ports, parameters and memories. Semantics of blocks must be provided in the design of block specifications and must comply with the requirements of Table 5.6. A block specification must verify some criterion as detailed in Table 5.7. Tooling used for the design of the block specification must be as accessible as possible for the designer as listed in Table 5.8. The successful design of block specification satisfying all these requirements will require some methodology. In the following we investigate the use of existing specification formalisms and methodolo-

Identifier	Decription	Justification, Purpose, Constraints
REQ-1.a	Block specification must provide a unique name for each block	A block unique identifier is its name
REQ-1.b	Block specification must provide informations about the number of each kind of input and output ports this block handles	A block can have a multiple number of input and output ports. Sets of ports may have the same functionality in the block specification and their number must be known

Table 5.1: Block structure specification requirements

Identifier	Decription	Justification, Purpose, Constraints
REQ-2.a	Input and output port specification must provide a name for each input and output	An input or output port unique identifier is its name
REQ-2.b	Input and output port specification must provide the allowed data types	Every port carries a data value. The allowed value data type must be provided. The allowed value data type is not necessarily fixed as multiple data types are possible for one port value
REQ-2.c	Input and output port specification must provide the allowed dimensions for each allowed data type of the input or output port	A port value can have one dimension (it is a scalar) or be multi-dimensioned (it is a vector or a matrix)

Table 5.2: Input/Output ports structure specification requirements

Identifier	Decription	Justification, Purpose, Constraints
REQ-3.a	Parameter specification must provide a name for the parameter	A parameter unique identifier is its name
REQ-3.b	Parameter specification must provide the allowed data type of the parameter	Every parameter carries a data value. The allowed value data types must be provided. The allowed value data type is not necessarily fixed as multiple data types might be possible for one parameter value
REQ-3.c	Parameter specification must provide the allowed dimensions for each allowed data type of the parameter	A parameter value can have one dimension (it is a scalar) or be multi-dimensioned (it is a vector or a matrix)

Table 5.3: Parameters structure specification requirements

gies for this purpose.

5.2 Block examples

5.2.1 MinMax block

informal specification

The *MinMax* Simulink block is a combinatorial block. Its purpose is to compute at each clock cycle the maximum or the minimum of the values of its input(s) signal(s) in the same cycle and to feed its output signal with this computed value. Switching between the computation of the minimum or the maximum is

Identifier	Decription	Justification, Purpose, Constraints
REQ-4.a	Memory specification must provide a name for the memory	A memory unique identifier is its name
REQ-4.b	Memory specification must provide the allowed data types for the memory by relying on the data type of an input port or a parameter	Every memory carries a data value. The allowed value and thus its data type must be provided according to an existing parameter or input port
REQ-4.c	Memory specification must provide the allowed dimensions for each allowed data type of the memory by relying on the dimensions of an input port or a parameter	A memory value can have one dimension (it is a scalar) or be multi-dimensioned (it is a vector or a matrix)

Table 5.4: Memories structure specification requirements

Identifier	Decription	Justification, Purpose, Constraints
REQ-5.a	Each input port, output port, parameter and memory specification must provide the restrictions on its allowed values. Such restrictions must be computable and expressed using a formal language	Input/output ports, parameters and memories allowed values might not be all the allowed values for its data type
REQ-5.b	Each input port, output port specification must provide the restrictions on its multiplicities. Such restrictions must be computable and expressed using a formal language	Input and output port multiplicities must be constrained. All multiplicities are not necessarily allowed

Table 5.5: Data type and dimensionality specification requirements

Identifier	Decription	Justification, Purpose, Constraints
REQ-6.a	A block specification must provide a formal specification for each phase of the execution of a block (initialisation, computation and update)	As a block is computable its executable specification should be provided
REQ-6.b	A block specification must provide for each phase of the execution of a block (initialisation, computation and update) its semantics variation points according to the inputs/outputs/parameters and memories values	As a block configuration might vary the resulting semantics will also vary.

Table 5.6: Semantics specification requirements

done according to the *Function* parameter having two allowed values: MIN and MAX.

The *MinMax* block applies its computation on values of data types on which an ordering relation has been defined. It allows to provide an order between all the possible values of the data types and thus apply the comparison. If one of the input values is smallest value according to the ordering relation – respectively the biggest – and the *Function* parameter value is MIN – respectively MAX –, then the output will be the smallest – respectively the biggest – value. Whereas this block has a quite simple basic semantics, it can be used in different settings.

Identifier	Decription	Justification, Purpose, Constraints
REQ-7.a	The block specification must provide means for an automatic verification of the specification structural correctness	Structural correctness check on a specification must ensure the verification of specifications instances structure (syntax and static semantics)
REQ-7.b	The block specification must provide means for an automatic verification of the exhaustiveness of the block configurations	This verification criterion must be referred to as the completeness of a block specification
REQ-7.c	A block specification must provide means for an automatic verification of the block configuration redundancy.	This verification criterion must be referred to as the disjointness of a block specification
REQ-8	A block specification must provide means for the verification of the semantics correctness of a block configuration.	An automatic verification of the semantics must be possible for all the specified block instances.

Table 5.7: Specification verification requirements

Identifier	Decription	Justification, Purpose, Constraints
REQ-9.a	Block specification tooling must provide a safe typing mechanism for formally defined constraints and semantics definitions	Ensuring the consistent typing of expressions is among the first checks to be done as it removes a large amount of errors
REQ-9.b	Block specification tooling must provide high level representations of the specification structure either by textual or graphical representation	Using high level representations of an ongoing design is known as a common means to provide perspectives and thus has the potential to improve design quality
REQ-9.c	Block specification formalism must allow for a convenient management of the block specification variability complexity	This can be done by relying on common software knowledge concepts (inheritance, ...) for the modeling of the specification elements. This might avoid the discovery bottleneck that is common while using a new DSML

Table 5.8: Related tooling requirements

Structural and Semantics Variation points

The *MinMax* block structure regarding its allowed interface (inputs and outputs) can be split in two structural variation points, both of them containing 3 variation points.

1. Only one input port is provided: in this setting, the computation is done on the value of each component (values contained) of the input.

   1.1 The only input is a scalar: the output value is equal to the input value.

   1.2 The only input is a vector: the output value is equal to the maximum/minimum value of the values contained in the input vector (the output value is a scalar).

   1.3 The input is a matrix: the output value is equal to the maximum/minimum value of the values contained in the input matrix (the output value is a scalar).

2. Multiple input ports: in this setting the computation is done on the value of each components (values contained) of the inputs and assigned to the corresponding component of the output.

   2.1 All the inputs are scalars: the output value is equal to the maximum/minimum of the inputs values (the output value is a scalar).

   2.2 All the inputs are either scalars or vectors: all the inputs vectors should have the same dimension. All the input scalars are expanded to vectors (each components of the resulting vectors are equals to the original scalar) of the same size than the other input vectors. The output is then a vector of the size of the input vectors. The Nth component of the output is equal to the maximum/minimum of the Nth components of each inputs.

   2.3 All the inputs are either scalars or matrices: all the input matrices should have the same size. All these input scalars are expanded to matrices of the same size than the other input matrices. The output is then a matrix of the size of the input matrices. The Nth component of the output is equal to the maximum/minimum value of the Nth components of each input values.

5.2.2 Delay block specification

Informal specification

The *Delay* Simulink block is a sequential block. Its purpose is to provide access to the value of an input signal (called the input data signal) from the previous clock cycles. The value provided on the output of the block will then be delayed according to a mandatory delay parameter (of value N). This block relies on a mandatory parameter called initial_value that specifies the initial value(s) of the output signal for the N first outputs of the block – it could thus be a vector if N is greater than 1. Without this parameter, this block would provide N undefined (the dataflow $\perp$ value) computed output signal values for the Nth first cycles of the block execution.

Structural and Semantics Variation points

This block is mostly used with this simple configuration, however there exists some allowed variations of its structure and semantics:

- variable delay: The output of the block will be the Nth preceding input data signal value. The N value is then provided as an input of the block. It can thus vary during the block execution.
- resettable: according to the value of an optional input port called reset, the block can be reseted (i.e. the output is set to the first component of the *initial_value* and the next output will be the current input). An additional optional parameter *reset_algo* will modify the behavior to adopt regarding the reset application condition.
  - NONE: deactivates the reset_input feature.
  - RISING_EDGE: activates the reset on a rising edge of the res_input signal value (when res_input signal value goes from a value lesser than zero to a value greater or equals to zero).
  - FALLING_EDGE: activates the reset on a falling edge of the res_input signal value (when res_input signal value goes from a value greater than zero to a value lesser or equal to zero).
  - EITHER: activates the reset on either a rising or a falling edge (when the value is not stable around zero).
  - LEVEL: activates the reset in both cases:
    * When res_input signal value is different from zero or
    * When the current output signal value is different from zero and the current res_input signal value is equal to zero.
  - LEVEL_HOLD: activates the reset when the res_input signal value is different from zero.

- external initial value: the *initial_value* parameter can be provided as an optional input of the block and not as a parameter.

These variations can result in sometime complex but usually numerous combinations of behaviors that need to be specified. These combinations may result in additional constraints on the input ports, parameters and memories data types, dimensions and values that must also be taken into account in the specification.

In the following, we will highlight our approach using a simplified version of the *Delay* block, its input value can only be a scalar value and we won't take into account the variable delay semantics variation point.

5.3 Mathematical notation for the specification of blocks

Some experiments were conducted together with domain experts in order to find formal means for the specification of blocks in a manner that is both acceptable by common system and software engineers and at the same time formal enough to allow automated treatment in different phases of code generation.

Our first experiment was to rely on a standard mathematical notation. Mathematics seemed to be a good starting point for our purpose as it is an universal formal language. In the following we depict a formalised specification for the *Sum* block – computing its output as the sum of its inputs – using a formalism close to the MathML[1] one. This is the usual formal specification provided for those kind of languages in the academic and industrial communities. Such typical specification could be structured as the following:

a) Definition of common notations:

- Specification data types (as in example in Figure 5.9).

  - $\mathbb{B}$: boolean data type.
  - $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}$: 8, 16 and 32 bits relative integers data types – a.k.a. signed integers.
  - $\mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}$: 8, 16 and 32 bits natural integers data types – a.k.a. unsigned integers.
  - $\mathbb{C}$: complex numbers data type.
  - $\mathbb{D}$ double precision floating-point numbers data type.
  - $\mathbb{T}$ data type gathering all the elements contained in all the previously defined data types.
  - $\mathcal{V}_n(\mathbb{T})$: a vector data type of size n of elements of type $\mathbb{T}$
  - $\mathcal{M}_{n,m}(\mathbb{T})$: a matrix data type of size n, m of elements of type $\mathbb{T}$

Figure 5.9: Common specification data types

- Block structural elements (as in example in Figure 5.10).

  - $\mathcal{I}$: the set of the block input signals.
  - $\mathcal{O}$: the set of the block output signals.
  - $\mathcal{P}$: the set of the block parameters.
  - $\mathcal{M}$: the set of the block memories.

Figure 5.10: Common specification for block structural elements

- $card(X)$: if X is a set then this returns the number of elements contained in X. This is the cardinal of the set.
- $value(e)$: the value of the block structural element e.
- $value(e)_j$: the component at index j of the value of the block structural element e.
- $value(e)_{j,k}$: the component at index j, k of the value of the block structural element e.
- $dt(e)$: the data type of the block structural element e.

Figure 5.11: Common specification operations of block structural elements

- Operations defined in block structural elements (as in example in Figure 5.11).

b) Allowed inputs definitions for the block (see Figure 5.12). This part models REQ-2.[a|b|c] and part of REQ-5.[a|b];

(a) $card(\mathcal{I}) > 0,\ \mathcal{I} = \{i : dt(i) \subset \mathbb{T}\backslash\mathbb{B}\}$

(b) $card(\mathcal{I}) = 1,\ \mathcal{I} = \{i : n \in \mathbb{N}_{32},\ dt(i) \subset \mathcal{V}_n(\mathbb{T}\backslash\mathbb{B})\}$

(c) $card(\mathcal{I}) = 1,\ \mathcal{I} = \{i : (n,m) \in \mathbb{N}_{32}^2,\ dt(i) \subset \mathcal{M}_{n,m}(\mathbb{T}\backslash\mathbb{B})\}$

(d) $card(\mathcal{I}) > 1,\ \mathcal{I} = \left\{ i : \left\{ \begin{array}{l} n \in \mathbb{N}_{32},\ i \in \mathcal{V}_n(\mathbb{T}\backslash\mathbb{B}) \\ dt(i) \subset \mathbb{T}\backslash\mathbb{B} \end{array} \right\} \right\}$

(e) $card(\mathcal{I}) > 1,\ \mathcal{I} = \left\{ i : \left\{ \begin{array}{l} (n,m) \in \mathbb{N}_{32}^2,\ i \in \mathcal{M}_{n,m}(\mathbb{T}\backslash\mathbb{B}) \\ dt(i) \subset \mathbb{T}\backslash\mathbb{B} \end{array} \right\} \right\}$

Figure 5.12: *Sum* allowed inputs specification

c) Allowed parameters definitions for the block (see Figure 5.13). This models REQ-3.[a|b|c] and part of REQ-5.a;

Parameter name	**Possible values**
Inputs	$\forall i \in [1, card(\mathcal{I})],\ \exists op_i \in [+,-],\ Inputs_i = \{op_i\}$
SumOver	"AllDimensions" (Default) "SpecifiedDimension"
Dimension	1 or 2

Table 5.13: *Sum* allowed parameters specification

d) Allowed memories definitions and initial value for the block. This will model REQ-4.[a|b|c] and part of REQ-5.a;

e) Outputs definitions for the block according to the inputs, parameters and memories (see Figure 5.14). This models part of REQ-6.[a|b] – compute phase of the semantics specification. In this specification the inputs are implicitly expanded (using the expansion mechanism) in order for all the input values to have the same dimension; and

[1] http://www.w3.org/Math/

$$
\begin{array}{l}
\text{(a)}\ \ card(\mathcal{O}) = 1,\ \mathcal{O} = \left\{ o = \sum_{j=1}^{card(\mathcal{I})} value(Inputs_j)\ value(i_j) \right\} \\
\text{(b)}\ \ card(\mathcal{O}) = 1,\ \mathcal{O} = \left\{ o = \sum_{j=1}^{n} value(Inputs_1)\ value(i_1)_j \right\} \\
\text{(c)}\ \ card(\mathcal{O}) = 1,\ \mathcal{O} = \left\{ \begin{array}{l}
\textit{if}\ (value(SumOver) = \text{“}AllDimensions\text{”})\ \textit{then} \\
\quad o = \sum_{j=1}^{n} \sum_{k=1}^{m} value(Inputs_1)\ value(i_1)_{j,k} \\
\textit{else} \\
\quad \textit{if}\ (value(Dimension) = 1)\ \textit{then} \\
\quad\quad \forall l \in [1,n], dt(o_l) \subset \mathcal{V}_n(n)\mathbb{T}\backslash\mathbb{B}, \\
\quad\quad o_l = \sum_{k=1}^{m} value(Inputs_1)\ value(i_1)_{l,k} \\
\quad \textit{else} \\
\quad\quad \forall l \in [1,m], dt(o_l) \subset \mathcal{M}_{n,1}(m)\mathbb{T}\backslash\mathbb{B}, \\
\quad\quad o_l = \sum_{j=1}^{n} value(Inputs_1)\ value(i_1)_{j,l}
\end{array} \right\} \\
\text{(d)}\ \ card(\mathcal{O}) = 1,\ \mathcal{O} = \left\{ o = \sum_{j=1}^{card(\mathcal{I})} value(Inputs_j)\ value(i_j) \right\} \\
\text{(e)}\ \ card(\mathcal{O}) = 1,\ \mathcal{O} = \left\{ o = \sum_{j=1}^{card(\mathcal{I})} value(Inputs_j)\ value(i_j) \right\}
\end{array}
$$

Figure 5.14: Sum semantics specification

f) Memories initialisation and update specification according to the block inputs, parameters, other memories and outputs values. This will model part of REQ-6.[a|b] – initialisation and update phases of the semantics specification. As the *Sum* does not expose memory constructs, these are not presented here.

The results of the typesetting of MathML or LaTeX mathematical notations are easy to read, quite similar to common natural language requirement documents and formal. Nevertheless the formalism used to write them is not structured enough regarding variability management (REQ-9) as we only list the input/-parameters/memories configuration variants and give the corresponding output computation variants and reuse for automatic verification (REQ-7 and REQ-8).

One of the major drawbacks of such a specification is on its maintainability, indeed the source that needs to be written in order to obtain our *Sum* specification is quite complex (5× more lines of source code than results line of specification). Maintainability can still be improved by largely commenting and logical structuring of the source code but its apprehension will remain quite difficult. It thus seems to be too complicated to ensure the specifications' sustainability.

As the lack of structure and variability management is the weak point of such specification methodologies, we looked at more naturally structured formalisms such as modeling languages. The UML is a common modeling language targeting universality and a natural candidate for this purpose.

In the following we will detail the specification for the *MinMax* and *Delay* blocks by relying first on the UML extended with a variability profile and then on the SPLE approaches. We finally conclude on the applicability of both specification methodologies.

5.4 UML for the specification of blocks

As already stated in Section 3.4.1, the UML has some semantics variation points leading to the necessity to do some choices prior to any use of this language. The *base semantics* defined in the fUML specification [4] has the advantage of defining these choices in its section 2.3. It is not our purpose here to discuss these choices, we rather want to rely on defined standards and thus we selected this predefined semantics.

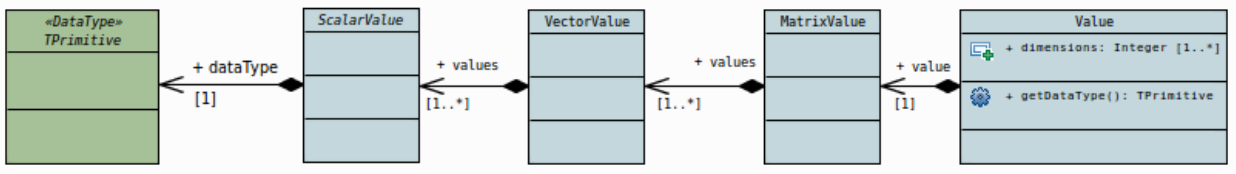


Figure 5.15: Specification of the block structural elements value using UML + OCL

5.4.1 UML BLOCK SPECIFICATION HARNESS

Modeling a block specification using the UML + OCL formalism requires first to select an UML diagram in order to hold the block specification. For this example, we choose to use the common UML class diagram as it is well suited for the specification of data structures. Using SysML Block diagrams is also an acceptable choice in this purpose. The following specifications consist of UML diagrams complemented with OCL constraints and definitions. In the specification section, we rely on the Eclipse OCL implementation and thus are using non-standard OCL operations such as the `oclContainer()` that is applied on an object and returns its container element if it exists and `oclInvalid` if it does not.

The specification of a block configuration is done through the specification of its structural elements: input and output ports (REQ-2), parameters (REQ-3) and memories (REQ-4). Each of these elements should hold a specification regarding its data type (REQ-2.b, REQ-3.b and REQ-4.b) and their related dimensions (REQ-2.c, REQ-3.c and REQ-4.c). The block specification must contain the semantics operations description for each semantics variation points of the block as required in REQ-6. In the following, we propose a specification for these elements using UML class diagrams and OCL constraints.

Values specification

In Figure 5.15 and Listing 5.16, we detail the specification for the Value metaclass. A Value instance has a reference to a MatrixValue element. This element is itself composed of a non-empty set of VectorValue which is in turn composed of a non-empty set of abstract ScalarValues. A scalar value has a reference to a TPrimitive data type class instance specifying the type of the scalar value. A hierarchy of types must be provided under TPrimitive like the one presented in Figure 4.6. It should be extended with primitive value classes inheriting from ScalarValue. The multiplicities (1..*) on the two values references allows to build scalar or vector only values. As MatrixValue (respectively VectorValue) elements are specifying matrix (respectively vector) data structures, additional constraints must be provided:

- *MatrixHasSameSizeVectors* (constraint *MatrixHasSameSizeVectors* in Listing 5.16): For any MatrixValue element, the size of its containing VectorValue elements should be the same.
- *HasSameDataTypeAllScalarValuesInMatrix* (constraint *HasSameDataTypeAllScalarValuesInMatrix* in Listing 5.16): For any MatrixValue element, all the contained ScalarValues element should be of the same primitive data type.
- *HasSameDataTypeAllScalarValuesInVector* (constraint *HasSameDataTypeAllScalarValuesInVector* in Listing 5.16): For any VectorValue element, all the contained ScalarValues element should be of the same primitive data type.

A Value instance contains a `dimensions` parameter. This parameter is a non empty sequence of non-unique integer values. These values provide a direct access to the dimensions of the object contained in the value reference. A Value element containing a Scalar should have a dimension parameter equals to [1] meaning that a MatrixValue `values` reference contains one VectorValue itself containing one ScalarValue in its `values` reference. A vector of size five will result in a dimension parameter equals to [5] (5 ScalarValue elements in reference `values` of a VectorValue element) and a matrix of size four times three will

```
context MatrixValue
inv MatrixHasSameSizeVectors:
values->forAll(v1, v2|
  v1.values->size() = v2.values->size()
)

inv HasSameDataTypeAllScalarValuesInMatrix:
values->collect(v| v.values)->forAll(s1,s2|
  s1.oclType() = s2.oclType()
)

context VectorValue
inv HasSameDataTypeAllScalarValuesInVector:
values->forAll(s1,s2|
  s1.oclType() = s2.oclType()
)
```

Listing 5.16: *MatrixValue* and *VectorValue* size and data type OCL constraints

result in a dimension parameter equals to $[4; 3]$ resulting in three VectorValue elements in MatrixValue `values` reference each one containing four scalar ScalarValue elements in each `values` references. The zero value in the dimension parameter is not allowed and the -1 (minus one) corresponds to an unbounded size of data structure. We first define some OCL operations in the context of the Value class in order to simplify the writing of constraints:

- `isUnbounded` operation returning true if the provided integer is equals to -1.

```
def: isUnbounded(i: Integer) : Boolean =
  i = -1
```

- `getScalarValue()`, `getVectorValue()` and `getMatrixValue()` operations are defined in order to ease the access to the various Value content (Listing 5.17).

```
context Value
def: getScalarValue() : ScalarValue =
  value.values->first().values->first()

def: getVectorValue() : VectorValue =
  value.values->first()

def: getMatrixValue() : MatrixValue =
  value
```

Listing 5.17: Accessors OCL operations

- We define some predicates accessors to get size configurations for a) scalar: `isScalarSize()`; b) vector: `isVectorSize()`; and c) matrix: `isMatrixSize()` in Listing 5.18.

```
context Value
def: isScalarSize() : Boolean =
  value.values->first().values->size() = 1 and value.values->size() = 1

def: isVectorSize() : Boolean =
  value.values->first().values->size() > 1 and value.values->size() = 1

def: isMatrixSize() : Boolean =
  value.values->size() > 1
```

Listing 5.18: OCL size predicates

```
context Value
def: isScalarDimension() : Boolean =
  dimensions->size() = 1 and dimensions->first() = 1

def: isVectorDimension() : Boolean =
  dimensions->size() = 1 and
  (dimensions->first() > 1 or isUnbounded(dimensions->first()))

def: isMatrixDimension() : Boolean =
  dimensions->size() = 2 and
  (dimensions->first() > 1 or isUnbounded(dimensions->first())) and
  (dimensions->last() > 1 or isUnbounded(dimensions->last()))
```

Listing 5.19: OCL dimension predicates

- Finally, the `dimension` attribute values predicates have been written for each possible combination of dimensions. They are detailed in Listing 5.19.

OCL invariants are defined on the dimension attribute according to the Value element:

- `DimensionValue` (Listing 5.20): the dimension attribute must be composed of only positive or null values
- `ScalarDimensions` (Listing 5.20): A scalar value should have a `dimension` sequence composed of only one element equals to One. The corresponding `value` reference is composed of one `MatrixValue` composed of one `VectorValue` composed of one `ScalarValue`.
- `VectorDimensions` (Listing 5.20): A vector value should have a `dimension` sequence composed of only one element different from one (equals to N). The corresponding `value` reference is composed of one `MatrixValue` composed of one `VectorValue` composed of N elements.
- `MatrixDimensions` (Listing 5.20): A matrix value should have a `dimension` table composed of two elements where the first element is different from one (the first one is equals to N and the second one to M). The corresponding `value` reference is composed of one `MatrixValue` composed of M `VectorValue` objects composed of N `ScalarValue` objects.

The `Value` object has an operation: `getDataType()` returning an `OclType` instance. Its returned value should be the common data type of the `ScalarValues` of the Value element.

Generic block specification

The blocks structure is a static structure containing the block itself, its ports, parameters and memories and a set of semantics specification elements. We model this structure in Figure 5.21. The elements of this model are inspired from the ones of Figure 4.5.

The classes presented here are the generic versions of each block structural elements. `Block`, `Port` (`InputPort` and `OutputPort`), `Parameter` and `Memory` classes have a `name` attribute inherited from the `NamedElement` class modeling REQ-1.a, REQ-2.a, REQ-3.a and REQ-4.a. The specification of parameters, ports and memories contains also a `value` attribute inherited from the `ValueElement` class referring to a Value object instance as specified in Figure 5.15. Ports specification holds informations about the kind of the port as defined in 4.2. In addition to these elements, the `Memory` class contains the `portDataTypeRef` reference. This reference refers to an `InputPort` element. This allows to specify the memory data type according to a previously defined `InputPort`. We define this relation using the `MemoryDataTypeFromInPort` OCL constraints (Listing 5.22). This reference does not constrain the dimensions of the memory as it may depend on additional informations. Indeed, we may need to store multiple values.

```
context Value
inv DimensionsValue:
  dimensions->forAll(d| d >= -1 and d <> 0)

inv ScalarDimensions:
  isScalarDimension() implies isScalarSize()

inv VectorDimensions:
  (isVectorDimension() implies isVectorSize())
  and
  ((dimensions->first() > 1 implies
    getVectorValue().values->size() = dimensions->first())
   and
   (isUnbounded(dimensions->first()) implies
    getVectorValue().values->size() > 1))

inv MatrixDimensions:
  isMatrixDimension()
  implies
  ((dimensions->last() > 1 implies
    value.values->size() = dimensions->last())
   and
   (isUnbounded(dimensions->last()) implies
    value.values->size() > 1)
   and
   (dimensions->first() > 1 implies
    getVectorValue().values->size() = dimensions->first())
   and
   (isUnbounded(dimensions->first()) implies
    getVectorValue().values->size() > 1))
```

Listing 5.20: Value dimension OCL constraints

```
context Value::getDataType() : OclType
body: getScalarValue().dataType.oclType()
```

- `MemoryDataTypeFromInPort` (Listing 5.22): If the `portDataTypeRef` reference is set then the memory value data type is the same than the one of the `InputPort` value targeted by the reference. This constraint model REQ-4.[b|c].

```
context Memory

inv MemoryDataTypeFromInPort:
  (not portDataTypeRef.oclIsUndefined())
  implies
  value.getDataType() = portDataTypeRef.value.getDataType()
```

Listing 5.22: MemoryDataTypeFromInPort OCL constraint

Each `Block` metaclass instance contains a collection of the abstract `Semantics` metaclass instances through the derived `semantics` reference. A `Semantics` element has a `compute()` operation holding the semantics definition of the specified block. This semantics is either a `CombinatorialSemantics` instance inheriting only the `compute()` operation or a `SequentialSemantics` instance inheriting the same operation and providing the `init()` and `update()` operations. Both methods are abstract to enforce their implementation. We provide additional constraints on the semantics part of a block specification:

- `CombinatorialSemantics` (Listing 5.23): If the `category` attribute (presented in Section 4.2) of the `Block` metaclass is set to `combinatorial` then every semantics specification for the block must be `CombinatorialSemantics` instances.

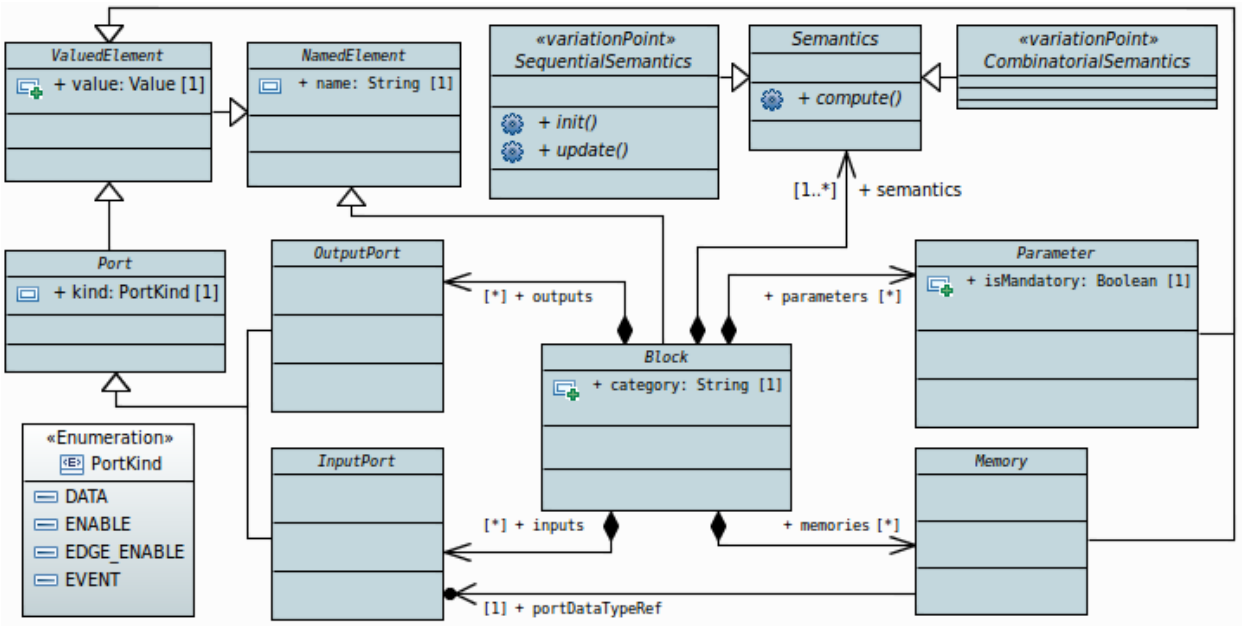


Figure 5.21: Generic block specification using UML + OCL

- `SequentialSemantics` (Listing 5.23): If the `category` attribute of the `Block` metaclass is set to `sequential` then every semantics specification for the block must be `SequentialSemantics` instances.

```
context Block

inv CombinatorialSemantics:
category = 'combinatorial' implies
  semantics->forAll(s| s.oclIsTypeOf(CombinatorialSemantics))

inv SequentialSemantics:
category = 'sequential' implies
  semantics->forAll(s| s.oclIsTypeOf(SequentialSemantics))
```

Listing 5.23: Block category OCL constraint

5.4.2 Injecting variability into UML

Whereas fUML is a good modeling language, it does not provide a dedicated variability management. In that sense, it does not allow to model a complete system with variation points and then to extract from a provided configuration of the system the corresponding fUML elements and constraints.

Some extensions of the classical UML language have been defined in this purpose. To our knowledge, the most advanced activities about the integration of variability capabilities in the UML are those from [162] and [152] where variability management is integrated in the UML through the definition of a UML profile for class diagrams and sequence diagrams – more recent works are applied to other kinds of diagrams like use case [31] or activity [74] diagrams. Management of variability for activity diagram is very handy as it adds to the structural variability management provided in class diagrams the ability to express behavioral variability.

SMarty variability UML profile

In order to model the variability in UML models, we have used the SMarty UML profile [81]. Our work is based on the implementation provided in the github website of the project [2]. We have extended the

[2] https://github.com/edipofederle/AGM_1

variability profile. Figure 5.24 provide an incomplete graphical overview.

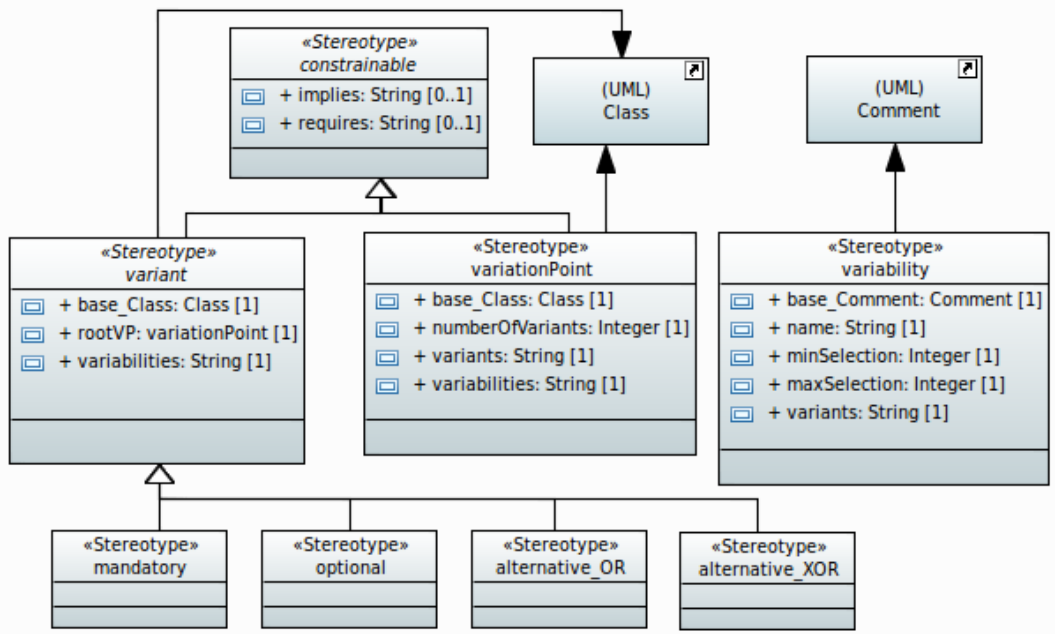


Figure 5.24: SMarty variability UML profile

The profile main element is the `variationPoint` defining a variation point in the variation tree. It applies on an UML class – `base_Class` relation; contains a certain number of `variant` – `numberOfVariants` integer attribute; a comma separated list of `variant` names that is the list of sub-features of the `variationPoint` – `variants` string attribute and; a comma separated list of `variabilities` element names containing the relationship informations – `variabilities` string attribute. The black arrow going from `variationPoint` to (UML) `Class` is the extension edge specifying the applicability of the stereotype. In this case, it is possible to apply the `variationPoint` stereotype to UML `Class`.

Each `variationPoint` stereotyped UML `Class` is the root feature for a `variant` stereotyped UML `Class`. `variant` elements have a reference to their container `variationPoint` – `rootVP` and; a comma separated list of `variabilities` element names containing the relationship informations of its parent `variantPoint` – `variabilities` string attribute. This stereotype is realized as four stereotypes: `mandatory`, `optional`, `alternative_OR` and `alternative_XOR` respectively modeling the mandatory, optional, or and alternative feature modeling relationships.

We have extended the original stereotype with an additional abstract class: `constrainable`. Both `variant` and `variationPoint` inherits from the abstract `constrainable` stereotype. It provides `implies` and `requires` strings containing a list of comma separated `variant` elements names. It allows to express feature modeling-like cross tree constraints (implies and requires) on any UML class stereotyped with `constrainable`.

The `variability` stereotype applies on UML `Comment` elements. It is named (`name` string attribute) and provides a cardinality relation – in the sense of cardinalities in feature models [54, 130] – implemented with its `minSelection` and `maxSelection` integer attributes. between the list of `variant` elements provided in the `variants` comma separated string list of `variant` stereotyped elements.

SMarty variability UML profile use

In order to define variability on a domain UML model by relying on the SMarty UML profile, it is mandatory to first define the `variationPoint` elements on the domain UML model elements. For each identified `variationPoint` one needs to define its `variant` as UML classes. The attribute `variants` of

the stereotyped `variationPoint` element must contains the complete list of names of the identified `variant` elements. By analogy with feature models, the `variationPoint` elements is the parent feature of its identified `variant` elements. The relation between them depends on the concrete stereotype used for the `variant` elements.

We use the previously defined generic UML model and the SMarty variability profile as a basis for the specification of blocks.

5.4.3 Profiled UML + OCL specification for *MinMax* block

MinMaxBlock class

The specification starts with a simple class: `MinMaxBlock`. This class is the main element of the specification, all the block specification elements should directly or indirectly be related to this Class. The diagram containing the specification is provided in Figure 5.26. The `MinMaxBlock` class implements the `Block` class from the generic specification of blocks (Section 5.4.1). The name informations required in REQ-1.a is the value of the inherited `name` attribute. We specify its value using the OCL constraint specified in Listing 5.25. This class is stereotyped as a `variationPoint`. Its variability is expressed in `semanticsVariability` comment.

```
context MinMaxBlock
inv BlockName:
  name = 'MinMax'
```

Listing 5.25: BlockName OCL constraint

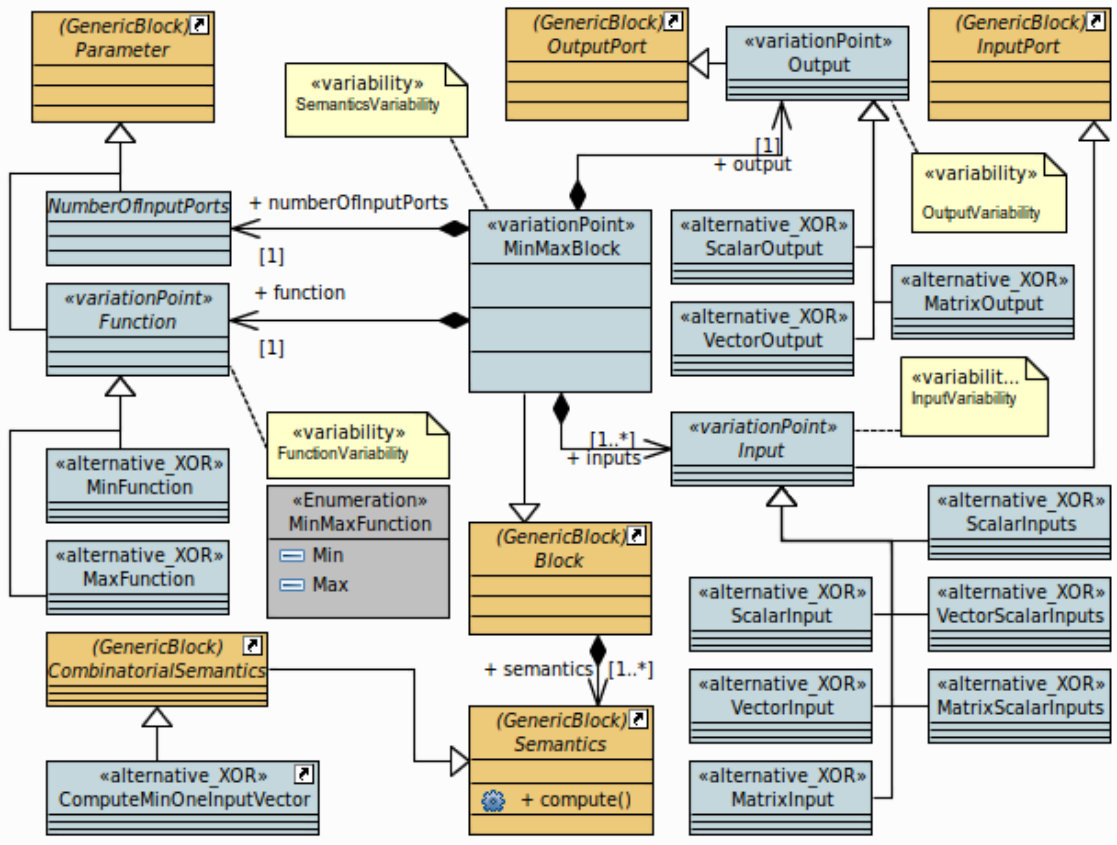


Figure 5.26: MinMax block specification using UML + variability profile + OCL

Input and Output ports specification

The `MinMaxBlock` class has two outgoing references directed to its right side: `inputs` that is a non-empty sequence of instances of the `Input` type; and `output` that is one instance of type `Output`. These references holds the specification for the set of input/output ports. The `Input` abstract class inherits from the generic `InputPort` class and the `Output` abstract class inherits from the generic `OutputPort` class.

`Input` class has six concrete realisations: `ScalarInput`, `VectorInput`, `MatrixInput` and `ScalarInputs`, `VectorInputs`, `MatrixInputs`. The first three are for block configurations with a single input port, the last three are for multiple input ports. It is stereotyped as `variationPoint`, the related relationship of cardinality one is detailed in the `inputVariability` comment; the result of these constraints is to make these the six concrete classes exclusives. Each of these concrete class is constrained with OCL constraints to provide the information required in REQ-1.b, REQ-2.b, REQ-2.c and REQ-5.b.

- `ScalarInput`: An input of this type should be a scalar. It is specified in constraint `ScalarInputValue` in Listing 5.27.
- `VectorInput`: An input of this type should be a vector. It is specified in constraint `VectorInputValue` in Listing 5.27.
- `MatrixInput`: An input of this type should be a matrix. it is specified in constraint `MatrixInputValue` in Listing 5.27.

```
context ScalarInput
inv ScalarInputValue:
  value.isScalarDimension()

context VectorInput
inv VectorInputValue:
  value.isVectorDimension()

context MatrixInput
inv MatrixinputValue:
  value.isMatrixDimension()
```

Listing 5.27: One input value OCL constraints

- `ScalarInputs`: For each of these inputs, their values are scalars. It is specified in constraint `ScalarInputsValue` in Listing 5.28.
- `VectorInputs`: For each of these inputs, their values are either scalars or vectors. It is specified in constraint `VectorInputsValue` in Listing 5.28.
- `MatrixInputs`: For each of these inputs, their values are either scalars or matrices. It is specified in constraint `MatrixInputsValue` in Listing 5.28.

Two OCL constraints (`InputDT` and `OutputDTRelationToInputDT` in Listing 5.29) have been specified to restrict the allowed Primitive data types of the inputs/outputs of the *MinMax* block. The first constraint targets the allowed inputs data types (in this case, the inputs can be of `boolean`, `int` or `double` data type) whereas the second one specifies the output data type according to the combinations of input data types. These constraints model REQ-2.b, REQ-2.c and REQ-5.a.

The output port in the block specification is modelled using the abstract `Output` class. It is implemented with three concrete classes: `ScalarOutput`, `VectorOutput` and `MatrixOutput`. Each of these three classes models one possible variant of the output dimensionality. The `variationPoint`

```
context ScalarInputs
inv ScalarInputsValue:
  oclContainer().inputs->forAll(i1 | i1.isScalarDimension())

context VectorInputs
inv VectorInputsValue:
  oclContainer().inputs->forAll(i1 |
    i1.value.dimensions->size() = 1
  ) and (
  let vectorInputs = oclContainer().inputs->select(i|
    (i.value.dimensions->first() > 1)
  ) in
    vectorInputs->size() > 1 implies
      vectorInputs->forAll(i1,i2|
        i1.value.dimensions->first() = i2.value.dimensions->first()
      )
  )

context MatrixInputs
inv MatrixInputsValue:
  (let scalarInputs = oclContainer().inputs->select(i|
    i.value.dimensions->size() = 1
  ) in
  scalarInputs->forAll(i |
    i.value.dimensions->first() = 1
  )) and (
  let matrixInputs = oclContainer().inputs->select(i|
    i.value.dimensions->size() = 2
  ) in
    matrixInputs->forAll(i1,i2|
      (i1.value.dimensions->first() = i2.value.dimensions->first()) and
      (i1.value.dimensions->last() = i2.value.dimensions->last())
    )
  )
```

Listing 5.28: Multiple input values OCL constraints

stereotype has been applied to the `Output` class, each of the three implemented classes have been stereotyped as `alternative_XOR` and a `variability` comment has been added constraining their relationship as an alternative. The following OCL constraints have been written constraining these three classes. They model REQ-2.[b|c].

- `ScalarOutput`: The contained value is a scalar value. It is constrained by the OCL constaint `ScalarOutputValue` provided in Listing 5.30.
- `VectorOutput`: The contained value is a vector value. It is constrained by the OCL constraint `VectorOutputValue` provided in Listing 5.30.
- `MatrixOutput`: The contained value is a matrix value. It is constrained by the OCL constraint `MatrixOutputValue` provided in Listing 5.30.

Parameters specification

The `MinMaxBlock` class has two references to classes inheriting from the generic Parameter class: `NumberOfInputsPorts` and `Function`. For each of these classes, we provide a structural specification as detailed in Figure 5.26 and OCL constraints for additionally constraining the values of the parameters.

The first parameter allows to set the number of input ports for the block. It is modelled as the `NumberOfInputPorts` class. The data type and dimensions of this parameter are constrained by the `NumberOfInputPortsParameterDT` OCL constraint and its value is constrained by the `NumberOfInputPortsValue` OCL constraint (Listing 5.31). A final OCL constraint: `NumberOfInputs` (same Listing) states that its value influences the number of Input class instances in

```
context MinMaxBlock
inv InputDT:
  inputs->forAll(i| i.value.getDataType().oclIsTypeOf(boolean)
              or
              i.value.getDataType().oclIsTypeOf(int)
              or
              i.value.getDataType().oclIsTypeOf(double))

inv OutputDTRelationToInputDT:
  (inputs->forAll(i| i.value.getDataType().oclIsTypeOf(boolean))
   implies output.value.getDataType().oclIsTypeOf(boolean)
  ) and
  (output.value.getDataType().oclIsTypeOf(boolean)
   implies inputs->forAll(i| i.value.getDataType().oclIsTypeOf(boolean))
  ) and (
   (inputs->forAll(i| i.value.getDataType().oclIsTypeOf(boolean) or
                i.value.getDataType().oclIsTypeOf(int)) and
   inputs->exists(i| i.value.getDataType().oclIsTypeOf(int)))
   implies
   output.value.getDataType().oclIsTypeOf(int)
  ) and (
  (inputs->forAll(i| i.value.getDataType().oclIsTypeOf(boolean) or
               i.value.getDataType().oclIsTypeOf(int) or
               i.value.getDataType().oclIsTypeOf(double)) and
   inputs->exists(i| i.value.getDataType().oclIsTypeOf(double)))
   implies
   output.value.getDataType().oclIsTypeOf(double))
```

Listing 5.29: Allowed data type combinations OCL constraints

```
context ScalarOutput
inv ScalarOutputValue:
  value.isScalarDimension()

context VectorOutput
inv VectorOutputValue:
  value.isVectorDimension()

context MatrixOutput
inv MatrixOutputValue:
  value.isMatrixDimension()
```

Listing 5.30: Output value dimensions OCL constraint

the `inputs` reference. The first constraints model REQ-3.b and REQ-3.c, whereas the two others model respectively REQ-5.b and REQ-5.a.

The second parameter holds the computation algorithm choice. This parameter is of type `MinMaxFunction` which is an Enumeration (constrained with `FunctionParameterDT` OCL constraint (Listing 5.32). The `Function` class is an abstract class with two concrete realisations: `MinFunction` and `MaxFunction`. The allowed values for these classes are respectively constrained in `MinFunctionValue` and `MaxFunctionValue` (same Listing). These constraints model REQ-3.a; REQ-3.b and REQ-3.c.

Semantics specification

According to REQ-6.a and REQ-6.b, the semantics specification for a block should be provided as a formal operation and its variability should be modelled according to inputs, outputs, parameters and memories configurations.

Declaration of a block semantics is done through the realisation of the `CombinatorialSemantics` or `SequentialSemantics` classes. Declaration of semantics in UML class diagrams is done by adding operations in a class. In our setting, each semantics variation point specification is provided as a class inheriting either from `CombinatorialSemantics` or `SequentialSemantics`. Each `Semantics` class is attached to the `MinMaxBlock` through the `semantics` relation and the inheritance relation as depicted

```
context NumberOfInputPorts
inv NumberOfInputPortsParameterDT:
  value.getDataType().oclIsKindOf(Integer) and
  value.isScalarDimension()

inv NumberOfInputPortsParameterValue:
  value.getScalarValue().oclAsType(Integer) >= 1

context MinMaxBlock
inv NumberOfInputsRelationToInput:
  numberOfInputPorts.value.getScalarValue().oclAsType(Integer) = inputs->size()
```

Listing 5.31: *NumberOfInputPorts* parameter OCL constraints

```
context Function
inv FunctionParameterDT:
  value.getDataType().oclIsKindOf(MinMaxFunction) and
  value.isScalarDimension()

context MinFunction
inv MinFunctionValue:
  value.getScalarValue().oclAsType(MinMaxFunction) = MinMaxFunction::Min

context MaxFunction
inv MaxFunctionValue:
  value.getScalarValue().oclAsType(MinMaxFunction) = MinMaxFunction::Max
```

Listing 5.32: `Function` parameter OCL constraint

in Figure 5.26.

The specification of the block semantics is then done by providing for each `init()`, `compute()` and `update()` operations their pre-conditions, post-conditions and body content. Pre and post conditions allows to specify the axiomatic semantics of the semantics variation points of the block whereas the body content specifies an operational semantics.

Semantics specification can be provided in class diagrams using several means:

- OCL constraints: it allows for the expression of complex constructs that may be useful for pre and post condition expressions but might be of difficult use for body expressions as it lacks imperative code constructs that are more natural for most users.
- UML activity diagrams are well suited for the specification of algorithms. The graphical concrete syntax does not allow for the specification of complex algorithms as it suffer from its verbosity.
- Textual notations like the one provided for the ALF action language. Targeting fUML allows, among others capabilities, the specification of activity diagrams based on a textual syntax. This overcomes the previous limitation but it actually lacks from implementation integration in standard UML modeling environments.
- Textual action languages like C++ or Java. Constraints can be specified using standard languages allowing to express algorithm specification using well known syntax and semantics. Using these adds a difficulty as they are disconnected from the modeling world and thus additional verification activities must be conducted.

All the previously provided semantics specification means relies on structured languages with a defined semantics. As all these means suffer from flaws regarding our targeted use, adaptations are required in order for them to be usable.

Semantics variability management

Regarding variability, each `Semantics` class is decorated with the `alternative_XOR` stereotype, and related to the `MinMaxBlock` class as its `variationPoint`. The `variability` comment specifies a

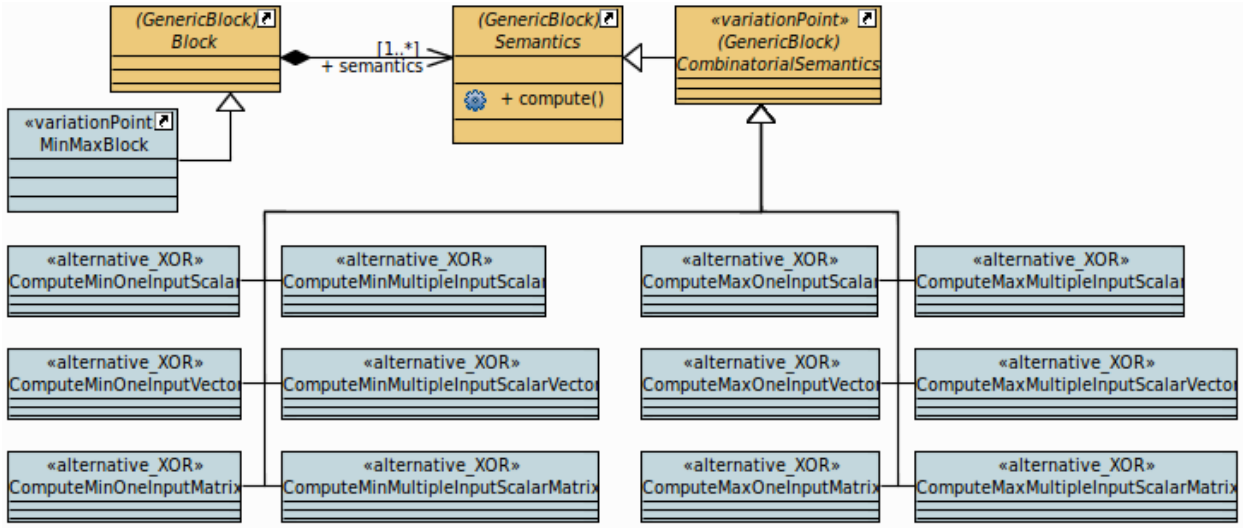


Figure 5.33: Semantics variants definition for the *MinMax* block

Semantics variation point class	Required variants	Implied variants
ComputeMinOneInputScalar	MinFunction, ScalarInput	ScalarOutput
ComputeMinOneInputVector	MinFunction, VectorInput	ScalarOutput
ComputeMinOneInputMatrix	MinFunction, MatrixInput	ScalarOutput
ComputeMinMultipleInputsScalar	MinFunction, ScalarInputs	ScalarOutput
ComputeMinMultipleInputsSclarVector	MinFunction, VectorInputs	VectorOutput
ComputeMinMultipleInputsScalarMatrix	MinFunction, MatrixInputs	MatrixOutput
ComputeMaxOneInputScalar	MaxFunction, ScalarInput	ScalarOutput
ComputeMaxOneInputVector	MaxFunction, VectorInput	ScalarOutput
ComputeMaxOneInputMatrix	MaxFunction, MatrixInput	ScalarOutput
ComputeMaxMultipleInputsScalar	MaxFunction, ScalarInputs	ScalarOutput
ComputeMaxMultipleInputsScalarVector	MaxFunction, VectorInputs	VectorOutput
ComputeMaxMultipleInputsScalarMatrix	MaxFunction, MatrixInputs	MatrixOutput

Table 5.34: Semantics variation point cross tree constraints

cardinality of 1 between each of them. Figure 5.33 show the complete set of semantics specification for the *MinMax* block.

This semantics configuration of the block supposes a certain configuration of the inputs, outputs and parameters of the block. Our provided extension of the SMarty profile including the `implies` and `requires` attributes allows to specify cross-tree constraints between concrete semantics definition classes and input, output and parameter definition classes (Table 5.34).

Semantics specification example

The previously presented dependencies between semantics variation points and input, output and parameter variants allow to extract a set of OCL constraints as a part of the pre and post conditions for the semantics operations. These constraints tackle the structural part of the specification as the `requires` relation provides input and parameter values and data type constraints and the `implies` relation provides the output values and data types constraints.

According to the previous elements, we choose to specify the pre and post conditions for the axiomatic semantics using OCL constraints and the operation alone using standard Java syntax. At the time of the experiment, no ALF implementations integrated in a modeling environment were available, it would however be an interesting language for our purpose. It is not mandatory to provide both axiomatic and opera-

tional semantics for the block but it may allow formal verification of the specified semantics as required in REQ-8. The complete set of all semantics definition classes model REQ-6.b.

We provide in the following an example specification for the `ComputeMinOneInputVector` semantics class. *MinMax* block.

- `computeMinOneInputVector_Pre` (Listing 5.35): the precondition in the axiomatic semantics definition. If this constraint is satisfied on a block instance then the block instance semantics should be the one specified in `computeMinOneInputVector_Body` (provided below). It is worth noting that the content of this constraint is already present in the variability specification cross tree constraints. Indeed, all the components of the pre-condition expression of Listing 5.35 are specified in the `MinFunction` and `VectorInput` related OCL invariants (Listings 5.32 and 5.28) and the cross tree constraints specified for the `ComputeMinOneInputVector` semantics variation point in Table 5.34 provides the link between semantics and structural elements.
- `computeMinOneInputVector_Post` (Listing 5.35): the postcondition of the axiomatic semantics. This constraint contains informations on the output of the block. In this case, it states that the output should be a scalar and its value is the minimum of the input vector components. It is worth noting that only the final part of the constraint (last three lines) is not provided by the variability specification through the cross tree constraints. The first part of the constraint expression is implies by the implies variants for the semantics variation point as defined in Table 5.34.

```
context ComputeMinOneInputVector::compute
pre:
  function = MinMaxFunction::Min and
  inputs->size() = 1 and
  inputs->first().value.isVectorDimension()

post:
  output.value.isScalarDimension() and
  inputs->first().value.getVectorValue().values->forAll(v|
    v->oclAsType(Real) >= output.value.getScalarValue()->oclAsType(Real)
  )
```

Listing 5.35: Pre and post conditions for computeMinOneInputVector axiomatic semantics using OCL

- `computeMinOneInputVector_Body` (Listing 5.36): the operational semantics of the block specified as a JAVA method. This operation assumes that we have previously defined the mandatory elements among which are the constructors for ScalarValue, VectorValue and MatrixValue, the list data type and finally the comparison operation (<=) on ScalarValue. The operational semantics purpose is to define the block output port value(s) according to the input port(s) value(s).

```
void computeMinOneInputVector_Body (Block block){
  List<ScalarValue> inputVectorValues =
    block.inputs.get(0).value.value.values.get(0).values;
  ScalarValue min = inputVectorValues.get(0);
  for (int i=1; i < inputVectorValues.size(); i++){
    if (inputVectorValues.get(i) <= min){
      min = inputVectorValues.get(i);
    }
  }
  VectorValue outVector = new VectorValue();
  outVector.addScalar(min);
  MatrixValue outMatrix = new MatrixValue();
  outMatrix.addVector(outVector);
  block.output.value = outMatrix;
```

Listing 5.36: computeMinOneInputVector_Body semantics expressed using a JAVA method

Semantics specification elements such as those given in these annotations provides the mandatory informations required in REQ-6.a and REQ-6.b. They also allow for the testing of the specification as it is possible to compare the specified operational semantics computation with the result of the simulation with SIMULINK. It is also possible to verify the axiomatization as the output resulting simulation values should verify the specification post-conditions.

SPECIFICATION VERIFICATION CONSIDERATIONS

REQ-7.a is partially automatically verified as UML diagrams (in general and in our context the UML class diagram) includes a set of structural correctness rules defined in the UML specification [120] for the verification of structural consistency of the model elements.

Regarding the other tooling requirements expressed in REQ-7.b, REQ-7.c and REQ-8, it is mandatory to develop an automatic verifications mechanism.

The first two criteria must be expressed according to the set of all the defined semantics variation points: on the one hand, the **completeness** criterion (REQ-7.b) aims at assessing that all the possible structural features variants combinations have been taken into account in the specification; on the other hand, the **disjointness** criterion (REQ-7.c) aims at assessing that all the defined semantics are unique and thus that they apply on different configurations of structural features variation points. If we declare *nbSpec* as the number of semantics variants for a block. From the ith semantics of the block: $computeSpec_Pre_i$, the pre-conditions (Required variant's constraints in Table 5.34); and $computeSpec_Post_i$, the post-conditions (Implied variants in Table 5.34); we then express the **completeness** criterion (REQ-7.b) as (5.1) and the **disjointness** criterion (REQ-7.c) as (5.2).

$$\bigwedge_{0 \leq i < nbSpec} computeSpec_Pre_i \tag{5.1}$$

$$\forall i, j, 0 \leq i < nbSpec, 0 \leq j < nbSpec, i \neq j \Rightarrow \neg \left(computeSpec_Pre_i \wedge computeSpec_Pre_j \right) \tag{5.2}$$

REQ-8 requires the correctness verification of the complete block semantics. As in the specification, we express the axiomatic specification of the semantics as OCL constraints and the operational specification as JAVA code, the expected verification can be expressed as Hoare triples. We provide an example of such Hoare triple for the compute semantics of the ith specification configuration of a block in (5.3). Such a verification should be provided for the three phases of the block semantics.

$$\forall i, 0 \leq i < nbSpec, \{computeSpec_Pre_i\}\ computeSpec_Body_i\ \{computeSpec_Post_i\} \tag{5.3}$$

The verification of the block semantics should then be done by proving the correct behavior of the program provided in Listing 5.37. These verifications can be done by relying on a mapping from the UML models with OCL constraints to a convenient formal domain [17] and from the mapping of JAVA (or ALF) programs to a convenient formal domain where automatic verification can be done as with the KRAKATOA tool or the KEY platform[3] [13]. An example of such a formal domain is the SMT solvers one where logical expressions and programs can be expressed and may be verified automatically. Such transformations and the related verifications will be the subject of Chapter 7.

5.4.4 PROFILED UML + OCL SPECIFICATION FOR *DELAY* BLOCK

We will detail here the *Delay* block specification using the previous approach. This will allow us to cover the specificities of the *Delay* block that are not present in the *MinMax* block.

The specification starts with a simple class: `DelayBlock`. This class is the main element of the UML specification. The block input, output, parameter and memory specification are similar to the one provided

[3] http://www.key-project.org/

```
assume(initSpec_Pre);
initSpec_Body();
assert(initSpec_Post);

while (true){
  assume(computeSpec_Pre);
  computeSpec_Body();
  assert(computeSpec_Post);

  assume(updateSpec_Pre);
  updateSpec_Body();
  assert(updateSpec_Post);
}
```

Listing 5.37: Generic block semantics expression

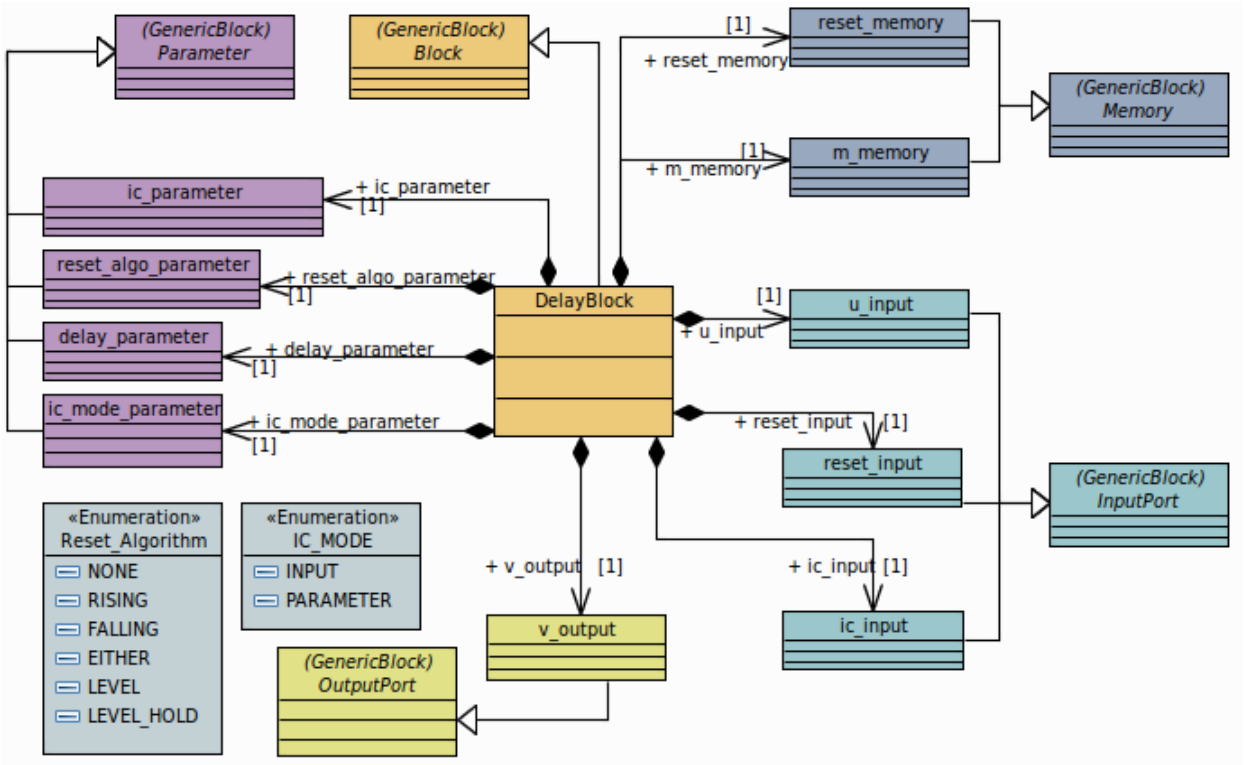


Figure 5.38: Delay block specification using the UML

for the *MinMax* block but adapted to the *Delay* block. A global view of the UML specification for the *Delay* block is provided in Figure 5.38.

Memories specification

The *Delay* block is a sequential block, it therefore uses some memories in order to compute its output value. It also uses a memory in its configuration when the `reset_algo` parameter takes the appropriate value. We need to provide the required specification elements for these two memories in order to complete their specification.

The `m_memory memory` is storing values from the `u_input` input port and provides the computed result as the `v_output` port value. Its allowed data types and dimensions are provided through OCL constraints given in Listing 5.39:

- The `M_Memory_PortDataTypeRef` OCL invariant constrains the `portDataTypeRef` reference to the `u_input` and the memory value data type is constrained to be the same as the one of the input value in the `M_Memory_DT` OCL invariant.

- The `M_Memory_Dimensions_SimpleDelay` and `M_Memory_Dimensions_MultipleDelay` constrain the memory value dimensions according to the `portDataTypeRef` value dimension and the `delay_parameter` value. It is indeed mandatory to add a new dimension to the memory if the `delay_parameter` value is greater than 1.

```
context DelayBlock
inv M_Memory_PortDataTypeRef:
  m_memory.portDataTypeRef = u_input

context m_memory
inv M_Memory_DT:
  value.getDataType() = portDataTypeRef.value.getDataType()

inv M_Memory_Dimensions_SimpleDelay:
  (oclContainer().delay_parameter.value.getScalarValue().oclAsType(Integer) = 1)
  implies
  (let refDimValueValue = portDataTypeRef.value.dimensions) in
    ((value.dimensions->size() = 1) implies
     value.dimensions->first() = refDimValue->first()) and
     ((value.dimensions->size() = 2) implies
      ((value.dimensions->first() = refDimValue->first()) and
       (value.dimensions->last() = refDimValue->last()))))

inv M_Memory_Dimensions_MultipleDelay:
  (oclContainer().delay_parameter.value.getScalarValue().oclAsType(Integer) > 1)
  implies
  (let refDimValueValue = portDataTypeRef.value.dimensions->append(
    delayValue.oclAsType(Integer)
  ) in
    ((value.dimensions->size() = 1) implies
     value.dimensions->first() = refDimValue->first()) and
     ((value.dimensions->size() = 2) implies
      ((value.dimensions->first() = refDimValue->first()) and
       (value.dimensions->last() = refDimValue->last()))))
```

Listing 5.39: M_Memory data type and dimensions constraining

The same methodology is used for the `reset_memory` memory. This time its data type and dimension are computed according to the `reset_input` input port. OCL constraints in Listing 5.40 provide these elements. We enforce the value of the `portDataTypeRef` reference value to the `u_input` input port element (OCL invariant `M_Memory_PortDataTypeRef`). We define the memory data type with the `M_Memory_DT` OCL invariant and the final OCL invariant: `M_Memory_Dimensions`

```
context DelayBlock
inv Reset_Memory_portDataTypeRef:
  reset_memory.portDataTypeRef = reset_input

context reset_memory
inv Reset_Memory_DT:
  value.getDataType() = portDataTypeRef.value.getDataType()

inv Reset_Memory_Dimensions:
  ((value.dimensions->size() = 1) implies
   value.dimensions->first() = portDataTypeRef.dimensions->first()) and
  ((value.dimensions->size() = 2) implies
    ((value.dimensions->first() = portDataTypeRef.dimensions->first()) and
    (value.dimensions->last() = portDataTypeRef.dimensions->last())))
```

Listing 5.40: Reset_Memory data type and dimensions constraining

5.4.5 Choices made regarding UML modeling

We detailed here one possible specification for the *MinMax* and *Delay* blocks using UML models. In order to do this specification, we needed to do some design choices as there is no recommended, standard and generic pattern for the writing of metamodels (data diagram representing languages).

The main UML model containing the generic specification for a block provided in Figure 5.21 (page 61) is a simple structural decomposition of a block with its content. This makes the model natural and easy to apprehend. The specialisation for the *MinMax* and *Delay* blocks provided in Figures 5.26 (page 63) and 5.38 (page 71) as long as the related semantics decompositions follow this same approach.

To our understanding, the structural approach is the beset for the specification of blocks as it allows to build, understand and maintain easily specifications. These characteristics are of primary interest for us as we want to ensure the specification correctness which is more easily achieved with a simple and straightforward modeling approach.

5.4.6 Limitations of the UML + profile + OCL specification approach

UML + OCL allows an accurate modeling of the block structural components. The main specification problems is the inability to efficiently express the variability of the block. Block structural variability may lead to at least a quadratic number of class definitions for each of which OCL constraints needs to be associated. This number of elements may increase the risk of mistakes in the writing/reading of the specification and make the management of variability more difficult (REQ-9).

Each block semantics variation point is specified with a separate UML operation in order to ensure their independent verification and the conditions for the execution of semantics operations needs to be added as pre-conditions of these operations. Each pre-condition should contain all the conditions for the semantics execution. This conditions will not be easy to build especially when the block complexity and semantics variability increase as the decomposition of the block specification features and the semantics variants definition and their relation to the structural features variants are done manually. Furthermore, this is going against REQ-9.

Regarding the automatic verification of UML + profile + OCL specification, the specifier is free to write it using any UML artifact and any UML diagram. Fortunately for the designer, the UML formalism is wide and expressive. Unfortunately for the verifier, it makes the automatic verification more complex as all the UML constructs must be taken into account. Regarding certification activities, this is again going against feasibility as it multiplies the number of verification activities and certification data to provide.

A methodology is then mandatory to restrict the number of allowed UML constructs and enforce the use of adequate patterns that can then be fed to tools for automatic analysis. This advocates for the use of a subset of UML with specific patterns focused on our specification purpose. Restrictions of the number of allowed UML constructs is difficult to do as each UML user has its own modeling habits, it is therefore not convenient for users to work with a subset of UML. These constraints usually leads to the definition of DSML oriented approaches [69, 107].

All these considerations are going against the use of this approach for the specification of blocks. We will now experiment the use of a SPLE approach and of their respective models for the specification of blocks.

5.5 SPLE for the specification of blocks

As we saw in 3.2.4, SPLE is a dedicated methodology for the management of software variability relying on a first phase on a detailed domain analysis leading to the definition of feature models. The feature models are then refined by giving a detailed definition of the identified features in the form of either specification artifacts like models or development artifacts like source code.

The use of SPLE methodology is the most advised formalism in the purpose of analysing and/or designing software exposing a potential large number of structural and behavioral variation points. Indeed it provides a convenient, rather simple, formally defined and tooled approach. We will illustrate the use

of SPLE, show its abilities and drawbacks and conclude on its usability for the specification of blocks. We will apply SPLE methodology to provide a specification for the *MinMax* and *Delay* blocks and show the applicability of the SPLE approach to our needs.

5.5.1 SPLE specification approach

Variability modeling of the possible *MinMax* block semantics has been experimented using SPLE techniques. The first step was the choice of a strategy for the feature modeling analysis of the block to be specified. There are three possible entry points for a specification using SPLE as advocated by Thüm in [145]: (a) feature-based analysis; (b) family-based analysis and; (c) product-based analysis. One of these three analysis is first chosen as an entry point of the analysis activity which is then conducted and extended by a second orientation point analysis. This allow first building a basis for the analysis and then completing it by analysing the problem using any other point of view.

Analysis of blocks in order to extract a feature model can be based on the same approaches. As it may be very difficult to know at the time of the specification the full set of valid block configuration (it is the purpose of writing the specification), a product-based analysis cannot be the starting point of the specification. Family-based analysis needs a knowledge of the structural and semantics invariant properties of the specified blocks in order to extract the families of properties present in the specification. Whereas some of these invariants might be known, it is not obvious to ensure that our knowledge of the block is complete enough to get all of these invariants. Feature-based analysis of the blocks on the other side can be done after a small analysis of the block parameters and inputs as their number is well known and their purpose can easily be found. We thus selected the feature-based approach.

Regarding the specification of dataflow block, we have already identified the features that cause the product variation (input ports number, input port dimensions and function parameters). It seems natural to use a feature-product-based analysis where we independently analyse the features of the product line and then we supplement this analysis by detailing the products using cross tree constraints.

5.5.2 SPLE specification of the *MinMax* block

We start our analysis by defining a feature model that will contain all the product line variation points. These variation points will be the potential input ports, output ports and parameters of the block; the semantics variation points will be modeled in order to attach a structural configuration for the block. We provide in Figure 5.41 such a feature model for the *MinMax* block specification.

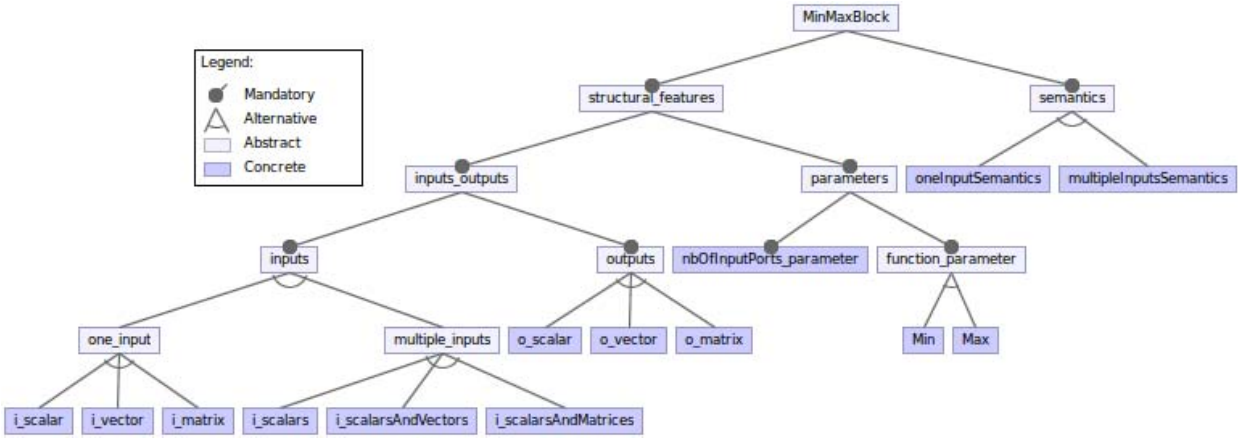


Figure 5.41: A feature model for the *MinMax* block structure and semantics

Detailed FM elements specification

The root feature of Figure 5.41, is the *MinMax* block itself, it is named after the specified block (REQ-1.a). Beneath the root feature, we provide all the mandatory features and their children features:

- `structural_features`: an abstract feature holding all the structural features of the block specification (REQ-2.a and REQ-3.a).
- `i_scalar`, `i_vector` and `i_matrix`: these three alternative features model the input port of the block, in this setting these is only one input port. Each feature models a different input port dimension (REQ-2.b and REQ-2.c).
- `i_scalars`, `i_scalarsAndVectors` and `i_scalarsAndMatrices`: these three alternative features model the input ports of the block. In this setting there are multiple input ports (REQ-2.b and REQ-2.c).
- `o_scalar`, `o_vector`, `o_matrix`: these three alternative features model the output port of the block (REQ-2.b and REQ-2.c).
- `nbOfInputPorts_parameter`: a mandatory feature modeling the `NumberOfInputPorts` parameter (REQ-3.a).
- `function_parameter`: a mandatory abstract feature modeling the `MinMaxFunction` parameter. Its two sub-feature model its different allowed values – `Min` and `Max` (REQ-3.a, REQ-3.b and REQ-3.c as it also specifies the data type and dimensions). It is not mandatory to define the two sub-features as it would be possible to define them through an enumerate type. However in this case, it is interesting to add them as the enumeration values are part of the variability of the specified block.
- `semantics`: a mandatory abstract feature holding the block semantics variation point specifications. Each semantics variation point is specified through a set of cross tree constraints. Each semantics description feature will be explicitly provided in a refinement specification phase.
  - `oneInputSemantics`: models the semantics of the block when there is only one input port. It is constrained with the 5.4 cross tree constraints.

$$\begin{aligned} oneInputSemantics &\Leftrightarrow one_input \\ one_input &\Rightarrow o_scalar \end{aligned} \tag{5.4}$$

  - `multipleInputsSemantics`: models the semantics of the block when there are multiple input ports. It is constrained with the 5.5 cross tree constraints.

$$\begin{aligned} multipleInputsSemantics &\Leftrightarrow multiple_inputs \\ i_scalars &\Rightarrow o_scalar \\ i_scalarsAndVectors &\Rightarrow o_vector \\ i_scalarsAndMatrices &\Rightarrow o_matrix \end{aligned} \tag{5.5}$$

Analysis of the feature model

Computations can be done on feature models providing metrics for the evaluation of the specification complexity among which is the calculus of the number of allowed products in the product line. In the model of Figure 5.41, the result of this computation is twelve. Meaning that there are twelve different configurations to specify for this block (the same number as in the UML + profile + OCL specification). The number of semantic pre/post conditions to write can be bigger than that because of the allowed data types of the inputs that may require to specify multiple times the same semantics with the use of different operations (for example for the comparison of values).

Multiplicities regarding the number of inputs that can be provided using cardinality in feature models. This provides the ability to specify the number of input ports (REQ-5.b).

Such feature model allows to structure the specification of a block using features and to express the dependencies between semantics variation points and structural features configuration as expected in REQ-9.

5.5.3 SPLE specification of the *Delay* block

In the same setting as for the *MinMax* block SPLE specification, we provide here the one for the *Delay* block. We provide in Figure 5.42 a feature model for its specification.

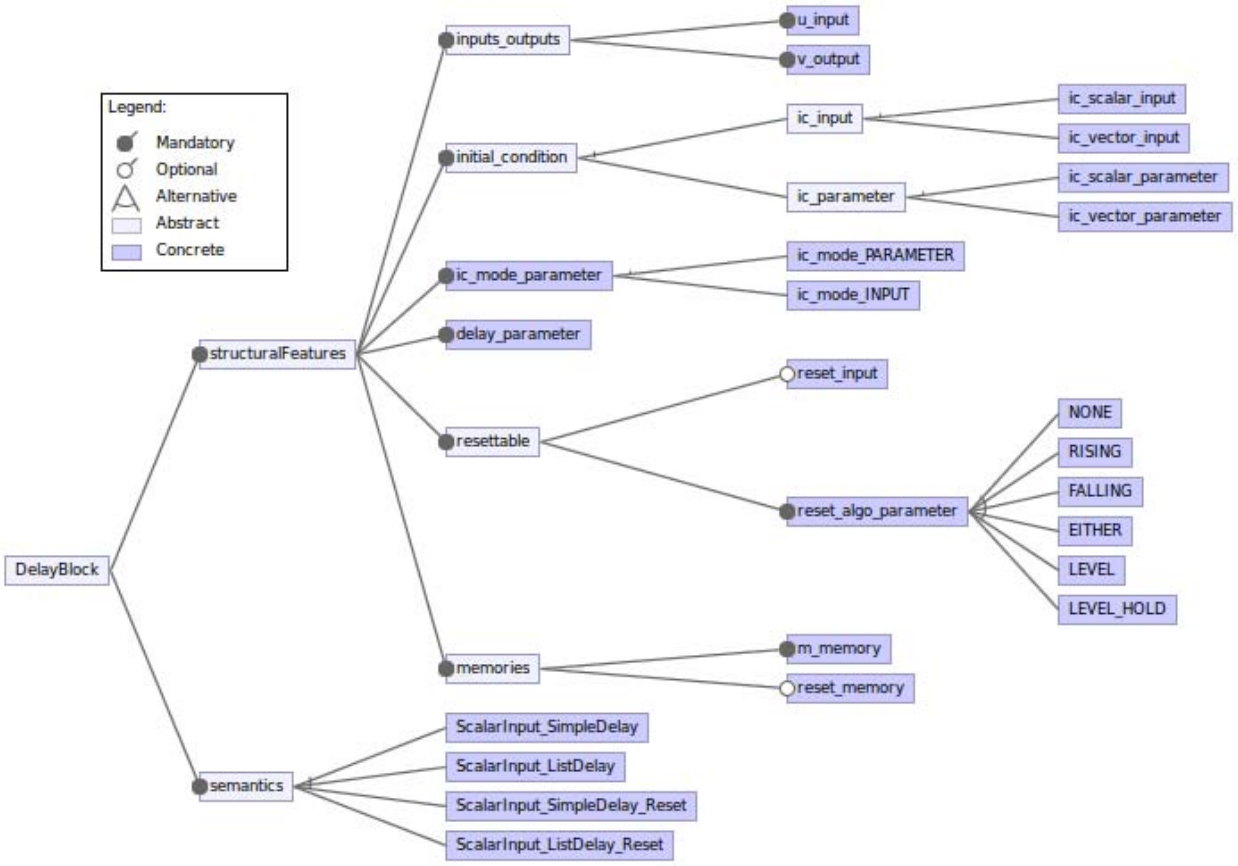


Figure 5.42: A feature model for the *Delay* block structure and semantics

The root element of Figure 5.42 is the product line itself. It is named after the block (DelayBlock) (REQ-1.a). Beneath the root element, we provide all the mandatory features and their children features below.

Structural features

The `structuralFeatures` feature is a mandatory abstract feature holding all the variable structural features of the block specification (REQ-2.a, REQ-3.a, REQ-4.a):

- `inputs_outputs`: a mandatory abstract feature holding the specification for the inputs and outputs ports of the block.
  - `v_output`: the output port of the block
  - `u_input`: the input data port of the block

- `initial_condition`: a mandatory abstract feature holding an alternative relation between two features: `ic_input` and `ic_parameter`. These two features represent the two possible configurations for providing the initial condition for the block, either dynamically as an input signal or statically as a parameter. Both variants are abstracts and contain alternative sub-features modeling the scalar or vector dimension of the `initial_condition`. It is mandatory for the `initial_condition` to be either scalar or vector as it is mandatory to store multiple values in it for a `delay` parameter value (N) greater than one.
- `ic_mode_parameter`: a mandatory feature modeling a parameter holding the choice between the dynamic or static way of providing the `initial_condition` value.
- `delay_parameter`: a mandatory abstract feature representing the previously specified N value.
- `resettable`: an optional abstract feature holding the `res_input` optional feature representing the reset input port and a mandatory parameter `reset_algo` indicating which algorithm is used to activate the reset according to the `res_input` input signal value as defined in Section 5.2.2.
- `memories`: this mandatory abstract feature holds the memories of the block. There are two memories defined in the block: `m_memory` that is mandatory and is the output-delaying memory (provides the output value) and the `reset_memory` that is optional and activated only if the `reset_algo_parameter` is set to `RISING_EDGE`, `FALLING_EDGE`, `EITHER` or `LEVEL` as these behavior needs a knowledge of the previous value of the `reset_input`.

Semantics

The `semantics` feature is a mandatory abstract feature holding the block semantics variation point specifications. Each semantics variation point is specified through a set of cross tree constraints

- `ScalarInput_SimpeDelay`: models the basic semantics of the *Delay* block with only one input port (`u_input`), the `delay` value set to 1 and the `reset` input is deactivated. It is constrained with the (5.6) cross tree constraints.

$$\begin{aligned} ScalarInput_SimpleDelay &\Rightarrow (ic_scalar_input \vee ic_scalar_parameter) \wedge NONE \\ reset_input &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \vee LEVEL_HOLD \\ reset_memory &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \\ ic_mode_PARAMETER &\Leftrightarrow ic_parameter \\ ic_mode_input &\Leftrightarrow ic_input \end{aligned} \tag{5.6}$$

- `ScalarInput_ListDelay`: models the basic semantics of the *Delay* block with only one input port (`u_input`), the `delay` value set to more than one and the `reset` input is deactivated. It is constrained with the (5.7) cross tree constraints.

$$\begin{aligned} ScalarInput_ListDelay &\Rightarrow (ic_vector_input \vee ic_vector_parameter) \wedge NONE \\ reset_input &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \vee LEVEL_HOLD \\ reset_memory &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \\ ic_mode_PARAMETER &\Leftrightarrow ic_parameter \\ ic_mode_input &\Leftrightarrow ic_input \end{aligned} \tag{5.7}$$

- `ScalarInput_SimpleDelay_Reset`: models the basic semantics of the *Delay* block with only one input port (`u_input`), the `delay` value set to 1 and the `reset` input is activated. It is constrained with the (5.8) cross tree constraints.

$$\begin{aligned} ScalarInput_SimpleDelay_Reset &\Rightarrow (ic_scalar_input \vee ic_scalar_parameter) \wedge \neg NONE \\ reset_input &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \vee LEVEL_HOLD \\ reset_memory &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \\ ic_mode_PARAMETER &\Leftrightarrow ic_parameter \\ ic_mode_input &\Leftrightarrow ic_input \end{aligned} \tag{5.8}$$

- `ScalarInput_ListDelay_Reset`: models the basic semantics of the *Delay* block with only one input port (`u_input`), the `delay` value set to more than 1 and the `reset` input is activated. It is constrained with the (5.9) cross tree constraints.

$$\begin{aligned} ScalarInput_ListDelay_Reset &\Rightarrow (ic_vector_input \vee ic_vector_parameter) \wedge \neg NONE \\ reset_input &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \vee LEVEL_HOLD \\ reset_memory &\Leftrightarrow RISING \vee FALLING \vee EITHER \vee LEVEL \\ ic_mode_PARAMETER &\Leftrightarrow ic_parameter \\ ic_mode_input &\Leftrightarrow ic_input \end{aligned} \tag{5.9}$$

Analysis of the feature model

The product line depicted in Figure 5.42 allows twenty four valid configurations. This number is computed based on the number of combinations allowed for the Non-reset semantics (two for each as the only variation is for the `initial_condition` that is either a parameter or an input), and the resetted semantics that have five combinations (five reset algorithms) and also the two combinations related to the `initial_condition`. The final number of valid product is then: $2 \times 2 + (5 \times 2) \times 2 = 24$.

In a block specification oriented reading of the feature model, the number of block variants is less than twenty four as the variability added by the `reset_algo` parameter can be managed by other means like the definition of a generic function for the reset input impact according to the `reset_algo` parameter value. The number of valid product can then be limited to eight (two products for each semantics sub-feature).

5.5.4 Limitation of the SPLE specification approach

SPLE does not allows to specify every dependencies and constraints that must be provided on every feature of the block. For example, it does not allow to specify that the value potentially held in the `nbOfInputPorts_Parameter` feature impacts on the concrete number of input ports instances that should be provided in the sub-features of the `multiple_inputs` feature. This kind of cross tree constraints linking feature attribute values and feature cardinalities are not standard constraints, they have been studied by Czarnecki et Al [51]. To our knowledge, work still needs to be done in order to handle the complexity of such constraints and their formal semantics especially when dealing with attributes with a potentially infinite number of values.

Every structural feature should be further specified in order to provide their allowed data types, dimensions (REQ-2.b, REQ-2.c, REQ-3.b, REQ-3.c, REQ-4.b, REQ-4.c) and constrained values (REQ-5.a and REQ-5.b) like for example only positive value are allowed for the `numberOfInputPorts` parameter. Semantics specification should also be provided (REQ-6.a, REQ-6.b).

The specialisation of SPLE for the specification of dataflow blocks needs some substantial tooling and adaptation in order to be usable as requested in REQ-5, REQ-6, REQ-7, REQ-8 and REQ-9.a. For more complex specifications, we might also need to refine the structural features to, for example, include more elements like data types attached to structural features or constraints that limit the allowed values for those data types. The resulting number of products in the product line would increase drastically making the specification difficult to manage and analyse.

5.6 Two complementary approaches

Use of SPLE methodologies and formalisms allows to improve the variability management and solves part of the problems raised in the UML + OCL specification proposal. However, it does not allow to handle completely our domain specification problems. We saw in this section two possible specification methodologies and formalisms. First, UML + SPLE profile + OCL allowing for an accurate specification of blocks structures along with block semantics and variability management at a certain granularity. It suffers from the complexity of the UML model impacting the implementation of automatic verifications. Second, SPLE allows for an accurate management of variability issues but its expression power is very limited regarding the structural and semantics specification of blocks.

5.6.1 Methodology proposal

An accurate specification formalism for our purpose would inherit from concepts taken from both approaches. We will illustrate a possible framework for the specification of the *Delay* block, using SPLE + UML + OCL. Our proposed methodology relies on the splitting of the block specification into 4 phases. Each phase is purpose oriented. In the first phase, the variability is analysed (REQ-9.b and REQ-9.c) by using SPLE and feature modeling, the block structural features are elicited, relations between them are provided. The overall complexity of the block is analysed. The second phase aims at matching the previously specified features to UML classes and to generate an UML model. This transformation can either be manual or generated, the advantage of the automatic generation is obviously about time gain but it is also about the standardisation of the produced output. In the next phase, the block specification is constrained by providing OCL constraints for data types and dimensions of structural features (REQ-1, REQ-2, REQ-3, REQ-4, REQ-5). Finally, the semantics specification can be provided for each generated semantics instance (REQ-6).

Automatic verification of the specification properties (REQ-7), semantics correctness (REQ-8) and typing verification of expressions (REQ-9.a) can be done by developing transformations based on both the feature model and the UML + OCL elements.

5.6.2 From SPLE analysis to UML model

Using the extended feature modeling [84], it is possible to provide additional informations for each feature. We suggest to attach to each feature a UML class name taken from the generic block specification model of Section 5.4.1. We provide a table for such a mapping in Table 5.43.

By following this mapping table, we can generate an UML model from the *Delay* block feature model. Such UML model would be the same as the one provided previously for the *Delay* block specification in Figure 5.38. We provide it here for reference in Figure 5.44. The advantage of generating the UML model from the feature model is to keep the variability management at the feature model level and to use the UML capabilities for the detailed modeling of the block structure and semantics. In this setting, the specifier can then use OCL for structural constraints of the block feature values and data types and use any action language listed in Section 5.4.3 for the semantics specification.

When the block specification is done, it is then possible to extract the valid products from the UML model according to the extracted models of the feature model.

SPLE feature	Mapped UML class
DelayBlock	Block
ScalarInput_SimpleDelay ScalarInput_ListDelay ScalarInput_SimpleDelay_Reset ScalarInput_ListDelay_Reset	SequentialSemantics
u_input ic_scalar_input ic_vector_input	Input
v_output	Output
ic_scalar_parameter ic_vector_parameter ic_mode_parameter delay_parameter reselt_algo_parameter	Parameter
m_memory reset_memory	Memory

Table 5.43: Mapping from *Delay* block feature mode elements to UML classes

5.6.3 Limitations of the methodology

On the specification part, this proposed approach fits the needs. As we have shown, it allows for an accurate modeling of the specification variability, structure and semantics. But it suffers from major drawbacks related to the tooling development required for the verification of block specification: first, the SPLE to UML transformation and then the specification verification.

SPLE to UML transformation

The first transformation is simple and straightforward. Its verification should be easy regarding its complexity, a simple traceability checking is enough and can even be done to some extend reliably by proof reading if the number of elements in the specification is not too big. As we saw previously with the *Delay* block example, the feature model can become quickly quite complex leading to difficulties in the transformation verification. This transformation target is an UML model, it should produce correct UML models and thus rely on an already existing UML implementation which have the drawback of being dependent of a specific platform or on the OMG-specified XMI format for the UML serialisation [5]. The second option is better as it allows to rely on common well accepted normative documents but need a consequent development. Of course current implementations of UML relying on this normative format is the simplest solution. While it is doable to implement this approach one needs to take care of the under-specified nature of the XMI format leading to incompatibility issues between tools.

Automatic verification

The second tooling development phase is about the verification of the specification itself after it has been implemented as an UML model and extended with OCL constraints.

REQ-7.a advocates for the structural correctness of the specification. Such correctness can be partially checked in the UML model itself as it does enforce some structural correctness. Additional structural verification should be implemented using OCL constraints to be applied on the modified model after generation.

REQ-7.b and REQ-7.c advocates for the completeness and disjointness of the specification. Such a verification relies first on the ability to extract from the specification all the valid block configurations. Such extraction of products from an UML specification is handled in [162]. In this work, the variability informations are extracted from the profiling informations of the UML model but the same informations can be extracted from the SPLE informations and the SPLE to UML transformation traceability informations.

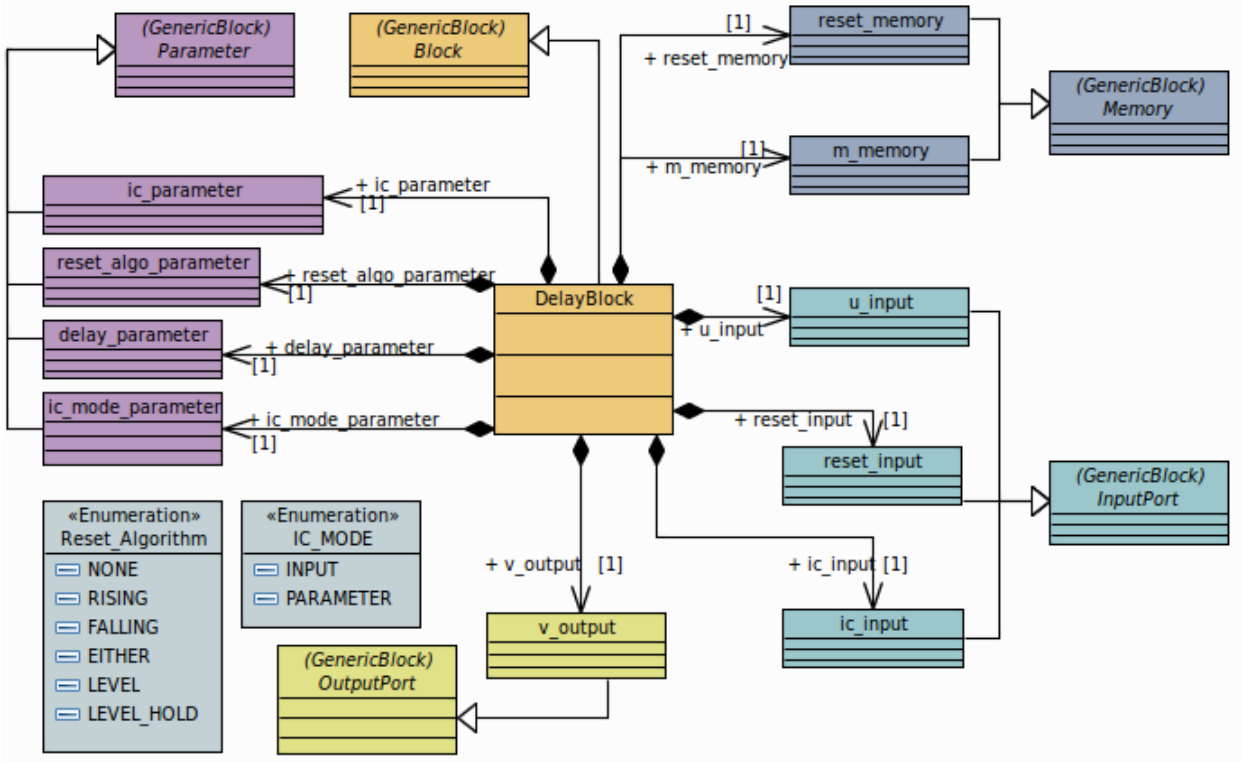


Figure 5.44: Delay block specification using UML + OCL

Once this product extraction done, it is then necessary to extract from the products the information for the completeness and disjointness assessment. We have provided in (5.1) and (5.2) the formalisation of such operations. Implementation of the verification can be done in many ways, we propose to translate the model elements (UML classes and references) and the OCL constraints to a formal domain such as SMT solvers, this approach was not implemented for UML + OCL but for a specific DSML presented in Chapter 6 and the verification approach from Chapter 7.

REQ-8 advocates the verification of the semantics specified for each possible block configuration. The structural configuration of the block is then the definition domain of the block containing its inputs, outputs, parameters and memories along with their specification. Specified semantics should then be translated to the same formal domain and verified. Again we propose to use SMT solvers and proof assistants to do this verification according to its formalisation in (5.3). This approach has been developed from a DSML presented in Chapter 6 and the verification approach in Chapter 7. The semantics specification language should be formalised in order to ensure its translation verification. This might not be easily done especially if the semantics specification language is a complex and expressive language like Java.

5.7 Synthesis

We detailed two propositions for both structural and semantics specification of dataflow blocks. Separated, each approach has a limited interest but their combinations showed interesting capabilities regarding expressiveness and verification. These capabilities are diminished by the complexity of the elements under specification leading to potentially over-constrained specifications (UML models) or surcharged feature models. We thus propose to rely on the DSML-based approach inspired on the SPLE and feature modeling principles. DSML are simpler as they are domain focused and it is easier to control the language specification and content. This leads to a simplification of the implementation of verification procedures and their implementation verification. Finally tooling definition is also simplified as we can rely on MDE techniques and automatise parts of the development effort. We will refer to this DSML as the BlockLibrary specification language and will detail it along with its associated tools in the next chapters.

6

A Domain Specific and Product Line experiment for language specification

As seen in Section 5, dataflow languages such as Simulink or Scade rely on polymorphic elements (blocks) with highly variable semantics. The high variability of these blocks makes the writing of precise and complete specifications as well as any implementation and verification based on it very challenging. In order to ease these activities, we advocate to specify dataflow blocks with a dedicated Domain Specific Modeling Language (DSML) based on Software Product Line Engineering (SPLE) principles and concepts.

In this chapter, we will first describe the domain on which our DSML must apply. Then in Section 6.2 we will describe a DSML based on a model driven approach: the BlockLibrary DSML. We will rely on the previously described *Delay* block for the elicitation and clarification of the specification language elements. The relation between SPLE methodologies and our specification metamodel will be shown in Section 6.3. Section 6.4 will detail how specific block configurations can be extracted from the BlockLibrary metamodel conforming models. Block specific semantics specification is detailed in Section 6.5. We will finally provide in Section 6.6 a formalisation for BlockLibrary verification properties.

6.1 Domain analysis

In the MDE methodology, a DSML definition starts from a domain analysis allowing to emphasize the key concepts of the domain and the relations between them. Relations are either structural (what are the properties of each concept, relations between concepts) or behavioral (related to the semantics of executable languages). From this analysis should emerge a metamodel along with, if it is required, additional OCL constraints.

6.1.1 Domain of study

The domain of study of our work is variable block specification for dataflow languages. A block specification is two-fold: a) the allowed structures of the block – or interfaces of the block – containing the allowed combinations of input ports, output ports, parameters and memories along with the allowed data types and values specification for each of these structural elements and; b) the semantics specifications that are attached to each possible block interface.

Structural elements specification should provide their allowed data types and values (REQ-[2|3|4].[b|c]). In order to do so, a data type must be provided for each `StructuralFeature` element of the block interface and logical constraints must enrich their specification.

A possible structured representation of the blocks has been provided using a class diagram in Section 5.4.1 and Figure 5.21 in pages 59 and 61. We relied on this in order to define the block specification but, as concluded previously, it does not provide appropriate and simple variability management capabilities.

For one block, all its possible structure variants have most of the time a common set of structure elements that remain unchanged – in the *MinMax* block the `function` parameter holding the function (min or max) value must be provided. It also seems that groups of structural elements are inextricably linked – in the *Delay* block, the `delay` parameter value impacts on the memory size. So, we suggest to group structural features in entities representing partial block interfaces. Building of more complete block interfaces should then be done by gathering already existing partial block interfaces. This gathering of partial block interfaces is a composition where overloading is forbidden, the content of the gathered interfaces are composed into a new interface. Semantics specification then will be attached to block interface(s).

Partial interface specifications are extended with logical properties on the gathered block interface elements. It is thus mandatory to allow for the expression of such properties in each (partial) interface specification.

Parameters and input ports may have the same semantic meaning for the block (as we saw for the *Delay* block regarding the `initial_condition` that can be expressed either as a parameter or as an input). Alternative block interfaces specification capabilities should then be provided and the same behavioral specification might be attached to different block interfaces.

6.1.2 Variability modeling

Dataflow blocks have potentially highly variable configurations of interfaces and semantics. This variability can be well handled using a SPLE approach. In SPLE, variability is tackled by allowing to express the hierarchy formed by the domain features – with a hierarchical relation between features – as long as cross-tree constraints between features. In our block specification, SPLE features will be referred to as the previously defined partial interface specifications.

The result of this domain analysis is the BlockLibrary metamodel specified using the Ecore modeling language. The metamodel is extended with static semantics informations provided as OCL constraints for the specification of additional correctness properties. It contains all the elements of interest to fulfill the requirements previously defined in Section 5.1. It is presented in Figure 6.1 where the default color for metaclasses background is yellow. Other colors are used to provide hints on the metaclasses inheritance relations (all white metaclasses inherit from the `StructuralFeature` metaclass and all grey metaclasses inherit from `SpecificationElement` metaclass).

In the following, we will first comment on a partial specification structure for the *Delay* block, then we will provide a textual example for it using the BlockLibrary DSML textual syntax associated to the metamodel to ease the writing of examples. Detailed description of the BlockLibrary metamodel will then follow and will be illustrated with the specification of the *Delay* block.

6.2 The BlockLibrary DSML

We provided a detailed SPLE specification for the *Delay* block in the previous chapter (Figure 5.42). We will rely on a simpler specification of the block where: a) the initial condition cannot be provided as an input of the block; b) the reset input is not allowed; and c) only scalar and vector double are available as input. Such simplification will ease our language introduction.

6.2.1 *Delay* block interfaces specification

We will detail the partial interfaces defined for this block specification. The interfaces hierarchy is displayed in Figure 6.2. In this figure, ellipse nodes are partial interfaces while square nodes are semantic specification.

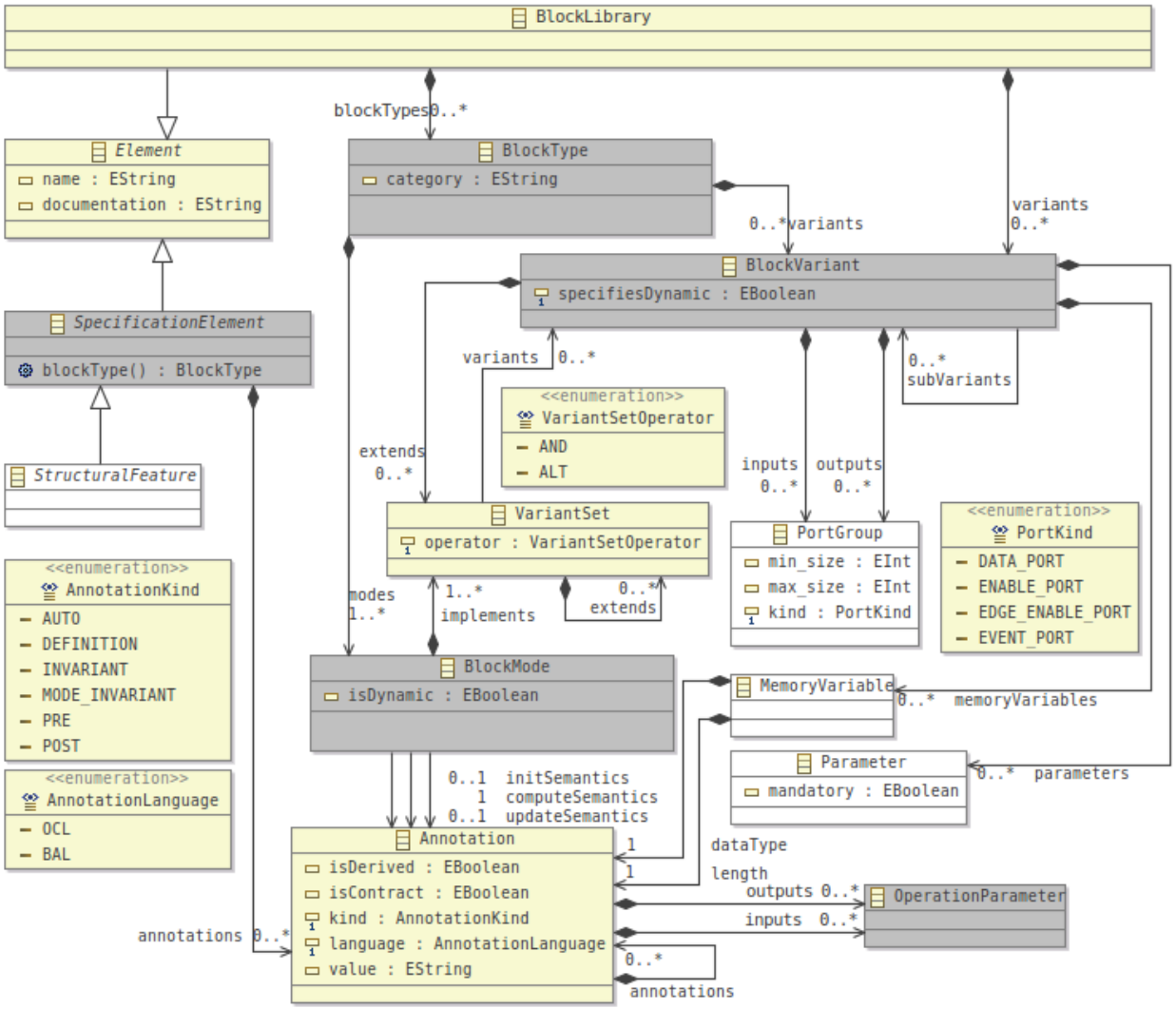


Figure 6.1: The *BlockLibrary* metamodel

Our simplified *Delay* block has two mandatory parameters: `delay` (providing a value containing the number of clock tick by which an input value is delayed to be provided as an output value) and `initial_condition` (the initial condition parameter used as the initial value(s) for the block output). The first one holds the delay length (the number of clock ticks by which the output value is delayed according to the input); the second one holds the initial value for the first output values (the number of values is equal to the `delay` value).

The `delay` parameter has a fixed data type and thus does not have any variation point whereas the `IC` one can either be a scalar, a vector or a matrix as it might be necessary to store either one or multiple input values. We decide to create a specific partial interface for the first parameter: `DelayParameter` and three specific ones for the three possible variants of the `IC` parameter: `ICScalar`, `ICVector` and `ICMatrix`.

The `U` input port (providing the input value to be delayed) is allowed to take either a scalar or a vector of double values, it is thus required to create two separated interfaces for these variants: `UScalar` and `UVector`. Dimensionality and data type of the output value depends on the input ones so we can provide the specification for the `V` output directly into the corresponding partial interfaces containing the `U` input specifications.

Each of those two partial interfaces are related to the `DelayParameter` one. This provides access to the `StructuralFeature` definitions held in `DelayParameter` in the relating partial interfaces.

We do not relate `UScalar` and `UVector` with `ICScalar`, `ICVector` and `ICMatrix` as `IC` dimensionality depends on the `Delay` parameter value and on the dimensionality of the `U` input. These values

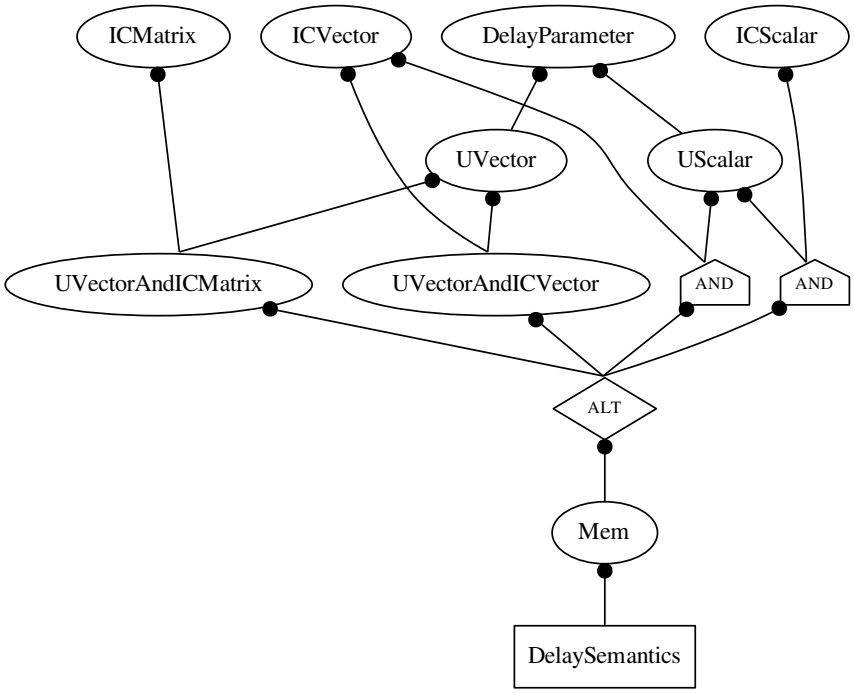


Figure 6.2: The *Delay* block specification hierarchy

also impact the size of the memory (`M`) that stores the delayed values. We detail this dimension relation in Table 6.3. According to these four possible combinations we see that `IC` and `M` dimensions are linked as `IC` semantics is to provide the initial value of the memory. We should additionally ensure the values sizes matching between `U` and `IC`. In order to do so, logical properties should be specified for the cases where `U` is not a scalar. We propose to insert two additional partial interfaces: `UVectorAndICVector` and `UVectorAndICMatrix`. The first one is related to both `UVector` and `IVCVector` and the second one to `UVector` and `ICMatrix`.

U dimension	Delay value	IC dimension	M dimension
Scalar	1	Scalar	Scalar
Vector	1	Vector	Vector
Scalar	> 1	Vector	Vector
Vector	> 1	Matrix	Matrix

Table 6.3: Relation between Delay value, U dimension, IC dimension and M dimension

A memory value should finally be specified according to the data it should store and the amount of these data, its specification should thus contain these informations. `M` memory declaration could have been provided along with the `initial_condition` interfaces as they are strongly related, but a memory data type should be specified according to the data it should hold and thus according to the input port values. We thus defined a partial interface for the memory specification: `Mem`. It specifies a memory storing multiple values and should extend either of the four configuration displayed in Table 6.3. These relations are provided in Figure 6.2 as the combinations of ALT nodes and AND nodes.

The ALT and AND nodes express the semantics according to whom the gathering of block interfaces is done: ALT nodes expresses an alternative composition whereas AND nodes expresses a mandatory composition of the related elements. In Figure 6.2 the composition is to be read bottom up, the element related by an edge to the lower part of an ALT or AND node is gathering the elements related by an edge to the upper part of an ALT or AND node. Edges directly linking partial interfaces and/or semantics specifications

have an implicit AND relation, they are not displayed in order to ease the readability of the Figure. The `UVectorAndICMatrix` partial interface is thus a composition of the `UVector` and the `ICMatrix` partial interfaces. We could also formalise it as in (6.1) with the = operator being the composition operator. Using the same formalism, the `Mem` partial interface definition would be as in (6.2).

$$UVectorAndICMatrix = AND(UVector, ICMatrix) \tag{6.1}$$

$$\begin{aligned} Mem = ALT(&AND(ICScalar, UScalar),\\ &UVectorAndICVector,\\ &AND(ICVector, UScalar),\\ &UVectorAndICMatrix) \end{aligned} \tag{6.2}$$

Finally we can attach a semantics to these interfaces specification. This is depicted with the `DelayMode` semantics specification element.

6.2.2 *Delay* block textual specification

Tooling development based on the use of metamodels typically starts with the definition of a concrete syntax. The kind of concrete syntax to be developed must depend on the user needs and capabilities but the certification needs must also be taken into account (see Section 2.1). We choose to use a textual syntax for our purpose as textual syntaxes have good qualities such as scaling, versioning and easy refactoring. We developed it using the XText framework that allows to define it in an Eclipse environment.

We present in Listing 6.4 a simplified specification of the *Delay* block. Where we only keep a part of the *Mem* partial interface dependencies in order to restrict the configurations where the memory has to store only scalar elements. Again, we choose to restrict the content of the specification for the sake of readability, interested reader can find the complete specification in Appendix A.

The specification starts with the declaration of the block library name (line 1): *BlockLibrary*. Some data types (TInt32, TDouble, TArrayDouble, TMatrixDouble and TString) are then defined in lines 2 to 6. The remaining elements of the listing (lines 8 to 58) gives the *Delay* block specification. We can note the declaration of the `delay` parameter on line 10 along with an invariant stating its allowed values. On line 26, we declare an invariant expression stating a structural condition on a partial interface. We will detail these constructs all along this chapter. The concrete textual syntax provided here is only an experimentation proposal that can be improved.

More advanced concrete syntaxes could have been used, for example a mix of both graphical and textual syntax allowing to first design a coarse block specification structure in a graphical manner and then to define for each element its detailed specification using a textual syntax. Our choice of a textual syntax was related to efficiency as textual syntaxes are more efficient to develop. In order to extend further the approach and to improve the user experience, an advanced concrete syntax is likely to provide better results for the product line part. The potential complexity of structural features constraints and semantics definition will only be handled efficiently with a textual syntax.

6.2.3 BlockLibrary metamodel abstract elements

The BlockLibrary metamodel contains three abstract metaclasses that gather common attributes for some of the metamodel metaclasses. They are presented in Figure 6.5.

Definition 6.2.3.1. *`StructuralFeature` is an abstract metaclass from which all the block specification structural feature should inherit (ports, parameters and memories). It does not provide any attribute or operations. Its purpose is to provide a common type for structural features used in various operations as parameters and return types.*

```
library BlockLibrary {
  type signed realInt TInt32 of 32 bits
  type realDouble TDouble
  type array TArrayDouble of TDouble [-1]
  type array TMatrixDouble of TDouble [-1,-1]
  type string TString

  blocktype Delay {
    variant DelayParameter {
      parameter Delay : TInt32 { invariant ocl { Delay.value > 0 } }
    }
    variant UScalar extends DelayParameter {
      in data U : TDouble { invariant ocl { U.isScalar() }}
      out data V : TDouble { invariant ocl { V.isScalar() }}
    }
    variant ICScalar {
      parameter IC isMandatory : TDouble { invariant ocl { IC.isScalar() }}
    }
    variant ICVector {
      parameter IC isMandatory : TArrayDouble { invariant ocl { IC.isVector() }}
    }
    variant Mem extends allof (UScalar, oneof(ICScalar, ICVector))
    {
      invariant ocl { IC.value.size() = Delay.value }
      memory Mem {
        datatype auto ocl {U.value}
        length auto ocl {Delay.value}
      }
    }
    mode DelaySemantics implements Mem {
      definition bal = init_Delay {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Delay {
        postcondition ocl { Output.value = Mem.value->first() }
        Output.value = Mem.value[0];
      }
      definition bal = update_Delay {
        postcondition ocl {
          Mem.value->last() = Input.value
        }
        postcondition ocl {
          Mem.value = (Mem.value@pre)->subList(2,Delay.value)->append(Input.value)
        }
        for (int i=0; i < (Delay.value - 1); i = i + 1) {
          Mem.value[i] = Mem.value[i + 1];
        }
        Mem.value[Delay.value - 1] = Input.value;
      }
      init init_Delay
      compute compute_Delay
      update update_Delay
    }
  }
}
```

Listing 6.4: Extract of the *Delay* block specification using BLOCKLIBRARY textual syntax

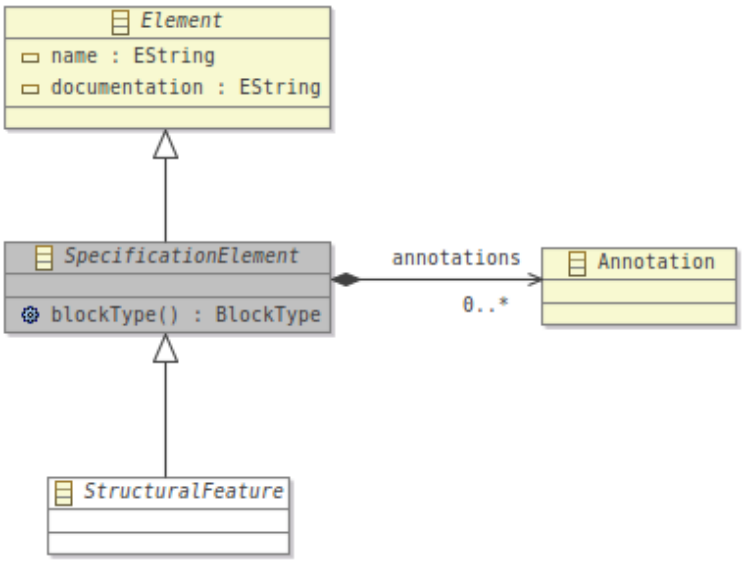


Figure 6.5: The BLOCKLIBRARY metamodel abstract metaclasses

Definition 6.2.3.2. *`SpecificationElement` is an abstract metaclass from which all the block specification elements must inherit (including `StructuralFeature`). It is a 2-tuple $(\{A\}, \text{SpecificationContainer})$, where: $\{A\}$ is a set of annotations constraining the `SpecificationElement` and SpecificationContainer is an operation returning the specification holder, i.e. the block specification or the block library – using a SPLE terminology, this would be the root feature of the current feature model. Its signature is: SpecificationContainer : `SpecificationElement` $\rightarrow$ BLOCKLIBRARY $\cup$ `BlockType`*

Definition 6.2.3.3. *`Element` is an abstract metaclass from which all the BLOCKLIBRARY metamodel metaclasses inherits from. It is a 2-tuple (N, D), where: N is the name of the element, provided as a String value. This attribute is related to requirement REQ-[1|2|3|4].a, as `StructuralFeature` inherits from `Element`. D a String value used to attach documentation to the `Element`.*

6.2.4 ANNOTATIONS

Annotations are of central interest in the BLOCKLIBRARY DSML. They either express constraints or operations. As a constraint, an annotation is used for structural element value restriction (kind of dependent type) or interface constraining. As an operation, an annotation is used for the semantics specification of a block interface. We provide an excerpt of the BLOCKLIBRARY metamodel focused on the `Annotation` metaclass in Figure 6.6.

The `Annotation` metaclass is meant to be extended. It is an entry point for the definition of an annotation as a constraint or an operation. `Annotation` instances should be a container for an AST for a constraint or an operation expressed using a predefined language.

Definition 6.2.4.1. *An `OperationParameter` is a specification for an operation parameter. It inherits from `StructuralFeature` (and so from `Element`) and thus has a name. It has one reference to a `DataType` element allowing to specify the parameter data type.*

Definition 6.2.4.2. *An `Annotation` is a 7-tuple $(K, L, V, C, \{Op\}_i, \{Op\}_o, \{A\})$, where: K (kind attribute) is the kind of the annotation, its value is taken among the `AnnotationKind` enumeration. L (`language` attribute) is the language used to write the annotation, its value is taken among the `AnnotationLanguage` enumeration. V (`value` attribute) is the value of the annotation as a String. This string must be a correct expression of the concrete syntax of the language selected by L. This expression can access the elements defined in the `Annotation` container and the container's linked `SpecificationElement`.*

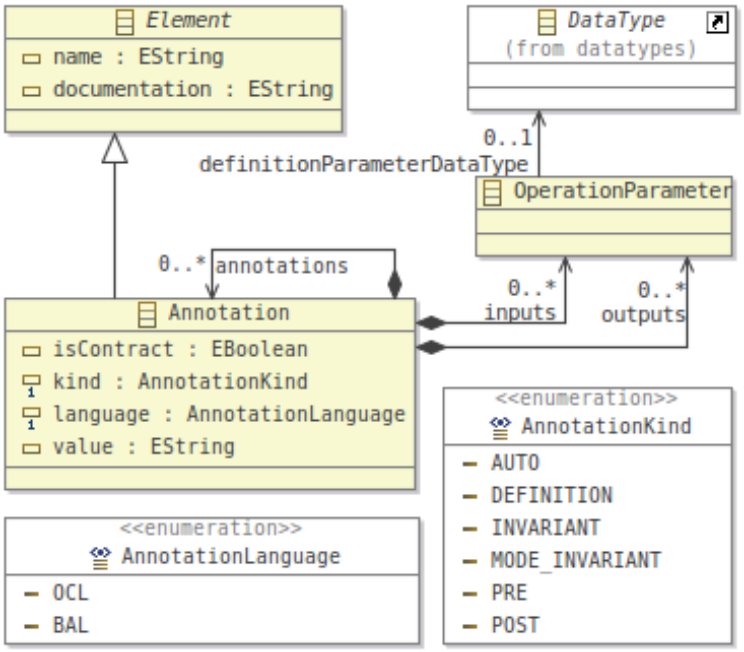


Figure 6.6: The *BlockLibrary* `Annotations` metaclass definition

C (`isContract` attribute) specifies whether the annotation specifies `DEFINITION` annotations only used in semantics contracts definition (pre/post annotations). $\{Op\}_i$ (`inputs` reference) and $\{Op\}_o$ (`outputs` reference) contains the definitions of `OperationParameter` elements, they are the input and output parameters for the specified `Annotation`. Finally, $\{A\}$ (`annotations` reference) is a set of `Annotation` elements. These sub-annotations are used as the contract of their containing `Annotation` parent.

Examples `Annotation` elements for the two possible annotation languages can be found in the *Delay* block specification of Listing 6.4: line 10 specifies an OCL invariant and line 33 to 36 defines a `BAL` definition.
`Annotation` should be expressed using specific languages. It is defined according to the `language` attribute. We currently support two languages in `Annotation` elements: a large subset of OCL as a general constraint language and a dedicated simple imperative language called BLOCKLIBRARY Action Language (`BAL`) used for a more convenient specification of the semantic functions operational semantics (we could have used an already defined language such as ALF, XTEND or JAVA but we chose to keep the action language simple for this experiment). We provide the specification for both of these languages in Section 7.1.

We distinguish between several kind of annotations according to the `kind` attribute value. This attribute defines the interpretation that should be made of the annotation.

- `DEFINITION Annotation` allows the definition of constants or functions. Constants `DEFINITION` are global constant declarations that can be used in other annotations. Function `DEFINITION` are meant to gather reusable expressions or parts of programs for the sake of simplification of the specification writing and reading. `DEFINITION` annotations are the only annotations allowed to have input parameters or outputs that are defined through the $\{Op\}_i$ and $\{Op\}_o$ references. This constraint is specified in the *DefAnnotKindRelToIORef* OCL invariant detailed in Listing 6.7. A `DEFINITION Annotation` taken from the *Delay* block specification would be:

```
definition bal = compute_Delay {
  postcondition ocl { Output.value = Mem.value->first() }
  Output.value = Mem.value[0];
}
```

- PRE/POST Annotation can only be included inside other DEFINITION annotations. They allow for the specification of their container definition contract – axiomatic semantics. This containment constraint is formalised in the PrePostDefAnnotContainedInDefAnnot OCL invariant detailed in Listing 6.7. A POST Annotation taken from the *Delay* block specification would be:

```
postcondition ocl {
  Mem.value->last() = Input.value
}
```

- INVARIANT Annotation are meant to express invariants on StructuralFeature of a block definition. They allow for data types and/or expression of StructuralFeature values constraints. They are thus to be contained only in StructuralFeature elements as formalised in the InvAnnotInStructFeatElem OCL invariant detailed in Listing 6.7. An INVARIANT Annotation taken from the *Delay* block specification would be:

```
parameter Delay : TInt32 { invariant ocl { Delay.value > 0 } }
```

- MODE_INVARIANT: These can only be contained in BlockMode and BlockVariant Element (formalised in the ModInvAnnotInBMorBV OCL invariant detailed in Listing 6.7). They are used as block interface constraints expressing conditionals on StructuralFeature elements. A MODE_INVARIANT Annotation taken from the *Delay* block specification would be:

```
invariant ocl { IC.value.size() = Delay.value }
```

- AUTO: These can be contained everywhere. By default it is used as a INVARIANT Annotation it can change according to the context of definition of the annotation: in a BlockMode or a BlockVariant it is interpreted as a MODE_INVARIANT Annotation.

The isContract attribute, targets the specialisation of DEFINITION Annotation elements that can be referenced only in axiomatic semantics definitions (see Section 6.2.7). Setting this attribute value to *true* changes the way the annotation is interpreted and thus its purpose in the specification. The first part of this definition is formalised in the IfIsContractThenDefinition OCL invariant detailed in Listing 6.7.

Finally, $\{Op\}_i$ and $\{Op\}_o$ specifies the OperationParameter elements for a DEFINITION Annotation. An OperationParameter is a specification for either an input parameter ($\{Op\}_i$) of the Annotation that contains it or a specification for an output of the Annotation ($\{Op\}_o$). Each OperationParameter instance has a name (inherited from the Element metaclass) and an optional reference to a data type. We do not enforce the data type setting for an operation parameter as it might be possible to infer it from the operation definition. According to the annotation language, the number of outputs OperationParameter elements should be constrained. In our setting, only action language operations are allowed to have multiple output values as we want to ensure the coherence with the OCL specification where operations cannot have multiple output values (this can be done via the use of the Tuple construct). This constraint is formalised in the NbOutputParamToAnnotLang OCL invariant in Listing 6.7.

6.2.5 Data types specification

According to requirement REQ-[2|3|4].[b|c], we want to be able to specify the data types and dimensions of the blocks structural features. It is therefore mandatory to provide a specification for the data types.

In this purpose, we decided to use the type system presented in Section 4.1.2. Data types instances must then be provided in the block library specification. A data type instance declaration taken from the *Delay* block specification would be the following defining a 32 bits integer data type:

```
type signed realInt TInt32 of 32 bits
```

```
context Annotation

inv DefAnnotKindRelToIORef:
  outputs->size() > 0 or inputs->size() > 0 implies
  kind = AnnotationKind::DEFINITION

inv PrePostDefAnnotContainedInDefAnnot:
  (self.kind = AnnotationKind::PRE or self.kind = AnnotationKind::POST)
  implies
  (self.oclContainer().oclIsKindOf(Annotation) and
   self.oclContainer().oclAsType(Annotation).kind = AnnotationKind::DEFINITION)

inv InvAnnotInStructFeatElem:
  self.kind = AnnotationKind::INVARIANT implies
  self.oclContainer().oclIsKindOf(StructuralFeature)

inv ModInvAnnotInBMorBV:
  self.kind = AnnotationKind::MODE_INVARIANT implies
  (self.oclContainer().oclIsKindOf(BlockMode) or
   self.oclContainer().oclIsKindOf(BlockVariant))

inv IfIsContractThenDefinition:
  self.isContract implies kind = AnnotationKind::DEFINITION

inv NbOutputParamToAnnotLang:
  language = AnnotationLanguage::OCL implies
  outputs->size() = 1
```

Listing 6.7: `Annotation` metamodel element OCL constraints

6.2.6 Block structural features

We define here the subset of elements from the BlockLibrary metamodel used in a block specification to hold the structural features of the specified block. A detailed diagram of these elements is provided in Figure 6.8. These structural features are contained in `BlockVariant` elements that will be detailed in Section 6.2.7.

Definition 6.2.6.1. *A `PortGroup` defines a group of similar ports with a common purpose that a block instance should be composed of. It would for example model the allowed inputs for an arithmetic operation. It is a 4-tuple* $(\{DT\}, Min, Max, K)$*, where:* $\{DT\}$ *is a set of allowed data types taken from the DataTypes metamodel, it references to the `allowedTypes` reference. This attribute is related to requirements REQ-2.[b|c]; Min and Max are integer values specifying how many ports of such a `PortGroup` may be used in a block instance, they respectively reference the `min_size` and `max_size` attributes. These attributes are additionally constrained according to OCL invariants formalised in Listing 6.9 stating that Min must be a natural number, Max must be either -1 or a natural number greater or equal to Min. The value -1 means that the bound value is not known and thus the maximum number of ports in this `PortGroup` is not limited. These Min and Max attributes are fulfilling the requirement REQ-1.b. K specifies the kind of port, it references the `kind` attribute, this value is an enumeration of type `PortKind`. According to this attribute value, the specified port behaves as a data carrying port (DATA_PORT), a port enabling the block computation according to the input boolean value of the port (ENABLE_PORT), the input rising/falling edge value of the port (EDGE_ENABLE_PORT) or a port carrying events (EVENT_PORT). A `PortGroup` is a dynamic element of the block specification as its value can only be concretely known when a block is executed.*

There is no example of this kind of multiple input ports in the *Delay* block. As described in the *MinMax* block specification in Chapter 5, all the input ports are used in the same purpose in the computation. In a BlockLibrary specification it would have been convenient to represent all the input port in a `PortGroup`. A possible declaration using the BlockLibrary syntax is provided in Listing 6.10. In this extract we define the input `PortGroup` named `Input` as a `double` value, this `PortGroup` model a sequence of at least 1 port (as the provided multiplicity is [1..*]).

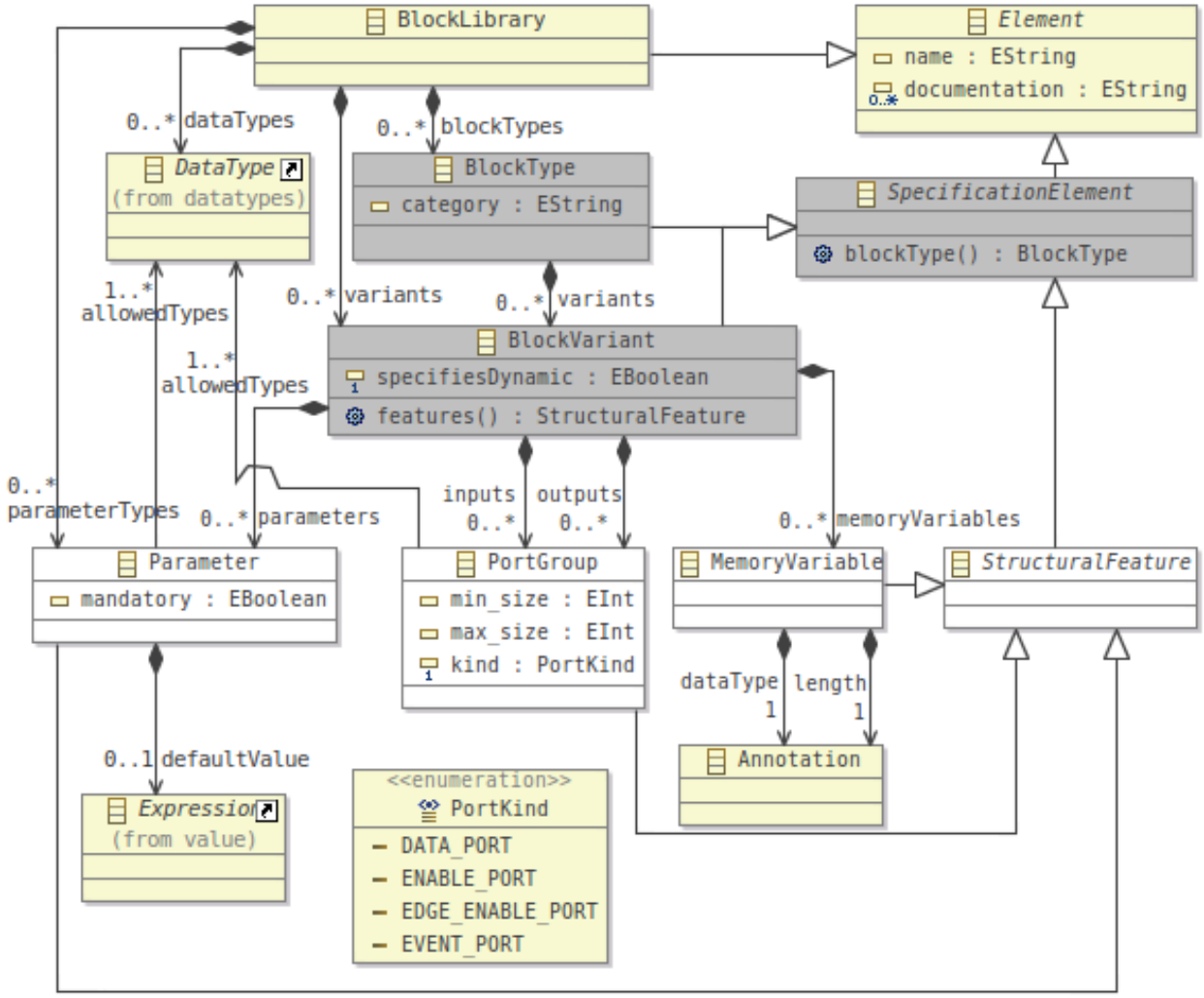


Figure 6.8: The BLOCKLIBRARY metamodel structural features definition elements

```
context PortGroup
inv MaxSizeOne:
  max_size = 1 implies min_size = 1

inv MinMaxSizeGTZero:
  min_size > 0 and (max_size > 0 or max_size = -1)

inv MaxSizeGTMinSize:
  max_size >= min_size or max_size = -1

inv OutputPortThenMinMaxValues:
  oclContainer().outputs->includes(self) implies
  (min_size = 1 and max_size = 1)
```

Listing 6.9: `PortGroup` `min_size` and `max_size` attributes OCL constraints

```
in data Input : TDouble [1 .. *]
```

Listing 6.10: *MinMax* block input specification using the BLOCKLIBRARY syntax

Definition 6.2.6.2. *A* `Parameter` *defines a parameter that a block instance may be composed of in order to configure the structure and behavior of the block. It is a 3-tuple* $(\{DT\}, M, D)$*, where:* $\{DT\}$ *is a set of allowed data types taken from the* DATATYPES *metamodel (REQ-3.[b|c]);* M *specifies, whether the parameter is mandatory or not; and* D *is a container for the default value specified as an* `Expression` *element (the* `Expression` *element contains a literal value stored as a String). A constraint is added specifying that if a* `Parameter` *is mandatory then its default value* D *should be provided. A* `Parameter` *is a static element of a block specification as its value is concretely known when the block is instantiated.*

An example of parameter definition for the `initial_condition` parameter of the *Delay* block is provided in the *Delay* block specification in line 27. In this setting, the `IC` parameter is declared as a `TDouble` value, that is mandatory – `isMandatory` keyword.

Definition 6.2.6.3. *A* `MemoryVariable` *defines a state variable that a block instance may be composed of in order to store data between cycles during the execution of sequential blocks. It is a 2-tuple (DT, L), where: DT is the data type (taken from the* DATATYPES *metamodel) of the* `MemoryVariable`*– modeled as the* `dataType` *reference to an* `Annotation` *element in the metamodel (REQ-4.b). L determines the dimension of the* `MemoryVariable` *(a dimension value greater than one allows to store multiple values in the memory) modeled as the* `length` *reference to an* `Annotation` *element in the metamodel (REQ-4-c). A memory specification is not acceptable if its initial value is not provided. This must be provided in the initialisation phase of the semantics specification of the block (detailed in Section 6.5). A* `MemoryVariable` *is a dynamic elements of a block specification.*

The *Delay* block specification needs a memory definition. This memory must store the input value. Its data type is then the one of the input value. The number of input values stored by this memory is dependent of the `delay` parameter that provides this information. A `MemoryVariable` example taken from the *Delay* block specification would be:

```
memory Mem {
  datatype auto ocl {U.value}
  length auto ocl {Delay.value}
}
```

6.2.7 BLOCKLIBRARY METAMODEL VARIABILITY STRUCTURE

A block specification is composed of both structure and behavior specification for one block configurations. These configurations are build from partial interfaces specifications: `BlockVariant`. `BlockVariant` are composed according to a logical language represented in the metamodel by `VariantSet` elements derived from SPLE.

A `BlockVariant` is a container of the information required to describe a correct block instance structure. Such a structure is referred to as the configuration of a block. A `BlockMode` element gathers a semantics specification for the block configuration. Each `BlockMode` can be associated to a set of configurations using the same `VariantSet`-based logical language. In the *Delay* block specification extract, we have one semantics definition element: `DelaySemantics`, some partial interfaces holding a `MemoryVariable` definition: `Mem`; `PortGroup` definitions: `UScalar` or `Parameter` definitions: `DelayParameter`, `ICScalar`, `ICVector`.

In the following, we will define the `BlockVariant`, `BlockMode` and `VariantSet` elements. We provide an extract of the BLOCKLIBRARY metamodel focused on these elements in Figure 6.11. In this figure, we provide the complete signatures for the `features`, `signatures` and `configurations` operations in a yellow note. We decided to do this as the graphical ECORE metamodel does not display correctly the return types of operations returning multiple elements.

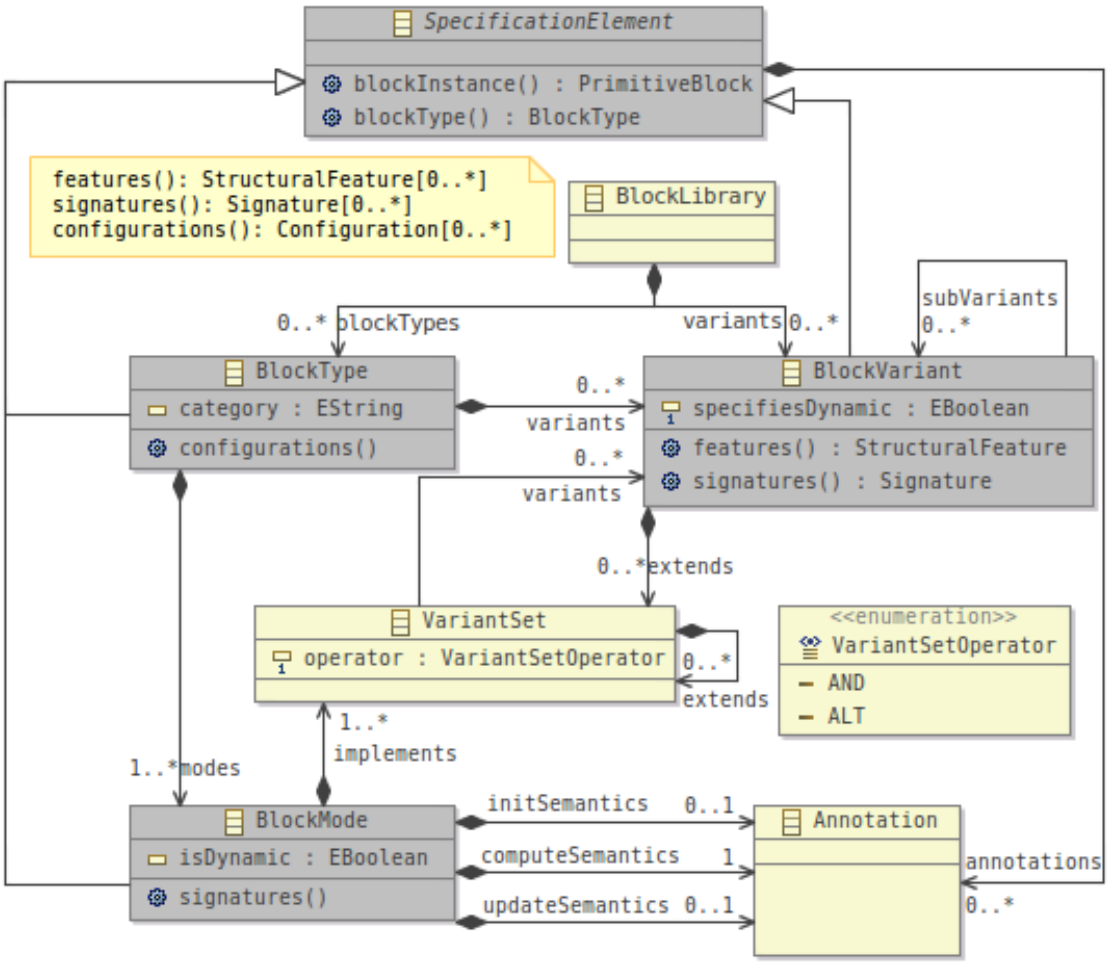


Figure 6.11: The BLOCKLIBRARY metamodel variability structure

Definition 6.2.7.1. *A* `BlockVariant` *is a partial interface specification for a block. It is a 10-tuple* $(\{pt\}, \{pg\}_i, \{pg\}_o, \{mv\}, \{vs\}, \{Inv_{mode}\}, Dyn, F, S, \{Def\})$*, where:* $\{pt\}$ *is a possibly empty set of* `Parameter`*;* $\{pg\}_i$ *is a possibly empty ordered set of input* `PortGroup`*;* $\{pg\}_o$ *is a possibly empty ordered set of* `PortGroup`*;* $\{mv\}$ *a possibly empty set of* `MemoryVariable`*;* $\{vs\}$ *a possibly empty set of* `VariantSet` *(we will detail this later);* $\{Inv_{mode}\}$ *is a possibly empty set of* `MODE_INVARIANT` *annotations constraining the structural elements values in the specification. Dyn specifies, whether the variant is dynamic meaning that its* `MODE_INVARIANT` *annotations are referring to the values of a dynamic element of the specification (*`PortGroup` *or* `MemoryVariable`*) this has an impact on the interpretation of the* `BlockVariant` *as detailed in the* `BlockMode` *element specification that follows.* F : `BlockVariant` $\rightarrow$ *Set(*`StructuralFeature`*) is the* `features()` *operation that returns the set of all* `StructuralFeature` *elements contained by self (see Section 6.4.1).* S : `BlockVariant` $\rightarrow$ *Set(*`Signature`*) the* `signatures()` *operation that returns the set of all* `Signature` *that can be extracted from self (see Section 6.4.2).* $\{Def\}$ *is a possibly empty set of* `DEFINITION` *annotations.*

In the *Delay* block specification, the `ICScalar` and `ICVector` `BlockVariant` defines the two partial interfaces for the block. In the first one, the parameter `initial_condition` is of type `TDouble` whereas on the second one, the same parameter is of type `TArrayDouble`. Using the tuples representation, this

example would be represented as (6.3).

$$\begin{aligned} \texttt{initial_condition}_1 &= (\mathit{TDouble}, 1, 1, \mathit{DATA_PORT}) \\ \texttt{initial_condition}_2 &= (\mathit{TArrayDouble}, 1, 1, \mathit{DATA_PORT}) \\ \mathit{ICScalar} &= (\{\texttt{initial_condition}_1\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \mathit{false}, \mathit{features}(), \mathit{signatures}(), \emptyset) \\ \mathit{ICVector} &= (\{\texttt{initial_condition}_2\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \mathit{false}, \mathit{features}(), \mathit{signatures}(), \emptyset) \end{aligned} \tag{6.3}$$

Definition 6.2.7.2. *A `BlockMode` represents one possible semantics configuration of a block specification. It is a 7-tuple* $(Init, Compute, Update, \{vs\}, Dyn, S, \{Inv_{mode}\})$*, where: Init, Compute and Update gathers the respective semantics phases functions of the block that are composed of at least one `DEFINITION Annotation` holding operational semantics and optionally composed of `PRE/POST Annotation` holding axiomatic specification;* $\{vs\}$ *is a non empty set of `VariantSet`; Dyn specifies, whether the specified semantics is dynamic – meaning that the specified semantics depends on the verification of a constraint on the value of a dynamic element (an input `PortGroup` or a `MemoryVariable`) (we will detail this just after this definition);* S : `BlockMode` $\rightarrow$ *Set(`Signature`) is the `signatures()` operation that returns the set of all `Signature` that can be extracted from self (see Section 6.4.2); and* $\{Inv_{mode}\}$ *are the `MODE_INVARIANT` annotations defined in the `BlockMode` context constraining the `StructuralFeature`.*

Init, *Compute* and *Update* provide the specification of the block configuration semantics. This semantics is provided as `DEFINITION Annotation` elements.

Dynamic semantics specifications

Dynamic semantics specifications are used in order to provide semantics definitions for specific executions of a block configuration. This will allow to split complex semantics specifications as distinct behaviors. Such segmentation should be done according to conditions expressed in the $\{Inv_{mode}\}$ `MODE_INVARIANT Annotation` elements.

An example of such a block specification is provided for the *Abs* block in Listing 6.12. This simple block outputs the absolute value of its only input port to its only output port. We first declare the block input and output ports in `BlockVariant Abs_Root` and then we provide two `BlockMode`, one for the case when the input is negative: `Abs_Neg`; and the other one when the input is positive or null: `Abs_PosOrNull`. Both `BlockMode` specification specifies dynamic behaviors of the block.

Partial interface building structure

In the *Delay* block specification, the `DelaySemantics BlockMode` defines one semantics for the block as provided in Listing 6.13.

It provides the three phases of a block execution semantics through the `DEFINITION Annotation`: `init_Delay`, `compute_Delay` and `update_Delay`.

Definition 6.2.7.3. *A `VariantSet` is a 3-tuple:* $(\{vs\}_{ext}, \{bv\}, Op)$*, where:* $\{vs\}_{ext} \in$ `VariantSet`* *is a possibly empty set of `VariantSet` that the current `VariantSet` extends;* $\{bv\} \in$ `BlockVariant`* *a set of contained `BlockVariant` and; Op = AND | ALT, is an operator stating whether the `VariantSet` referred in* $\{vs\}_{ext}$ *and the `BlockVariant` referred in* $\{bv\}$ *are necessarily (`AND`) related or alternatively (`ALT`) – meaning only one – related to the current `VariantSet`. `AND` and `ALT` relations are the n-ary versions of the `and` and `xor` logical relations applied to a set of `BlockVariant` and `VariantSet`. `VariantSet` relations link to other `VariantSet` relations via the `extends` reference making it possible to build complex relations.*

We will illustrate the `VariantSet` elements declaration and use based on the `Mem BlockVariant` declaration of the *Delay* block specification:

```
library SimpleBlocks {
  // Primitive types
  type realDouble TDouble

  blocktype Abs is Combinatorial {
    variant Abs_Root isDynamic{
      in data E1 : TDouble
      out data S1 : TDouble
    }
    mode Abs_Neg implements Abs_Root {
      modeinvariant ocl { E1.value < 0.0 }
      definition bal = compute_Abs_Neg {
        postcondition ocl { S1.value = - E1.value }
        postcondition ocl { S1.value >= 0.0 }
        S1.value = - E1.value;
      }
      compute compute_Abs_Neg
    }
    mode Abs_PosOrNull implements Abs_Root {
      modeinvariant ocl { E1.value >= 0.0 }
      definition bal = compute_Abs_PosOrNul {
        postcondition ocl { S1.value >= 0.0 }
        postcondition ocl { S1.value = E1.value }
        S1.value = E1.value;
      }
      compute compute_Abs_PosOrNul
    }
  }
}
```

Listing 6.12: The *Abs* block specification using BLOCKLIBRARY textual syntax

```
    mode DelaySemantics implements Mem {
      definition bal = init_Delay {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Delay {
        postcondition ocl { Output.value = Mem.value->first() }
        Output.value = Mem.value[0];
      }
      definition bal = update_Delay {
        postcondition ocl {
          Mem.value->last() = Input.value
        }
        postcondition ocl {
          Mem.value = (Mem.value@pre)->subList(2,Delay.value)->append(Input.value)
        }
        for (int i=0; i < (Delay.value - 1); i = i + 1) {
          Mem.value[i] = Mem.value[i + 1];
        }
        Mem.value[Delay.value - 1] = Input.value;
      }
      init init_Delay
      compute compute_Delay
      update update_Delay
    }
```

Listing 6.13: The `DelaySemantics BlockMode`

```
variant Mem extends allof (UScalar, oneof(ICScalar, ICVector))
{
  invariant ocl { IC.value.size() = Delay.value }
  memory Mem {
    datatype auto ocl {U.value}
    length auto ocl {Delay.value}
  }
}
```

Extension relations modeled by `VariantSet` link `BlockVariant` elements with other `BlockVariant` elements or `BlockMode` with other `BlockVariant` elements. These are expressed in the textual syntax using the `extends` and `implements` keywords. These keywords, taken from the JAVA language, are followed by either one `BlockVariant` element name – in this case the modeled `VariantSet` has an AND operator and only one `BlockVariant` in its `variants` relation – or a combination of `oneof` and `allof` keywords respectively modeling AND and ALT `VariantSet` elements – in this case, each `allof` or `oneof` keyword model a `VariantSet` with respectively an AND or an ALT operator and the following `BlockVariant` referenced by their name as the content of their `variant` relation. It is possible to combine these constructs to build complex compositions of `VariantSet` such as the one previously provided in Figure 6.2.

In the *Delay* block specification, the `MemSimple` `BlockVariant` (lines 36 to 45) is specified as extending either the `UVectorAndICVector` `BlockVariant` or extending both `UScalar` and `ICScalar`. The `allof(...)` construct (line 38) refers to a `VariantSet` with a AND operator whereas the `oneof(...)` construct (lines 36 to 39) refers to a `VariantSet` with an ALT operator. This construction allows to build two possible partial block interfaces. The first configuration gathers `SimpleDelay` and `UVectorAndICVector`. As these `BlockVariant` have also been built from other `BlockVariant`, it recursively includes `UVector` and `ICVector` (line 33) and `DelayParameter` (line 19). The mechanism for the resolution of the configurations that we have detailed here will be formalised in Section 6.4.

6.2.8 BLOCKLIBRARY METAMODEL SPECIFICATION CONTAINERS

Definition 6.2.8.1. *A `BlockType` holds the full specification for one block. The `BlockMode` and `BlockVariant` elements are combined to specify `BlockType`. It is a 3-tuple:* $(\{bv\}, \{bm\}, C)$*, where:* $\{bv\}$ *is a non empty set of `BlockVariant` elements,* $\{bm\}$ *is non empty set of `BlockMode` elements and* C : `BlockType` $\rightarrow$ `SetConfiguration` *the function defined in Section 6.4.5 extracting a set of `Configuration` element from a `BlockType` specification.*

It is important to ensure that a `BlockType` element holds `BlockMode` elements with different names. As such, we will ensure that it is possible to refer to a `BlockMode` according to its name in the context of a `BlockType`. This is formalised in the `AllBlockModesHaveDifferentNames` OCL invariant in Listing 6.14.

```
context BlockType

inv AllBlockModesHaveDifferentNames:
self.modes->forAll(m1, m2 | m1 <> m2 implies m1.name <> m2.name)
```

Listing 6.14: AllBlockModesHaveDifferentNames OCL constraint

Definition 6.2.8.2. *A BLOCKLIBRARY is a 2-tuple:* $(\{bt\}, \{bv\})$*, where:* $\{bt\} \in$ `BlockType`$^+$ *a set of `BlockType` elements.* $\{bv\}$ *is a non empty set of `BlockVariant` referred to as global `BlockVariant`. Such elements can be declared to hold `StructuralFeature` elements specification that can then be used in all `BlockType` specifications from the same block library.*

As previously stated for `BlockMode`, it is important in a BLOCKLIBRARY context to ensure that all the `BlockVariant` elements have a distinct name relatively to the `BlockVariant` contained in each `BlockType` specification. The same property should apply to `BlockType` names. Thus, is is possible, in the context of a `BlockType` to refer to a `BlockVariant` using its name, and in the BLOCKLIBRARY context to refer to a `BlockType` element from its names. We formalise them in the

```
context BlockLibrary

inv AllBlockVariantsHaveDifferentNamesInBlockType:
  self.blockTypes->forAll(bt|
    bt.variants->union(self.variants)->forAll(bv1,bv2|
      bv1 <> bv2 implies bv1.name <> bv2.name
    )
  )

inv AllBlockTypesHaveDifferentNames:
  self.blockTypes->forAll(bt1, bt2 |
    bt1 <> bt2 implies bt1.name <> bt2.name
  )
```

Listing 6.15: BLOCKLIBRARY metaclass OCL constraint

`AllBlockVariantsHaveDifferentNamesInBlockType` OCL invariant and the `AllBlockTypesHaveDifferentNames` OCL invariant provided in Listing 6.15.

6.3 RELATION TO FEATURE MODELING

A BLOCKLIBRARY element is – on a first approximation – a placeholder for `BlockType` elements. Each of these `BlockType` are meant to contain all the possible structural configurations for a block and their relative semantics. We have shown the ability – by using the BLOCKLIBRARY DSML– to segment a block specification into partial interfaces of blocks (`BlockVariant`) and to combine these interfaces into more complex interfaces through the `VariantSet` relations. We will show how the `BlockVariant` hierarchy, their attached `BlockMode` and their container `BlockType` can be related to features in the SPLE approach. The `VariantSet` thus corresponds to a set of constraint edges as presented in the FODA terminology by Kang in 1990 [83] and to the consists-of relations as proposed by Schobbens in 2007 [135].

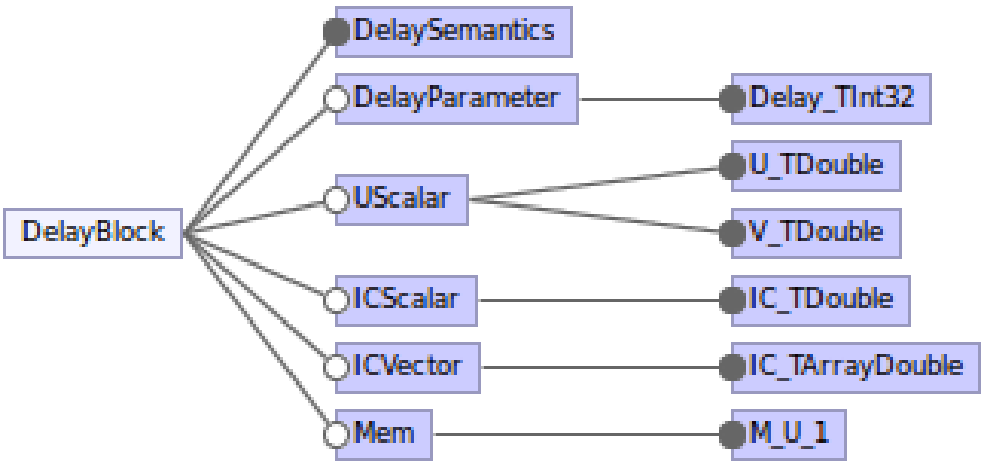
6.3.1 CONVERSION OF A BLOCKTYPE TO A FEATURE MODEL

If a `BlockType` instance is considered as the root feature for a feature model, then its contained `BlockVariant` and `BlockMode` should then be considered as its sub-features. Each `BlockVariant` `StructuralFeature` can be considered as a sub-feature of their containing `BlockVariant`. We provide an informal specification for the transformation taking as input a BLOCKLIBRARY instance specification and providing a FM for each of its `BlockType` elements.

1. A `BlockType` element is transformed to the root feature of a new FM.
2. A `BlockMode` elements are transformed to alternative sub-features of the root `BlockType` feature.
3. A `BlockVariant` (both the ones contained in the `BlockType` and the globally used ones) are transformed to optional sub-features of the root `BlockType` feature.
4. A `StructuralFeature` contained in `BlockVariant` are transformed to mandatory sub-features of their respective `BlockVariant` feature.
5. Every relation expressed using the `VariantSet` logical language is converted to a cross-tree constraint.

The application of this transformation by hand to the *Delay* block specification builds the FM provided in Figure 6.16 along with the cross tree constraints provided in (6.4).

Such a derived FM is not a typical feature model – at least not as it would have been designed by a human being. Indeed if we compare it to the one provided for the SPLE specification of the *Delay* block in Figure 5.42, we can see that it does not provide a convenient structuring of the block elements nor a model



$$\begin{aligned} UScalar &\Rightarrow DelayParameter \\ Mem &\Rightarrow UScalar \wedge (ICScalar \oplus ICVector) \\ DelaySemantics &\Rightarrow Mem \end{aligned} \tag{6.4}$$

Figure 6.16: FM extracted from the *Delay* block `BlockType` specification

whose meaning is easy to apprehend for a human being. Refactoring of the model should be done in order to transform a significant part of the cross-tree constraints to hierarchical relations between features. We are confident that parts of these refactoring can be done automatically by transforming the `VariantSet`-based expressions to hierarchical relations. This would provide the user with a more human readable FM and help the block specifier on his/her specification design.

Although this raw FM has no interest for human reading, many can be found for automated analysis as is explained in the next subsection.

6.3.2 Automatic feature model analysis

A number of analysis techniques for FM have been developed in the SPLE community over the past 20 years. A summary has been presented by Benavides 2010 [20]. Among these analysis techniques, not all are relevant for a BlockLibrary instance analysis. We comment here on some of the most appropriate ones and provide hints on their use on BlockLibrary instances:

- **Computation of the set of all products**. Extracting all the possible products defined through a FM allows to "*identify new valid requirements combinations not considered in the initial scope of the product line*" [20]. This would allow finding all the possible configurations for a `BlockType` and extracting them. Such set of configurations could then be used as a basis for the verification of some properties over the specification such as its disjointness as expected in REQ-7.b or its completeness as expected in REQ-7.c. Semantics correctness of the specification, as required in REQ-8, must also use this kind of computations as each extracted block configuration would provide a context for the evaluation of its contained semantics specification. We have implemented such an automated extraction mechanism for the BlockLibrary instances (details are provided in Section 6.4).
- **Assessment of the conformance of a product to its FM**. We must be able to decide whether there is a `BlockMode` that corresponds to a given block instance. Such mechanism can be used for the verification of block configurations according to their specification. Such assessment provides for each input block a unique possible configuration stored in a `Configuration` element as specified in Section 6.4.2. A failure in the matching of the block instance to a `Configuration` will then be the result of either flaws in the specification or an invalid block instance writing. As a result, ensuring the correctness of the specification will grant the ability to check the block instances.

- **Detection of anomalies in the products specification**. In SPLE and feature modeling, the hierarchy of features along with the cross-tree constraints might cause the apparition of anomalies in the model. An example of anomaly in FM is the impossibility for a feature to be allowed in any product of the feature model (dead feature). Leaving such anomalies in a FM does not prevent from the extraction of all products but it is important to ensure their detection as they might cause flaws in the specification due to the fact that the elements specified in a dead feature are not taken into account in any product. We apply this detection to our BlockLibrary instances, it allows us to find if some `BlockVariant` are not implemented. The detection of anomalies verification has been implemented as a verification of the BlockLibrary instances on the BlockLibrary editor. Such verifications are based on the configuration extraction calculus described in Section 6.4 and are formalised in Section 6.6.
- **Filtering**. Filtering of a product line allows to extract from a feature model and a partial configuration the set of all matching products configurations that contains the provided partial configuration. This allows for example to extract from a product line the set of all products containing specific features. Filtering can be applied on the BlockLibrary elements in order to find the potential configurations from partial block instance informations. We refer to such filtering in Section 8 while dealing with automatic code generation verification.

6.4 From block specification to configurations

From a block specification, it is mandatory to express the set of all possible configurations. As previously claimed, this will allow extracting valuable informations in order to perform verifications and data extraction.

In this section, we provide the specification for the function extracting all the block configurations specified in a `BlockType` element. We first give some definitions for operations used in this function. We describe the `Configuration` construct and the operations applied to it. We then provide an algorithm for the extraction of `Configuration` elements and finally we give some computation examples on the *Delay* block specification.

6.4.1 Preliminary operations definitions on BlockLibrary elements

- $|X|$: Set() $\rightarrow$ integer

  Compute the cardinal of a collection.

- $SF(e)$: `BlockVariant` $\rightarrow$ Set(`StructuralFeature`)

  Compute the set of `StructuralFeature` elements held in e. This is implemented as the *features()* operation in the `BlockVariant` metaclass.

  Returns a possibly empty set of structural features.

$$SF(e) = \{e.inputs\} \cup \{e.outputs\} \cup \{e.parameters\} \cup \{e.memoryVariables\} \tag{6.5}$$

- $SC_{\texttt{INVARIANT}}(e)$: `StructuralFeature` $\rightarrow$ Set(`Annotation`)

  Compute the set of structural constraints of e. Stuctural constraints are the `INVARIANT` `Annotation` gathered through the *annotation* reference of e.

  Returns a possibly empty set of `Annotation` elements.

$$\begin{aligned} &\forall e, e : \texttt{StructuralFeature}, \\ &SC_{\texttt{INVARIANT}}(e) = \{a \in e.annotations | a.kind = \texttt{INVARIANT}\} \end{aligned} \tag{6.6}$$

- $SC_{\texttt{MODE_INVARIANT}}(e)$: $\texttt{BlockMode} \cup \texttt{BlockVariant} \rightarrow \text{Set}(\texttt{Annotation})$

  Compute the set of structural constraints of *e*. Structural constraints are the `MODE_INVARIANT` `Annotation` gathered through the *annotations* reference of *e*.
  Returns a possibly empty set of `Annotation` elements.

$$\begin{aligned} &\forall e, e : \texttt{BlockMode} \vee e : \texttt{BlockVariant}, \\ &SC_{\texttt{MODE_INVARIANT}}(e) = \{a \in e.annotations | a.kind = \texttt{MODE_INVARIANT}\} \end{aligned} \tag{6.7}$$

- $VS^X(e)$: $\texttt{BlockMode} \cup \texttt{BlockVariant} \cup \texttt{VariantSet} \rightarrow \text{Set}(\texttt{VariantSet})$

  Compute the set of `VariantSet` elements contained in *e* through the *implements* reference (if *e* is a `BlockMode`) or the *extends* reference (if *e* is a `BlockVariant` or a `VariantSet`). Each `VariantSet` element should have its *operator* attribute set to **X**.
  Returns a possibly empty set of `VariantSet`.

$$\begin{aligned} &\forall X, X = AND \vee X = ALT \rightarrow \forall e.(\\ &(e : \texttt{BlockMode} &&\rightarrow VS^X(e) = \{vs \in e.implements | vs.operator = X\}) \\ &\wedge (e : \texttt{BlockVariant} &&\rightarrow VS^X(e) = \{vs \in e.extends | vs.operator = X\}) \\ &\wedge (e : \texttt{VariantSet} &&\rightarrow VS^X(e) = \{vs \in e.extends | vs.operator = X\})) \end{aligned} \tag{6.8}$$

- $BV(v)$: $\texttt{VariantSet} \rightarrow \text{Set}(\texttt{BlockVariant})$

  Compute the set of `BlockVariant` elements contained in *vs*. These are the `BlockVariant` elements referred through the *variants* reference of the `VariantSet` metaclass.
  Returns a possibly empty set of `BlockVariant`.

$$\forall v, v : \texttt{VariantSet}, BV(v) = v.variants \tag{6.9}$$

6.4.2 Configuration and Signature constructs

From a BlockLibrary instance, we define two data structures: `Signature` and `Configuration` both used in order to store the result of the extraction of all the block specifications from a `BlockType` element. These elements are defined in a separate package in the BlockLibrary metamodel. An extract of the metamodel package content is provided in Figure 6.17. We will provide in the following the definitions for these two elements.

Definition 6.4.2.1. *The `Signature` metaclass gathers the mandatory elements for a block interface specification. It is a 4-tuple $(D, bm, \{bv\}, F)$, where D is a derived attribute specifying whether the signature is dynamic or not (*isDynamic *attribute); bm is an optional `BlockMode` element holding a semantics specifications for the signature; $\{bv\}$ is a set of `BlockVariant` elements that are the partial interfaces specifications of the signature; F is the* features() *operation returning all the `StructuralFeature` elements contained in the $\{bv\}$ elements (defined in Section 6.4.1). `Signature` instances are created by calling the* signatures() *operation on either a `BlockVariant` or a `BlockMode` (defined in Section 6.4.4).*

A dynamic `Signature` is a `Signature` element holding at least one `BlockVariant` element that has its *isDynamic* flag set to true. Leading to the interpretation of the `Signature` as one interface of a block for which at least one constraint deals with the value of a dynamic `StructuralFeature`. The *isDynamic* derived attribute is defined using the OCL language in Listing 6.18.

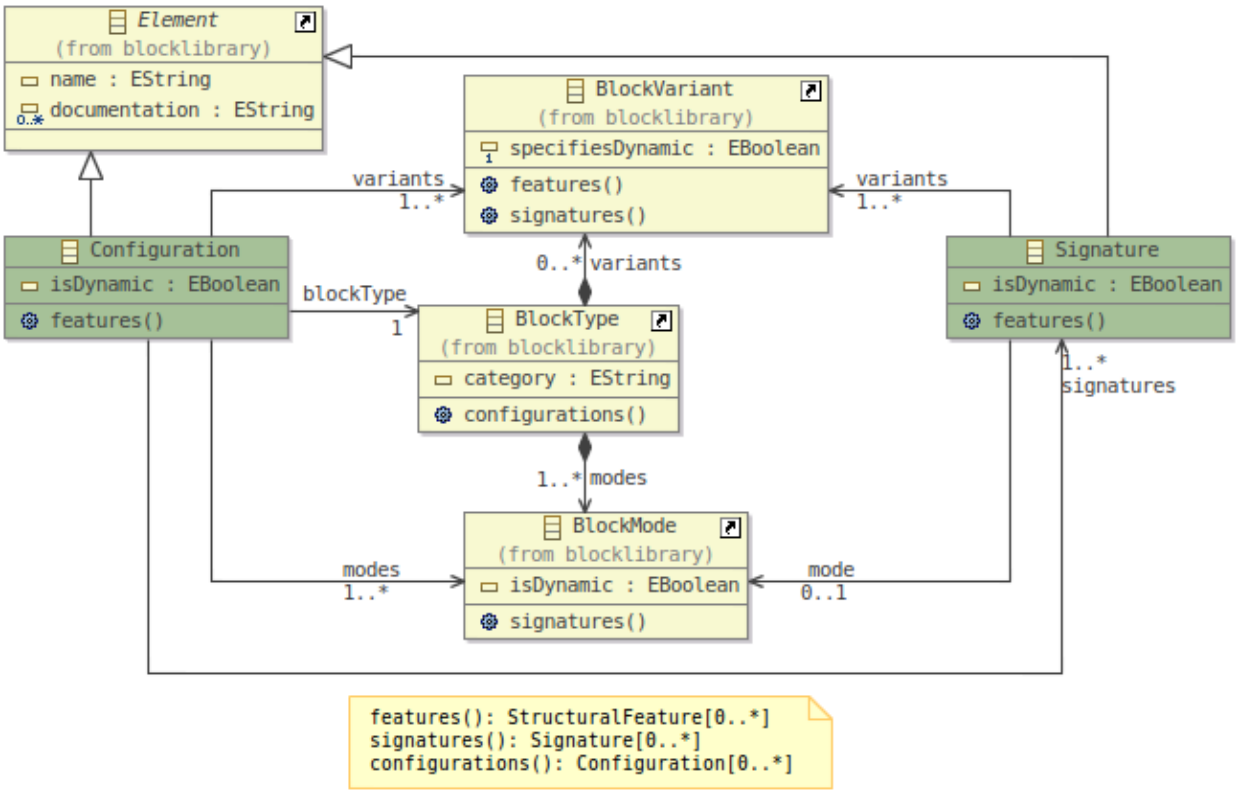


Figure 6.17: The `Configuration` metaclass

```
context Signature::isDynamic : Boolean
derive: variants->exists(var| var.isDynamic)

context Signature
inv SFNamesConflictInSignature:
  self.variants->collect(bv|
    bv.features())->forAll(p1,p2|
      p1 <> p2 implies p1.name <> p2.name
    )

inv MaxOneUnboundedInPGinSignature:
  let inputs = self.variants->collect(v| v.inputs) in
  inputs->select(in| in.max_size = -1)->size() <= 1
```

Listing 6.18: MaxOneUnboundedInputPortGroupinSignature OCL constraint

For a `Signature` element we add a constraint on the uniqueness of the names of the `StructuralFeature` it contains. This will allow to ensure the possibility to refer to `StructuralFeature` elements from their name in the context of a `Signature`. We formalise this as the *SFNamesConflictInSignature* OCL constraint in Listing 6.18.

Multiple input `PortGroup` elements are allowed in a single `Signature` `BlockVariant` elements. If more than one of these input `PortGroup` is unbounded (*max_size* attribute set to -1) then it will not be possible anymore to ensure that there is only one match for a block instance according to this `Signature`. We thus ensure that there is only one unbounded input `PortGroup` element in a `Signature`. This is formalised as the *MaxOneUnboundedInPGinSignature* OCL constraints in Listing 6.18.

Definition 6.4.2.2. *The `Configuration` metaclass gathers the required elements for a block configuration specification. It is a 5-tuple* $(D, T, \{bm\}, \{bv\}, F)$*, where* D *specifies whether the configuration is dynamic or not* (isDynamic *attribute*)*;* T *is the reference* blockType *to a `BlockType` element;* $\{bm\}$ *is a set of `BlockMode` elements that are the different behaviors specifications for the configuration;* $\{bv\}$ *is a set of `BlockVariant`*

```
context Configuration::isDynamic : Boolean
derive: variants->exists(var| var.isDynamic)

context Configuration
inv SFNamesConflictInConfiguration:
  self.variants->collect(bv|
    bv.features()
  )->forAll(p1,p2|
    p1 <> p2 implies p1.name <> p2.name
  )
```

Listing 6.19: `Configuration` isDynamic definition using OCL

```
/* pre: sigs->collect(s| s.mode)->asSet()->size() = 1
   post: result->forAll(s| s.variants->includesAll(bvs))
   post: result->forAll(s| s.mode = sigs->first().mode)
   post: let varSigs: Collection(Collection(BlockVariant)) =
           sigs->collect(s|s.variants) in
             varSigs->forAll(s|
               result->select(r| r.variants)->exists(r|
                 r->includesAll(s)
               )
             )
*/
includeBV(sigs, bvs) =
  sigs' = Set<Signature>{};
  foreach sig : Signature in sigs do
    s = new Signature();
    s.mode = sig.mode;
    s.variants.addAll(sig.variants);
    s.variants.addAll(bvs);
    sigs'.add(s);
  done;
  return sigs';
```

Listing 6.20: includeBV algorithm specification

elements that are the structural elements of the configuration; F is the features() *operation returning all the* `StructuralFeature` *elements contained in the* $\{bv\}$ *elements.* `Configuration` *instances are created by calling the* configurations() *operation on a* `BlockType` *element (defined in Section 6.4.5).*

A dynamic `Configuration` holds at least one `Signature` element that has its *isDynamic* flag set to true. The *isDynamic* derived attribute is defined using the OCL language in Listing 6.19.

For each configuration extracted from the `BlockMode` elements contained in a `BlockType`, the `StructuralFeature` elements name should be unique in order to be able to refer to these elements by their name. This property is ensured in the *SFNamesConflictInConfiguration* OCL invariant formalised in Listing 6.19.

6.4.3 Operations based on Signature constructs

We detail here the operations that allow the manipulation of sets of `Signature` constructs.

- *includeBV*(*sigs*, *bvs*): Set(`Signature`) → Set(`BlockVariant`) → Set(`Signature`)

  appends the second argument collection of `BlockVariant` elements in each `Signature` of the first argument collection. We define this operation with the algorithm provided in Listing 6.20.

  Returns a possibly empty set of `Signature`.

- *distributeBV*(*sigs*, *bvs*): Set(`Signature`) → Set(`BlockVariant`) → Set(`Signature`)

```
/* pre: sigs->collect(s| s.mode)->asSet()->size() = 1
   post: result->collect(r| r.variants) = bvs->collect(bv|
           sigs.variants->including(bv)
         )
   post: result->forAll(s| s.mode = sigs->first().mode)
*/
post: result->forAll(r| r.mode = sigs->first().mode)
distributeBV(sigs, bvs) =
  sigs' = Set<Signature>{};
  foreach sig : Signature in sigs do
    foreach bv : BlockVariant in bvs do
      s = new Signature();
      s.mode = sig.mode;
      s.variants.addAll(sig.variants);
      s.variants.add(bv);
      sigs'.add(s);
    done;
  done;
  return sigs';
```

Listing 6.21: distributeBV algorithm specification

distributes all the `BlockVariant` elements of the second argument into the `Signature` provided as first argument. We define this operation with the algorithm provided in Listing 6.21.

Returns a possibly empty set of sets of `Signature`.

6.4.4 Extraction of Signature elements

We detail here the algorithms for the extraction of `Signature` instances as they are specified in Figure 6.17. These are created by a call to the *signatures()* operation. There are two implementations of the *signatures()* operation, one is specified on the `BlockVariant` metaclass and the other on `BlockMode` metaclasses. We will use a single algorithm for both implementations.

- *Signatures*(e): `BlockVariant` $\cup$ `BlockMode` $\rightarrow$ Set(`Signature`)

  Extracts a set of `Signature` elements starting from e. Each `Signature` holds a unique path through the `BlockType` instance specification tree going from e to the root of the tree via the relations defined by the `VariantSet` elements. Detailed specification of this algorithm is provided in Listing 6.22. We depend on the `VS_AND` and `VS_ALT` operations in this algorithm, they match to their respective VS^X operations defined in (6.8).

6.4.5 Extraction of Configuration elements

- *Configurations*(e): `BlockType` $\rightarrow$ Set(`Configuration`)

  Extracts a set of `Configuration` elements. Each `Configuration` holds a set of at least one `Signature` element. The collection of `Configuration` elements constitutes the complete set of `BlockType` instance specifications. Detailed specification of this algorithm is provided in Listing 6.23.

REQ-7.[b|c] verifications are based on the *Configuration(`BlockType`)* algorithm that extracts the complete set of `Configuration` elements from a `BlockType` instance. These verifications are detailed in Section 6.6.

```
/* post: e.oclIsTypeOf(BlockMode) implies result->forAll(s| s.mode = e)
   post: e.oclIsTypeOf(BlockVariant) implies result->forAll(s|
           s.mode.oclIsUndefined())
   post: result->collectNested(s| s.variants) =
         VS_ALT(e)->collectNested(vsALT|
           vsALT.variants->union(VS_AND(e)->collect(vsAND| vsAND.variants))) */
Signatures_init(e) =
  sigs = Set<Signature>{};
  Signature s = new Signature();
  sigs.add(s);

  if (e instanceof BlockMode) then
    s.mode = e;

  foreach vs : VariantSet in vs_AND(e)
    sigs = includeBV(sigs, vs.variants);
  done;

  foreach vs : VariantSet in vs_ALT(e)
    sigs = distributeBV(sigs, vs.variants);
  done;
  return sigs;

/* post: Signature_init(e)->collectNested(s|
           s.variants->collect(sub| Signature(sub))->collectNested(s2|
             s2.variants->union(s.variants)
           )
         )->isEmpty() implies
         (result->collectNested(s| s.variants) =
         Signature_init(e)->collectNested(s| s.variants))
   post: not (Signature_init(e)->collectNested(s|
           s.variants->collect(sub| Signature(sub))->collectNested(s2|
             s2.variants->union(s.variants)
           )
         )->isEmpty()) implies
         result->collectNested(s| s.variants) =
         Signature_init(e)->collectNested(s|
           s.variants->collect(sub| Signature(sub))->collectNested(s2|
             s2.variants->union(s.variants))) */
Signatures(e) =
  sigs = Signatures_init(e);

  result = Set<Signature>{};
  foreach sig : Signature in sigs do
    innerSigs = Set<Signature>{};
    foreach innerBV : BlockVariant in sig.variants do
      innerSigs.addAll(Signatures(innerBV));
    done;
    foreach inSig : Signature in innerSigs do
      inSig.variants.addAll(sig.variants);
      result.add(inSig);
    done;
  done;

  if (result.size() = 0) then
    return sigs;
  else
    return result;
  endif
```

Listing 6.22: BlockMode and BlockVariant signature extraction initialisation algorithm specification

```
/* post: return->forAll(c|
           c.modes->collect(m| Signatures(m))->oclAsSet()->size() = 1
         ) */
Configurations(bt) =
  sigs = Set<Signature>{};
  foreach bm : BlockMode in bt.modes do
    sigs.addAll(Signatures(bm));
  done;

  configs = Set<Configuration>{};
  foreach sig : Signature in sigs do
    c = new Configuration(bt);
    if (sig.mode != null) then
      c.modes.add(sig.mode);
    endif
    c.variants.addAll(sig.variants);
    configs.add(c);
  done;

  foreach pair: (c1: Configuration, c2: Configuration) in configs do
    if (c1.variants = c2.variants) then
      c1.modes.addAll(c2.modes);
      c2.delete;
    endif
  done;

  return configs;
```

Listing 6.23: BlockType configuration extraction algorithm specification

6.5 Semantics modeling

In a BlockLibrary instance, extracted configurations hold semantics phases definitions provided through `DEFINITION Annotation`. Each semantic `DEFINITION Annotation` needs to be provided either as an axiomatic semantics – defining the pre-conditions and post-conditions for each phase of the semantics – or as an operational semantics definition or even both, that must be correct one against the other. The advantage of providing both axiomatic and operational specification is on the verification capabilities regarding the specified semantics.

The verification of a block semantics specification cannot be achieved by only relying on the content of the axiomatic and operational semantics definition. We should also rely on the `StructuralFeature` elements defined in the `BlockVariant` contained in the block configuration. In a configuration, each `StructuralFeature` has a defined data type and some additional constraints on its value. Regarding configurations themselves, `BlockVariant` elements are extended with `MODE_INVARIANT` annotations corresponding to additional constraints on the block configuration. These elements must be used as additional pre-conditions of the semantics functions.

General purpose programming languages can be extended with annotations like C with ACSL, Java with JML, C# or F# with Spec#, Ada 2012, Eiffel or Why/WhyML that allows to express programs and related annotations specifying properties, assertions or contracts to be verified on the language constructs. As a `Configuration` element holds at least one semantics definition operations and variables declarations for ports, parameters and memories and their associated data types, we can envision a `Configuration` structure operational semantics definition to be translated as a function, its axiomatic semantics as a contract on the generated function, the `StructuralFeature INVARIANT Annotation` as annotations on the variables definitions and the `BlockVariant` and `BlockMode MODE_INVARIANT` as additional contract informations. We provide in Listing 6.24 such a hand-extracted function using the C programming language and ACSL as an annotation language.

In this section we will provide clarifications on the interpretation of a BlockLibrary specification in terms of executable functions. We will rely on the C language complemented with ACSL annotations to give a formalisation of the BlockLibrary specifications as function contracts. Simulink blocks specified

```
/*@ requires *delay > 0;
    requires *delay = 1; // 1 is taken from the size of iC
    requires \separated(mem, input, delay, output, iC);
    assigns *mem;
    ensures *mem == *iC; */
void init_Delay (
  double *mem, double *input, int *delay, double *output, double *iC)
{ *mem = *iC; }

/*@ requires *mem == *iC;
    requires *delay > 0;
    requires \separated(mem, input, delay, output, iC);
    assigns *output;
    ensures *output == *mem;  */
void compute_Delay (
  double *mem, double *input, int *delay, double *output, double *iC)
{ *output = *mem; }

/*@ requires *output == *mem;
    requires *delay > 0 ;
    requires \separated(mem, input, delay, output, iC);
    assigns *mem;
    ensures *mem == *input; */
void update_Delay (
  double *mem, double *input, int *delay, double *output, double *iC)
{ *mem = *input; }
```

Listing 6.24: Extracted annotation contract and function for the DelaySemantics `BlockMode` semantics phases

using the BlockLibrary approach are meant to be used for the verification of automatically generated code, in this purpose we will reuse the mapping provided here in Chapter 8.

6.5.1 Block semantics phases contracts

From the specification written for the *Delay* block provided in Listing 6.4, we shall be able to extract for each `Configuration` a function. Listing 6.24 provides an example of such contract for one `Configuration` extracted from the *Delay* `BlockType` (here the `initial_condition` parameter value is set to be a scalar). A function is extracted for each semantic phases declared in the *DelaySemantics* `BlockMode`: *init_Delay*, *compute_Delay* and *update_Delay*. Each function contract is then expressed using pre-conditions (*requires* ACSL annotations as in line 1) expressed from the `INVARIANT` `Annotation` of the
`StructuralFeature` elements and the `MODE_INVARIANT` of the `Configuration` `BlockVariant` (*requires* ACSL annotations as in line 2). The contract is completed with the pre/post conditions provided as axiomatic semantics of the `BlockMode` semantics phases specifications (*ensures* ACSL annotations as in line 3).

It is worth noting at this point of the specification that some `INVARIANT` have not been taken into account in our extracted annotation contracts: a) **Data types**: In the specification, the *input* port group is of type *TDouble*. In our translation as a contract clause, we only define this as declaring the *input* as a function parameter of type *double*. This should be specified carefully as the definition of this data type should be provided including its boundaries (minimum and maximum values) and its allowed precision (number of digits in the decimal part). Similar informations should be provided for any data type used in the specification. b) **Dimensions**: We specify two `MODE_INVARIANT` `Annotation` in the *UVector* `BlockVariant`. These constraints seem to be redundant in our specification as they specify that both input *U* and output *V* are vectors elements but they are declared as arrays of double values. This problem is implementation related and will be detailed in Chapter 7.

From this example we offer a template for the generation of function and their contracts annotation from a BlockLibrary specification in Listing 6.25. This shows the generic extracted function contract and code from configuration *ConfZ* of `BlockMode` *BlockModeA* initialisation computation phase.

```
/*@ requires ConfZ.BlockVariant1.input1.inv1; ...
    requires ConfZ.BlockVariant1.output1.inv1; ...
    requires ConfZ.BlockVariant1.memory1.inv1; ...
    requires ConfZ.BlockVariantP....
    requires ConfZ.BlockVariant1.mode_invariant1; ...
    requires ConfZ.BlockVariantP....
    requires \separated(input_1,...,input_m,ouput_1,...,output_n,memory_1,...,memory_o);
    requires ConfZ.BlockModeA.init.pre1;
    ...
    requires ConfZ.BlockModeA.init.preN;
    assigns ...;
    ensures ConfZ.BlockModeA.init.post1;
    ...
    ensures ConfZ.BlockModeA.init.postM;
*/
void BlockX_BlockModeA_ConfZ_Init_Semantics(
  input_1, ..., input_m,
  output_1, ..., output_n,
  memory_1, ..., memory_o)
{
  <initialisation code>;
}
```

Listing 6.25: Extracted annotation contract and function body for an initialisation phase

6.5.2 Block semantics contract encoding with dynamic behaviors

As seen previously, some `MODE_INVARIANT` may apply on dynamic values of input `PortGroup` or `MemoryVariable` producing dynamic `Configuration`. These may also be specified through multiple `BlockMode`. This allows for a more fine grained specification of blocks semantics specially on the specification of behaviors according to input `PortGroup` values. The BlockLibrary structure then allows to split the specification of a block semantics between multiple `BlockMode`, each one having a distinct `MODE_INVARIANT`. This is translated as distinct behaviors in function contracts.

From the specification of the *Abs* block in Listing 6.12 (page 97) we can extract only one `Configuration` element containing both `BlockMode` and one `BlockVariant`. The corresponding C + ACSL code for this `Configuration` must be as provided in Listing 6.26. In this listing we see that we define one *behavior* for each `BlockMode`, each one containing *assumes* clauses (lines 2 and 6) specifying the condition for the *behavior* to be considered applicable (these are the behaviors' pre-conditions). The *assumes* clauses are extracted from the `BlockMode` `MODE_INVARIANT`. For each *behavior* the post-conditions are extracted from the post-conditions specified in the semantics phases axiomatic definition and inserted as *ensures* clauses (lines 3, 4, 7 and 8). Finally the code for the semantics phase is extracted according to both the dynamic `MODE_INVARIANT` (involving the *if* conditionals) and the operational semantics for the considered semantics phase (involving the code inside each *then* branch).

We model a generic full dynamic semantics for a block based on the previous examples in Listing 6.27.

The phase function global preconditions (line 1 and 2) are taken from the `StructuralFeature` elements `INVARIANT`. As a `Configuration` is composed of a set of `Signature` elements, each `Signature` holds a behavior definition for the overall block semantics.

Each behavior is characterised according to the `MODE_INVARIANT` constraints extracted from the `Signature` `BlockMode` and its list of `BlockVariant`. Constraints from the `BlockVariant` are common to all `BlockMode` according to the `Configuration` definition, thus these constraints are to be included in the global pre-condition specification (as *requires* clauses – lines 3 and 4).

`MODE_INVARIANT` `Annotation` held in each `BlockMode` are used as pre-conditions for their respective behaviors, thus they are used as *assumes* clauses as in line 6 and 11. Pre-conditions specified in the semantic phase axiomatic specifications are converted to *assumes* clauses as in line 7. Post-conditions specified in the semantic phase are used as *ensures* clauses of their respective behavior as shown in line 8.

The function body is a combination of nested *if-then-else* constructs. Each *then* branch holds the code for a specific behavior. Each *if* condition is set to the conjunction of the `MODE_INVARIANT` constraints

```
/*@ requires \separated(e1, s1);
    assigns *s1;
    behavior abs_Neg:
      assumes *e1 < 0.0;
      ensures *s1 >= 0.0;
      ensures *s1 == - *e1;
    behavior abs_PosOrNull:
      assumes *e1 >= 0.0;
      ensures *s1 >= 0.0;
      ensures *s1 == *e1;
*/
void compute_Abs_Neg_Abs_PosOrNull (double *e1, double *s1){
  if (*e1 < 0.0){
    *s1 = - *e1;
  } else if (*e1 >= 0.0){
    *s1 = *e1;
  }
}
```

Listing 6.26: Extracted annotation contract and function for the Abs `BlockType` semantics

taken from the `BlockMode` (lines 20 to 22), the *then* branch content is then the operational semantics code related to this behavior (line 23).

```
/*@ requires ConfZ.BlockVariant1.structuralFeature1.inv1; ...
    requires ConfZ.BlockVariantP....
    requires \separated(input_1,...,input_m,ouput_1,...,output_n,memory_1,...,memory_o);
    requires ConfZ.BlockVariant1.mode_invariant1; ...
    requires ConfZ.BlockVariantP.mode_invariant1; ...
    assigns ...;
    behavior ConfZ_Step_BlockModeA:
      assumes ConfZ.BlockModeA.mode_invariant1; ...
      assumes ConfZ.BlockModeA.pre1; ...
      assigns ...;
      ensures ConfZ.BlockModeA.post1; ...
    ...
    behavior ConfZ_Step_BlockModeQ:
      assumes ConfZ.BlockModeQ.mode_invariant1; ...
      assigns ...;
      ...
*/
void BlockX.ConfZ_Semantics (a
  input_1, ..., input_m,
  output_1, ..., output_n,
  memory_1, ..., memory_p)
{
  if (ConfZ.BlockModeA.mode_invariant1 && ... &&
     ConfZ.BlockVariant1.mode_invariant1 && ... &&
     ConfZ.BlockVariantP.mode_invariant1 && ...){
    <ConfZ.BlockModeA.code>
    //@ assert ConfZ.BlockModeA.post1 && ...
  } else if (...) {
    ...
  } else if (ConfZ.BlockModeQ.mode_invariant1 && ... ){
    ...
  }
}
```

Listing 6.27: Generic specification for a block dynamic semantics phase for one `Configuration` (ConfZ)

The transformation depicted here has been implemented in order to translate BlockLibrary instance models to semantics functions in a formalism allowing to automatically and formally verify their correction. This is detailed in Chapter 7 and model REQ-8.

Whereas this phase-specific verification allows to verify each phase of the block semantics correctness, it does not model the full semantics of the block. In dataflow languages semantics all the initialisation phases are executed once and then at every clock tick, all the compute phases and then all the update phases are executed. This verification is equivalent to verifying the generated code for a complete Simulink instance model and will be tackled in the second part of this manuscript.

6.6 Specification verification properties

As previously stated, the quality of the specification should be ensured (according to REQ-7.[a|b|c] and REQ-8) in order for it to be usable for the verification of block instances. Such a verification must ensure properties based on three different criteria: well-formedness (REQ-7.a), completeness (REQ-7.b) and disjointness (REQ-7.c) of the block specification variability. In addition to these, semantics correctness (REQ-8) of the specification must be ensured. In this section, we will focus on the formalisation of the REQ-7.[a|b|c] verifications while the semantics verification is left aside and will be tackled in Chapter 7.

6.6.1 Well-formedness

Well-formedness of a BlockLibrary instance is verified according to the defined concrete and abstract syntaxes of the BlockLibrary. As our DSML relies on MDE foundations, it is straightforward to ensure its syntactic correctness. Indeed, the generated editors are based on the metamodel defined for the language and so instances will conform to the metamodel. OCL constraints are provided in order to ensure additional constraints as the one provided all along the current section. These checks ensures REQ-7.a.

6.6.2 Variability coverage

The variability criterion for the verification of a BlockLibrary instance is twofold. We should ensure: a) the completeness of every block specification according to requirement REQ-7.b; and b) the consistency implied by the disjointness of all the block specification according to requirement REQ-7.c.

Each `Configuration` element gathers the definition domain for a block instance. This definition domain is made up of definitions of `StructuralFeature` elements contained in the `Configuration`, their data types and additional constraints on their values. In the following, we will refer to such a domain for a configuration C as $\mathcal{D}_C$.

In a configuration C, each `MODE_INVARIANT` contained in the `BlockVariant` and `BlockMode` elements is a constraint that applies on $\mathcal{D}_C$. In that sense, they are logical predicates applying on `StructuralFeature` elements of the domain. As previously defined, this collection of structural constraints is provided by (6.7) for each `BlockMode` and `BlockVariant` of the configuration.

A configuration can then also be expressed as a predicate using the $SC_{\texttt{MODE_INVARIANT}}$ of every `BlockVariant` and `BlockMode` of a `Configuration` (6.10). We refer to $CP(c)$ as the configuration predicate for configuration c.

$$\forall c : \mathit{Configuration}, CP(c) = \left(\bigwedge_{1 \leq i \leq |c.\mathit{variants}|} SC_{\texttt{MODE_INVARIANT}}(c.\mathit{variants}_i) \right) \wedge \left(\bigvee_{1 \leq j \leq |c.\mathit{modes}|} SC_{\texttt{MODE_INVARIANT}}(c.\mathit{mode}_j) \right) \tag{6.10}$$

Completeness and disjointness criteria are properties applying on a block `Configuration`. Their assessment can thus be done in the context of the block specification itself and not only on configuration's. Such a domain on which all configurations can be expressed for one block b is referred to as $\mathcal{D}_b$ and is formally defined in (6.11).

$$\forall b, b : \texttt{BlockType}, \mathcal{D}_b = \bigcup_{1 \leq i \leq |b.\mathit{modes}|} \left(\bigcup_{c \in \mathit{Conf}(m_i)} \mathcal{D}_c \right) \tag{6.11}$$

Completeness of the specification

The completeness criterion (REQ-7.b) aims at ensuring, on the definition domain of a block, that all the possible configurations have been expressed. This is equivalent to ensuring that the conjunction of all the possible block configuration predicates can always be satisfied on the block domain. We propose to formalise this criterion as (6.12).

$$\forall b, b : \texttt{BlockType}, \bigvee_{1 \leq i \leq |b.modes|} \left(\bigvee_{1 \leq i \leq |Conf(m_i)|} CP(c_i) \right) \tag{6.12}$$

Disjointness of the specification

The disjointness criterion (REQ-7.c) aims at ensuring, on the definition domain of a block, that all pairs of possible configurations are disjoint and thus cannot be satisfied on the same block instance. This is equivalent to ensuring that every conjunction of two different configuration predicates cannot be verified. We propose to express this criterion as (6.13).

$$\forall b, b : \texttt{BlockType}, \bigwedge_{1 \leq i \leq |b.modes|} \left(\bigwedge_{\substack{1 \leq j,k \leq |Conf(m_i)| \\ j<k}} \neg(CP(c_j) \wedge CP(c_k)) \right) \tag{6.13}$$

We provide a specification and the related implementation for the transformation between a BlockLibrary instance to a verification formalism in Chapter 7. We will specify in this chapter the transformation from BlockLibrary instances to the previously presented predicates. We will highlight some tools allowing to automatically verify the generated predicates and quantify the ability for our approach to be applied on real size block specifications. We will finally provide hints on the capability of our automated approach to manually find errors in the specification through the automated proof mechanism.

7

BlockLibrary specifications formal verification

We previously presented the BlockLibrary language. As a specification dedicated language, confidence must be provided on its use. This is done through the verification of BlockLibrary instance correctness, completeness and consistency according to the defined language structure and semantics.

In the previous chapters, we specified the structure of the BlockLibrary conforming models by defining its metamodel and the operations that can be applied on it in order to extract meaningful informations from it: `Configuration` elements extraction from a `BlockType` instance. While this provides informations on the BlockLibrary specification content and the assurance of their conformance to the metamodel and to its static semantics (OCL constraints), it does not ensures its correctness.

BlockLibrary conforming models correctness must be expressed according to criteria. These criteria are detailed in REQ-7.[b|c] and REQ-8 defined in page 52. These requirements states: a) from a block specification, the set of extracted `Configuration` elements is disjoint and complete (REQ-7.[b|c]); and b) for each `Configuration` element, its semantics definition can be verified (REQ-8).

Asserting the correctness of a BlockLibrary instance can be done by language experts and through experimentation over time or by providing formal foundations for their understanding and analysis. Using the first approach, test cases could be provided as blocks in an environment model allowing to test all the `Configuration` according to the criteria and thus providing confidence on the provided specification. It has the drawback of not being exhaustive. Using the second approach, the criteria can be expressed using a formal verification language and thus be formally verified and be exhaustive. It is then required to provide a translational semantics for the BlockLibrary language using a transformation to a formal language.

In this chapter, we will apply the second approach as it provides immediate confidence on the BlockLibrary instances and its formal nature makes it unlikely to be a posteriori proven wrong. In order to do so, we choose to provide a transformation from BlockLibrary conforming models to the Why3 platform on which formal verification can be done thanks to SMT solvers and/or proof assistants. We will detail here the aformentioned verification strategy that consists in translating the language conforming models to a formal domain; extraction of variability correctness properties and block semantics phase function along with their verification. Verification will be experimented on realistic block examples; and scalability of the verification approach will be discussed. Finally, limitations of our specification language capabilities will be highlighted.

7.1 Verification prerequisites

The BlockLibrary language aims at structuring block specification and their variability. There are two aspects regarding the verification of BlockLibrary instances: 1) variability verification as modeled in

REQ-7.[b|c]; and 2) semantics verification correctness/consistency of the various semantics aspects as modeled in REQ-8. We offer to translate BlockLibrary instances to a formal logical data structure on which formal reasoning can be performed.

It is thus possible to apply various verification strategies among which are automatic proof via SMT solvers or the use of proof assistant like Coq. We choose to rely on the Why3 platform as it allows from a single formalism, the Why language and its WhyML extension, to target both verification strategies.

In the BlockLibrary language, `Annotation` elements are specialized as OCL constraints or BAL operations. Both of these languages AST are specified according to a metamodel. In the following we will quickly describe both languages implementations.

7.1.1 OCL specification

A tight integration of the OCL language in the BlockLibrary language has been done by relying on a pre-defined metamodel and grammar for the OCL language [160]. This OCL implementation covers an extensive part of the OMG OCL standard and includes the TOCL [163] extension done in Faiez Zalila PhD. As we have no need for the time-related constructs of the TOCL, we decided to lighten the metamodel and removed the time-related constructs. In a first attempt we planned to reuse the existing XText OCL grammars from the standard Eclipse tools. Most of them rely on the Essential OCL grammar which itself relies on other XText grammars in various Eclipse plugins. Those must be included if one wants to embed Essential OCL in a custom language. This solution revealed itself to be too heavy to maintain and we concluded it was easier to rely on a custom light XText grammar for our work.

`Annotation` elements are used in various BlockLibrary language constructs. The `Annotation` container is the scope or context of the OCL expressions. Its *kind* attribute is used for the definition of the specification purpose for the `Annotation`: for example whether it is specifying a pre/post condition or an invariant. These informations provided at the BlockLibrary level allows us to restrict the OCL language constructs supported in the minimal OCL expressions subset. We provide in Appendix B the XText grammar we have implemented.

7.1.2 BAL specification

The block semantics is expressed in a BlockLibrary specification using BAL. This language is a rather simple imperative language. We could have used general purpose action languages like ALF or fUML, or more computation related languages like EmbeddedMatlab or Scilab but we choose to rely on a very restricted imperative language in order to avoid to cover a too large language. This simple language also has the advantage of restricting the possibilities regarding the writing of the semantics functions and thus must enforce the specifier to explicitly detail its implementation. The language constructs are:

- Blocks of imperative code. These can contain:
  - Variable declarations: new variables declared and accessible only in the scope of the block containing the declaration. Variables can be shaped as scalar or multidimentional arrays.
  - Variable assignments: assign a value to a locally declared variable or to a `StructuralFeature` element.
  - Conditional constructs: if-then-else conditional statements.
  - Loop constructs: for and while loop statements.
- Logical, relational, arithmetic and literal expressions. Literal expressions values can be defined among integer, double, string , boolean or enumeration values. These expressions have only been provided a meaning through scalar values.

We provide a grammar for this language in Appendix C. We defined here a syntax and the structure of the AST for this language but no semantics. We choose to define this one in a translational way by providing a translation to the WhyML language. We will detail this transformation and thus provide a translational semantics for BAL language in Section 7.5.

7.1.3 Why3 platform

According to our use, the principal advantages of the Why and WhyML languages are their expressiveness – with both logical specification and programs specifications (along with their code) – and their associated tooling – providing transformation capabilities to build a bridge between specifications and automated or interactive theorem provers.

G. Babin, M. Carton and myself developed an XText grammar and a metamodel for Why and WhyML languages. In this PhD work, I relied on this generated tooling to develop a MDE-based translation of BlockLibrary conforming models to Why theories and WhyML modules.

7.1.4 Transformation technology choice

Our DSML has been implemented using MDE methodologies and tools. We used Ecore as a formalism for the expression of the OCL and BAL metamodels. Based on these metamodels, services can be defined and applied on the DSML instances using MDE techniques.

In our case, the transformation targets are the Why and WhyML languages. Both are quite complex languages. This leads us to rely on the use EMF framework combined with XText for the transformation implementation.

Using the XText framework capability to act as a bridge between textual and model representations and the EMF source and target model manipulation API capability to manipulate models, it is thus possible to develop a model to model transformation. The MDE architecture of the transformation is provided in Figure 7.1.

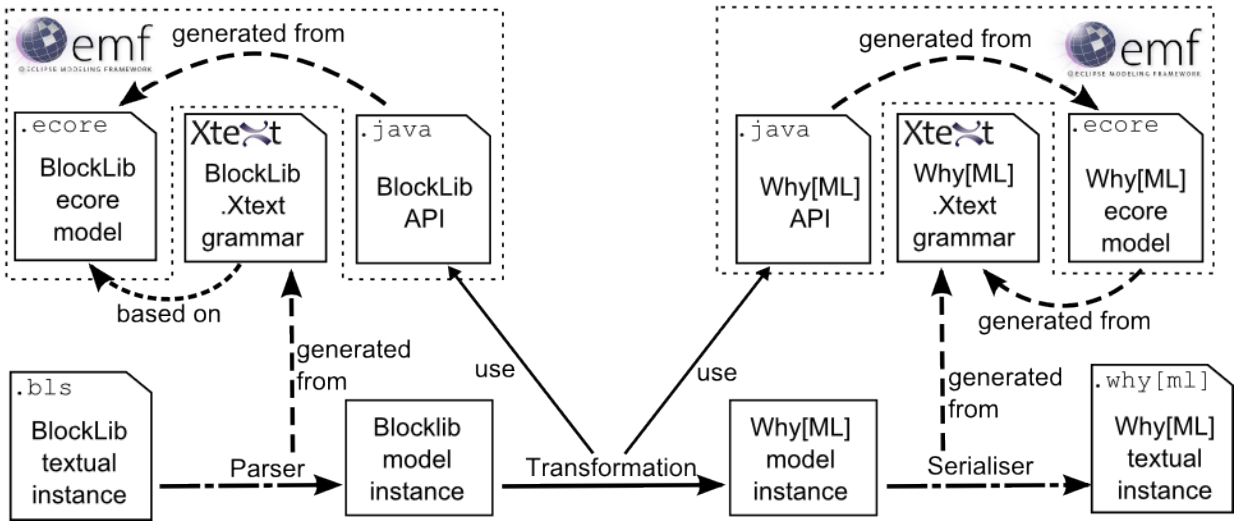


Figure 7.1: MDE architecture of the transformation

- On the BlockLibrary side (left side of Figure 7.1: we generate a parser and a textual code editor based on the BlockLibrary metamodel for the loading and edition of BlockLibrary conforming models.
- On the Why and WhyML side (right side of Figure 7.1): we generate a serialiser for saving as text files the generated Why and WhyML conforming models generated with the model to model transformation.

In the following, we will detail the implementation of the transformation by first providing static Why3 libraries of theories modeling BlockLibrary elements, data types and languages constructs. Then, we will express the two verification approaches: a) transformation from a BlockLibrary conforming model to theories held in Why files for the assessment of the variability verification criteria of REQ-7.[b|c]; and

b) transformation from a BlockLibrary instance to modules held in WhyML files for the assessment of the semantics verification criterion of REQ-8.

7.2 BlockLibrary specification example

All over this chapter the transformation mechanism will be illustrated based on the *MinMax* block specification example. An extract of its specification can be found in Listing 7.2. This example is focused on two configurations of the block where the number of inputs ports is greater than one, all input and output ports are of type `TDouble` and the function parameter value can be Min or Max.

7.3 Why3 libraries

Our implementation of the transformation relies on static libraries specifying: a) Why3 type definitions modeling the type system on which the BlockLibrary specification instances relies on (Section 4.1.2; b) Why3 record type definitions modeling the BlockLibrary `StructuralFeature` elements; c) Why3 function interface definitions and attached lemmas modeling OCL.

7.3.1 Primitive data types theory

The Why3 platform and languages provide a library for primitives data types. This library can be found on the tool web-site[1]. We specialized it for the modeling of our type system. We provide in Table 7.3 the mapping between the concrete metaclasses of our type system metamodel, its corresponding type name, its containing theory, and if applicable the standard Why3 type and theory it relies on.

Meta-class name	Why3 type name and theory	Why3 standard library theory
TBoolean	boolean_type from blocklibrary_scalar.Boolean	Bool.bool from bool.Bool
TRealSingle	tRealSingle from blocklibrary_scalar.RealSingle	real from real.RealInfix
TRealDouble	tRealDouble from blocklibrary_scalar.RealDouble	real from real.RealInfix
TRealInteger	tRealInteger from blocklibrary_scalar.RealInteger	int from int.Int
TString	string_type from blocklibrary_string.String	–

Table 7.3: Mapping between our type system and the Why3 types

Numeric data types definitions

The numeric data types definition from the standard library are appropriate for our needs. We decided to mostly rely on them in our implementation. *TBooolean*, *TRealSingle*, and *TRealDouble* are thus directly mapped to already existing standard Why3 types. The data types names choice have been done according to the GeneAuto data types terminology.

In the `TRealInteger` metaclass, the value of an instance is dependent on the `nBits` and `signed` attributes. In dataflow languages like Simulink, the type system allows the use of a specific definition of integer data types. We distinguished between multiple implementations of the integer data type according to the number of bits required for their representation (8, 16 or 32) and if they are signed of not. As an example of type definition, we give the declaration for a 32 bits signed `TRealInteger` in the `SignedInt32` theory detailed in Listing 7.4. The `RealInteger` theory is holding the definition for the `TRealInteger` data type as modeled in the data types metamodel, whereas the `SignedInt32` theory is the concrete definition for the 32 bits signed integer. We add a predicate: `limit_tRealSignedInt32` in the theory allowing to constrain the allowed maximum and minimum values for an element of this type.

[1] http://why3.lri.fr

```
library MinMaxExtractInv {
  type signed realInt TInt16 of 16 bits
  type realDouble TDouble
  type enum MinMaxFunction {Min,Max}

  blocktype MinMax {
    variant MinMaxParameters {
      parameter FunctionParam : MinMaxFunction
      parameter NbInputs : TInt16 { invariant ocl { NbInputs.value >= 1 } }
    }
    variant MinMaxInScalars extends MinMaxParameters {
      in data In1 : TDouble [1 .. *] { invariant ocl { In1->size() = NbInputs.value } }
    }
    variant MinMaxOutScalar {
      out data Out : TDouble
    }
    mode MinOutputScalarMultipleInputsScalars implements allof(
      MinMaxOutScalar,MinMaxInScalars)
    {
      modeinvariant ocl { NbInputs.value > 1 }
      modeinvariant ocl { FunctionParam.value = MinMaxFunction::Min }
      definition bal = compute_MinOutScalarMultipleInputsScalars {
        postcondition ocl {
          In1->forAll(i| i.value >= Out.value)
        }
        var res = In1[0].value;
        for (var i = 1; i < (size(In1)); i = i + 1){
          if (res > In1[i].value){
            res = In1[i].value;
          }
        }
        Out.value = res;
      }
      compute compute_MinOutScalarMultipleInputsScalars
    }
    mode MaxOutputScalarMultipleInputsScalars implements allof(MinMaxOutScalar,
        MinMaxInScalars) {
      modeinvariant ocl { NbInputs.value > 1 }
      modeinvariant ocl { FunctionParam.value = MinMaxFunction::Max }
      definition bal = compute_MaxOutScalarMultipleInputsScalars {
        postcondition ocl {
          In1->forAll(i| i.value <= Out.value)
        }
        var res = In1[0].value;
        for (var i = 0; i < (size(In1)); i = i + 1){
          if (res < In1[i].value){
            res = In1[i].value;
          }
        }
        Out.value = res;
      }
      compute compute_MaxOutScalarMultipleInputsScalars
    }
  }
}
```

Listing 7.2: *MinMax* block textual specification extract

```
theory RealInteger
  use import bool.Bool
  use import int.Int

  type tRealInteger

  constant nBits : int
  constant signed : bool
  constant max_RealInteger : int
end

theory SignedInt32
  use import int.Int
  use import bool.Bool

  constant nBits_signed_32 : int = 32
  constant signed_signed_32 : bool = True
  constant max_RealInteger_signed_32 : int = 2147483648

  type tRealSignedInt32 = int

  clone export RealInteger with
    constant nBits = nBits_signed_32,
    constant signed = signed_signed_32,
    constant max_RealInteger = max_RealInteger_signed_32,
    type tRealInteger = tRealSignedInt32

  predicate limit_tRealSignedInt32 (x : tRealSignedInt32) =
    (-max_RealInteger_signed_32) <= x <= (max_RealInteger_signed_32 - 1)
end
```

Listing 7.4: TRealInteger 32 bits signed definition in Why3

Interested reader can find our complete specification for data types in Appendix D.1.1.

String related data types definitions

At the time we implemented that part, in the Why3 standard library, the `String` data type was only available as a WhyML module. We needed for our purpose to be able to use it also in theories. We thus developed three theories in this purpose:

- `Char` theory defining the `tChar` record type holding the simple definition for a character as a record type whose single field is an integer value named `code`. This code refers to the UTF8 table values for the corresponding character. This theory is detailed in Appendix D.1.3. We define two functions applicable on `tChar` typed elements:
  - *toLower_char* : *tChar* $\rightarrow$ *tChar* returning the value of the parameter as a lower case character. Its attached lemmas define the conditions for its application, the operation applies only if the *code* value is between 32 and 90 (included).
  - *toUpper_char* : *tChar* $\rightarrow$ *tChar* returning the value of the parameter as an upper case character. As previously its attached lemmas define the conditions for its application.
- `Utf8` theory containing relevant UTF8 values mapping as constants. This theory is partly detailed in Appendix D.1.3.
- `String` theory containing the definition for the `string_type` type composed of a list of `tChar` elements. `String` theory is defined in Listings 7.5 and 7.6. We define two specific functions that can be used on `string_type` typed elements:
  - *concat* : *string_type* $\rightarrow$ *string_type* $\rightarrow$ *string_type* returning the concatenation of the two parameters (detailed in Listing 7.5). This Why theory includes both the function definition and its formalisation through lemmas.

```
theory String
  (* see module string.String *)
  use import int.Int
  use import Char
  use import list.Length
  use import list.Append
  use import list.List
  use import list.Mem
  use import list.NthNoOpt
  use import blocklibrary_common.CommonFunctions

  type string_type = list tChar

  function concat (s1 s2: string_type) : string_type = s1 ++ s2

  lemma concat_length: forall s1, s2: string_type.
    length (concat s1 s2) = length s1 + length s2

  lemma concat_l_cons: forall s1, s2: string_type, c1: tChar.
    concat (Cons c1 s1) s2 = Cons c1 (concat s1 s2)

  lemma concat_r_cons: forall s1, s2: string_type, c1: tChar.
    concat s1 (Cons c1 s2) = concat (concat s1 (Cons c1 Nil)) s2

  lemma concat_l_nil: forall s1, s2: string_type.
    (s1 = Nil -> concat s1 s2 = s2)

  lemma concat_r_nil: forall s1, s2: string_type.
    (s2 = Nil -> concat s1 s2 = s1)

  lemma concat_l_mem: forall s1, s2: string_type, c1: tChar.
    mem c1 s1 -> mem c1 (concat s1 s2)

  lemma concat_r_mem: forall s1, s2: string_type, c1: tChar.
    mem c1 s2 -> mem c1 (concat s1 s2)

  function toLower (s1: string_type) : string_type

  axiom toLower_content: forall s1: string_type, i: int.
    0 <= i < length s1 -> nth i (toLower s1) = toLower_char (nth i s1)

  function toUpper (s1: string_type) : string_type

  axiom toUpper_content: forall s1: string_type, i: int.
    0 <= i < length s1 -> nth i (toUpper s1) = toUpper_char (nth i s1)
```

Listing 7.5: String theory definition in WHY

– *subString* : *string_type* → *int* → *int* → *string_type* returning the subset of elements contained between two indexes (both included) of the first argument (detailed in Listing 7.6). This WHY theory includes both the function definition and its formalisation through lemmas.

`Char` and `String` theories are highly inspired from their namesake standard library modules available on the WHY3 platform website.

7.3.2 BLOCKLIBRARY STRUCTURALFEATURE THEORY

Block interfaces are compositions of `StructuralFeature` elements with constrained data types and values. In order to model BLOCKLIBRARY conforming models with WHY3, we need to be able to express those elements at a WHY3 theory level. We choose to model the `StructuralFeature` elements with specific WHY3 record types:

```
  function subString (s: string_type) (lo up:int) : string_type =
    if lo >= length s \/ lo < 0 \/ up < 0 \/
       up >= length s \/ up < lo then Nil
    else match s with
         | Nil -> Nil
         | Cons hd tl ->
             if lo = 0 then
               if up = 0 then
                 Cons hd Nil
               else Cons hd (subString tl 0 (up-1))
             else subString tl (lo-1) (up-1)
         end

  lemma subString_nil: forall x,y: int.
    subString Nil x y = Nil

  lemma subString_length_nil: forall x,y: int.
    length (subString Nil x y) = 0

  lemma subString_0_0: forall s: string_type, c: tChar.
    subString (Cons c s) 0 0 = Cons c Nil

  lemma subString_length_0_0: forall s: string_type, c: tChar.
    length (subString (Cons c s) 0 0) = 1

  lemma subString_0_x: forall s: string_type, c: tChar, x: int.
    (0 <= x < length s) ->
    subString (Cons c s) 0 x = Cons c (subString s 0 (x-1))

  lemma subString_length_0_x: forall s: string_type, c: tChar, x: int.
    (0 <= x < length s) ->
    length (subString (Cons c s) 0 x) = 1 + length (subString s 0 (x-1))

  lemma subString_x_y: forall s: string_type, c: tChar, x,y: int.
    (0 < x <= y < length s) ->
    subString (Cons c s) x y = subString s (x-1) (y-1)

  lemma subString_length_x_y: forall s: string_type, c: tChar, x,y: int.
    (0 < x <= y < length s) ->
    length (subString (Cons c s) x y) = length (subString s (x-1) (y-1))

  lemma subString_OutOfBound:  forall l: string_type, lo up: int.
    (lo >= length l -> (subString l lo up) = Nil) /\
    (lo < 0 -> (subString l lo up) = Nil) /\
    (up < 0 -> (subString l lo up) = Nil) /\
    (up >= length l -> (subString l lo up) = Nil) /\
    (up < lo -> (subString l lo up) = Nil)

  lemma length_one: forall l: list 'a, e: 'a.
    length (Cons e l) = 1 + length l
end
```

Listing 7.6: String theory definition in WHY (continued)

```
theory InPortGroup
  use import String.String
  use import int.Int

  type tInPortGroup 'a

  function name_inpg (tInPortGroup 'a) : string_type
  function min_size_inpg (tInPortGroup 'a) : int
  function max_size_inpg (tInPortGroup 'a) : int
  function value_inpg (tInPortGroup 'a) : 'a

  axiom tInPortGroup_min_max_one: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = one -> pg.min_size_inpg = one

  axiom tInPortGroup_min_max_value: forall pg: tInPortGroup 'a.
    pg.min_size_inpg >= zero /\ pg.max_size_inpg >= zero

  axiom tInPortGroup_min_max_size: forall pg: tInPortGroup 'a.
    pg.min_size_inpg <= pg.max_size_inpg \/
    pg.max_size_inpg = zero

  function size_inpg (pg: tInPortGroup 'a) : int =
    if pg.max_size_inpg = zero then zero else
    if pg.max_size_inpg = one then one else
    pg.max_size_inpg - pg.min_size_inpg

  lemma size_inpg_max_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = zero -> size_inpg pg = zero

  lemma size_inpg_min_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg <> zero /\ pg.min_size_inpg = zero ->
    size_inpg pg = pg.max_size_inpg

  lemma size_inpg_max_one: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = one -> size_inpg pg = one

  lemma size_inpg_min_non_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg <> zero /\ pg.min_size_inpg <> zero ->
    size_inpg pg = pg.max_size_inpg - pg.min_size_inpg
end
```

Listing 7.7: Input PortGroup definition in Why3

Input PortGroup

Input `PortGroup` is modeled as the `tInPortGroup` type. Listing 7.7 contains its formalisation. We formalise some of the Input `PortGroup` metaclass attributes as uninterpreted function: `name_inpg`, `min_size_inpg`, `max_size_inpg` and `value_inpg` respectively formalizing the `name`, `min_size`, `max_size` and `value` attributes. By relying on uninterpreted function, we are then able to provide additional *axioms* expressing constraints on the size values for a port group. These *axioms* express the same constraints as the one given at the metamodel level. Finally, we provide the lemmas for the `size_inpg` function retrieving the actual size of an input port.

Output PortGroup

Output `PortGroup` is modeled as the `tOutPortGroup` type. As previously, the Output `PortGroup` attributes are formalised as uninterpreted functions. Its definition is very close to the one of `tInPortGroup`. Only one *predicate* is expressed on this record type constraining the minimum and maximum size of the group to one. This constraint is extracted from the BlockLibrary metamodel OCL ones. Listing 7.8 contains the `tOutPortGroup` definition.

```
theory OutPortGroup
  use import String.String
  use import Scalar.Boolean
  use import int.Int

  type tOutPortGroup 'a

  function name_outpg (tOutPortGroup 'a) : string_type
  function min_size_outpg (tOutPortGroup 'a) : int
  function max_size_outpg (tOutPortGroup 'a) : int
  function value_outpg (tOutPortGroup 'a) : 'a

  axiom tOutPortGroup_min_max_one: forall pg: tOutPortGroup 'a.
     pg.max_size_outpg = one /\ pg.min_size_outpg = one
end
```

Listing 7.8: Output `PortGroup` definition in Why3

```
theory Parameter
  use import String.String

  type tParameter 'a = {
    name_pt : string_type ;
    isMandatory_pt : boolean_type ;
    value_pt : 'a
  }
end
```

Listing 7.9: `Parameter` definition in Why3

PARAMETER

`Parameter` is modeled as the `tParameter` type-parametrised record type. Listing 7.9 contains its formalisation. We defined the `tParameter` record type with fields corresponding to the `Parameter` metaclass attributes.

MEMORYVARIABLE

`MemoryVariable` are modeled as the `tMemoryVariable` type-parametrised record type. Listing 7.10 contains its formalisation. We defined the `tMemoryVariable` record type with fields corresponding to the
`MemoryVariable` metaclass attributes.

For each `StructuralFeature` type definition, we have declared a `value_XX` field. This field is typed according to the type parameter of its containing record type. In a BLOCKLIBRARY specification, we refer to this `value_XX` field on a `StructuralFeature` instance `sf` by calling `sf.value`.

According to our experiments, it looks like there elements could be generated automatically from the metamodel definition along with its OCL constraints. Early experiments were conducted by M. Carton and are a perspective to ease some of our work.

```
theory MemoryVariable
  use import String.String

  type tMemoryVariable 'a = {
    name_mv : string_type ;
    value_mv : 'a
  }
end
```

Listing 7.10: `MemoryVariable` definition in Why3

	Multiple occurrences	Ordered content
Set	No	No
Bag	Yes	No
OrderedSet	No	Yes
Sequence	Yes	Yes

Table 7.11: OCL collections characteristics

7.4 OCL EXPRESSIONS TRANSFORMATION

OCL is a constraint expression language using first order logic and model navigation constructs. The WHY language also allows for the expression of first order logic and as such is sufficiently expressive to model OCL constraints.

OCL is built on a simple set of types called the primitive data types that is a subset of the type system we used in our BLOCKLIBRARY language. Using OCL, it is possible to gather values through the use of collections. There are four kind of collections defined in OCL: these collection types differ regarding their ability to handle (or not) multiple occurrences of the same value and if these values are ordered or not – when a value is added to the collection, the insertion is done at the right place in the collection allowing to keep the order of the collection correct. Table 7.11 provides the collection type names according to these two conditions. Regarding values held in a block, collections can be used for example to gather multiple values for a memory value. The vectors can contain several time the same value and the contained value order matters. It is thus mandatory to use an OCL collection type that provides the same behavior. In the WHY3 standard library, collections are modeled as lists allowing multiple occurrences of the same unordered value. This makes them a direct translation for *Bag* collections. In our implementation of OCL, we do not take into account the type of the collections and make the assumption that every collection is a *Bag* collection, this allows to simplify the management of collections in our implementation of OCL: elements are not removed from the collections as multiple occurrences are allowed and elements are not moved inside the collection as it is not ordered. We defined a WHY theory called *OCLCollectionOperation* containing the definition for some basic list accessors and operations. The other three kind of OCL collections could have been implemented in a similar way.

We provide the definition for the basic list getter operator in Listing 7.12. This definition has been extracted from the WHY3 array module from the standard WHY3 library.

```
function ([]) (a: list 'a) (i: int) : 'a = nth i a
```

Listing 7.12: List getter definition as an operator

OCL defines multiple operations that are to be applied either on primitive values – referred to as standard language operations – or on collection values – referred to as collection and iteration operations. These operations are not supposed to be used on *Bag* collections and explicit conversions between collections types must be done. We do not enforce this in our implementation of OCL as we provide only one collection type.

We only implement support for OCL expressions. Indeed, our OCL expressions are defined on various elements of the BLOCKLIBRARY language that provides the required context informations for the expressions. We also decided not to support messaging related constructs and tuples constructs. Messaging is related to sequence and state machine diagrams which do not make sense in our case. Tuples could have been implemented with records.

The translation of OCL expressions to the WHY language can be done using two strategies: either operations are translated to basic first order logic expressions and thus can directly be used in WHY or the definition for the operations are axiomatized using WHY functions declarations and the OCL constraints are translated as expressions using these functions.

In our implementation, we choose to use a combination of the two. List getter is used for simple col-

lection accesses, standard type operations have been defined as functions in WHY, simple collections operations are directly mapped to their logical expression equivalent. This approach has the advantages of easing the translation work as the semantics of the OCL standard types operations is already defined and thus avoid to generate too complex expressions. This has the pleasant side effect of easing the transformation verification activities as the translation itself is simpler.

In the following, we provide the mapping between the source OCL constructs and operations and the target WHY predicates, functions and expressions.

7.4.1 OCL STANDARD LIBRARY OPERATIONS

In OCL expressions operations can be applied on primitive OCL types. These operations are classical handling of primitive data types and are gathered in the OCL standard library. They have already been formalised in the WHY3 library. We thus rely on these formalisation for our translation. Table 7.13 sum-ups these mappings.

We did not implement the transformation for the *div* and *mod* operations on OCL *Double* elements. The *div* operation would have been of particular interest as its behavior on the OCL and on the WHY language are not the same. Indeed where the OCL implementation of *div* applied to any number and 0 (division by zero) returns a *null* value, the WHY version is simply not defined. The management of specific values like *null* and *undefined* is not done in this thesis and is highlighted as a current limitation of our work in Section 7.4.4. OCL implementation comprises the *round* and *floor* operations that we did not implement either. String operations on Table 7.14 are the one detailed in our String theory provided in Listing 7.5.

OCL expression	OCL context	Target WHY code	WHY3 theory name
`i.abs()`	i: Integer	`abs i`	int.Abs
`d.abs()`	d: Double	`abs d`	real.Abs
`i.div(j)`	i,j: Integer	`div i j`	int.EuclideanDivision
`i.mod(j)`	i,j: Integer	`mod i j`	int.EuclideanDivision
`i.min(j)`	i,j: Integer	`min i j`	int.MinMax
`d.min(j)`	d,j: Double	`min d j`	real.MinMax
`i.max(j)`	i,j: Integer	`max i j`	int.MinMax
`d.max(j)`	d,j: Double	`max d j`	real.MinMax

Table 7.13: OCL primitive numeric types operations mapping to WHY theories functions

OCL expression	OCL context	Target WHY code	WHY3 theory name
`s.size()`	s: String	`length s`	list.List
`s1.concat(s2)`	s1,s2: String	`concat s1 s2`	blocklibrary_string.String
`s.subString(i,j)`	i,j: Integer; s: String	`subString s i j`	blocklibrary_string.String

Table 7.14: OCL String operations mapping to WHY theories functions

Boolean OCL expressions are implemented using boolean expressions in WHY. *xor* operator has been implemented with *and, or* and *not* operators. Table 7.15 sum-ups these operators translations. Numeric operations have been implemented using the standard WHY arithmetic theories and their operators. We detail OCL numeric operations in Table 7.16. Finally relational operators are also based on standard WHY constructs. Their mapping can be found in Table 7.17.

OCL expression	OCL context	Target WHY code
`a and b`	a,b: Boolean Expression	`a /\ b`
`a or b`	a,b: Boolean Expression	`a \/ b`
`not a`	a: Boolean expression	`not a`
`a xor b`	a,b: Boolean Expression	`(a \/ b) /\ not (a /\ b)`
`a = b`	a,b: Boolean expression	`a = b`
`a <> b`	a,b: Boolean expression	`a <> b`
`a implies b`	a,b: Boolean Expression	`a -> b`

Table 7.15: OCL logical operators mapping to WHY operators

OCL expression	OCL context	Target WHY code
`a + b`	a,b: Integer expression	`a + b`
	a,b: Double expression	`a +. b`
`a - b`	a,b: Integer expression	`a - b`
	a,b: Double expression	`a -. b`
`a * b`	a,b: Integer expression	`a * b`
	a,b: Double expression	`a *. b`
`a / b`	a,b: Integer expression	`a / b`
	a,b: Double expression	`a /. b`
`-a`	a: Integer expression	`-a`
	a: Double expression	`-.a`

Table 7.16: OCL arithmetic operators mapping to WHY operators

7.4.2 COLLECTION OPERATIONS

As previously mentioned, there are four types of OCL collections. We only consider the use of the *bag* collections in our BLOCKLIBRARY expressions. In Table 7.18, we detail the mapping between standard OCL collections operations and WHY functions. In our handling of OCL collection operations, we do not handle OCL generic nature nor subtyping of elements. If the same operation is to be expressed on different types it is then developed separately for each different type. Whereas this may seem to be a limited way of handling this problem, in practice, our support restriction of OCL makes this easier as only a few operations needs to be encoded several times.

The restrictions on the kind of collection we rely on for our constraints have an impact on the translation we provide for some collection operations. The *append* and *including* operations have the same implementations and so have *subOrderedSet* and *subSequence*. According to the OCL specification, some collections operations are not allowed on *bag* collections: *append, at, first, indexOf, indexAt, last, prepend, subOrderedSet, subSequence*. We decided to allow their use in our implementation of OCL. This is a major drift from the OCL standard but it has the advantage of greatly simplifying the translation mechanism without restraining the expressiveness of the language. The formalisation of the usual OCL collections is of additional complexity as studied by Mentré in 2012 [106] but was not of primary interest for our current work so we decided not to address the related issues.

Each of these functions are defined through a set of axioms specifying their context of use: on which element type they are defined and if required restrictions are needed for their definitions; and the result of their computation according to the provided input values. As sake of example, we provide the definition for the *union* collection operation in Listing 7.19.

```
function union (l1 l2: list 'a) : list 'a

lemma union_Presence: forall l1 l2: list 'a, e: 'a.
  mem e l1 \/ mem e l2 <-> mem e (union l1 l2)
```

OCL expression	OCL context	Target WHY code
a = b	a,b: Numeric expression	a = b
a <> b	a,b: Numeric expression	a <> b
a < b	a,b: Integer expressions	a < b
	a,b: Double expressions	a <. b
a > b	a,b: Integer expressions	a > b
	a,b: Double expressions	a >. b
a <= b	a,b: Integer expressions	a <= b
	a,b: Double expressions	a <=. b
a >= b	a,b: Integer expressions	a >= b
	a,b: Double expressions	a >=. b

Table 7.17: OCL relational operators mapping to WHY operators

OCL expression	Target WHY code	OCL expression	Target WHY code
a->count(o)	count o a	a->append(o)	append a (Cons o Nil)
a->excludes(o)	not mem o a	a->prepend(o)	Cons o a
a->excludesAll(o)	list_not_mem o a	a->including(o)	append a (Cons o Nil)
a->includes(o)	mem o a	a->excluding(o)	excluding a o
a->includesAll(o)	list_mem o a	a->indexOf(o)	indexOf a o
a->isEmpty()	length a = 0	a->insertAt(i,o)	insertAt a o i
a->notEmpty()	length a <> 0	a->intersection(c)	intersection a c
a->size()	length a	a->union(c)	union a c
a->first()	a[0]	a->subOrderedSet(l,u)	subList a l u
a->last()	a[length a]	a->subSequence(l,u)	subList a l u
a->at(i)	a[i]	a->sum(), a: Bag(Integer)	sumInt a
		a->sum(), a: Bag(Double)	sumReal a

Table 7.18: OCL collection operations mapping to WHY operators

```
lemma union_Empty: forall l1 l2: list 'a.
  (union l1 l2) = Nil <-> l1 = Nil /\ l2 = Nil
```

Listing 7.19: *union* collection operation formalisation in WHY

7.4.3 LOGICAL PROPERTY ASSESSMENT ITERATION OPERATIONS

Iteration operations are the main operations used on OCL collections. They allow to assess a logical property verification on: every element of a list (forAll), at least one element of a list (exists), exactly one element of a list (one). They can express the uniqueness of the result of the application of a function on every element of a list (isUnique). All these operations returns a boolean value.

In Table 7.20 we provide the translation for the *forAll*, *exists*, *one* and *isUnique* OCL operations on collections. Unlike the previous translations, we do not rely on predefined functions but we rather map these operations to simple first order logic expressions. Regarding the translation of the condition expression: *exp*, the defined OCL iterators: *it*, *it1* and *it2* are mapped to a call to the position of the element in the collection via the list getter operator. In practice, references to *it* or *it1* are replaced by a[i] and references to *it2* are replaced by a[j] in *exp*. This is expressed using the function application: $[a/b]c$ that does a substitution of b by a in c.

OCL expression	Target WHY code
a→**forAll**(it: DT | exp)	∀ i: int. 0 ≤ i < length a → [a[i]/it]exp
a→**forAll**(it1, it2: DT | exp)	∀ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a → [a[i]/it1,a[j]/it2]exp
a→**exists**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ [a[i]/it]exp
a→**exists**(it1, it2: DT | exp)	∃ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ [a[i]/it1,a[j]/it2]exp
a→**one**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ $[a[i]/it]exp$ ∧ (∀ j: int. 0 ≤ j < length a ∧ j <> i → $[a[i]/it]exp$ <> $[a[j]/it]exp$)
a→**isUnique**(it: DT | exp)	∀ i,j : int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ i <> j → $[a[i]/it]exp$ <> $[a[j]/it]exp$

Table 7.20: OCL logical property verification operations mapping to WHY expressions

7.4.4 Value extraction iteration operations

Iteration operations also allow the extraction of values from a collection: according to their verification (select) or non-verification (reject) of a property; or by applying a treatment on every element of a list (collect). Value extraction operations are more complex to model as they do not only provide a single boolean value as output, they actually compute a list of elements.

Iteration operations semantics

Iteration operations apply on collections and compute filtering of the collection values and/or mapping of functions on the collection values. We decide thus to represent a collection as the $\langle c, p, f \rangle$ tuple. Its semantics is provided in (7.1).

$$[\![\langle c, p, f \rangle]\!] = \{f(v) | v \in c, p(v)\} \tag{7.1}$$

According to the previous notation, we define the initial value of a collection as in (7.2) where $\top$ is the predicate returning *true* and `id` the identity relation.

$$c = \{v_1, ..., v_n\} = \langle c, \top, \texttt{id} \rangle \tag{7.2}$$

The definition of iteration operations is hence specified as in (7.3). From these definitions, we extract implementations for iteration operations using the WHY language. We provide in the following sections two possible implementations.

$$\begin{array}{rrrr} \langle c,p,f\rangle & \rightarrow & select(e|\phi) = & \langle c, p \wedge [f/e]\phi, f\rangle \\ \langle c,p,f\rangle & \rightarrow & reject(e|\phi) = & \langle c, p \wedge \neg[f/e]\phi, f\rangle \\ \langle c,p,f\rangle & \rightarrow & any(e|\phi) = & \langle c,p,f\rangle \rightarrow select(e|\phi) \rightarrow first() \\ \langle c,p,f\rangle & \rightarrow & collect(g) = & \langle c, p, f \circ g\rangle \end{array} \tag{7.3}$$

Iteration operations as first order logic constructs

Providing an implementation for iteration operations can be done using first order logic constructs. It is then required to generate a different function for each iteration operation call and then in the translated OCL operation code, to write a call to the generated function. Generated function for the *select*, *reject*, *any* and *collect* value extraction operations are provided in Table 7.21. Each generated function name is post-fixed with an "_iterator" elements. This must be replaced by a unique value allowing to ensure that the name of each generated collection function is unique. The first argument to the generated iteration function is the collection on which the operation applies, the others corresponds to the variables (others than the iterator itself) used in the iteration operation body.

OCL expression	Target WHY code

a->select(it:DT| exp)	`function select_UID (a: list 'b) (var1: type) ... : list 'b =`
	`  match a with`
	`  | Nil -> Nil`
	`  | Cons hd tl ->`
	`  if ([hd/it]exp) then Cons hd select_iterator tl`
	`  else select_iterator tl`
	`end`

a->reject(it:DT| exp)	`function reject_UID (a: list 'b) (var1: type) ... : list 'b =`
	`  match a with`
	`  | Nil -> Nil`
	`  | Cons hd tl ->`
	`  if not ([hd/it]exp) then Cons hd select_iterator tl`
	`  else select_iterator tl`
	`end`

a->any(it:DT| exp)	`function any_UID (a: list 'b) (var1: type) ... : 'b =`
	`  match a with`
	`  | Nil -> Nil`
	`  | Cons hd tl ->`
	`  if ([hd/it]exp) then hd`
	`  else any_iterator tl`
	`end`

a->collect(it:DT| exp)	`function collect_UID (a: list 'b) (var1: type) ... : 'b =`
	`  match a with`
	`  | Nil -> Nil`
	`  | Cons hd tl -> union ([hd/it]exp) (collect_UID tl)`
	`end`

Table 7.21: OCL iteration operations mapping to WHY expressions

Whereas using first order logic to provide an implementation for iteration operations is quite straightforward, the implementation and verification of the generation might be quite complex.

ITERATION OPERATIONS AS HIGHER ORDER LOGIC CONSTRUCT

As an alternative approach, we propose to use higher order logic to represent the operations in WHY. Table 7.22 contains the iteration operations mappings to their respective function defined in WHY. An example of such formalisation is provided in Listing 7.23 for the *select* iteration operation. The *select* function in WHY takes two arguments, the first one is the collection on which the operation is applied and the second one is the predicate that must be used in order to select the elements of the collection. In addition to the function definition, we provide a set of lemmas verified by relying on the WHY3 platform generating proof obligations discharged using SMT solvers or proof assistants. In the case of the *select* function, part of the lemmas are verified using SMT solvers and others have been verified using the COQ proof assistant. We provide in Figure 7.24 a report generated with the WHY3 toolset for these verifications. We detail these verifications in Appendix D.

OCL expression	Target WHY code
a→**collect**(it: DT | exp)	**collect** a fexp
a→**reject**(it: DT | exp)	**reject** a fexp
a→**select**(it: DT | exp)	**select** a fexp
a→**any**(it: DT | exp)	**anyAs** a fexp

Table 7.22: OCL iteration operations mapping to WHY high order logic functions

In order to use iteration expressions, one must provide a body containing the condition or the operation to apply on each element of the collection. We refer to this body in Table 7.22 as *fexp*. The condition body

```
function select (l: list oclType) (p: HO.pred oclType) : list oclType =
    match l with
    | Nil -> Nil
    | Cons hd tl -> if p hd then Cons hd (select tl p)
                            else select tl p
    end

  lemma select_nil: forall p: HO.pred oclType.
    select Nil p = Nil

  lemma select_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
    p e -> select (Cons e Nil) p = Cons e Nil

  lemma select_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
    not (p e) -> select (Cons e Nil) p = Nil

  lemma select_cons_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
    p e -> select (Cons e l) p = Cons e (select l p)

  lemma select_cons_not_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
    not (p e) -> select (Cons e l) p = select l p

  lemma select_mem_reduc: forall l: list oclType, b: oclType, p: HO.pred oclType.
    mem b (select l p) -> mem b l

  lemma select_mem: forall l: list oclType, b: oclType, p: HO.pred oclType.
    (mem b l /\ p b) -> mem b (select l p)

  lemma select_not_mem: forall l: list oclType, b: oclType, p: HO.pred oclType.
    (mem b l /\ not (p b)) -> not (mem b (select l p))
```

Listing 7.23: Select iteration operation formalisation in WHY using higher order logic

Proof obligations	Alt-Ergo-Pro (1.0.0)	Coq (8.4pl3)
lemma select_nil	0.03	
lemma select_cons_nil_verified	0.04	
lemma select_cons_nil_not_verified	0.05	
lemma select_cons_verified	0.03	
lemma select_cons_not_verified	0.04	
lemma select_mem_reduc		2.40
lemma select_mem		2.31
lemma select_not_mem		2.01

Figure 7.24: Select lemmas verification with WHY3 through SMT solvers and proof assistants

of the *reject*, *select* and *anyAs* operations is translated as an inlined predicate whereas the function body of the *collect* operation is translated as an in-lined function. Inlined predicates and functions are provided as the second argument of their respective WhyML function call. Contrary to the first order operations, there is no need here to keep track of the variables used in the iteration operation as the inlined nature of the function call makes their definition directly available.

In the *MinMax* block specification provided in Listing 7.2, a postcondition is provided for the *MinOutputScalarMultipleInputsScalars* compute semantics phase definition:

```
postcondition ocl {
  In1->forAll(i | i.value >= Out.value)
}
```

This post-condition is equivalent to:

```
postcondition ocl {
  In1->select(i| i.value < Out.value)->isEmpty()
}
```

Which could be translated in WhyML using higher order logic as a call to the *select* function:

```
let in1_0 = select in1 ((\ bind_i: tInPortGroup (tRealDouble).
                          bind_i.value_inpg >. out.value_outpg)) in
  length in1_0 = 0
```

The declarations for the *collect*, *reject* and *any* operations are provided in Appendix D.4.

Additional limitations on the support of the OCL language

In the previous sections, we give the translation rules for a subset of the OCL language. OCL provides a well known syntax for software engineers. In addition, this subset of the language eases and secures the constraints writing process relying on first order logic by including only well known and standard functions and operations on classical data types.

A major limitation in our implementation of the OCL language is the absence of the specific values for data: *undefined* and *null*. This enforces the block designer to explicitly set the values for the block structural features. In a block specification, some feature may be optional for some signatures. In these cases, the specifier must provide a default value for the feature. The *reset_algo* parameter is an example of such a feature: its special value *NONE* is the explicit value of the feature when the parameter is not used.

Undefined and *null* values can be implemented in Why by relying on *option* types. Adding the support for these specific values will have a strong impact on the Why implementation of the OCL constructs as these specific values will need to be taken into account in the implementation of the functions and in their correctness proofs.

7.5 The BAL expressions transformations

The BAL is a simple imperative language. The WhyML language allows for the expression of programs in an imperative form. The expressiveness of BAL is a subset of WhyML one's. As such, programs written using BAL can be converted to WhyML programs. We detail in Table 7.25, the transformation rules we have defined in order to transform a BAL function body into a WhyML program.

The provided translation rules have been arbitrarily defined. To our understanding, it is the most natural way of translating BAL expressions as WhyML functions expressions. This translation is not the only possible mapping and is highly dependent of the choices made in previous translations. As an example, the *value* attribute access translation is provided as a field access in WhyML but this is related to the fact that *e* (which is a `StructuralFeature` element) is formalised as a record type in Why3. Regarding *for loop* expressions we only support the provided form of *for loop* with an iterator increasing (or decreasing) of one at each iteration, this has the advantage of allowing a direct mapping to the WhyML for loop allowed constructs and simplifies the verification of loops.

Logical, relation, arithmetic and literal expressions are translated according to the same transformation rule as in the OCL translation.

BAL expression	Target WHY code
`sf.value = exp`	`sf.value_XX <- exp`
`var e = exp`	`let e = ref exp in`
`e[i][j] = exp`	`e := setAt !e (setAt !e[i] exp j) i`
`sf.value[i][j] = exp`	`sf.value <- setAt sf.value (setAt sf.value[i] exp j) i`
`if (cExp) then tExp else eExp`	`if cExp then tExp else eExp`
`for (var i = iExp; i < topExp; i = i + 1) { exp }`	`for i = iExp to topExp - 1 do exp done;`
`for (var i = iExp; i > bottomExp; i = i - 1) { exp }`	`for i = iExp downto topExp + 1 do exp done;`
`while (cExp) { exp }`	`while cExp do exp done;`

Table 7.25: BAL expressions to WHYML translation rules

7.6 BLOCKLIBRARY VERIFICATION TRANSFORMATIONS

The translation from BLOCKLIBRARY conforming models to WHY aims at ensuring the verification of the variability completeness and disjointness properties as modeled in REQ-7.[b|c] and the semantics correctness as modeled in REQ-8. In order to do so, we decided to translate each `BlockType` specification to a set of WHY theories for the first verification, to WHYML modules for the second one and to generate proof goals to ensure these properties. Figure 7.26 provides an overview of these transformations that will be detailed in the following sections.

7.6.1 VARIABILITY VERIFICATION TRANSFORMATION

Variability verification targets the assessment of the completeness and consistency properties. In order to ease this assessment, we check the stronger disjointness property instead of consistency properties. In order to ensure the verifiability of the properties, we need to express the domain on which the verification needs to be applied. In our case, a different domain is described for each `BlockType` specification.

This first transformation is split in three steps: block domain transformation, `Signature` domain transformation and variability verification goals generation.

BLOCK DOMAIN TRANSFORMATION STEP

The block domain is composed of `StructuralFeature` element definitions composed of their name and data type. Each `StructuralFeature` definition is translated to a new type expressed as an alias of the standard data types we defined in Section 7.3.1. An example of translation from the `StructuralFeature` defined in the *MinMax* BLOCKLIBRARY specification is provided in Figure 7.27. The last two types generated in the *MinMax_FeaturesDT* theory are generated from the *In1* and *Out* `StructuralFeature`.

These alias types should be constrained according to the `INVARIANT` expressed on each `StructuralFeature`. We chose to implement this by creating a theory for each `BlockVariant` element containing at least one `StructuralFeature` definition that is constrained by an `INVARIANT`. This theory will contain a *predicate* definition for each `INVARIANT` defined for a `StructuralFeature` in this `BlockVariant`. We will refer to these theories as the `StructuralFeature INVARIANT` theories. Each generated predicate will have as parameters all the types corresponding to the `StructuralFeature` ac-

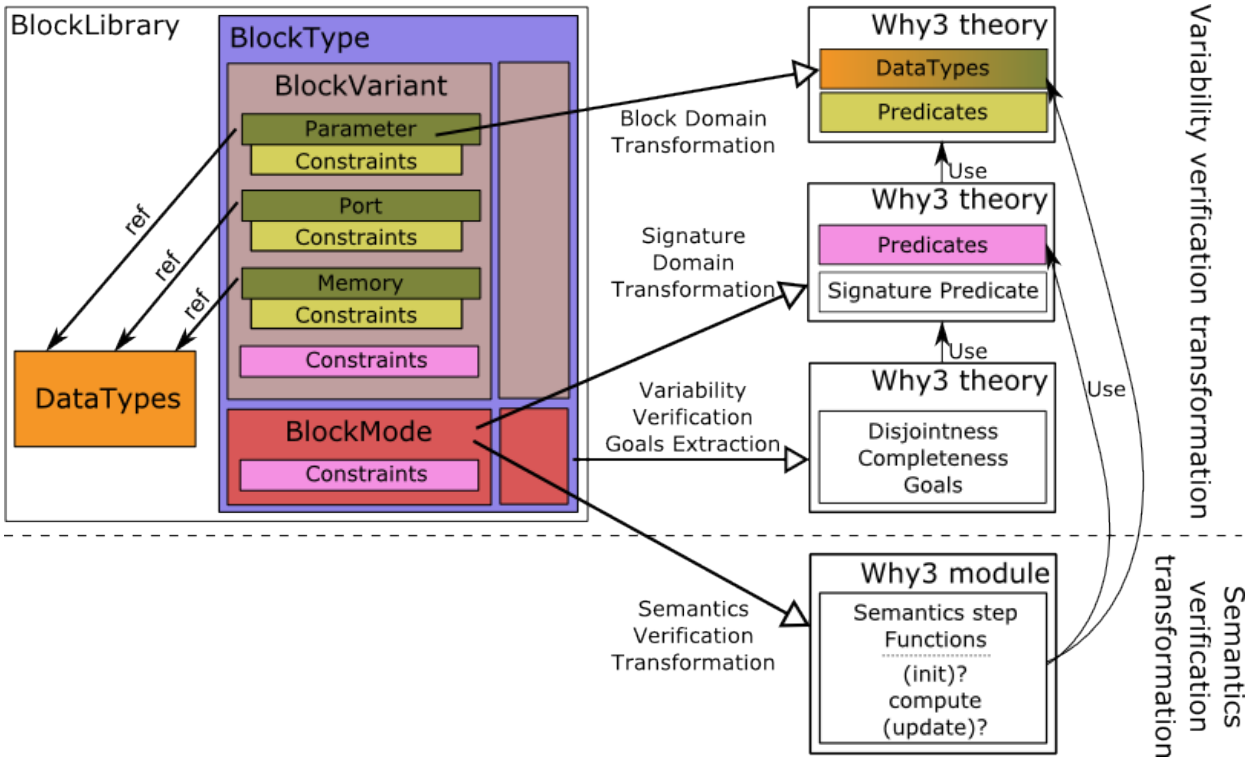


Figure 7.26: Overview of the BLOCKLIBRARY to WHY3/WHYML transformation

```
variant MinMaxParameters {
  parameter FunctionParam : MinMaxFunction
  parameter NbInputs : TInt16 { invariant ocl { NbInputs.value >= 1 } }
}
```

is translated to

```
theory MinMax_FeaturesDT
  use export BlockLibrary_Main.BlockLibraryDataTypes

  type minMaxFunction = Min | Max

  type tFunctionParam_MinMaxParameters_MinMaxFunction =
         tParameterType (minMaxFunction)
  type tNbInputs_MinMaxParameters_TInt16 = tParameterType (tRealSignedInt16)
  type tIn1_MinMaxInScalars_TDouble = list (tInPortGroup (tRealDouble))
  type tOut_MinMaxOutScalar_TDouble = tOutPortGroup (tRealDouble)
end
```

Figure 7.27: Block domain extraction of `StructuralFeature` data types

```
variant MinMaxParameters {
  parameter FunctionParam : MinMaxFunction
  parameter NbInputs : TInt16 { invariant ocl { NbInputs.value >= 1 } }
}
variant MinMaxInScalars extends MinMaxParameters {
  in data In1 : TDouble [1 .. 0] { invariant ocl { In1->size() = NbInputs.value } }
}
```

is translated to

```
theory MinMaxParameters_PreConditions
  use export MinMax_FeaturesDT

  predicate minMaxParameters_inv0
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    nbInputs.value_pt >= 1

  predicate minMaxParameters_NbInputs_limit
      (nbInputs: tNbInputs_MinMaxParameters_TInt16) =
    limit_tRealSignedInt16 (nbInputs.value_pt)

  predicate minMaxParameters_PreConditions
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    minMaxParameters_inv0 nbInputs functionParam /\
    minMaxParameters_NbInputs_limit nbInputs functionParam
end

theory MinMaxInScalars_PreConditions
  use export MinMax_FeaturesDT

  predicate minMaxInScalars_inv_0
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (in1: tIn1_MinMaxInScalars_TDouble)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    length in1 = nbInputs.value_pt

  predicate minMaxInScalars_PreConditions
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (in1: tIn1_MinMaxInScalars_TDouble)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    minMaxInScalars_inv_0 nbInputs in1 functionParam
end
```

Figure 7.28: Block domain extraction of `StructuralFeature INVARIANT Annotation`

cessible through the current `BlockVariant Signature`. The body of the predicate will be the expression provided in the `INVARIANT Annotation`. An example for this translation from the *MinMaxParameters* and *MinMaxInScalars* `BlockVariant` is provided in Figure 7.28.

Signature domain transformation step

The `Signature` domain corresponds to the set of constraints that can be extracted from a `Signature` element. We choose to work at the `Signature` level as each `Signature` is holding a block specification that should be verified and as a `Signature` being behavior-focused, extracted informations should be easier to manage and thus to automatically verify by opposition to `Configuration` elements who may contain multiple behavior and thus will contain more complex function definitions.

From each `Signature`, we thus generate a theory that imports the definitions provided by the `StructuralFeature INVARIANT` theories of the `BlockVariant` contained in the `Signature`. In this theory, we translate every `MODE_INVARIANT` contained in the `Signature` (in all its `BlockVariant` and its unique `BlockMode`), to a *predicate* definition under the same principle as previously. Finally a `Signature` *predicate* is generated as defined in Listing 6.22 by calling the `Signature MODE_INVARIANT`

predicates. These theories will be referred to as the `Signature` theories. An example of such translation is provided for the `Signature` extracted from the *MinOutputScalarMultipleInputsScalars* `BlockMode` in Figure 7.29.

```
mode MinOutputScalarMultipleInputsScalars implements allof(
  MinMaxOutScalar,MinMaxInScalars)
{
  modeinvariant ocl { NbInputs.value > 1 }
  modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
}
```

is translated to

```
theory MinOutputScalarMultipleInputsScalars_sig0
  use export MinMax_FeaturesDT

  predicate minoutputscalarmultipleinputsscalars_modeInv_1
      (out: tOut_MinMaxOutScalar_TDouble)
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (in1: tIn1_MinMaxInScalars_TDouble)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    nbInputs.value_pt > 1

  predicate minoutputscalarmultipleinputsscalars_modeInv_2
      (out: tOut_MinMaxOutScalar_TDouble)
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (in1: tIn1_MinMaxInScalars_TDouble)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    functionParam.value_pt = Min

  predicate minOutputScalarMultipleInputsScalars_sig0
      (out: tOut_MinMaxOutScalar_TDouble)
      (nbInputs: tNbInputs_MinMaxParameters_TInt16)
      (in1: tIn1_MinMaxInScalars_TDouble)
      (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction) =
    minoutputscalarmultipleinputsscalars_modeInv_1 out nbInputs in1 functionParam /\
    minoutputscalarmultipleinputsscalars_modeInv_2 out nbInputs in1 functionParam
end
```

Figure 7.29: `Signature` domain extraction of `MODE_INVARIANT` Annotation

Variability verification goals extraction step

Completeness and disjointness of the specification have to be ensured for the entire `BlockType` specification, meaning that it must be expressed according to every `Signature` and thus according to every generated
`Signature` predicates. It should also take into account the domain on which the verification is done.

A new theory is generated, it will hold the required goal declarations for the expression of the verification properties. This theory must import all the `StructuralFeature INVARIANT` theories extracted from the
`BlockType` as long as the `Signature` theories extracted for each `Signature`.

Both of the verification goals are built on an implication:

- its premise is the domain defined in the `StructuralFeature INVARIANT` theories, it is thus a conjunction of all the predicates declared in these theories.
- its conclusion is the conjunction of the `Signature` predicate taken from the `Signature` theories.

We show the generated *goals* for the *MinMax* block specification in Figure 7.30.

```
theory MinMax_Verif
  use import MinOutputScalarMultipleInputsScalars_sig0
  use import MaxOutputScalarMultipleInputsScalars_sig0
  use import MinMaxParameters_PreConditions
  use import MinMaxInScalars_PreConditions

  goal MinMax_completeness :
    forall out: tOut_MinMaxOutScalar_TDouble,
    nbInputs: tNbInputs_MinMaxParameters_TInt16,
    in1: tIn1_MinMaxInScalars_TDouble,
    functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction
  minMaxParameters_PreConditions nbInputs functionParam /\
  minMaxInScalars_PreConditions nbInputs in1 functionParam
  ->
  minOutputScalarMultipleInputsScalars_sig0 out nbInputs in1 functionParam \/
  maxOutputScalarMultipleInputsScalars_sig0 out nbInputs in1 functionParam

  goal MinMax_disjointness :
    forall out: tOut_MinMaxOutScalar_TDouble,
    nbInputs: tNbInputs_MinMaxParameters_TInt16,
    in1: tIn1_MinMaxInScalars_TDouble,
    functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction
  minMaxParameters_PreConditions nbInputs functionParam /\
  minMaxInScalars_PreConditions nbInputs in1 functionParam
  ->
  not (minOutputScalarMultipleInputsScalars_sig0 out nbInputs in1 functionParam /\
      maxOutputScalarMultipleInputsScalars_sig0 out nbInputs in1 functionParam)
end
```

Listing 7.30: Completeness and disjointness verification goals

7.6.2 Semantics verification transformation

Semantics verification targets the assessment of the correctness of the semantics definition provided in the `BlockMode` elements in order to assess the correctness of the operational specification with respect to the axiomatic one provided in the pre/post conditions. We do not enforce `BlockMode` semantics definition to contain both axiomatic and operational semantics but it is mandatory to provide it during the specification phase in order to be able to perform this verification.

For each `BlockMode` specification and each `Signature` that can be extracted from it, we have shown in Section 6.5 that we can extract functions with contracts. Our semantics verification mechanism is inspired from this translation but is different as its target language is the WhyML language.

For each `Signature` extracted from each `BlockMode` extracted from each `BlockType`, we can build one Hoare triple for each semantics phase. Each Hoare triple can then be translated to a WhyML function and its contract. The function is composed of the operational semantics provided in each semantics phase definition and of its contract composed of:

- Pre-conditions extracted from: all the `INVARIANT` provided by all the `StructuralFeature`; all the `MODE_INVARIANT` contained in the `Signature` `BlockVariant` elements and the `BlockMode` element; and from the pre-conditions contained in the semantics phase definition.
- Post-conditions extracted from: the post-conditions contained in the semantics phase definition.

We provide in Figure 7.31 the translation for one of the *MinMax* block semantics definitions.

7.7 Variability verification through SMT solving

The previously generated Why theories are fed into the Why3 tool taking care of their translation to suitable formats for verification through SMT solvers or proof assistants. Here we will focus on the use of SMT solvers as we are targeting fully automated verification activities to ease the writing and verification

```
definition bal = compute_MinOutScalarMultipleInputsScalars {
  postcondition ocl {
    In1->forAll(i| i.value >= Out.value)
  }
  var res = In1[0].value;
  for (var i = 1; i < (size(In1)); i = i + 1){
    if (res > In1[i].value){
      res = In1[i].value;
    }
  }
  Out.value = res;
}
```

is translated to

```
let compute_MinOutScalarMultipleInputsScalars
    (out: tOut_MinMaxOutScalar_TDouble)
    (nbInputs: tNbInputs_MinMaxParameters_TInt16)
    (in1: tIn1_MinMaxInScalars_TDouble)
    (functionParam: tFunctionParam_MinMaxParameters_MinMaxFunction)
  requires { nbInputs.value_pt >= 1 }
  requires { length in1 = nbInputs.value_pt }
  requires { nbInputs.value_pt > 1 }
  requires { functionParam.value_pt = Min }
  ensures { forall i0: int.
            0 <= i0 < length in1 ->
            in1[i0].value_inpg >=. out.value_outpg
  } =
  let res_outer = in1[0].value_inpg in
  let res = ref res_outer in
  for i = 1 to length in1 - 1 do
    if ! res >. in1[i].value_inpg then
      res := in1[i].value_inpg
  done ;
  out.value_outpg <- ! res
```

Figure 7.31: `Signature` function with contract extraction of `BlockMode` specification

of specification by domain experts. However, in case of automatic verification failure, we could still fall back on proof assistant for the most complex blocks under the guidance of a proof expert.

7.7.1 Specification extract variability verification

In the *MinMax* block specification provided in Listing 7.2, we only extract a part of the specification. Applying the previously defined extraction mechanism provides the two goals detailed in Listing 7.30.

Completeness goal verification

If we simplify the completeness goal and express it using only expressions based on `StructuralFeature` elements values (by replacing the predicates calls by their definition and if we remove the duplicates expressions that can be removed), we obtain the goal of Listing 7.32.

```
goal MinMax_completeness :
    1 <= value_pt nbInputs /\ length in1 = value_pt nbInputs ->
     1 < value_pt nbInputs /\ value_pt functionParam = Min \/
      1 < value_pt nbInputs /\ value_pt functionParam = Max
```

Listing 7.32: Simplified completeness and disjointness goals

It is impossible to prove correct the `MinMax_completeness` goal to be true. Indeed, in the premises of the implication, the `value_pt nbInputs` value is supposed to be greater or equals to 1 and equals to the length of the `in1 PortGroup` (the number of these ports groups) but in the conclusion, we must prove the `value_pt nbInputs` to be greater than 1.

```
1  goal MinMax_disjointness :
       1 <= value_pt nbInputs /\ length in1 = value_pt nbInputs ->
        not ((1 < value_pt nbInputs /\ value_pt functionParam = Min) /\
              1 < value_pt nbInputs /\ value_pt functionParam = Max)
```

Listing 7.33: Simplified completeness and disjointness goals

This problem is due to the content of the *MinMax* block specification part we based our generation on. This specification only provides `BlockMode` for the cases where the number of inputs is greater than 1. If we want to verify this part of the specification, we have to restrain the specification of the `NbInputs` parameter to be strictly greater than 1. In this case the verification is done automatically in a few tenth of a second (0.09 seconds to be exact) by the Alt-Ergo SMT solver[2]. The example provided here whereas being simple is very representative of the typical block specification errors.

Disjointness goal verification

Regarding the proof for the `MinMax_disjointness`, we can also apply the same simplification as previously and obtain its expression based only on `StructuralFeature` values expressions as provided in Listing 7.33. This goal is verified by SMT solvers in 9 tenth of a second using the Alt-Ergo SMT solver.

Verification of the disjointness goal is done easily as the second components of every configuration predicates are exclusive (`value_pt functionParam`).

Of course, as our goal is not to verify partial configurations of a block, we applied the verification transformation to full block specifications. Results obtained are discussed in the following section.

7.7.2 Entire specification verification

We provide the entire specification in a textual form in Appendix A.2 for the *MinMax* block. Table 7.34 gives the verification times for some blocks we have written a BlockLibrary specification for. These verifications have been done using the Alt-Ergo SMT solver. According to what was previously stated, it is expected that according to the number of `Signature` element computed from a block specification, the verification time increases. This is mostly due to the exponential size of the disjointness goal. With the goals size increase comes an increase in the verification time and in the system resources needs. It is worth noting that despite the complexity of the *Delay* block example, the time required for the verification remains reasonable.

Block name	$\#_{\texttt{BlockMode}}$	$\#_{\texttt{Signature}}$	$Time_{Completeness}(s)$	$Time_{Disjointness}(s)$	$Memory_{Maxuse}(KiB)$
MinMax	10	10	0.09	0.09	123,960
Sum	8	8	0.11	0.13	130,844
Lookup	6	6	0.26	1.36	136,808
Delay	12	144	2.45	8.06	1,507,972

Table 7.34: Some blocks specification verification performances

7.7.3 Goals transformation as a mean to ease the verification

Using SMT solvers through the Why3 platform has some advantages. Among these we can cite the possibility to apply transformations on the theories to prove, before translating these ones to SMT solvers or proof assistants inputs.

[2]The verification has been processed on a Linux Mint laptop with a 2.4 Ghz dual core processor and 4 Gb of Ram

Transformations

Transformations allows manipulating the expressions in order to simplify the goals to prove. For example, transformations allows to produce one goal for each component of a conjunction; in-line predicates definitions (replace a predicate call with its definition) or terms definitions, remove *let* definitions by replacing the defined variable by its definitions in each of its use and many other ones. Goals simplifications done previously are the result of using this transformation mechanism. Further details about available transformations are provided in the Why3 manual [3].

Application to the completeness verification

We take as example the translation for the erroneous part of the *MinMax* block specification. We will focus on the `MinMax_completeness` goal. On this goal, we apply three *inline_goal* transformations. These lead to the simplified result that was detailed in Listing 7.32. We then apply the *introduce_premises* transformation leading to the introduction of both conjunction components of the premise as *axioms*. We finally apply the *split_goal_full* transformation distributing the logical conjunctions over the logical disjunction leading to four different goals to prove. We give the final goals details in Listing 7.35. The last goal is trivially proven as it aims at proving the definition of the `minMaxFunction` type. The other three are not provable from the informations provided in the H and H1 axioms.

```
constant out : tOutPortGroup real
constant nbInputs : tParameterType int
constant in1 : list (tInPortGroup real)
constant functionParam : tParameterType minMaxFunction

axiom H : 1 <= value_pt nbInputs
axiom H1 : length in1 = value_pt nbInputs

goal MinMax_completeness_1 : 1 < value_pt nbInputs \/ 1 < value_pt nbInputs
goal MinMax_completeness_2 : 1 < value_pt nbInputs \/ value_pt functionParam = Max
goal MinMax_completeness_3 : value_pt functionParam = Min \/ 1 < value_pt nbInputs
goal MinMax_completeness_4 : value_pt functionParam = Min \/
                             value_pt functionParam = Max
```

Listing 7.35: Inlined and split resulting goals for the completeness goal

Interpretation of the results

One main advantage of the use of this tool is the ability, with a minimal knowledge of first order logic and its manipulation, to actually apply the transformations, see their results and from this either a) manage to do the proof with SMT solvers and eventually with proof assistant; or b) guess informations on the reasons why the verification is failing and thus modify the block specification with this new knowledge. We like to call this a proof "debugging" process. This implies the ability to provide feedback information at the BlockLibrary conforming models level.

General transformation application methodology process

The completeness and disjointness goals have specific shapes, it is possible to define transformation application methodologies in order to simplify them and then obtain simpler sub-goals that one may try to prove with SMT solvers or if SMT solvers fails to conclude by relying on proof assistants.

Both completeness and disjointness goals are implications. Their premises are shaped as conjunctions and their conclusions are shaped according to the goal. We detail in Figure 7.36 (respectively in Figure 7.37) the transformation application methodologies that can be used on the completeness goal (respectively on the disjointness goal). In these figures, arrows denote the application of one or more transformations. The content of the rectangular notes is the general pattern of the goal on which the transformation is applied.

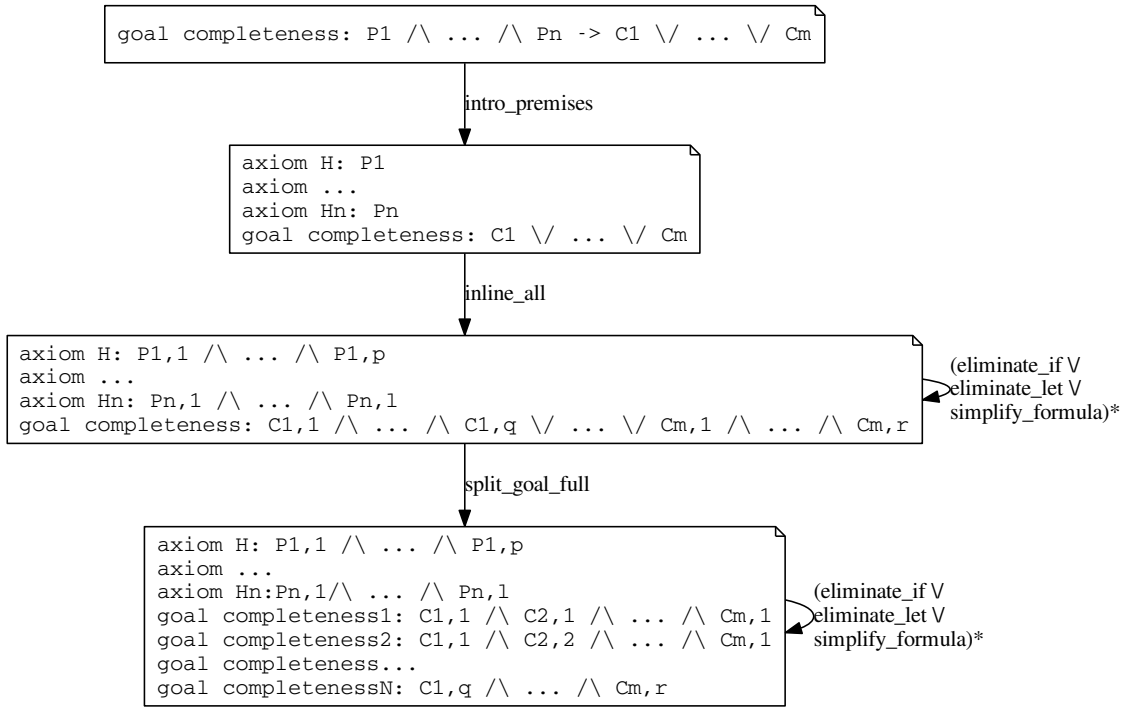


Figure 7.36: Completeness goal transformation application methodology

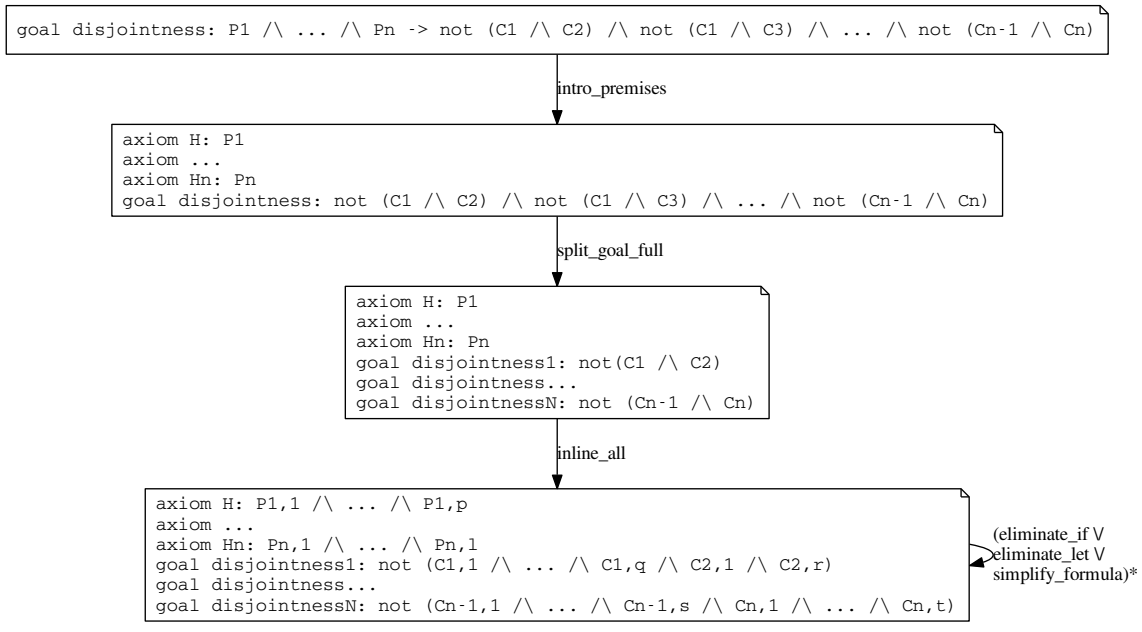


Figure 7.37: Disjointness goal transformation application methodology

The proposed methodologies allow to split complex goals as multiple simpler goals. This is an application of the "*Divide and conquer*" approach that suits well SMT solvers as they perform better on smaller goals. This approach is in fact close to the one applied in the first place by SMT solvers to achieve their verification. The specific shape of our generated goal makes plausible that no SMT solver inner strategy implements the provided methodologies.

For our work, we used Why3 in its 0.83 version. In more recent versions (0.85), other transformations are available. These new transformations may allow for a more efficient process.

Verification methodology process interest for BlockLibrary instance verification

Sub-goals obtained through the applications of our methodology can be fed to SMT solvers that may prove them. If this is not a success, then the sub-goals may still be too complex for an SMT solver to solve. In this case, there are three general solutions available to assess the correctness of the goal: a) we give more time and memory to the SMT solver to achieve the proof. This may succeed as computing power plays an important role in the success of SMT solving; b) we try to verify the negation of the goal. The verification of this goal will then show that the original goal is false; and c) we use a proof assistant that may allow to discharge some goals and even finish the proof.

If none of these three techniques provides a positive answer, the transformation methodology application or the proof assistant results may provide a discrepancy information on the specification that we can use as verification feedback:

- Discrepancy informations might be provided by some SMT solvers in the form of counter examples but it is not always possible depending on the toolset.
- The application of the transformations is generating simpler goals, these goals may be simple enough for the toolset user to detect discrepancies in the specification and then guess a correction to apply on the specification.
- Proof assistants feedbacks are based on the knowledge the human doing the proof have regarding the proof result such as a missing hypothesis that may be linked to missing informations (such as missing invariants for example) on the block specification or a false assumption caused by a faulty specification. The human operator can then provide the necessary feedbacks and correct the block specification accordingly.

Verification feedbacks should be expressed on the BlockLibrary instance specification allowing to correct it accordingly, one must ensure a tight integration of this feedback in the tooling and especially in the BlockLibrary editor environment.

We target to provide an automation of this feedback mechanism but timing constraints made this impossible. According to our experiments, it looks possible to provide it but the development work for this task is not small and thus needs a consequent timing investment.

7.8 Semantics verification through SMT solving

WhyML modules generated from a BlockLibrary instance contains the definition for a function extracted from a `Signature` instance. We have provided in Figure 7.31 the specification of a *MinMax* block `BlockMode` and its translation as a WhyML function with its contract. Here, we will tackle the verification of this function through the verification of its contract correctness.

7.8.1 Hoare triple verification

This verification targets to show that if the pre-conditions expressed on the function contract of the `compute_MinOutScalarMultipleInputsScalars` function are satisfied then the provided function implementation will satisfy the provided post-conditions. For this function, there is only one post-condition: the output is smaller or equals to any input.

```
goal WP_parameter_compute_MinOutScalarMultipleInputsScalars :
  forall nbInputs:int, in1:list (tInPortGroup real), functionParam: minMaxFunction.
  nbInputs >= 1 /\ length in1 = nbInputs /\ nbInputs > 1 /\ functionParam = Min ->
  (let o = length in1 - 1 in
    (1 > o ->
      (forall out:real. out = value_inpg (nth 0 in1) ->
        (forall i0:int. 0 <= i0 /\ i0 < length in1 ->
                        value_inpg (nth i0 in1) >=. out))) /\
    (1 <= o ->
      (forall res:real. forall out:real. out = res ->
        (forall i0:int. 0 <= i0 /\ i0 < length in1 ->
                        value_inpg (nth i0 in1) >=. out))))
```

Listing 7.38: Weakest pre-condition for the `compute_MinOutScalarMultipleInputsScalars` semantics

From the function contract and specification, the WHY3 tool computes a WP producing a proof obligation. Its successful verification implies that the function post-condition is verified according to the function code and its pre-conditions. The WP computed for this `BlockMode` semantics function is provided in Listing 7.38. In this WP, we clearly distinguish the pre-conditions (line 3), two cases are then given: one where `in1` is small enough ($1 > o$) for the loop not to be computed and the other case where `in1` is big enough for the loop to be computed ($l <= o$). For each of these cases the WP expresses that the computed code must imply the function post-condition.

The first case is trivially verified as its pre-condition is false (the length of *in1* cannot be smaller than 1). Regarding the second case the provided pre-conditions are not sufficient to allow the verification. The use of SMT solvers provides the same result. The impossibility to prove this contract is not a surprise as the function code contains a loop and no loop invariant is provided to characterise it.

Verification of function code containing loops is usually done using both loop variants and loop invariants. The loop variant provides a condition expressing the finite nature of the loop and the invariants express properties ensured before, during and after the execution of the loop. Using WHYML `for` loops, it is not required to provide a loop variant as it is directly inferred from the loop declaration. We thus only need to provide the loop invariant.

7.8.2 ADDING LOOP INVARIANTS FOR THE VERIFICATION

The loop invariant that needs to be provided must ensure that at each step of the loop, the actual `res` value is smaller than all the values of input ports that have already been compared. We provide the correct `BlockMode` semantics phase function code with the loop invariant in line 7 of Listing 7.39.

```
definition bal = compute_MinOutScalarMultipleInputsScalars {
  postcondition ocl {
    In1->forAll(i| i.value >= Out.value)
  }
  var res = In1[0].value;
  for (var i = 1; i < (size(In1)); i = i + 1){
    invariant { In1->subSequence(0,i-1)->forAll(e| res <= e.value) }
    if (res > In1[i].value){
      res = In1[i].value;
    }
  }
  Out.value = res;
}
```

Listing 7.39: Compute semantics phase with loop invariant

Providing loop invariants for the verification of code containing loop constructs requires the ability for the language to support annotation writing. We added the support for the writing of simple annotations in the BAL language loop constructs. Loop invariant annotations body is expressed with OCL. It is advised to use this language as it allows to express quantified expressions that are usually required in loop invariants.

Technical difficulties linked to the language scoping mechanism provided by the XTEXT platform made impossible to achieve in time the development of the transformation of BAL invariants as loop invariants

in the generated WhyML code. This is ongoing work and is expected to be solved shortly.

The addition of loop invariants for the verification of code raises the question of the ability for the block specifier to actually write these loop invariants. The code to be written in the block specification is supposed to be simple code but this does not necessarily make the loop invariant writing simple.

7.8.3 Automatic generation of invariants

The code constructs handled in the BAL language are limited and we do not allow for too complex data structures. This advocates for the possibility to augment the BlockLibrary translation to Why3 with a loop invariant generation capability.

This could be done by relying on pre-proven algorithm patterns for which loop invariants are already provided (this approach is also known as a template based approach [46]). If such patterns are detected in the code used for the specification then the loop invariants can automatically be integrated in the code. While this approach has the advantage of being quite straightforward and simple to implement (it still needs to define the patterns and to provide their loop invariants), it does not guarantee the usefulness of the loop invariant regarding the verification of the code as the loop invariant is generic for the loop and might be too weak for the verification of the overall code.

The use of backward propagation of the post-conditions with automatic inference of loop variants and loop invariants by relying on abstraction [127] or a combination of least and greatest fixed points [147] might also be applicable to our goal.

We did not experiment on these lines of approach but current works [133] provides interesting techniques and insights on how it can be dealt with.

7.9 Scalability

As was shown previously, the complexity of the *MinMax* block specification is not really challenging and does not cause specific problems (as soon as the required loop invariants are provided).

Regarding semantics specification verification, it is our purpose in the BlockLibrary DSML to limit the expressibility of the BAL language to simple constructs as it makes the block specifier more cautious about what he will express and restrict him in its way to express it.

Regarding the variability analysis, scalability can be at stake as the generated goals size can quickly become important. The completeness goal size is linear in the size of the BlockLibrary instance as it is a conjunction of all the configuration predicates. The generated disjointness goal size on the other hand is not linear in the size of the BlockLibrary instance structure. Indeed as we are doing two-by-two comparison of configuration predicates, the goal size is exponential. The verification of a full block specification is thus expensive but the specification of blocks is supposed to be an iterative process and thus it is expected for the verifications not to be done again every time the specification is modified as some parts of the specification will not change.

As an illustration of this goal size problem, we will use the full *Delay* block specification as provided in Appendix 6.5. In this block specification, we have defined 12 `BlockMode`. For each `BlockMode` we can extract 12 different `Signature` constructs. This leads to 144 different signatures. The generated completeness goal is a disjunction of 144 `Signature` predicates and is discharged by the CVC4 SMT solver in about 6 seconds. The generated disjointness goal is a conjunction of 10296 comparisons of two `Signature` predicates. This complete goal is discharged in about 150 seconds with the CVC4 SMT solver.

As we can see, the size of the generated goals can quickly grow and so is the verification time. By relying on the application of transformations, it is possible to split the main completeness and disjointness goal as multiple goals. Each generated goal is independent and thus can be verified on a different computer or processor core and thus the intuition of the ability to do this verification in a concurrent way is coming to mind. Works by Wintersteiger [158] or Déharbe [63] pave the way toward the use of SMT solvers in a distributed way. In both work, they bring forward the expected but impressive potential performance gain

brought by this approach.

7.10 Limitations

As we have shown in this chapter, our BlockLibrary language have some expressiveness and capabilities limitation. We will highlight here some of these and explain our choice of not tackling them in this PhD.

7.10.1 Dataflow languages capabilities limitations

We presented dataflow languages in Chapter 3. In this description we mentioned the kind of a port: DATA, ENABLE, EDGE_ENABLE or EVENT. In our BlockLibrary DSML, `PortGroup` are DATA ports.

ENABLE and EDGE_ENABLE ports allow to control the computation of the block according to the port value. We did not implement specific support for this kind of port as it is possible to implement them by adding: a) a new `MemoryVariable` storing the output value ; b) a dynamic `BlockMode` activated only if the ENABLE port value is greater or equal to zero and for which the defined compute semantics will be to output the previously produced output value. Of course this memory asks for a default value and thus the block configuration should provide it via an additional `Parameter`. An EDGE_ENABLE port implementation will be based on the previous one with the notable difference of declaring yet another `MemoryVariable` storing the port value in order to compare it and thus detect the rising or falling edge.

7.10.2 OCL and BAL Expressiveness limitation

We did limit the capabilities of the constraint and action languages in order to ensure the time related constraint of the PhD work. We did not implement all the rounding operations such as *floor* and *round* but we could have done so by reusing the already defined functions provided in the Why3 standard library. We do not allow the use of casting operations in BAL. Such operations can be quite complex to formalise and needs the development of type checking mechanism to ensure their correct use. Regarding data type management, we limited the range of data types allowed in our tool as we do not handle types like complex numbers. Such types could have been managed but no block used by our industrial partners in Projet-P do rely on complex numbers.

7.10.3 BlockLibrary limitations

As shown in the different BlockLibrary specification examples provided in this document, every `StructuralFeature` can be declared with only one attached data type. Whereas this forces the designer to model its block specifications carefully and take into account the difficulties related to the various allowed data types, it is not really convenient as we may want to specify a `StructuralFeature` value as being of any possible numeric data type. Adding support for multiple data types in the BlockLibrary language is not a problem but it introduces issues on the verification side as every possible combinations of allowed data types for all the `StructuralFeature` in a `Signature` should be verified. According to the strategy used in order to handle this, the result could be a combinatorial explosion of the number of generated `Signature` or a serious increase regarding the complexity of the generated code. Verifying all the data types combinations must also require to provide a feedback to the user on the forbidden combinations of data types in the block specification he is writing.

7.11 Synthesis

In this chapter we have show how we handle the verification of BlockLibrary conforming models. We provide for them a translational semantics based on the Why language. We also express in this same language the verification criteria for the completeness and disjointness of a block specification and show how they can be verified. We assess the block's semantics specification by relying on a translation of the block specification to WhyML functions with contracts.

Variability related verifications provides good results on full block specifications but semantics verification suffers from a weakness due to the lack of loop invariants expressed on the semantics. This problem might be solved by translating manually written invariants but might also be dealt with using automatic techniques.

In the first part of this PhD, we have analysed and tackled the problem of highly variable dataflow languages block libraries specification by developing a domain specific specification language. The specification verification is done by translating the specification to a formal domain where automatic verification techniques can be applied to ensure correctness properties. These verifications are backed up by efficient and reliable tooling.

In the second part of this PhD, we rely on the formal specification of blocks for the verification of automatically generated code. We will show how reliable formal specifications can be used for the verification of low and high level design properties on the generated code. In the context of tool qualification, we will show in which cases the formal specification can be considered as a source for the automatic generation of qualification data for safety critical systems development tools.

Part III

Automatic code generation verification based on the block library specification

8 Verification of generated code

Automatic code generation is one of the key benefit of the use of model-based development for embedded critical systems. But, in order to rely on its advantages, one first needs to ensure its correctness. Code generation verification can either be done by ensuring the correctness of the generator itself or by applying a translation validation approach (i.e. verification of the correctness of the generated code at each use). Verification can also be done by using either formal or more traditional techniques.

In this chapter we detail our use of formal methods in a translation validation approach for the verification of automatically generated code. We show how the formal informations gathered through the previously presented BlockLibrary specification approach can be used in order to decorate the generated code with behavioral annotations. Verification of the generated code according to the annotations is done using partly automated deductive verification techniques. Verification of system level properties based on previously generated annotations is investigated and its concrete implementation detailed. Improvements in the verification of automatically generated code is tackled by these approaches and their limitations is finally discussed.

8.1 Annotations for code verification

A correct BlockLibrary conforming model contains the detailed and verified, i.e. complete, exclusive and behaviorally correct, specification for dataflow blocks. For each block, it contains its structural variation points and for each one, its corresponding semantics. This semantics is expressed as Hoare triples including pre/post conditions and related actions. In this part, we focus on the pre/post conditions. Regarding ACG, the code generated for a block must comply with the BlockLibrary pre/post conditions of the semantics specification provided in the corresponding variation point for the block. Compliance assessment must then result in the verification of the generated code for this block. By translating each block configuration as annotations on the generated code, we target automatically generated code verification in a translation validation manner. Our proposal can be related to proof-carrying code (PCC) [113] in the sense that the generated code will contain annotations required to verify safety properties on the generated code.

8.1.1 Configuration matching of block

A block in a dataflow model has a fixed number of input and output ports, parameters and memories. Each of these have an attached datatype and parameters have a value. According to the block information, there is a unique correct configuration that applies for this block as the block specifications are exclusive.

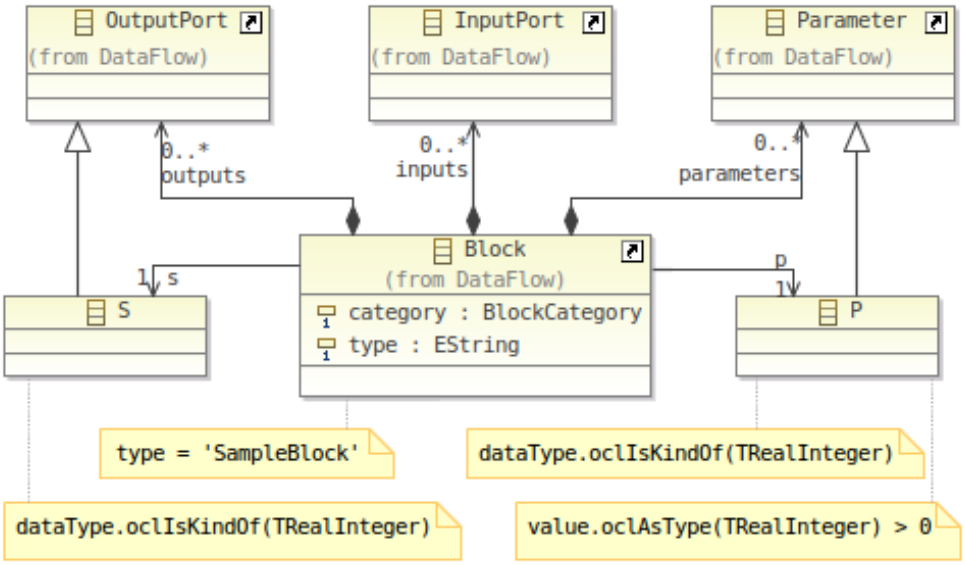


Figure 8.1: `Configuration`-specific generated metamodel example

Any block extracted from a dataflow model can be represented as a model conforming to the `Block` metaclass from the dataflow metamodel provided in Figure 4.5. This model should then conform first to the dataflow metamodel itself and to the constraints expressed on its matching configuration. The configuration must thus be expressed using the dataflow metamodel elements.

We propose a practical way to deal with this `Configuration` matching check by relying on a model transformation generating for each `Configuration` structure, a metamodel implementing the dataflow metamodel (the `Configuration`-specific metamodel) including the `Configuration` Hoare triple preconditions as additional OCL constraints expressed on the metamodel elements. We then have to check the conformance of a block (a model conforming to the dataflow metamodel), to the generated `Configuration`-specific metamodel and its associated constraints.

We provide a simplified example in Figure 8.1 for a simple block `Configuration`. In this example, we generate a `Configuration`-specific dataflow metamodel from the following `Configuration` structure: the `BlockType` name is *SampleBlock*, the block has one integer output port (the `S` output port) and one parameter (the `P` parameter). In addition to these `StructuralFeature` definitions, an `INVARIANT Annotation` is provided on the parameters specifying that its value can only be greater than zero.

Checking the block model according to this generated `Configuration`-specific metamodel can be done only if the block model actually conforms to the generated metamodel. The block model conforms to the dataflow metamodel and not to the `Configuration`-specific one. It is thus mandatory to adapt the block model by transforming its `StructuralFeature` elements to their matching generated metaclasses in the `Configuration`-specific metamodel.

We did not implement the `Configuration` matching mechanism including the `Configuration`-specific metamodel generation because of timing constraints. We did not identified any blocking points regarding its implementation, it shall thus be implemented without major difficulties.

8.1.2 Annotation generation

According to our purpose, we need to generate code annotations corresponding to the informations contained in each block `Configuration`. Annotation generation activity may be integrated with the ACG or done independently of the ACG with a delayed fusion mechanism. Generated code must then be verified with respect to the annotations.

According to our ability to modify the code generator and the independence criterion required by the

qualification activities, we identified two possible approaches to annotate the ACG generated code whether the ACG is considered as a white box or as a block box. We will detail these two approaches and the resulting annotated code.

Annotation generation integration in a "*white box*" ACG

Considering the ACG as a white box means that we have access to its implementation and can modify it. This one must be extended in order to: a) handle BlockLibrary conforming models and extract all the possible `Configuration` from it; b) match for each of the input model blocks its corresponding `Configuration`; and c) generate annotations along with the generated code containing the `Configuration` informations. This "*three phases*" way to integrate the use of the BlockLibrary specification into the ACG can be lightened by relying only on the third activity. Indeed, one can implement the annotation generation process without having to implement the BlockLibrary matching mechanism depicted in the first two points. This may lighten the ACG development process but is more likely to be error prone as it includes human in the loop development. Of course, our proposed approach relies on human development for the previously described three phases but this development must be done only once and can then be verified extensively.

Annotation generation integration in a "*black box*" ACG

Considering the ACG as a black box means that we are not able to access its implementation and thus to modify it. The use of an external tool shall be necessary in order to annotate the code. This tool must be able, based on the generated code and a BlockLibrary conforming model, to identify which part of the code matches which input block. This leads to the requirement for the ACG itself to provide traceability links between the input dataflow model elements and the output generated code. Traceability is a qualification prerequisite for the use of an ACG in critical embedded system development activities.

Assuming the presence of traceability informations in the generated code, our external tool must provide the following capabilities: a) identify for each block its corresponding `Configuration` specification; b) generate annotations based on the `Configuration` informations; and c) weave the annotations with the previously generated code relying on the traceability data.

Each `Configuration` elements must be matched to the generated code content, pre and post conditions must be added in the code and potential invariants/variants integrated on their corresponding loops constructs. Whereas matching the configuration's elements, transforming the annotations to cope with this matching and integrating pre/post conditions is not complex, the integration of variants/invariants is more complex. Indeed, if the generated code does not cope with the expected generated code (the one provided as operational semantics in the block specification), it seems difficult to automatically insert the loop annotations.

Whereas the generated code shape is not fixed for all ACG, the data structures allowed for use in safety-critical software generated code, the data types and data structures required to model dataflow models (scalar, vectors, matrices and if allowed bus structures) are limited. Loop annotations are required for operations applied on multi-dimensional structures likes matrices, these loops must be decorated by the developer with the strongest loop invariant and a variant ensuring the loop termination. This could be done by relying on known loop or multi-dimensional structures operations templates with predefined loop annotations.

Such an annotation integration suffers of a very limited applicability as soon as optimisations are present during the code generation; indeed code optimisations may break the structure of the code and the block segmentation of the generated code. The wide scope of this work made it difficult to tackle during this PhD and was thus kept as a future work. However, optimisations should also provide traceability elements that links the model to the optimised code including elements about the applied optimisations.

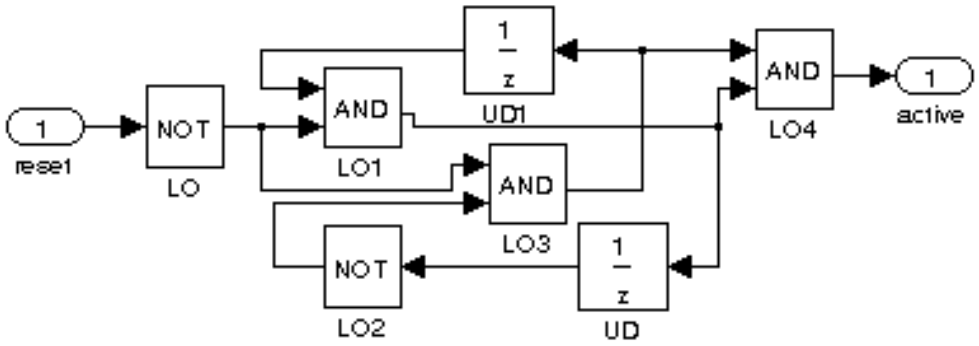


Figure 8.2: Counter SIMULINK model

```
typedef struct {
    BOOL reset;
    BOOL active;
} t_Counter_io;

typedef struct {
    BOOL UD1_memory;
    BOOL UD_memory;
} t_Counter_state;
```

Listing 8.3: GENEAUTO generated code for the *counter* model data structures declaration

RESULTING ANNOTATED CODE

From the previous step, we get source code files containing functions organised according to the expected ACG output structure. According to the ACG used, generated code has a specific shape. In the following examples, we used the GENEAUTO ACG. Adaptations are required for each ACG. However, as these ones target safety critical systems, we are confident that the safety critical traceability requirements will ease this aspect.

For each atomic *Sub-System*, three functions are generated. Two optional containing the initialisation and update semantics phase code for the memories variables initialisation and update; and the compute phase code that is mandatory. In each function, blocks of source code are generated for each block of the enclosing *Sub-System*. Each block of code is surrounded by traceability informations allowing to link it to the dataflow block it was generated for.

Figure 8.2 depicts the *Counter* model that was presented in Chapter 3. It models a modulo 3 counter for which the output (the *active* output block) is set to *true* every three clock tick.

The corresponding code generated with the GENEAUTO ACG for the system data structures is provided in Listing 8.3. Generated code along with its traceability informations is provided for the initialisation

```
void Counter_init(t_Counter_state *_state_) {
  /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
  _state_->UD1_memory = FALSE;
  /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
  /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
  _state_->UD_memory = FALSE;
  /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
}
```

Listing 8.4: GENEAUTO generated initialisation code for the *counter* model

```
void Counter_compute(t_Counter_io *_io_, t_Counter_state *_state_) {
  BOOL reset;
  BOOL LO;
  BOOL LO2;
  BOOL LO4;
  /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
  Counter_UD1 = _state_->UD1_memory;
  /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
  /*  START Block: <SystemBlock: name=Counter>/<SourceBlock: name=reset>  */
  reset = _io_->reset;
  /*  END Block: <SystemBlock: name=Counter>/<SourceBlock: name=reset>  */
  /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO>  */
  LO = !reset;
  /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO>  */
  /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO1>  */
  Counter_LO1 = Counter_UD1 && LO;
  /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO1>  */
  /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
  Counter_UD = _state_->UD_memory;
  /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
  /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO2>  */
  LO2 = !Counter_UD;
  /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO2>  */
  /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO3>  */
  Counter_LO3 = LO && LO2;
  /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO3>  */
  /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO4>  */
  LO4 = Counter_LO3 && Counter_LO1;
  /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO4>  */
  /*  START Block: <SystemBlock: name=Counter>/<SinkBlock: name=active>  */
  _io_->active = LO4;
  /*  END Block: <SystemBlock: name=Counter>/<SinkBlock: name=active>  */
  /*  START Block memory write: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
  _state_->UD_memory = Counter_LO1;
  /*  END Block memory write: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
  /*  START Block memory write: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
  _state_->UD1_memory = Counter_LO3;
  /*  END Block memory write: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
}
```

Listing 8.5: GeneAuto generated compute and update code for the *counter* model

phase in Listing 8.4 and for the compute and update phases in Listing 8.5.

8.1.3 Annotation verification

In the previously provided example, we decided to rely on C code along with ACSL annotations for the following reasons: this language is widely used as target in the safety-critical systems industry, the GeneAuto ACG has mainly be used and evaluated in this context and the ACSL annotation language is supported by formal analysis tools like the code analysis framework Frama-C. This could have been done by relying on Ada code with Spark annotations and the Spark Examiner tool.

Using the Frama-C framework, one can analyse the C code in order to extract informations provided by various plugins. These plugins allow to do static analysis of the source code to extract informations like variables ranges and scope, code metrics, detect dead code, and many others [1]. The Frama-C framework analyses ACSL annotations and is also able to automatically generate additional annotations regarding potential runtime errors that may happen during the analysed program. This mechanism is provided in the *RTE* plugin.

We use the Frama-C framework in order to analyse ACSL annotations and as a bridge toward the use of SMT solvers and proof assistants for the verification of source code. The framework provides two plugins: *WP* and *Jessie* fitting this purpose. Both plugins implement a weakest precondition calculus on the C source code annotated using ACSL. These generated logical formula can then be assessed using SMT solvers or sent to the previously presented Why3 platform to rely also on SMT solvers and proof assistants. Proof assistant are used in order to tackle difficult proofs when the SMT solvers fails to achieve the proof.

The provided code verification is done quite easily as the generated code is simple and only implies the use of scalar *Relational Operator* and *Unit Delay* blocks. We experimented on the generation of code for a model with the same blocks but with vector values. This implies the generation of loops and defining loop invariants. The result was still automatically verifiable.

8.1.4 Tool support

The aforementioned automatic annotation generation proposals requires a strong tooling development effort. Regarding the white box approach, the ACG needs to be fitted for annotation generation whereas regarding the black box approach an external verification tool needs to be developed.

White box approach in GeneAuto

GeneAuto is our main experimentation platform on ACG development. We added support for annotations manipulation [2]. For this purpose, we developed a metamodel based on a subset of the ACSL specification [1] covering annotations, ghost code and function contracts elements. We give a graphical representation of this metamodel in Figure 8.6.

In this metamodel, the main element is the *VAElement* (standing for *Verification Annotation Element*) extending the classical GeneAuto *Annotation* element. *VAElement* is refined as *SimpleVAElement* and *CompoundVAElement*. Expressions in *VAElement*s are expressed as *AnnotationStatement* statements extending GeneAuto's *Statement* metaclass. Sub metaclasses of *AnnotationStatement* represents the various annotation clauses among which are *Assign*, *Ensures* (post-condition), *Requires* (pre-condition), *LoopInvariant*, ... The metaclasses names are clearly inspired from the ACSL terminology.

We relied on the EMF framework to generate the corresponding Java classes. We hence integrated those into GeneAuto's code model. In order to print the annotations on the generated code, we implemented a printer for ACSL annotations in the GeneAuto tool. These elements allow developing GeneAuto *code backend* generating annotations along with the code.

[1] Visit the Frama-C framework website for detailed informations: http://frama-c.org

[2] This work has been done as a partnership with Timothy Wang from Georgia Tech University and with the help of Andres Toom from *IB KRATES* and *Technical University of Tallinn*, one of the main developer of GeneAuto

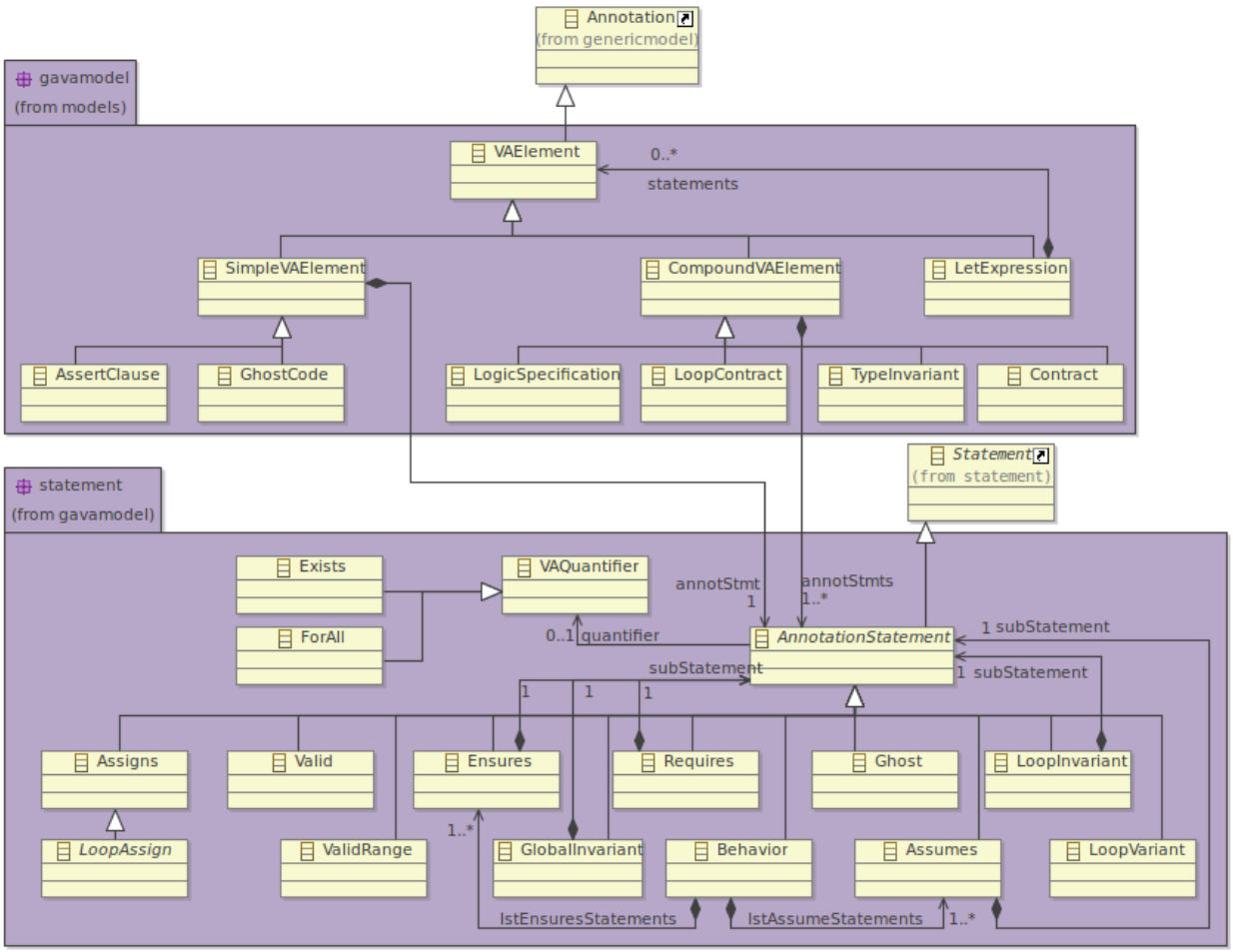


Figure 8.6: GeneAuto annotations extension metamodel

Black box approach through Frama-C

The Frama-C toolset provides C code manipulation facilities and is extensible by defining plugins. It would thus be possible to define a new Frama-C plugin to conduct automated annotation of the generated source code.

This plugin must extract the configurations from a BlockLibrary specification. The specification is available as an EMF model. Developing a parser for the BlockLibrary models and extracting informations from these directly in the plugin would be a very time consuming activity. Instead of this approach we propose to provide, within the BlockLibrary toolbox, a transformation extracting the required informations as a simple data structure easier to use in a Frama-C plugin. Such an approach was used to bridge the Java XML reader/writer in GeneAuto and the CaML source code extracted from the Coq development of the block sequencer (see the work done around the GeneAuto toolset by Izerrouken et Al [76, 78, 149]).

Regarding the configuration matching mechanism, the plugin must be able to extract from the source code the required informations about the blocks `StructuralFeature` elements. This can be done relying on the traceability comments provided by the ACG: one can find the corresponding block in the model source file and thus extract the required informations. Matching between the block configuration and extracted BlockLibrary block configurations must then be conducted.

Finally, from the matched configuration, pre and post conditions are attached as a contract to the block of code and loop variants/invariants to the loops. It is worth noticing that a translation mechanism must be developed in order to transform BlockLibrary `Annotation` expressions to a formalism that can be used for the verification of code. A first mean will be to generate annotations on the generated code (ACSL

annotations on C code for example) or to rely on other initiatives like the one currently under development at *CEA* targeting the use of WHY3 language constructs through ACSL annotations. Regardless of the chosen approach, this mechanism must provide translations services to:

- Match input and output ports names to generated source code variables names.
- Remove calls to the value attribute.
- Unfold quantified expressions on `PortGroup` constructs. Indeed, OCL collection expressions applied on `PortGroup` elements cannot be used as such. Each input of the block is a separated variable. Assuming I as a `PortGroup` containing n ports, we provide in Table 8.7 the transformations rules for collection operations applied on I. In this table, $[x/it]exp$ means the substitution of all occurrences of *it* in *exp* by x.

OCL expression	ACSL expression
$I \rightarrow forAll(it\|exp)$	$\bigwedge_{i\in[1..n]} [I_i/it]exp$
$I \rightarrow exists(it\|exp)$	$\bigvee_{i\in[1..n]} [I_i/it]exp$
$I \rightarrow one(it\|exp)$	$\bigoplus_{j\in[1..n]} [I_j/it]exp$ With $\bigoplus$ meaning the xor logical operation
$I \rightarrow isUnique(it\|exp)$	$\bigwedge_{j\in[1..n]\wedge k\in[1..n]\wedge j<k} [I_j/it]exp \neq [I_k/it]exp$

Table 8.7: OCL to ACSL translation rules

This development could not be conducted during the time frame of this PhD. It is thus kept as a future work as its application is of wide interest, as will be presented in the following section. The generated annotations allow assessing the correctness of the generated code with respect to the language specification. They are also needed to ease the verification of more complex functional and integration properties and thus corresponds to a kind of unit properties. The previously provided translation from BLOCKLIBRARY to WHY is targeting a first order logic language and thus provides a good starting point for the transformation to annotations on code. A similar work has been done by *ATOS ORIGIN*[3] and ONERA during the PhD of A. Fernandez-Pires. This work generates ACSL functions contract from OCL annotated UML class operation. This transformation is provided as an ECLIPSE plugin.

8.2 FORMAL VERIFICATION

Software verification must be made according to the requirements expressed during its design phase. Starting from high level requirements (*HLR*) that express the problem to be solved, refinement techniques are used to design a satisfying solution for these requirements. These refined requirements are often referred to as low level requirements (*LLR*). *LLR* can then be translated to development artifacts.

During common development cycles, when the system has been developed, it is first unit tested , based on these *LLR* (this can be related to the previous section annotations use). Then verification goes up to integration and functional phases in the requirements level up to the *HLR*.

In this section, we will investigate the use of Synchronous Observers [73] (*SO*) as a means for the expression of the system *HLR* requirement during the design phase and their automatic verification on the generated code level relying partly on the previously generated block semantics annotations to ease the verification.

[3] https://code.google.com/a/eclipselabs.org/p/ocl2acsl/

8.2.1 Synchronous observers

Using the Simulink toolset, we created a masked sub-system block: the *Observer* block. A masked block can be considered as a specialised sub-system block on which it is possible to attach additional informations as `Parameter`. This masked block has one `Parameter: AnnotationType`. This `Parameter` value is of an enumerated type with two possible values: `pre-condition` or `post-condition`. An overview of the *Observer* block is given in Figure 8.8. In this figure the top-left part is the view of the *Observer* block itself, lower-left part is the content of the *Observer* block and the right part is its parameter window.

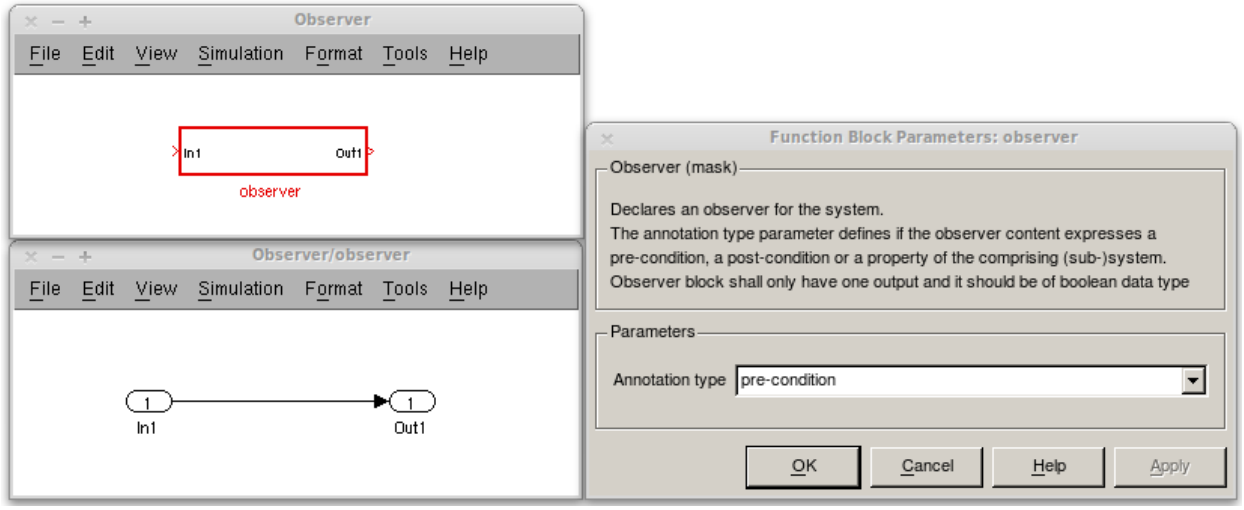


Figure 8.8: The *Observer* block, its content and its parameters view in the Simulink environment

The number of inputs for a *SO* block is not limited and must at least be 1. The observers inputs must be plugged onto outputs of the observed system blocks.

In the modeling environment, the block behaves exactly as any other *Sub-System* block. But, its interpretation in the ACG is different and leads to the generation of annotations.

In the following we propose an example of *SO* for the *Counter* system, its interpretation as logical properties and its translation as ACSL annotations. We will then generalise the approach and propose a generic methodology for the extraction of logical expressions from *SO* blocks based on BlockLibrary specification.

8.2.2 Concrete application on the *Counter* system

As sake of example, we propose a *SO* for the *Counter* example model. We depict the use of the observer on the *Counter* model in Figure 8.9 and detail its content in Figure 8.10. This block has two inputs plugged on the observed system input and output. The observer `AnnotationType Parameter` value is set to `post-condition`. In the following, we will detail the observer semantics, its use for verification through simulation, one possible translation to code annotations and its practical verification.

SO formal semantics and exploitation

The formal semantics for the *SO* provided in Figure 8.10 is provided by relying on mathematical notations in (8.1). We set s_t as being the value of the output of the *Switch* block at time t and thus s_{t-1} as the value of

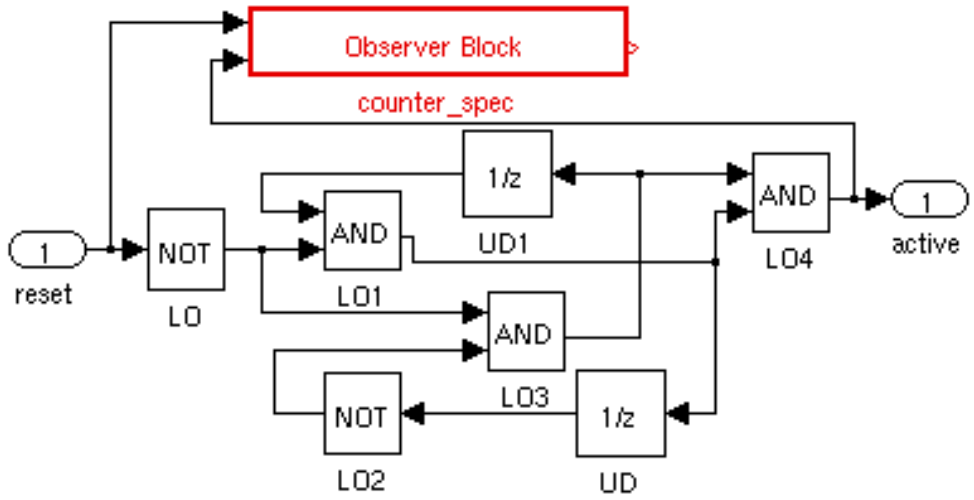


Figure 8.9: The *Counter* model with its *counter_spec* synchronous observer

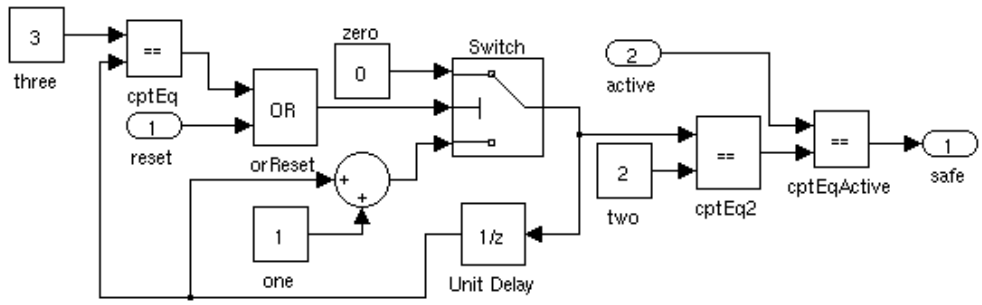


Figure 8.10: The *counter_spec* synchronous observer content

the output of the *UnitDelay* block at the same time t.

$$\begin{aligned} s_0 &= 1 \\ s_t &= 0 \text{ if } ((s_{t-1} == 3) \vee reset_t) \\ &\quad\ 1 + s_{t-1} \text{ else} \\ safe_t &= (active_t == (s_t == 2)) \end{aligned} \tag{8.1}$$

In this semantics formalisation, we can see that s_t can never be greater than 3, that its value is resetted to zero when *reset* is true or its previous value is 3 and that it is increased by 1 in every other cases. The observer output value: *safe* is the comparison between the value of *active* – which is plugged into the output of the *Counter* system and is a boolean value – and the comparison between the value of s and 2. From this we can conclude that the *safe* value is true only once every 4 time steps (if *reset* is not set to true during these 4 time steps) and 4 time steps after the last time *reset* is set to true. With this observer we thus are observing a modulo 3 counter which is the expected semantics for the *Counter* system.

The *safe* value is the output of the *SO* and as such if we consider the *SO* as a property it must be converted to a logical expression. The *SO* semantics is twofold as it contains two alternative branches (the s_t value in (8.1)). In order to express this semantics we must expose the alternative branches into the logical expression as a conjunction. The *SO* expression is to be expressed according to the *SO* inputs: *safe* and *reset* but care must also be taken on the fact that the *Unit Delay* block has a memory so it is holding a value that must be considered as a variable of the expression. We thus provide the expression as the body of a Why predicate in Listing 8.11.

```
predicate counter_spec (reset: bool) (active: bool) (s: int) =
  (((3 = s) \/ reset) -> ((0 = 2) = active)) /\
  ((not((3 = s) \/ reset)) -> ((s+1 = 2) = active))
```

Listing 8.11: Counter_obs observer expression as a predicate

By expressing this observer as a *Sub-System* block in a simulation tool like Simulink (as in Figures 8.9 and 8.10), it is possible to first do some testing activities by providing values for the inputs of the system and by asserting that the *SO* output value is true for these test values. According to atomic blocks semantics, test cases (test vectors) are automatically generated and can be used in order to verify the design properties. Design properties and assertions can be expressed on the model with verification blocks. Verification blocks are special blocks holding properties expressed using Simulink blocks (in the same way as a *SO*). Verification of the verification blocks properties is done according to automatically generated test cases. Formal verifications can be applied on the model using Simulink design verifier allowing to formally check (using abstract interpretation) the model for the absence of classical run-time errors such as division by zero or integer overflow.

Translation to code annotations

In order to translate the *SO* as an ACSL annotation, we need to compute the post-condition for each of its containing blocks implementation. As specifications take too much space to be displayed here, we provide only the exact post-condition for each block in Table 8.12. Each of these blocks have been specified in a BlockLibrary and are provided on our website [2].

Block name	Block post-condition
reset, active	out == reset, out == active
zero, one, two, three	out == 0, out == 1, out == 2, out == 3
cptEq, cptEq2, cptEqActive	out == (In1 == In2)
orReset	out == In1 || In2
Sum	out == In1 + In2
Switch	if (In2) then out == In1 else out == In3
UnitDelay	init: Unit_Delay_Memory == IC compute: out == Unit_Delay_Memory update: Unit_Delay_Memory == In1

Table 8.12: *counter_obs* blocks post-conditions

Listing 8.13 shows the three predicates generated from the *counter_spec SO*. Each predicate corresponds to a semantics phase of the *SO*.

In this example, the extraction of expressions is quite straightforward as we rely on blocks applied only on scalar elements. This approach can be extended to multi-dimensioned expressions, but time constraints did not allow to conduct the experiment in the scope of this PhD.

Practical verification

The verification of the previously generated *SO* may not be successfully done fully automatically using the Frama-C tool. Indeed, in the verification of the compute and update semantics phases, additional informations about memories initial values are needed from the initialisation phase.

In order to ensure the verifiability of the overall system, including memories, we must provide the encompassing code including the call to the three semantics phase functions. The initialisation and update semantics phases for the *SO* must also be provided. In the next section, we give an example of such an automatically generated code and insights on its automatic verification.

```
/*  START Block: <SystemBlock: name=Counter>/<SystemBlock: name=counter_spec>  */
/*@ predicate counter_spec_init (t_counter_spec_loc *obsState) =
      obsState->Unit_Delay_memory == 0;
*/
/*@ predicate counter_spec_compute (t_counter_spec_io *obsInput,
                                    t_counter_spec_loc *obsState) =
      ((3 == obsState->Unit_Delay_memory) || obsInput->reset) ==>
      ((0 == 2) == obsInput->active) &&
      (!((3 == obsState->Unit_Delay_memory) || obsInput->reset)) ==>
      (((obsState->Unit_Delay_memory + 1) == 2) == obsInput->active);
*/
/*@ predicate counter_spec_update (t_counter_spec_io *obsInput,
                                   t_counter_spec_loc *obsState) =
      ((3 == obsState->Unit_Delay_memory) || obsInput->reset) ==>
      obsState->Unit_Delay_memory == 0
      &&
      (!((3 == obsState->Unit_Delay_memory) || obsInput->reset)) ==>
      obsState->Unit_Delay_memory == obsState->Unit_Delay_memory + 1 ;
*/
/*  END Block: <SystemBlock: name=Counter>/<SystemBlock: name=counter_spec>  */
```

Listing 8.13: *counter_spec* synchronous observer as a ACSL predicates

8.2.3 Logical expression extraction

The blocks composing any *SO*-s must be defined in the block library. This provides their formal specification including both their operational and axiomatic semantics. The purpose of a *SO* is to be interpreted as a logical expression formulating the *SO* semantics and the system *HLR*. In dataflow languages, the semantics of a model is held in the blocks. An expression of the *SO* semantics must thus be based on the semantics of its contained blocks.

From the BlockLibrary specification, we obtain the blocks axiomatic semantics, we hence propose to rely on the expression of this one to express the semantics of the whole *SO*. We detail in the following an algorithm for this extraction and provide an example of its application.

Synchronous observer semantics extraction

The output port of a *SO* provides its final value. The expression of the semantics of the *SO* is thus the expression of the post-condition of the output block. For every block, the post-condition expression is provided as a function of the block inputs, parameters and memories. As a block input port must be linked to another block output port through a signal, this input port value must be expressible according to the preceding block output port value.

Lets assume the *SO* provided in Figure 8.14. We can express the output ports values of a block according to a function of the blocks inputs and internal values (parameters and memories). This function is an operational semantics of the block. In (8.2) we give functions definitions for the three B1, B2 and B3 blocks composing the *SO*. The last parameter of the function call $(PM(X))$ models the parameter and memory values of the block X.

$$\begin{aligned}(b1out1, b1out2) &= f_{B1}(b1in1, b1in2, PM(B1))\\ b2out &= f_{B2}(b2in, PM(B2))\\ b3out &= f_{B3}(b3in1, b3in2, b3in3, PM(B3))\end{aligned} \tag{8.2}$$

The *SO* output value must be a boolean value as it is supposed to represent a logical expression. The $b3out$ value can then be considered as a logical expression. The only element to be resolved is then the definition of the signal's semantics which was previously presented as a copy of its input value to the output value.

The definition of the *SO* as a logical expression can be expressed according to the inputs of the B1 and B2 blocks and thus according to the *SO* inputs. The result is detailed in (8.3) where π_n is the projection

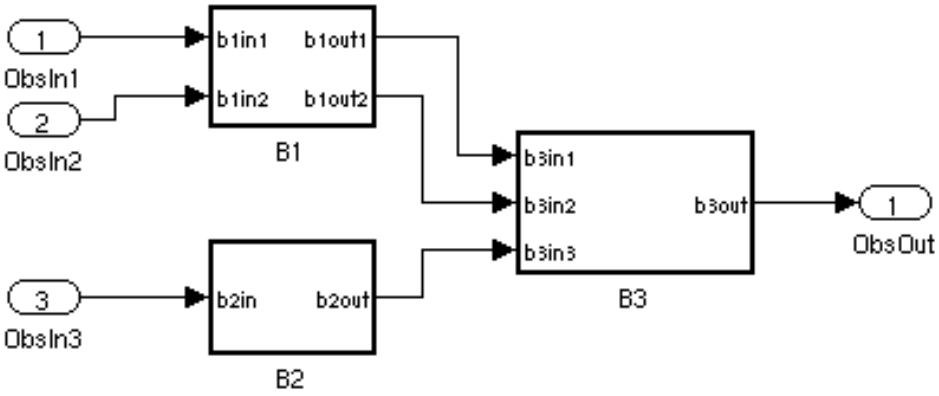


Figure 8.14: An abstract synchronous observer

operator providing the nth element of a tuple.

$$\begin{aligned} f_{B3}(\quad & \pi_1(f_{B1}(ObsIn1, ObsIn2, PM(B1))), \\ & \pi_2(f_{B1}(ObsIn1, ObsIn2, PM(B1))), \\ & f_{B2}(ObsIn3, PM(B2)), \\ & PM(B3)) \end{aligned} \tag{8.3}$$

Finally, each *SO* inputs is plugged onto the observed system inputs and outputs. It is thus possible to define the (8.3) expression using only values of the observed system.

In the BlockLibrary specification, we have at least one logical expression expressed as a post-condition for each f_{BX} function. We can thus replace these ones in the previous expression. If multiple post-conditions are specified for a block, the f_{BX} function is the conjunction of these expressions. The transformation must include the application of the translation rules detailed in Section 8.1.4. The result provides us with the logical expression for the *SO* that can be used as its semantics expression.

Management of memories

In the previous paragraph, we deliberately omitted the management of memories. Indeed, regarding memories values, we did not handle the setting of their initial value nor the update of these values.

As for the observed system, the *SO* memories need to be initialised once during its containing system initialisation phase and the memory value must be accessible in the whole source code generated for the observer system. We thus decide to declare specific *SO* memory variables containing the *SO* memories values. Those are initialised after the observed system initialisation code. As this is only used for verification purpose, this *SO* initialisation code must not impact the system semantics and thus is held in *ghost* code. In the BlockLibrary, the post-condition expressed on the memory block initialisation phase specification is used as an additional post-condition of the initialisation phase function as it is mandatory to ensure that the *SO* initialisation is done right after the initialisation of the observed system.

Regarding the update semantics phase of the *SO* memories, its computation is done after the system update phase code. Again this *SO* update phase code is expressed as *ghost* code to avoid any impact on the semantics of the observed system. The post-condition expressed on the memory block update phase specification in the BlockLibrary is used as an additional post-conditions of the update phase function as it is mandatory to ensure that the *SO* update is done at each update of the observed system.

The generated code for both initialisation and update semantics phase is the one that would have been generated by an ACG but it is written on the generated software as ghost code. Its position in the generated code is the one that is required by the dataflow. Ghost code is supposed not to impact regular code, this is not yet proven correct but works are in progress in this sense [67]. In addition to that, it is required to

```
/* Observers data structures */
/*@ ghost typedef struct {
    BOOL reset;
    BOOL active;
    BOOL safe;
} t_counter_spec_io;
*/

/*@ ghost typedef struct {
    UINT8 Unit_Delay_memory;
} t_counter_spec_state;
*/
```

Listing 8.15: *SO* generated data structure

```
//@ ghost t_counter_spec_io * _counter_spec_input;
//@ ghost t_counter_spec_state * _counter_spec_state;
```

Listing 8.16: *SO* generated data structure variables

provide a verification ensuring that the *SO* memories are independent of other memories.

8.2.4 Main module generation

Embedded safety-critical systems are systems that must be executed periodically. Using dataflow languages for their development constrains their structure and enforce the use of the three phases semantics pattern. We can thus automatically infer from this the software for the whole system initialisation and execution. In our work, we decided to generate a function in a separate module.

We provide in the following the generated *main_module_counter* for the *counter* model and its *counter_spec SO* using our extension of the GeneAuto toolset:

- The required data structures for the *SO* are declared. *t_counter_spec_io* contains the *SO* input and output variable declarations; and *t_counter_spec_state* contains the *SO* memory variable declarations. Generated elements are provided in Listing 8.15.
- Ghost variables are created for the two previous data structures. Generated elements are provided in Listing 8.16.
- *SO* semantics is expressed as three predicates, one for each *SO* semantics phase. These three predicates declarations, as functional generated code, are traceable to the source model via traceability comments. Generated elements have already been provided in Listing 8.13.
- The *main_module_compute* module is generated and its code is provided in Listing 8.17. The modules has two inputs, these are system data structures. In the main function body, a call to the initialisation function is done and then the *SO* initialisation code is inserted as ghost code. Using an *assert* annotation, we ensure the correctness of the ghost initialisation code. The main module loop is annotated with a simple loop assignation clauses specifying the variables assigned during the loop computation. In the loop body, the main computation function is called; the *SO* input values (held in the _counter_spec_input data structure) are updated according to the new values in the system in ghost code and; the update semantics phase of the *SO* is added as ghost code. The correctness of both compute and update *SO* semantics phases is finally assessed. With the verification of all the *assert* clauses, the correctness of the system code will be assessed.

The main module verification is done using Frama-C and is successful in a few seconds for this example. This *Counter* system is representative of scalar only systems as it contains both sequential and combinatorial blocks. The complete code for this example is provided in Appendix E.

```
/*@ requires \valid(_state_) && \valid(_io_);
    requires \valid(_counter_spec_state) && \valid(_counter_spec_input);
    requires \separated(_state_, _io_);
    assigns _counter_spec_state->Unit_Delay_memory, _io_->active,
            _state_->UD_memory, _state_->UD1_memory;
*/
void main(t_Counter_state *_state_, t_Counter_io *_io_) {
  Counter_init(_state_);
  //@ ghost _counter_spec_state->Unit_Delay_memory = 0;
  //@ assert counter_spec_init (_counter_spec_state);
  /*@ loop assigns _counter_spec_state->Unit_Delay_memory, _io_->active
                   _state_->UD_memory, _state_->UD1_memory; */
  while (TRUE) {
    Counter_compute(_io_, _state_);
    //@ ghost _counter_spec_input->reset = _io_->reset;
    //@ ghost _counter_spec_input->active = _io_->active;
    /*@ ghost
      if ((3 == _counter_spec_state->Unit_Delay_memory) ||
          _counter_spec_input->reset)
        _counter_spec_state->Unit_Delay_memory = 0 ;
      else
        _counter_spec_state->Unit_Delay_memory =
          _counter_spec_state->Unit_Delay_memory + 1 ;
    */
    //@ assert counter_spec_compute (_counter_spec_input, _counter_spec_state);
    //@ assert counter_spec_update (_counter_spec_input, _counter_spec_state);
  }
}
```

Listing 8.17: *Counter* main module

Regarding the current status of the prototype development, the automatic generation of the main module code, ghost predicates, ghost structures and their instantiation is implemented. The future step of this development is to add support for the synthesizing of the predicates and ghost code body. This implementation is the more delicate one as it implies to implement the translation of the *SO* to a predicate according to the *SO* block's `Configuration` extracted from a BlockLibrary specification. It thus necessitate a rather consequent time investment.

8.2.5 From specific kind of properties to annotations

Using the annotation handling extensions to the GeneAuto ACG, work has been done by Wang et al [154] in order to handle the verification of control and command algorithms via the assessment of high level properties of their control laws like robustness. This work tackles the automatic verification of Lyapunov stability performance measures and is currently being extended for a broader range of applications. In this work, the verification is also done using *SO*. Property-specific *SO* have been created and an extension of GeneAuto providing the automatic generation of annotations has been developed. The automatic verification of the properties on the generated code is tackled by a dedicated PVS backend for linear algebra.

8.2.6 A parallel work based on *SO*

The *SO* approach is the subject of a publication [61] on which we developed an automatic code generator derived from GeneAuto from Simulink to Lustre and then from Lustre to C targeting the verification of the generated code and of its attached properties (the *SO*).

The first part of this work has been developed as a model transformation based extension of the GeneAuto toolset. A Lustre extension adding support for annotations has been developed. In our translation, Simulink models and *SO* are translated as Lustre nodes, additional annotations are generated to ensure the link between the system generated nodes and the *SO* ones. The Lustre to C translation, based on the translation scheme by Biernacki et Al [24], handles both Lustre code and annotations and translates them to C source code with ACSL annotations.

Generated Lustre code is verified against *SO* annotations using SMT-based model checking techniques allowing to prove the validity of the generated observers. Verification of the C generated code is done by relying on the Frama-C toolset.

8.3 Gain wrt classical verification activities

The approach detailed here provides the ability to express verification properties on high level design of systems. Such properties are expressed directly on the model and are then translated as code annotations. Their verification can be done using the Frama-C toolset, SMT solvers and if needed proof assistants.

8.3.1 A complement to state of the art design verification

State of the art embedded systems design verification is done by relying first on simulation with tools like Simulink. As was previously detailed, formal verifications can be applied on the design using Simulink design verifier. Automatic test cases generation can also be done. Test cases can also be used on the system executable code in order to ensure the respect of the behavior previously assessed. Whereas this approach provides good insights on the correctness of the design, early verification of the design and allows detection of errors, it does not provides a formal verification of the properties as it is only test based and suffers from exhaustiveness.

In most safety-critical industries, Scade is used to express the system after its design and early verification using Simulink. At the Scade level, properties can be expressed using *SO* and as Scade relies on Lustre, formal verifications can be done using model checking with for example the Prover plugin[4] also usable for the verification of systems specified in languages such as C, Simulink or UML.

These design verification approaches does provide formal verification regarding the system correctness at a design level. They allow also to generate tests that can be used on the manually written or automatically generated code but does not provides formal assurance of the properties verification on the code.

On the contrary, by relying on our approach, design level properties (both *HLR* and *LLR*) can be gathered on the code level and formally verified. Translation and verification on the code is made possible by the providing of our formally specified block library.

8.3.2 Current limitations and perspectives

The core of our verification approach is based on the ability to generate both *LLR* and *HLR* expressed at the design level as annotations to be verified at the code level. This generation relies on the formal specification of blocks provided by the BlockLibrary specification approach. Both *LLR* and *HLR* are embedded as code annotations, *LLR* as blocks of code contracts and *HLR* as predicates used in specific locations of the system computation. We have shown the ability for our annotation generation mechanism to provide properties verifiable on concrete examples.

As was previously stated, the annotation generation mechanism is currently not fully implemented. We presented in Chapter 7 a translation mechanism converting OCL constraints as Why expressions. On going work at *CEA* aim at providing the ability to rely on Why3 formalisation (theories and modules) in ACSL verification. Providing such a functionality in a tool like Frama-C could ease our translation work as it might then be possible to rely on the theories generated in Chapter 7 in that purpose.

The block specification and *SO* are providing the expected behavior of the generated code. As such they may be used as a source for the automatic generation of test cases. The tests will be used for the runtime verification of the system ensuring fault detections or to complete the formal verification of the generated code when the automatic proof fails. We will rely on these in the following for the automatic generation of certification and qualification data.

[4] http://www.prover.com/products/prover_plugin/

9

Certification/Qualification data generation

Software certification activities aims at increasing the confidence a user can have on the use of a software system. Such confidence is often achieved by providing data detailing: the system development activities and processes; the results of the software verification activities; or data produced by reliable (i.e. qualified) third party tools used in the development activities.

State of the art techniques for confidence assessment on a software is usually produced by relying on proof reading and intensive testing. By nature, these approaches, while being widely used, cannot be exhaustive and thus cannot be fully trusted especially while developing safety-critical software.

In this chapter, we will show how the BlockLibrary DSML and related tools presented in this document could be integrated in the qualification process of an ACG. We will detail the possible uses we have identified for BlockLibrary instances in order to generate certification data and the (formal) methods that can be used along these data in order to gain certification credits.

9.1 BlockLibrary for qualification

BlockLibrary specification writing aims at providing detailed and formal specification for an ACG input language. In order for our approach to be valuable for an ACG qualification use, one should ensure that the formalism and underlying technologies used are likely to be accepted by certification authorities.

As detailed in this document, our BlockLibrary DSML has been designed using a model based development approach, its semantics has been defined by formalisation and by model transformation, and its correctness is ensured using formal methods. Each of these three elements, as the building blocks of our approach, must be considered in the context of certification.

The BlockLibrary DSML may provide confidence in the code generated by an ACG. One may use the BlockLibrary DSML as means to verify the generated code and thus may avoid some verification activities on the generator.

There is no explicit text in *DO-330* stating whether it is possible to apply *DO-178C* technology related 'supplements' in the qualification activities of a tool. Indeed, the 'supplements' are meant to be applicable on *DO-178C* related qualification and *DO-330* is meant to be domain agnostic. As this applicability is not explicit, it is recommended that their application should be justified through the elicitation of the *DO-330* document guidance to be satisfied through the use of 'supplement' documents [126]. It is important to put an emphasis on the fact that as a specialisation of the *DO-178* document for tools qualification, 'supplement' documents application seems likely to be doable.

In the following sections, we will detail how the BlockLibrary DSML definition, formalisation and verification applies in the context of the certification of an ACG according to *DO-178C* and *DO-330* espe-

cially by relying on *DO-178C* 'supplement' documents *DO-331* and *DO-333*.

9.1.1 *DO-331*: Model-based technology

We have shown previously that our BlockLibrary approach is appropriate for the specification and verification of block libraries for dataflow languages and for the verification of the generated code with respect to the language semantics and the user requirements. We claim that our BlockLibrary approach has the required characteristics expressed in *DO-331* (detailed in Section 2.1) in order for a model to be used for qualification activities:

"*The model is completely described using an explicitly identified modeling notation*". The BlockLibrary metamodel is specified using Ecore that is a standard (defined after the MOF notation) and a well accepted modeling notation (highly used in industrial applications). Building models from standard metamodels (i.e. Ecore) and additional standard constraint languages (i.e. OCL) is one of the strengths of the model-driven approach as it allows to formally specify a data structure (the model) from an already specified and accepted formalism.

"*The modeling notation has a precise grammar (also called "syntax") and meaning (also called "semantics"). The modeling notation may be graphical and/or textual.*" From the BlockLibrary, a grammar has been defined in order to provide a syntax for the definition of BlockLibrary instances. In Chapter 6, we defined the content of the language and all its components. For each of them, we detailed their meaning and semantics. In this document, we have defined the overall BlockLibrary semantics definition as being twofold: a) the structural semantics composed of the definition of the `BlockVariant` and `StructuralFeature` elements and their composition through the `BlockVariant` elements. The former elements structure is defined and constrained using OCL constraints (Section 6.2). The latter elements have been specified in Sections 6.4.4 and 6.4.5. Formal verifications are applied to them as depicted in Section 6.6 and successfully done as detailed in Section 7.7; b) the behavioral semantics of the BlockLibrary specification is given through the definition of the `BlockMode` elements containing functions definition as Hoare triples as depicted in Section 6.5, their formal verification is provided in Section 7.8. We thus have defined a precise syntax and a formal semantics to our BlockLibrary modeling notation.

"*The model contains software requirements and/or software architecture definition.*" It is the purpose of the BlockLibrary specification model to provide the specification for dataflow blocks through the elicitation of the structural variability and their corresponding semantics. These are related to software requirements as they are the source for automatic code generation of embedded systems design models. We detail in Section 9.2 the uses for BlockLibrary models as a source for automatic generation of requirement and software development artifacts.

"*The model is of a form and type that are used to direct analysis or behavioral evaluation as supported by the software development process or the software verification process.*" As we are working in the context of a tool qualification, reference to 'software' in this definition must be replaced by 'tool'. A BlockLibrary instance contains specification for the input elements of an ACG and the expected behavioral informations of the generated code, it is thus likely to be used as an information source for both tool development and tool verification process. We will detail these in Section 9.2.

9.1.2 *DO-333*: Formal methods

According to the *DO-333* document, the combination of formal modeling and formal analysis can be used to produce a formal method that might be used for the replacement of some of the traditionally conducted verification activities. As we presented previously, BlockLibrary model instances qualifies for being considered as a formal model.

The use of theorem provers have already been proven trustworthy and usable as a formal analysis mean. On the other side, works are in progress in order to bring such a confidence on the use of: SMT solvers by the formalisation of Alt-Ergo SMT solver for example; and on toolsets like Frama-C and Why3 in the U3CAT project[1]. Early works on the use of formal methods for the verification of embedded systems

[1] http://frama-c.com/u3cat/

like the one of Rushby [131] have built the basis of the methods adoption. Formal methods integration in modern certification/qualification activities are carried on and discussed in seminars grouping both industrials and academic domain experts like the *Dagstuhl* seminars[2].

Integrating formal methods in the development of embedded safety-critical systems may be used for leveraging verification activities and for the fulfilment of *DO-178C* certification objectives like: a) requirements correctness and consistency verification; b) source code review; c) test case replacement; d) low-level requirements verification; and e) system development process effort reduction (documentation generation).

As for *DO-178C* certification objectives, it is our belief that *DO-330* qualification activities might also be leveraged by the use of formal methods for: a) requirements completeness verification; b) requirement coverage verification; c) tool source code review; d) tool testing effort; and e) tool development process effort reduction.

We have shown the applicability of formal analysis on BlockLibrary instances models and the confidence they bring on the BlockLibrary specification. We will then show in the following sections how such a model instance can be used for the generation of certification data helping in the qualification and verification of an ACG development or the code it generates.

9.2 BlockLibrary use for the certification of an ACG tool

As a development tool, an ACG that generates safety-critical system code must be qualified. Qualification must be done according to the *DO-330* document. The BlockLibrary approach has been designed and developed with the objective of formally specifying ACG dataflow input language semantics. As such it must be used as an input language documentation and specification provider.

In the following we provide credible uses for a BlockLibrary specification in the context of the qualification of an ACG. We sum-up these uses in Figure 9.1. In this figure, the left part represents a BlockLibrary model from which automatic generations can be done (green arrows) and on the left side, the inner structure of the Projet-P ACG (details on the Projet-P ACG can be found in Section 2.3.2). In *DO-331*, "*potential uses*" for models are provided as: "*Providing unambiguous expression of requirements and architecture; Supporting the use of automated code generation; Supporting the use of automated test generation; Supporting the use of analysis tools for verification of requirements and architecture; Supporting the use of simulation for partial verification of requirements, architecture, and/or Executable Object Code*" [6]. We will illustrate these uses in the context of the Projet-P ACG.

9.2.1 Providing unambiguous expression of requirements and architecture

A correct BlockLibrary instance provides the specification for a set of blocks. For each block, it contains the complete and disjoint set of its `Configuration` elements. From a `Configuration`, we have shown in Chapter 7 that we can extract and verify the block semantics as a function contract (a Hoare triple).

In the context of a development tool qualification, it is mandatory to provide certification data among which are the Tool Operational Requirements (*TOR*). *TOR* for an ACG are supposed to provide for each possible element of the input language the expected output of the ACG tool. The semantics verification transformation provides this information but the generated output code is written using WhyML code which is not a classical embedded systems source code like C or Ada.

We propose to use the BlockLibrary semantics transformation as a basis for the development of an automated code generation providing for a specific `Configuration` element its expected embedded system generated source code.

In addition to the link between the BlockLibrary specification and the expected generated embedded code, a BlockLibrary `Configuration` element can be mapped to a concrete dataflow block with a specific configuration. Indeed, the `Configuration` element pre-conditions will provide the parameters

[2] http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=15182

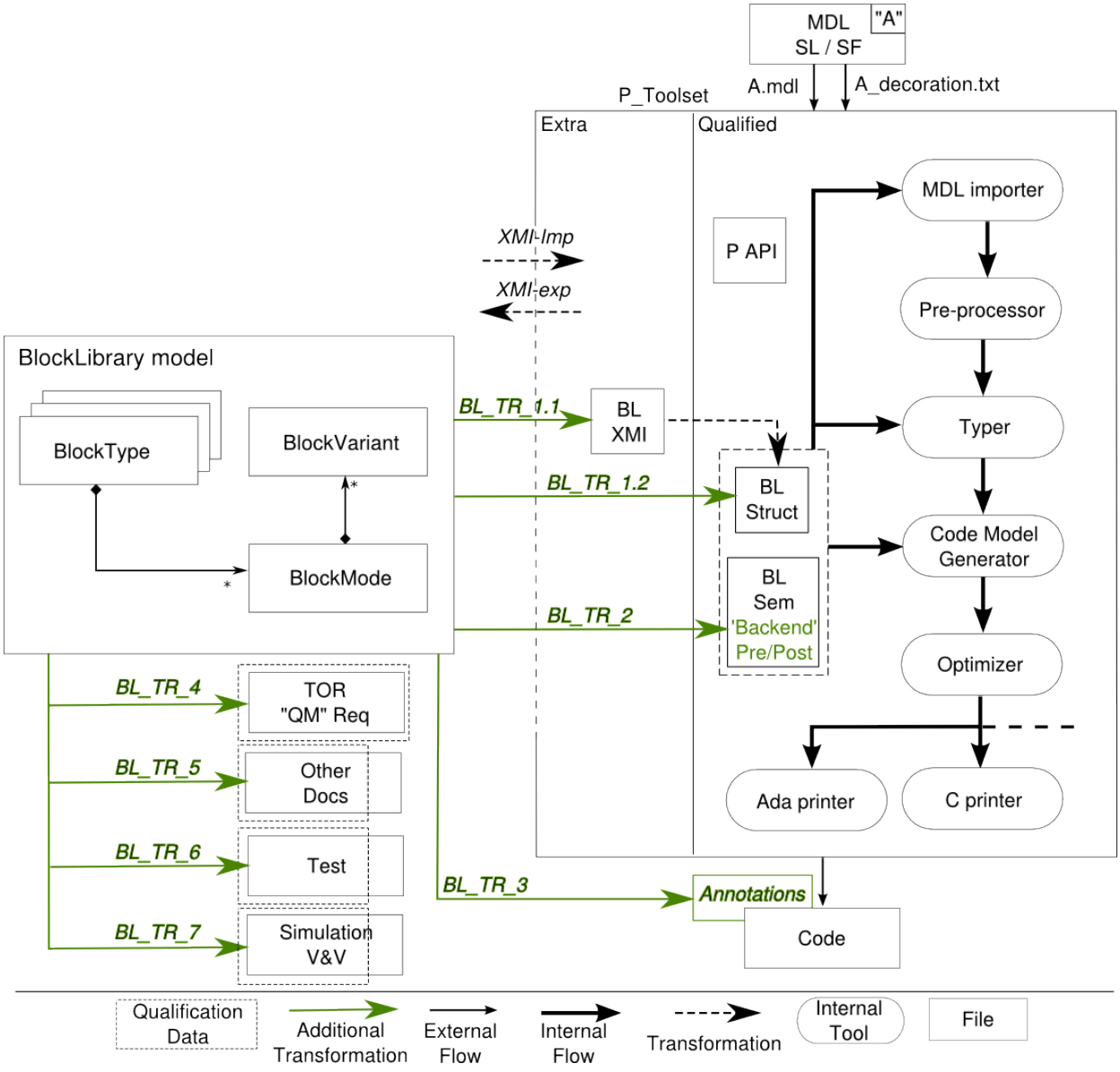


Figure 9.1: BLOCKLIBRARY use for ACG verification and development

values for this block configuration and thus can be mapped to specific block instances.

From the original `Configuration` semantics verification we obtain the assurance of the semantics specification correctness (Chapter 7). This assurance, given by the use of annotations must then be provided on the C code level. This can be done by translating the `Annotation` elements provided in the `Configuration` semantics definition to annotations on the generated source code (ACSL annotations on C code or SPARK annotations on ADA code). As annotation languages are defined using propositional logic semantics, the translation of the `Configuration` pre and post conditions is close to the one provided for the translation to WHYML. The annotated embedded system source code with annotations can then be verified using a deductive verification approach through tools like FRAMA-C for C source code verification or the SPARK EXAMINER for ADA source code.

Combination of block instance configuration and expected generated code is the exact content expected for the *TOR* of an ACG. We thus would be able to automatically produce such elements in various forms as for example: a model containing the informations that can be used in specific requirements management tools like *the qualifying machine*[3] or a textual documentation used as a reference documentation of

[3] http://www.open-do.org/projects/qualifying-machine/

the supported blocks for the ACG developers. These generation of documentations are referred to as the *BL_TR_4* and *BL_TR_5* transformations in Figure 9.1.

9.2.2 Supporting the use of automatically generated code

The Projet-P/Hi-MoCo and GeneAuto ACG research projects architecture are sensibly the same as the first one was strongly inspired by the second one. The ACG parses the model, it sequences and types the model (only in the GeneAuto ACG) or extracts these data from the model execution (for the QGen toolset), for each input block it matches the generated code according to the *code backends* containing the code to generate (the code model in Section 2.3.1) and finally it prints the code to the expected code formalism (C or Ada languages in our case). Some of these code generation steps rely on the information contained in the block library: parsing of the input model informations (parameters values of the blocks), matching the block instances with their allowed configurations in order to verify the block typing and generate the corresponding block code.

In our example ACG projects, each *code backend* is a file (a Java class in the GeneAuto ACG or an Ada module in the Projet-P ACG). In a *code backend*, the block configuration is checked and according to it, the corresponding code model is generated.

A BlockLibrary `Configuration` contains the expected generated code as a simple function code (a `BAL` operation specification). For each `Configuration`, a set of `StructuralFeature` are specified along with some `INVARIANT` annotations and `MODE_INVARIANT` additional configurations. These constraints along with the `StructuralFeature` definitions are the only information required in order to match the block instance to its expected configuration and thus its corresponding expected generated code contained in the `BlockMode` semantics specification(s). The `BAL` function body is the expected generated code for a block `Configuration` element. From this, it would be possible to generate automatically the corresponding code model structure that must be generated in the *code backend* (transformation *BL_TR_2.2* in Figure 9.1). The choice between *code backends* will be conditioned by the `Configuration Annotation` and the `StructuralFeature` definition (type and allowed values defined by their attached `INVARIANT Annotation`). Likewise, the automatic generation of annotations on the generated code as prescribed in Section 8.1 can be done automatically but this will be done according to the annotations written on the `BAL` code.

Formal verification of completeness and disjointness done on the `Configuration` elements for a block specification ensures that if the *code backend* is generated correctly, then the generated code will not contain any dead code and would be traceable to each `Configuration` element it was generated from. This will ease the verification of the *code backend* and the overall code generated providing additional automatic traceability information from *TOR* to developed (generated) ACG code.

Code backend generation is one example of ACG code that might be generated automatically from a BlockLibrary instance. We do not provide an implementation example for this generation but we are confident on its feasibility as the `BAL` is quite similar to software behavior models.

According to the structure of the ACG tool one might be able to generate different modules of the ACG used for example to check block typing; or generate ACG configuration files like the block library file (as provided in Figure 2.3) through the *BL_TR_1.1* or *BL_TR_1.2* transformations in Figure 9.1.

9.2.3 Supporting the use of analysis tools for verification of requirements and architecture.

As previously stated, it is the purpose of the BlockLibrary approach to ensure the correctness of the ACG input block specification. It is therefore a formal model for the definition of blocks specification and a formal verification of requirements.

9.2.4 Supporting the use of simulation for partial verification of requirements, architecture, and/or Executable Object Code.

Each BlockLibrary instance `Configuration` is composed of a set of `StructuralFeature` elements with a defined data type and dimensionality and a set of constraints on the `StructuralFeature` values.

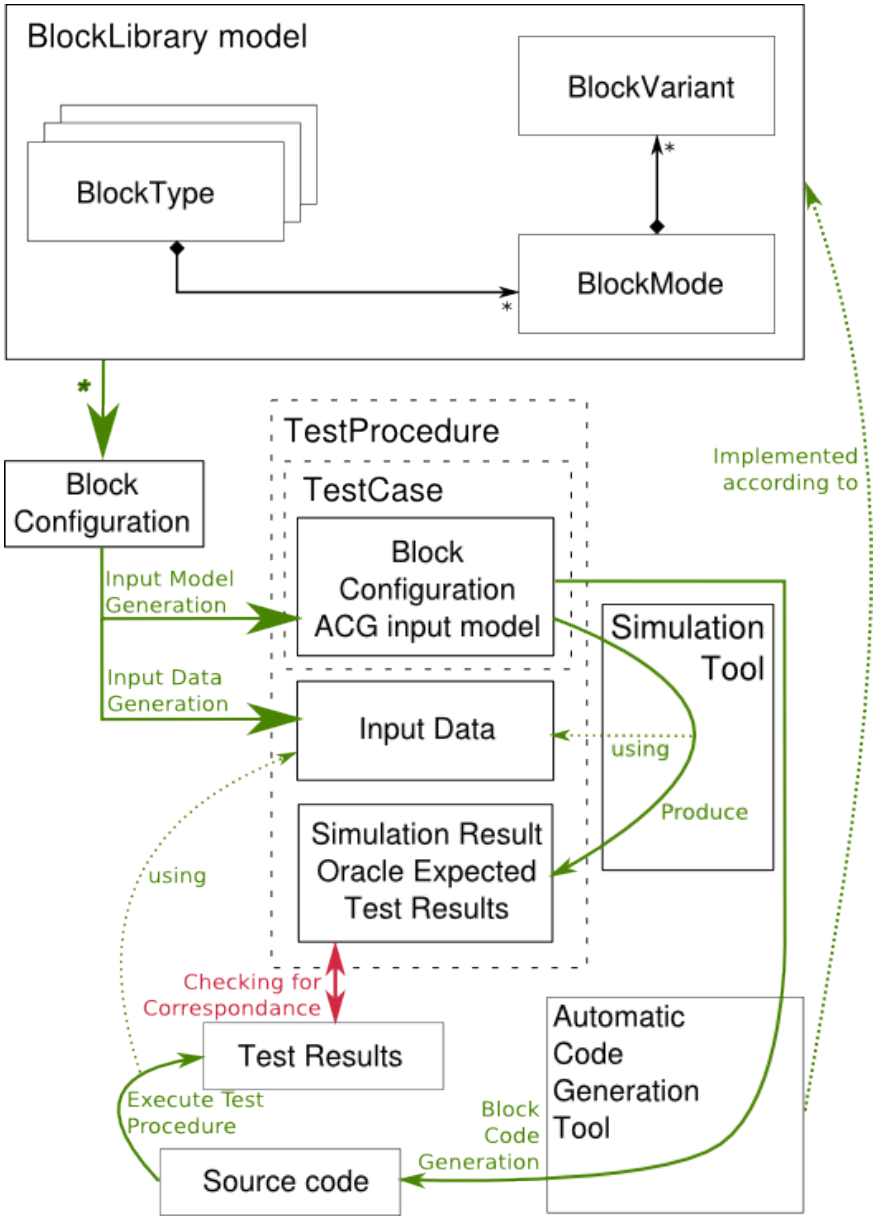


Figure 9.2: BLOCKLIBRARY use for test procedures generation and verification

In this section, we will show that these elements can be used in order to achieve automatic generation of test cases and for the verification of ACG development activities.

Each `Configuration` element can be considered as a complete source for test case generation. Indeed each `Configuration` element is a distinct configuration for a block instance. From each `Configuration`, according to the `StructuralFeature` specifications (data types and `INVARIANT Annotation`) and the `Configuration Annotation` (`MODE_INVARIANT`), we can extract a Constraint Satisfaction Problem (CSP). From this CSP, solvers such as MINIZINC[4] or CHOCO[5] can extract matching set of values for each `StructuralFeature` that can be used as the input values for a block execution.

According to a block `Configuration` element and its previously generated parameters and input values, we can generate an input model containing a block instance corresponding to the specified block `Configuration` element. This model can then be used: a) in a simulation tool (for example the one used for the usual development of input models) and then be used to simulate the block behavior; and b)

[4] http://www.minizinc.com
[5] http://choco-solver.org

in an ACG and then be used in order to generate code.

The simulation execution result can be considered as an oracle (expected result) for the test procedure and be compared with the result of the corresponding execution of the generated code (relying on the generated input data). This verification transformation is depicted in Figure 9.1 as the *BL_TR_7* transformation.

9.2.5 Supporting the use of automated test generation

Using the input model generated from each block `Configuration` as depicted in Figure 9.2, it is possible to trace the executions of the ACG for each generated `Configuration` test procedure of a block. Such test procedures will allow to verify the following properties: all the possible configurations for a block are handled in the code generation; each generation is functional for each configuration and; no dead code exists in the block-specific ACG code. This testing method will be part of the testing approach of transformation *BL_TR_6* of Figure 9.1.

9.3 Additional required verifications

ACG development is an activity where reliability is of primary importance. Development of the ACG by relying on automated techniques thus requires to ensure the correctness of the automated techniques.

Most of our proposed credible uses of the BlockLibrary instance helping the verification or development of an ACG are relying on automatic generations. These generations must themselves be verified in order to be able to benefit from the informations they provide for the gain of certification credits. In contrary to the ACG itself, these code generations are quite simple and purpose specific. Their complexity is limited and their formal verification can be expected to be simpler.

Formal and model-based partial verification of an ACG by relying on the formalisation of its input language provides additional confidence in the development of safety-critical embedded systems ACG. Their use in an industrial context is first conditioned to the acceptation of the approach by industrial practitioners and must then be prepared for qualification. To our knowledge, both conditions can be fulfilled as industrial practitioners are more and more accepting and using model-based approaches and to some extends formal methods; and certification bodies are expected to accept well-founded and justified tooled approaches.

10 Conclusion & Future work

In this PhD, we targeted a formal and tooled approach for the specification of highly variable languages structure and semantics. We experimented this approach for block diagrams, a very common DSML for the design of safety critical systems. We have given formal means to ensure both structural and semantics specification correctness of block diagrams, and we have shown the usefullness of such specification for automatic code generators (ACG) development activities like specification, verification, implementation and qualification activities. Our experiments were conducted on Simulink, a highly variable dataflow language, that is widely used in concrete, safety-critical and real-size industrial applications. Specification of these language components is complex in the sense that their semantics and structure may vary according to their parametrisation and the context they are used on. Every aspect of our proposal was implemented in the Eclipse framework and is freely available in open source[1].

In this final chapter, we first sumarize the key elements from our work and review the fulfillment of the research objectives detailed in Chapter 1. We detail in Section 10.2 its concrete results. Section 10.3 gathers the future works regarding both the BlockLibrary approach and wider scale research directions.

10.1 Research objectives fulfillment

10.1.1 Research objective 1: Formal specification and verification of highly variable languages

In order to achieve this objective, we have shown how a combination of modeling and formal approaches can be used for the definition of a dedicated specification language: the BlockLibrary specification language. By relying on SPLE methodologies, concepts and techniques, the dedicated specification language approach provides a more accurate handling of the specification complexity than state of the art approaches that are either too expressive or targets a too wide application domain like the UML or SPLE.

Relying on MDE technologies eases the specification language definition, its formalisation and the development of mandatory tools for editing and verifying specifications. The MDE approach for the definition of the BlockLibrary language grants access to many technologies that ease the manipulation and transformation of models. We have used these ones in order to provide a model-based transformation framework for the verification of the correctness of the block library specification. Correctness of a BlockLibrary instance is based on three distinct and complementary criteria:

a **Structural correctness** provided by the MDE formalisation of the specification language. According to the specification language requirements expressed using an Ecore metamodel and OCL constraints, the appropriate tools have been developed in order to ensure the structural consistency of

[1] http://blocklibrary.enseeiht.fr/html/

the language instances with respect to this structural specification.

b **Variability correctness** provided by the model transformation of BlockLibrary instances to the Why3 platform. This transformation provides a translational formal semantics to the specification language by relying on the formal semantics provided by the Why3 toolset languages. We expressed variability correctness with two criterion: completeness and disjointness. Based on the previous formalisation of the blocks specification we define these criterion using Why3:

- Completeness: ensure that the set of all expressed structural block configuration variants are exhaustive relatively to the allowed blocks structural features definitions and their allowed combinations.
- Disjointness: ensure that each pair of expressed structural block configuration is disjoint and thus does not specify twice differently the same configuration.

c **Semantics specification correctness** ensured by the model transformation of BlockLibrary configurations to their representation as a WhyML function with function contract expressed using annotations in Why.

For each criteria a verification technique is used. Structural correctness is ensured by relying on the usual MDE conformance mechanism and OCL constraints verification. Variability and semantics specification correctness are ensured by relying on the verification capabilities of the Why3 toolset. Variability criterion are expressed as goals to be proven based on the formalisation of the block specification expressed using the Why language; semantics specification is also expressed using Why. Both verification are then tackled by relying on automated SMT solvers or manual proof assistants to formally assess the verification goals or the function contract correctness.

10.1.2 Research objective 2: Uses of highly variable language formal specification for automated generated code verification

As dataflow model semantics is mostly contained in the blocks, their formal specifications may be used as a formal reference for the verification of the code generated from the block instances.

We detailed an approach for the automatic generation of annotations on the code produced by an ACG. These annotations will contain each block instance specification embedded in the block-specific generated code. We have shown the feasibility of such an automated formal verification by relying on language-specific source code verification tools like Frama-C for the verification of C code or Spark for Ada code. This approach relies on the translation validation proposal [125] as advocated by Pnueli and applied to the verification of automated code generators. This formal verification of generated code provides a block-level low level requirements (*LLR*) verification of the generated code with respect to the language semantics.

This *LLR* verification on the code has then been shown useful for the verification of high level requirements (*HLR*). In our setting, *HLR* are expressed on the ACG input model using synchronous observers (*SO*). These ones being expressed using the same language as the observed model, can benefit from their formal specification. Relying on the axiomatic semantics of the blocks, we provided a methodology for the conversion of *SO* as logical properties embeddable as annotations in the generated code.

10.1.3 Research objective 3: Uses of highly variable language formal specification for ACG qualification

We finally have shown how formal block specifications and code generation verification mechanism could be used for leveraging tool qualification activities. From a correct and verified BlockLibrary instance, we have demonstrated the possibility for the automated generation of safe and formal artifacts used in ACG qualification activities such as elements of the ACG requirements and architecture or ACG development artifacts such as block specific code generation modules; and, in the end, ACG test cases generation according to all possible block configurations.

10.2 Concrete productions

The BlockLibrary metamodel, editors and tools have been developed using the Eclipse platform. The complete set of tools and their associated source code are freely available on our project website[2] and can be installed from our update site[3]. Because of the variety of Eclipse platforms configurations, it was not possible for us to test the deployment of the plugins on every Eclipse release. We thus provide the Eclipse configuration on which the toolset installation has been tested and all the instructions on how to install it.

The Eclipse update site provides the following features including the plugins developed during this PhD.

- **BlockLibrary editors**: contains the BlockLibrary text and hierarchical editors for BlockLibrary models. It also contains an Eclipse view (the BlockLibrary Signature view) providing a structured dynamic overview of the `Signature` content (`BlockVariant`, `BlockMode`, `StructuralFeature`) during BlockLibrary edition.
- **BlockLibrary Examples**: contains a set of BlockLibrary instance models in textual form for a set of blocks including the ones used in this PhD.
- **BlockLibrary model2text generators**: contains the model transformations presented in this PhD including the BlockLibrary to Why3 transformations and the BlockLibrary to DOT format transformation that provides a graphical view of models.

These productions are provided including the source code and are licensed according to the terms of the GPL V3 license provided with the plugins.

10.3 Future research directions

10.3.1 BlockLibrary-related activities

Throughout our PhD work, the BlockLibrary DSML application domain has been restricted as choices were made in order to be able to reach the final stage of the approach and to experiment on the various concrete applications enabled by this approach. The actual work done on the BlockLibrary DSML must go on in order to ensure its scaling to full size block libraries specification and thus the verification of full size industrial use cases. The work we have done currently was applied on real size blocks. The verification of each block is independent, we are confident that their verification cost will be comparable to the one done on the *Delay* and *MinMax* blocks.

We propose here some development lines for the BlockLibrary specification approach improvement.

BlockLibrary edition capabilities

BlockLibrary edition is done by relying on a textual syntax. While this syntax is easy to read, it is possible that on a large scale such a representation will not be easy to maintain.

As BlockLibrary can be seen as a specification of complex block structural features on which it is possible to express variability information enriched with block semantics informations, we propose to define a more sophisticated editor mixing both graphical variability management and textual specification.

Graphical variability management will be handled as a classical feature model editor with the possibility to express complex set of block features and relations between these sets. Each feature of our model, will be textually editable in order to provide its containing `StructuralFeature` definitions and, for each `StructuralFeature` its additional typing and constraints informations. At this point of the specification, the specifier can benefit from the automatic verification of both completeness and disjointness properties and gain early feedbacks on the structural correctness of the specification.

[2] http://blocklibrary.enseeiht.fr/html
[3] http://dieumegard.perso.enseeiht.fr/plugins/blocklibrary/

From the feature model, The extraction of all the products will provide all the products for which a semantics description must be provided. For each product (or a subset of them if semantics definitions can apply to multiple ones), the block specifier will have to write the semantics specifications.

We think that such an approach will provide a clear methodology for the definition of the variability model and a better handling of the block specification complexity relying on the graphical overview of the block structural variability and the tight integration ot the verification results. In addition to these, the use of a feature model in the early block specification stages will allow to rely on feature modeling approaches and verification capabilities as we detailed in Section 6.3.

Enhance coverage of dataflow specification

We decided to limit the amount of informations managed in a BlockLibrary instance as we wanted to ensure the applicability of our specification approach.

We limited the data types handling to simple ones and avoided the use of most structured data types. We thus removed support of complex numbers, floating points numbers and buses (structured data types). In the specification of `StructuralFeature`, we decided to limit the number of allowed data types to 1. This was a huge simplification that enforces us to decompose data types allowance of `StructuralFeature` into multiple definitions of the same `StructuralFeature` held in different variability structure (`BlockVariant`).

While these limitations were interesting and allowed us to show the applicability of the BlockLibrary approach, it will be mandatory to handle every capabilities of a block in order to ensure a large handling of all possible block specification:

- **Allowed data types**. The addition of the missing data types (complex and floating point numbers and buses) in the BlockLibrary language does not implies deep modification but the difficulty lies in the BlockLibrary to Why3 translation as: on the one hand it is mandatory to provide the required data types definitions and axioms (that are mostly provided in the Why3 standard library) but also the operations (operator and manipulation operations) definitions and the axioms they rely on; and on the other hand, it makes more complex the translation development itself.

- **Multiply typed features**. In the BlockLibrary to Why3 translation, each `StructuralFeature` definition is translated as a type declaration and its invariants as predicates on its values. If a `StructuralFeature` is specified to have multiple potential data types, it implies that different type declarations would be generated for each data type. The result of this generation would be to produce for each `StructuralFeature` invariant a different predicate for each of its declared type. Complexity arise when any other `BlockVariant` extends a `BlockVariant` in which such a `StructuralFeature` is defined, in this case each `StructuralFeature` data type would be considered as a variant of the `BlockVariant` (it is exponential in the number of generated predicates if multiple `StructuralFeature` have multiple data types in the same `BlockVariant`). It will also be mandatory to consider only the allowed combinations of data types according to the `INVARIANT` and `MODE_INVARIANT` expressed on the specification. In the end, regarding the semantics definition, verification must be handled in order to ensure only allowed and correctly typed semantics specification and thus filter on the available `StructuralFeature` types combinations.

  As the purpose of the BlockLibrary is to ensure specification correctness, such a wide freedom of `StructuralFeature` data types specification must not be allowed and additional rules might be necessary in order to ensure : a) to manage the data types compositions, (for example by allowing `StructuralFeature` definitions with multiple data types only if the data types are from the same family (integers, floats, ...)); b) ensure typing verification in `Annotation` expressions for early feedback to the user; c) for each data type family ensure the complete definition and axiomatization of all the possible cross-types operations (for example, the division of an Int16 by an Int32).

Languages transformation verification

We detailed in this PhD the transformation of the BlockLibrary language and its embedded OCL and BAL languages to the Why and WhyML languages structures. We provided the translation rules and we implemented them.

The verification of this implementation has been made by relying only on testing. While we have a good confidence on the translation correctness, its verification cannot be completely trusted as it is not exhaustive.

Formal verification approaches applied to model transformations must be used in order to ensure this translation correctness. A huge amount of work has been done in the past year on the verification of model transformations [32]. We also provided an approach on the verification of transformation [150] that may be applied for the verification of our translations.

10.3.2 Overall approach future works

This PhD work focused on the formal specification of dataflow languages, its verification and application in embedded critical systems development. From the work done in this PhD, we identified several ways forward in the domain of languages variability specification, software verification and MDE formalisation that will be detailed further here.

Languages variability specification

We produced our BlockLibrary metamodel from the analysis of the dataflow languages domain and by relying on a SPLE methodology in order to structure the metamodel. This analysis of our specific domain variability led us to the distinction of two sources of variability: structural and semantics.

It is natural for system design languages to expose a certain degree of variability. Indeed, design languages requires to provide a certain abstraction and expressiveness level in order to be usable in real size industrial developments. High level design languages such as UML or SysML are examples of these. The formalisation of such high level languages is complex and needs to be tackled in the same way we have conducted in this PhD for the Simulink block libraries. For example, state machines have been the subject of this kind of studies at the implementation level in the Polyglot [14] project without relying on SPLE technologies. The reification of that work would be a good starting point for a new case study.

To illustrate how we could reify our approach and apply it to other use cases, we abstract the BlockLibrary specification metamodel (Figure 10.1). `BlockVariant` and `BlockMode` are respectively replaced by StructuralSpecificationElement and SemanticsSpecificationElement; `BlockType` is replaced by its generic pendent: DomainModel. This generic language specification metamodel still relies on a SPLE approach and thus provides the same capabilities as the BlockLibrary one in order to formally specify language variability.

This metamodel must be specialised for each language to be specified; from such a specialisation of the metamodel, the verification approaches provided in this PhD thesis shall be applicable and reusable and provide formal confidence in the language specification. According to the language to be specified (UML, StateMachines, ...), its application domain (network calculus, system/architecture design, hybrid systems design, ...) or the primitive elements to be used (specific operations, data types, ...) and the structural features on which to use them (ports, parameters, interfaces, ...) a support for additional formal language (TLA, Fiacre, ...) might have to be provided, this would allow to use adapted formal language leading to more convenient specification writing.

Embedded systems automatic verification

We have provided in Chapter 8 an approach to the verification of *HLR* on generated code from their expression using synchronous observers in an ACG input language. We have provided and demonstrated their verifiability on concrete but rather small examples.

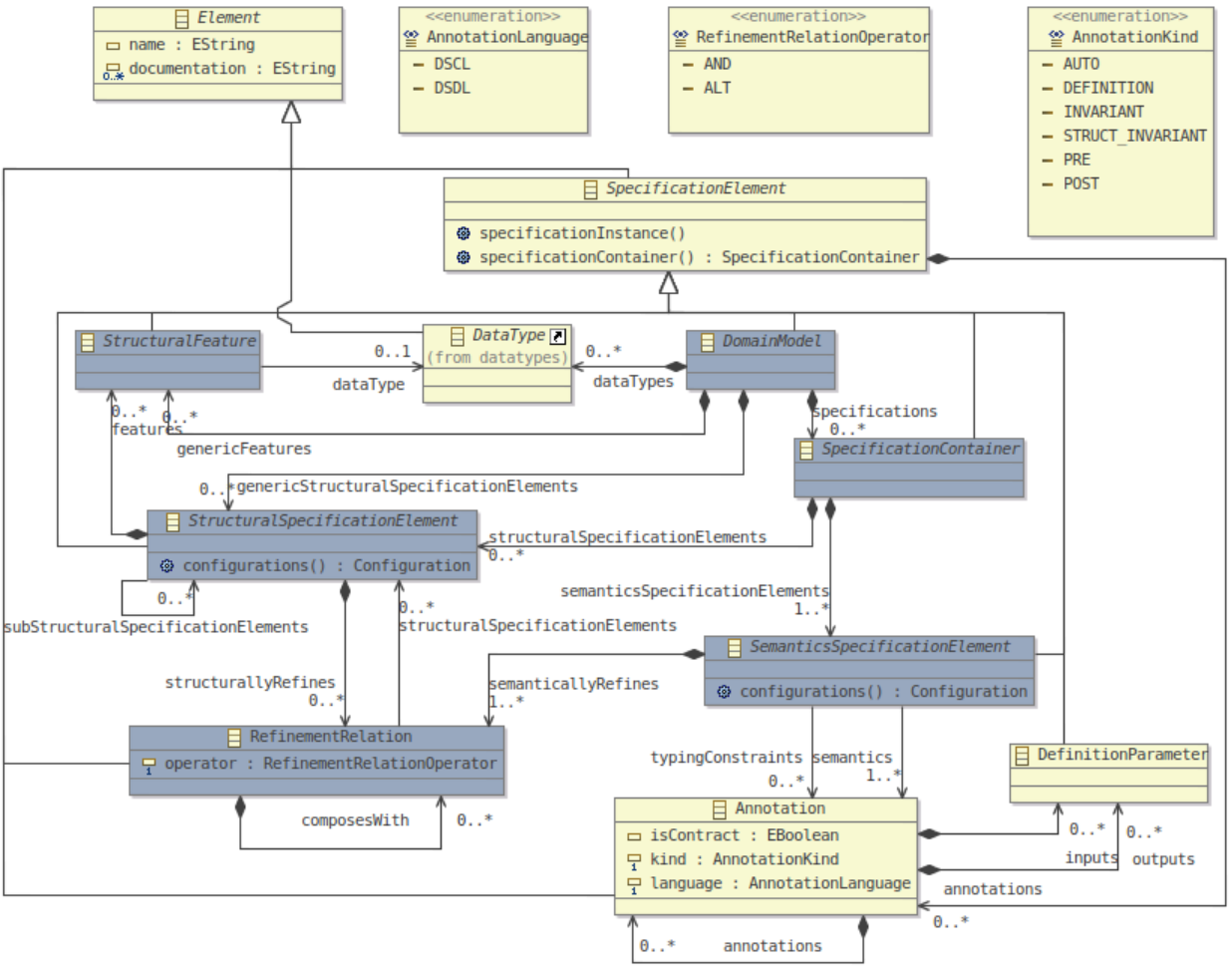


Figure 10.1: Generic language variability specification metamodel

This work must go further on in order to provide a more complete set of formal blocks specification using the BlockLibrary language. The expression of more complex high level properties must be assessed on more complex systems based on the newly provided block specification in order to ensure the scalability of the automatic verification approach relying on code annotation and static analysis.

Concrete industrial embedded critical software applications are not only designed using dataflow languages such as Simulink, they most of the time rely on a combinations of dataflow and state chart allowing to design not only the function but also the various modes in which the software must operate. Our specification and annotation translation approach should be extended to state chart models and combinations of data and state flow for the verification of their corresponding generated code.

Industrial certification/qualification

We detailed in Chapter 9 a set of applications for the BlockLibrary specification language (requirements specification, ACG components generation, generated code verification tools development, simulation for verification, automatic test generation). We aim in a close future to be able to work on their concrete applicability in an industrial certification and/or qualification context in close relation with both industrial users and certification authorities. This work would contribute to the diffusion of the formal methods and modeling research approaches to the industrial users.

The content of the BlockLibrary being the most important part of the ACG (the *TOR* and their formal verification) we envision to extend these experiment and to rely on the block specification for the automatic generation of the code generator itself.

Appendices

A

Complete block specifications

A.1 *Delay* block specifiation

```
library DelayComplet {
  // Primitive types
  type boolean TBoolean
  type realInt TUInt8 of 8 bits
  type realInt TUInt16 of 16 bits
  type realInt TUInt32 of 32 bits
  type signed realInt TInt8 of 8 bits
  type signed realInt TInt16 of 16 bits
  type signed realInt TInt32 of 32 bits
  type realDouble TDouble
  type realSingle TSingle
  type string TString
  // Arrays
  type array TArrayUInt8 of TUInt8 [0]
  type array TArrayDouble of TDouble [0]
  type array TMatrixDouble of TDouble [0,0]
  type array TListMatrixDouble of TDouble [0,0,0]
  // Delay Enumerations
  type enum TResetAlgo {NONE, RISING, FALLING, EITHER, LEVEL, LEVEL_HOLD}

  blocktype Delay {
    variant ResetParam {
      parameter Reset_Algo : TResetAlgo default !!TResetAlgo::NONE
    }
    variant InputScalar {
      modeinvariant ocl {
        Input.value.isScalar()
      }
      modeinvariant ocl {
        Output.value.isScalar()
      }
      in data Input : TDouble
      out data Output : TDouble
    }
    variant InputVector {
      modeinvariant ocl {
        Input.value.isVector()
      }
      modeinvariant ocl {
        Output.value.isVector()
      }
      in data Input : TArrayDouble
      out data Output : TArrayDouble
    }
    variant InputMatrix {
```

```
      modeinvariant ocl {
47      Input.value.isMatrix()
      }
      modeinvariant ocl {
        Output.value.isMatrix()
      }
52    in data Input : TMatrixDouble
      out data Output : TMatrixDouble
    }
    variant InternalICScalar {
      modeinvariant ocl { IC.value.isScalar() }
57    parameter IC : TDouble default 0.0
    }
    variant ExternalICScalar {
      modeinvariant ocl { IC.value.isScalar() }
      in data IC : TDouble
62  }
    variant InternalICVector {
      modeinvariant ocl { IC.value.isVector() }
      parameter IC : TArrayDouble
    }
67  variant ExternalICVector {
      modeinvariant ocl { IC.value.isVector() }
      in data IC : TArrayDouble
    }
    variant InternalICMatrix {
72    modeinvariant ocl { IC.value.isMatrix() }
      parameter IC : TMatrixDouble
    }
    variant ExternalICMatrix {
      modeinvariant ocl { IC.value.isMatrix() }
77    in data IC : TMatrixDouble
    }
    variant InternalICListMatrix {
      modeinvariant ocl { IC.value->forAll(e| e.isMatrix()) }
      parameter IC : TListMatrixDouble
82  }
    variant ExternalICListMatrix {
      modeinvariant ocl { IC.value->forAll(e| e.isMatrix()) }
      in data IC : TListMatrixDouble
    }
87  variant InternalDelay {
      parameter Delay : TInt32 {
        invariant ocl { Delay.value > 0 }
      }
    }
92  variant ExternalDelay {
      in data Delay : TInt32 {
        invariant ocl { Delay.value > 0 }
      }
    }
97  variant ListDelay_ScalarInput extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
      InputScalar,
      oneof (InternalICVector, ExternalICVector)
102 ) {
      modeinvariant ocl {
        Delay.value > 1
      }
      modeinvariant ocl {
107     IC.value.size() = Delay.value
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {0}
112   }
    }
    variant ListDelay_VectorInput extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
117   InputVector,
      oneof (InternalICMatrix, ExternalICMatrix)
    ) {
```

```
      modeinvariant ocl {
        Delay.value > 1
      }
      modeinvariant ocl {
        IC.value.size() = Delay.value
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {0}
      }
    }
    variant ListDelay_MatrixInput extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
      InputMatrix,
      oneof (InternalICListMatrix, ExternalICListMatrix)
    ) {
      modeinvariant ocl {
        Delay.value > 1
      }
      modeinvariant ocl {
        IC.value.size() = Delay.value
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {0}
      }
    }
    variant SimpleDelay_Scalar extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
      InputScalar,
      oneof (InternalICScalar, ExternalICScalar)
    ) {
      modeinvariant ocl {
        Delay.value = 1
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {1}
      }
    }
    variant SimpleDelay_Vector extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
      InputVector,
      oneof (InternalICVector, ExternalICVector)
    ) {
      modeinvariant ocl {
        Delay.value = 1
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {1}
      }
    }
    variant SimpleDelay_Matrix extends allof (
      ResetParam,
      oneof (InternalDelay, ExternalDelay),
      InputMatrix,
      oneof (InternalICMatrix, ExternalICMatrix)
    ) {
      modeinvariant ocl {
        Delay.value = 1
      }
      memory Mem {
        datatype auto ocl {Input.value}
        length auto ocl {1}
      }
    }
    variant ResetInput {
      in data Reset : TDouble
      memory MemReset {
        datatype auto ocl {Reset.value}
```

```
      length auto ocl {1}
    }
  }
  mode DelayMode_Simple implements oneof (SimpleDelay_Scalar,
                                          SimpleDelay_Vector,
                                          SimpleDelay_Matrix) {
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::NONE
    }
    definition bal = init_Delay_Simple {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Delay_Simple {
      postcondition ocl { Output.value = Mem.value }
      Output.value = Mem.value;
    }
    definition bal = update_Delay_Simple {
      postcondition ocl {
        Mem.value = Input.value
      }
      Mem.value = Input.value;
    }
    init init_Delay_Simple
    compute compute_Delay_Simple
    update update_Delay_Simple
  }
  mode DelayMode_List implements oneof (ListDelay_ScalarInput,
                                        ListDelay_VectorInput,
                                        ListDelay_MatrixInput) {
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::NONE
    }
    definition bal = init_Delay_List {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Delay_List {
      postcondition ocl { Output.value = Mem.value->first() }
      Output.value = Mem.value[0];
    }
    definition bal = update_Delay_List {
      postcondition ocl {
        Mem.value = Mem.value->excluding(
          Mem.value->first()
        )->append(Input.value)
      }
      var iter = 0;
      while (iter < (Delay.value - 1)){
        Mem.value[iter] = Mem.value[iter + 1];
        iter = iter + 1;
      }
      Mem.value[Delay.value - 1] = Input.value;
    }
    init init_Delay_List
    compute compute_Delay_List
    update update_Delay_List
  }
  mode DelayReset_Simple_RISING implements allof (
    ResetInput,
    oneof (SimpleDelay_Scalar, SimpleDelay_Vector, SimpleDelay_Matrix)
  ){
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::RISING
    }
    definition bal = init_Resettable_Simple_RISING {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Resettable_Simple_RISING {
      postcondition ocl {
        Output.value = Mem.value or Output.value = IC.value
      }
      postcondition ocl {
```

```
        (MemReset.value <= 0.0 and 0.0 < Reset.value) implies
        Output.value = IC.value
      }
      postcondition ocl {
        (not (MemReset.value <= 0.0 and 0.0 < Reset.value)) implies
        Output.value = Mem.value
      }
      if (MemReset.value <= 0.0 && 0.0 < Reset.value) {
        Output.value = IC.value;
      } else {
        Output.value = Mem.value;
      }
    }
    definition bal = update_Resettable_Simple_RISING {
      postcondition ocl {
        Mem.value = Input.value
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      Mem.value = Input.value;
      MemReset.value = Reset.value;
    }
    init init_Resettable_Simple_RISING
    compute compute_Resettable_Simple_RISING
    update update_Resettable_Simple_RISING
  }
  mode DelayReset_Simple_FALLING implements allof (
    ResetInput,
    oneof (SimpleDelay_Scalar, SimpleDelay_Vector, SimpleDelay_Matrix)
  ){
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::FALLING
    }
    definition bal = init_Resettable_Simple_FALLING {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Resettable_Simple_FALLING {
      postcondition ocl {
        Output.value = Mem.value or Output.value = IC.value
      }
      postcondition ocl {
        (MemReset.value >= 0.0 and 0.0 > Reset.value) implies
        Output.value = IC.value
      }
      postcondition ocl {
        (not (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
        Output.value = Mem.value
      }
      if (MemReset.value >= 0.0 && 0.0 > Reset.value) {
        Output.value = IC.value;
      } else {
        Output.value = Mem.value;
      }
    }
    definition bal = update_Resettable_Simple_FALLING {
      postcondition ocl {
        Mem.value = Input.value
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      Mem.value = Input.value;
      MemReset.value = Reset.value;
    }
    init init_Resettable_Simple_FALLING
    compute compute_Resettable_Simple_FALLING
    update update_Resettable_Simple_FALLING
  }
  mode DelayReset_Simple_EITHER implements allof (
    ResetInput,
    oneof (SimpleDelay_Scalar, SimpleDelay_Vector, SimpleDelay_Matrix)
  ){
```

```
342       modeinvariant ocl {
            Reset_Algo.value = !!TResetAlgo::EITHER
          }
          definition bal = init_Resettable_Simple_EITHER {
            postcondition ocl { Mem.value = IC.value }
347         Mem.value = IC.value;
          }
          definition bal = compute_Resettable_Simple_EITHER {
            postcondition ocl {
              Output.value = Mem.value or Output.value = IC.value
352         }
            postcondition ocl {
              ((MemReset.value <= 0.0 and 0.0 < Reset.value) or
               (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
              Output.value = IC.value
357         }
            postcondition ocl {
              ((not (MemReset.value <= 0.0 and 0.0 < Reset.value)) or
               (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
              Output.value = Mem.value
362         }
            if (MemReset.value <= 0.0 && 0.0 < Reset.value) {
              Output.value = IC.value;
            } else {
              Output.value = Mem.value;
367         }
          }
          definition bal = update_Resettable_Simple_EITHER {
            postcondition ocl {
              Mem.value = Input.value
372         }
            postcondition ocl {
              MemReset.value = Reset.value
            }
            Mem.value = Input.value;
377         MemReset.value = Reset.value;
          }
          init init_Resettable_Simple_EITHER
          compute compute_Resettable_Simple_EITHER
          update update_Resettable_Simple_EITHER
382     }
        mode DelayReset_Simple_LEVEL implements allof (
          ResetInput,
          oneof (SimpleDelay_Scalar, SimpleDelay_Vector, SimpleDelay_Matrix)
        ){
387       modeinvariant ocl {
            Reset_Algo.value = !!TResetAlgo::LEVEL
          }
          definition bal = init_Resettable_Simple_LEVEL {
            postcondition ocl { Mem.value = IC.value }
392         Mem.value = IC.value;
          }
          definition bal = compute_Resettable_Simple_LEVEL {
            postcondition ocl {
              Output.value = Mem.value or Output.value = IC.value
397         }
            postcondition ocl {
              ((Reset.value <> 0.0) or
               (Reset.value = 0.0 and MemReset.value <> 0.0)) implies
              Output.value = IC.value
402         }
            postcondition ocl {
              (not ((Reset.value <> 0.0) or
                    (Reset.value = 0.0 and MemReset.value <> 0.0))) implies
              Output.value = Mem.value
407         }
            if ((Reset.value != 0.0) || ((Reset.value == 0.0) &&
                (MemReset.value != 0.0))) {
              Output.value = IC.value;
            } else {
412           Output.value = Mem.value;
            }
          }
          definition bal = update_Resettable_Simple_LEVEL {
```

```
        postcondition ocl {
          Mem.value = Input.value
        }
        postcondition ocl {
          MemReset.value = Reset.value
        }
        Mem.value = Input.value;
        MemReset.value = Reset.value;
      }
      init init_Resettable_Simple_LEVEL
      compute compute_Resettable_Simple_LEVEL
      update update_Resettable_Simple_LEVEL
    }
    mode DelayReset_Simple_LEVEL_HOLD implements allof (
      ResetInput,
      oneof (SimpleDelay_Scalar, SimpleDelay_Vector, SimpleDelay_Matrix)
    ){
      modeinvariant ocl {
        Reset_Algo.value = !!TResetAlgo::LEVEL_HOLD
      }
      definition bal = init_Resettable_Simple_LEVEL_HOLD {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Resettable_Simple_LEVEL_HOLD {
        postcondition ocl {
          Output.value = Mem.value or Output.value = IC.value
        }
        postcondition ocl {
          (Reset.value <> 0.0) implies Output.value = IC.value
        }
        postcondition ocl {
          (Reset.value = 0.0) implies Output.value = Mem.value
        }
        if (Reset.value != 0.0) {
          Output.value = IC.value;
        } else {
          Output.value = Mem.value;
        }
      }
      definition bal = update_Resettable_Simple_LEVEL_HOLD {
        postcondition ocl {
          Mem.value = Input.value
        }
        postcondition ocl {
          MemReset.value = Reset.value
        }
        Mem.value = Input.value;
        MemReset.value = Reset.value;
      }
      init init_Resettable_Simple_LEVEL_HOLD
      compute compute_Resettable_Simple_LEVEL_HOLD
      update update_Resettable_Simple_LEVEL_HOLD
    }
    mode DelayReset_List_RISING implements allof (
      ResetInput,
      oneof (ListDelay_ScalarInput, ListDelay_VectorInput, ListDelay_MatrixInput)
    ){
      modeinvariant ocl {
        Reset_Algo.value = !!TResetAlgo::RISING
      }
      definition bal = init_Resettable_List_RISING {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Resettable_List_RISING {
        postcondition ocl {
          Output.value = Mem.value->first() or Output.value = IC.value->first()
        }
        postcondition ocl {
          (MemReset.value <= 0.0 and 0.0 < Reset.value) implies
          Output.value = IC.value->first()
        }
        postcondition ocl {
```

```
        (not (MemReset.value <= 0.0 and 0.0 < Reset.value)) implies
        Output.value = Mem.value->first()
      }
      if (MemReset.value <= 0.0 && 0.0 < Reset.value) {
        Output.value = IC.value[0];
      } else {
        Output.value = Mem.value[0];
      }
    }
    definition bal = update_Resettable_List_RISING {
      postcondition ocl {
        Mem.value = Mem.value->excluding(
          Mem.value->first()
        )->append(Input.value)
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      var iter = 0;
      while (iter < (Delay.value - 1)){
        Mem.value[iter] = Mem.value[iter + 1];
        iter = iter + 1;
      }
      Mem.value[Delay.value - 1] = Input.value;
      MemReset.value = Reset.value;
    }
    init init_Resettable_List_RISING
    compute compute_Resettable_List_RISING
    update update_Resettable_List_RISING
  }
  mode DelayReset_List_FALLING implements allof (
    ResetInput,
    oneof (ListDelay_ScalarInput, ListDelay_VectorInput, ListDelay_MatrixInput)
  ){
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::FALLING
    }
    definition bal = init_Resettable_List_FALLING {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Resettable_List_FALLING {
      postcondition ocl {
        Output.value = Mem.value->first() or Output.value = IC.value->first()
      }
      postcondition ocl {
        (MemReset.value >= 0.0 and 0.0 > Reset.value) implies
        Output.value = IC.value->first()
      }
      postcondition ocl {
        (not (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
        Output.value = Mem.value->first()
      }
      if (MemReset.value >= 0.0 && 0.0 > Reset.value) {
        Output.value = IC.value[0];
      } else {
        Output.value = Mem.value[0];
      }
    }
    definition bal = update_Resettable_List_FALLING {
      postcondition ocl {
        Mem.value = Mem.value->excluding(
          Mem.value->first()
        )->append(Input.value)
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      var iter = 0;
      while (iter < (Delay.value - 1)){
        Mem.value[iter] = Mem.value[iter + 1];
        iter = iter + 1;
      }
      Mem.value[Delay.value - 1] = Input.value;
```

```
        MemReset.value = Reset.value;
      }
      init init_Resettable_List_FALLING
      compute compute_Resettable_List_FALLING
      update update_Resettable_List_FALLING
    }
    mode DelayReset_List_EITHER implements allof (
      ResetInput,
      oneof (ListDelay_ScalarInput, ListDelay_VectorInput, ListDelay_MatrixInput)
    ){
      modeinvariant ocl {
        Reset_Algo.value = !!TResetAlgo::EITHER
      }
      definition bal = init_Resettable_List_EITHER {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Resettable_List_EITHER {
        postcondition ocl {
          Output.value = Mem.value->first() or Output.value = IC.value->first()
        }
        postcondition ocl {
          ((MemReset.value <= 0.0 and 0.0 < Reset.value) or
           (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
          Output.value = IC.value->first()
        }
        postcondition ocl {
          ((not (MemReset.value <= 0.0 and 0.0 < Reset.value)) or
           (MemReset.value >= 0.0 and 0.0 > Reset.value)) implies
          Output.value = Mem.value->first()
        }
        if (MemReset.value >= 0.0 && 0.0 > Reset.value) {
          Output.value = IC.value[0];
        } else {
          Output.value = Mem.value[0];
        }
      }
      definition bal = update_Resettable_List_EITHER {
        postcondition ocl {
          Mem.value = Mem.value->excluding(
            Mem.value->first()
          )->append(Input.value)
        }
        postcondition ocl {
          MemReset.value = Reset.value
        }
        var iter = 0;
        while (iter < (Delay.value - 1)){
          Mem.value[iter] = Mem.value[iter + 1];
          iter = iter + 1;
        }
        Mem.value[Delay.value - 1] = Input.value;
        MemReset.value = Reset.value;
      }
      init init_Resettable_List_EITHER
      compute compute_Resettable_List_EITHER
      update update_Resettable_List_EITHER
    }
    mode DelayReset_List_LEVEL implements allof (
      ResetInput,
      oneof (ListDelay_ScalarInput, ListDelay_VectorInput, ListDelay_MatrixInput)
    ){
      modeinvariant ocl {
        Reset_Algo.value = !!TResetAlgo::LEVEL
      }
      definition bal = init_Resettable_List_LEVEL {
        postcondition ocl { Mem.value = IC.value }
        Mem.value = IC.value;
      }
      definition bal = compute_Resettable_List_LEVEL {
        postcondition ocl {
          Output.value = Mem.value->first() or Output.value = IC.value->first()
        }
        postcondition ocl {
```

```
        ((Reset.value <> 0.0) or
         (Reset.value = 0.0 and MemReset.value <> 0.0)) implies
        Output.value = IC.value->first()
      }
      postcondition ocl {
        (not ((Reset.value <> 0.0) or
              (Reset.value = 0.0 and MemReset.value <> 0.0))) implies
        Output.value = Mem.value->first()
      }
      if ((Reset.value != 0.0) || ((Reset.value == 0.0) &&
          (MemReset.value != 0.0))) {
        Output.value = IC.value[0];
      } else {
        Output.value = Mem.value[0];
      }
    }
    definition bal = update_Resettable_List_LEVEL {
      postcondition ocl {
        Mem.value = Mem.value->excluding(
          Mem.value->first()
        )->append(Input.value)
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      var iter = 0;
      while (iter < (Delay.value - 1)){
        Mem.value[iter] = Mem.value[iter + 1];
        iter = iter + 1;
      }
      Mem.value[Delay.value - 1] = Input.value;
      MemReset.value = Reset.value;
    }
    init init_Resettable_List_LEVEL
    compute compute_Resettable_List_LEVEL
    update update_Resettable_List_LEVEL
  }
  mode DelayReset_List_LEVEL_HOLD implements allof (
    ResetInput,
    oneof (ListDelay_ScalarInput, ListDelay_VectorInput, ListDelay_MatrixInput)
  ){
    modeinvariant ocl {
      Reset_Algo.value = !!TResetAlgo::LEVEL_HOLD
    }
    definition bal = init_Resettable_List_LEVEL_HOLD {
      postcondition ocl { Mem.value = IC.value }
      Mem.value = IC.value;
    }
    definition bal = compute_Resettable_List_LEVEL_HOLD {
      postcondition ocl {
        Output.value = Mem.value->first() or Output.value = IC.value->first()
      }
      postcondition ocl {
        (Reset.value <> 0.0) implies Output.value = IC.value->first()
      }
      postcondition ocl {
        (not (Reset.value <> 0.0)) implies Output.value = IC.value->first()
      }
      if (Reset.value != 0.0) {
        Output.value = IC.value[0];
      } else {
        Output.value = Mem.value[0];
      }
    }
    definition bal = update_Resettable_List_LEVEL_HOLD {
      postcondition ocl {
        Mem.value = Mem.value->excluding(
          Mem.value->first()
        )->append(Input.value)
      }
      postcondition ocl {
        MemReset.value = Reset.value
      }
      var iter = 0;
```

```
712         while (iter < (Delay.value - 1)){
              Mem.value[iter] = Mem.value[iter + 1];
              iter = iter + 1;
            }
            Mem.value[Delay.value - 1] = Input.value;
717         MemReset.value = Reset.value;
          }
          init init_Resettable_List_LEVEL_HOLD
          compute compute_Resettable_List_LEVEL_HOLD
          update update_Resettable_List_LEVEL_HOLD
722     }
    }
  }
```

Listing A.1: *Delay* block specification using the BLOCKLIBRARY language

A.2 *MINMAX* BLOCK SPECIFICATION

```
1   library MinMaxLib {
      // Scalar data types
      type realDouble TDouble
      type realInt TInt16 of 16 bits
      // Multi-dimensional data types
6     type array TArrayInt16 of TInt16 [0]
      type array TArrayDouble of TDouble [0]
      type array TMatrixDouble of TDouble [0,0]

      type string TString
11    // Enumerations
      type enum MinMaxFunction {Min,Max}

      blocktype MinMax {
        variant MinMaxParameters {
16        parameter FunctionParam : MinMaxFunction
          parameter NbInputs : TInt16 { invariant ocl { NbInputs.value >= 1 } }
        }
        variant MinMaxInScalars extends MinMaxParameters {
          in data In1 : TDouble [1 .. 0]
21      }
        variant MinMaxInVectors extends MinMaxParameters {
          in data In1 : TArrayDouble [1 .. 0]
        }
        variant MinMaxInMatrices extends MinMaxParameters {
26        in data In1 : TMatrixDouble [1 .. 0]
        }
        variant MinMaxOutScalar {
          out data Out : TDouble
        }
31      variant MinMaxOutVector {
          out data Out : TArrayDouble
        }
        variant MinMaxOutMatrix {
          out data Out : TMatrixDouble
36      }
        mode MinOutputScalarMultipleInputsScalars implements allof(MinMaxOutScalar,
                                                                   MinMaxInScalars) {
          modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
          modeinvariant ocl { NbInputs.value >= 1 }
41        definition bal = compute_MinOutScalarMultipleInputsScalars {
            postcondition ocl {
              In1->forAll(i| i.value >= Out.value)
            }
            var res = In1[0].value;
46          for (var i = 0; i < (size(In1)); i = i + 1){
              if (res > In1[i].value){
                res = In1[i].value;
              }
            }
51          Out.value = res;
          }
          compute compute_MinOutScalarMultipleInputsScalars
        }
        mode MaxOutputScalarMultipleInputsScalars implements allof(MinMaxOutScalar,
56                                                                 MinMaxInScalars) {
```

```
    modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Max }
    modeinvariant ocl { NbInputs.value >= 1 }
    definition bal = compute_MaxOutScalarMultipleInputsScalars {
      postcondition ocl {
61      In1->forAll(i| i.value <= Out.value)
      }
      var res = In1[0].value;
      for (var i = 0; i < (size(In1)); i = i + 1){
        if (res < In1[i].value){
66        res = In1[i].value;
        }
      }
      Out.value = res;
    }
71  compute compute_MaxOutScalarMultipleInputsScalars
  }
  mode MinOutputScalarOneInputVector implements allof(MinMaxOutScalar,
                                                      MinMaxInVectors) {
    modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
76  modeinvariant ocl { NbInputs.value = 1 }
    definition bal = compute_MinOutputScalarOneInputVector {
      postcondition ocl {
        In1->forAll(i| i.value->forAll(v|v >= Out.value))
      }
81    var res = In1[0].value[0];
      for (var i = 0; i < (size(In1[0].value)); i = i + 1){
        if (res > In1[0].value[i]){
          res = In1[0].value[i];
        }
86    }
      Out.value = res;
    }
    compute compute_MinOutputScalarOneInputVector
91  }
  mode MaxOutputScalarOneInputVector implements allof(MinMaxOutScalar,
                                                      MinMaxInVectors) {
    modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Max }
    modeinvariant ocl { NbInputs.value = 1 }
    definition bal = compute_MaxOutputScalarOneInputVector {
96    postcondition ocl {
        In1->forAll(i| i.value->forAll(v|v <= Out.value))
      }
      var res = In1[0].value[0];
      for (var i = 0; i < (size(In1[0].value)); i = i + 1){
101     if (res < In1[0].value[i]){
          res = In1[0].value[i];
        }
      }
      Out.value = res;
106   }
    compute compute_MaxOutputScalarOneInputVector
  }
  mode MinOutputScalarOneInputMatrix implements allof(MinMaxOutScalar,
                                                      MinMaxInMatrices) {
111   modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
    modeinvariant ocl { NbInputs.value = 1 }
    definition bal = compute_MinOutputScalarOneInputMatrix {
      postcondition ocl {
        In1->forAll(i| i.value->forAll(v|v->forAll(s| s >= Out.value)))
116   }
      var res = In1[0].value[0][0];
      for (var i = 0; i < (size(In1[0].value)); i = i + 1){
        for (var j = 0; j < (size(In1[0].value[0])); j = j + 1){
          if (res > In1[0].value[i][j]){
121         res = In1[0].value[i][j];
          }
        }
      }
      Out.value = res;
126   }
    compute compute_MinOutputScalarOneInputMatrix
  }
  mode MaxOutputScalarOneInputMatrix implements allof(MinMaxOutScalar,
                                                      MinMaxInMatrices) {
```

```
131       modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Max }
          modeinvariant ocl { NbInputs.value = 1 }
          definition bal = compute_MaxOutputScalarOneInputMatrix {
            postcondition ocl {
              In1->forAll(i| i.value->forAll(v|v->forAll(s| s <= Out.value)))
136         }
            var res = In1[0].value[0][0];
            for (var i = 0; i < (size(In1[0].value)); i = i + 1){
              for (var j = 0; j < (size(In1[0].value[0])); j = j + 1){
                if (res < In1[0].value[i][j]){
141               res = In1[0].value[i][j];
                }
              }
            }
            Out.value = res;
146       }
          compute compute_MaxOutputScalarOneInputMatrix
        }
        mode MinOutputVectorMultipleInputsVectors implements allof(MinMaxOutVector,
                                                                   MinMaxInVectors) {
151       modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
          modeinvariant ocl { NbInputs.value > 1 }
          definition bal = compute_MinOutVectorMultipleInputsVectors {
            postcondition ocl {
              Out.value->forAll(o| In1->forAll(i| i.value->at(Out.value->indexOf(o)) >= o))
156         }
            var res = In1[0].value;
            for (var i = 0; i < (size(In1)); i = i + 1){
              for (var j = 0; j < (size(In1[i].value)); j = j + 1){
                if (res[j] > In1[i].value[j]){
161               res[j] = In1[i].value[j];
                }
              }
            }
            Out.value = res;
166       }
          compute compute_MinOutVectorMultipleInputsVectors
        }
        mode MaxOutputVectorMultipleInputsVectors implements allof(MinMaxOutVector,
                                                                   MinMaxInVectors) {
171       modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Max }
          modeinvariant ocl { NbInputs.value > 1 }
          definition bal = compute_MaxOutVectorMultipleInputsVectors {
            postcondition ocl {
              Out.value->forAll(o| In1->forAll(i| i.value->at(Out.value->indexOf(o)) <= o))
176         }
            var res = In1[0].value;
            for (var i = 0; i < (size(In1)); i = i + 1){
              for (var j = 0; j < (size(In1[i].value)); j = j + 1){
                if (res[j] < In1[i].value[j]){
181               res[j] = In1[i].value[j];
                }
              }
            }
            Out.value = res;
186       }
          compute compute_MaxOutVectorMultipleInputsVectors
        }

        mode MinOutputMatrixMultipleInputsMatrices implements allof(MinMaxOutMatrix,
191                                                                 MinMaxInMatrices) {
          modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Min }
          modeinvariant ocl { NbInputs.value > 1 }
          definition bal = compute_MinOutScalarMultipleInputsMatrices {
            postcondition ocl {
196           Out.value->forAll(ov|
                ov->forAll(os|
                  In1->forAll(i|
                    i.value->at(Out.value->indexOf(ov))->at(ov->indexOf(os)) >= os
                  )
201             )
              )
            }
            var res = In1[0].value;
```

```
        for (var i = 0; i < (size(In1)); i = i + 1){
          for (var j = 0; j < (size(In1[i].value)); j = j + 1){
            for (var k = 0; k < (size(In1[i].value[j])); k = k + 1){
              if (res[j][k] > In1[i].value[j][k]){
                res[j][k] = In1[i].value[j][k];
              }
            }
          }
        }
        Out.value = res;
      }
      compute compute_MinOutScalarMultipleInputsMatrices
    }
    mode MaxOutputMatrixMultipleInputsMatrices implements allof(MinMaxOutMatrix,
                                                                MinMaxInMatrices) {
      modeinvariant ocl { FunctionParam.value = !!MinMaxFunction::Max }
      modeinvariant ocl { NbInputs.value > 1 }
      definition bal = compute_MaxOutScalarMultipleInputsMatrices {
        postcondition ocl {
          Out.value->forAll(ov|
            ov->forAll(os|
              In1->forAll(i|
                i.value->at(Out.value->indexOf(ov))->at(ov->indexOf(os)) <= os
              )
            )
          )
        }
        var res = In1[0].value;
        for (var i = 0; i < (size(In1)); i = i + 1){
          for (var j = 0; j < (size(In1[i].value)); j = j + 1){
            for (var k = 0; k < (size(In1[i].value[j])); k = k + 1){
              if (res[j][k] < In1[i].value[j][k]){
                res[j][k] = In1[i].value[j][k];
              }
            }
          }
        }
        Out.value = res;
      }
      compute compute_MaxOutScalarMultipleInputsMatrices
    }
  }
}
```

Listing A.2: *MinMax* block specification using the BLOCKLIBRARY language

B

OCL grammar

We provide in the following the XTEXT grammar for the OCL we embedded in the BLOCKLIBRARY specification language for the specification of constraints in the specification of blocks.

```
OclExpression returns ocl::OclExpression :
  BoolOpCallExp|LetExp;

OclModelElementExp returns ocl::OclModelElementExp:
  elem=[Element|UIDENT]
;

DefinitionCallExp returns ocl::DefinitionCallExp:
  annot=[Annotation|LIDENT] '('( arguments+=OclExpression (',' arguments+=OclExpression)* )
      ?')'
;

BoolOpCallExp returns ocl::OclExpression :
  EqOpCallExp (({ocl::BoolOpCallExp.source=current} operationName=BOOLOP)   argument=
      EqOpCallExp )*
;
BOOLOP      : 'and'|'or'|'xor'|'implies'|'equivalent';

EqOpCallExp returns ocl::OperatorCallExp :
  RelOpCallExp (({ocl::EqOpCallExp.source=current} operationName=EQOP)  argument=RelOpCallExp
       )*
;
EQOP      : '='|'<>';

RelOpCallExp returns ocl::OperatorCallExp :
  AddOpCallExp (({ocl::RelOpCallExp.source=current} operationName=RELOP)  argument=
      AddOpCallExp )*
;
RELOP     : '>'|'<'|'>='|'<=' ;

AddOpCallExp returns ocl::OperatorCallExp :
  MulOpCallExp (({ocl::AddOpCallExp.source=current} operationName=ADDOP)  argument=
      MulOpCallExp )*
;
ADDOP : '-'|'+';

MulOpCallExp returns ocl::OperatorCallExp :
  NotOpCallExp (({ocl::MulOpCallExp.source=current} operationName=MULOP)  argument=
      NotOpCallExp )*
;
MULOP: '*'|'/';

NotOpCallExp returns ocl::OperatorCallExp :
  ({ocl::NotOpCallExp} operationName=UnaryOP source=NotOpCallExp)
  | PropertyCallExp
```

```
;
UnaryOP: NOTOP | '-';
NOTOP: 'not';

PropertyCallExp returns ocl::PropertyCallExp:
  source = Primary_OclExpression (calls+=PropertyCall)*
;

Primary_OclExpression returns ocl::OclExpression:
  VariableExp
  |SuperExp
  |SelfExp
  |ResultExp
  |StringExp
  |BooleanExp
  |NumericExp
  |CollectionExp
  |EnumLiteralExp
  |OclUndefinedExp
  |IfExp
  |BraceExp
  |OclModelElementExp
  |DefinitionCallExp;

VariableReferencePivot returns pivot::VariableReferencePivot:
  Iterator
  |LocalVariable
  |ActionLocalVariable
  |Annotation
;

VariableExp returns ocl::VariableExp:
  referredVariable=[pivot::VariableReferencePivot|LIDENT]
;

SuperExp returns ocl::SuperExp:
  {ocl::SuperExp} 'super'
;

SelfExp returns ocl::SelfExp:
  {ocl::SelfExp}  'self'
;

ResultExp returns ocl::ResultExp:
  {ocl::ResultExp} 'result'
;

StringExp returns ocl::StringExp:
  {ocl::StringExp}
  stringSymbol=SINGLE_QUOTED_STRING
;

NumericExp returns ocl::NumericExp:
  RealExp|IntegerExp
;

RealExp returns ocl::RealExp: realSymbol=DOUBLE;

REAL hidden(): INT '.' (EXT_INT | INT);
terminal EXT_INT: INT ('e'|'E')('-'|'+') INT;

IntegerExp returns ocl::IntegerExp:
  {ocl::IntegerExp}
  integerSymbol=(INT|DIGIT)
;

CollectionExp returns ocl::CollectionExp:
  SimpleCollectionExp
  |IntRangeCollection
  |TupleExp
  |MapExp
;

SimpleCollectionExp returns ocl::SimpleCollectionExp:
```

```
  BagExp
  |OrderedSetExp
  |SequenceExp
  |SetExp
;

BagExp returns ocl::BagExp:
  {ocl::BagExp}
  'Bag' '{' (elements+=OclExpression ("," elements+=OclExpression)*)? '}'
;

OrderedSetExp returns ocl::OrderedSetExp:
  {ocl::OrderedSetExp}
  'OrderedSet' '{' (elements+=OclExpression ("," elements+=OclExpression)*)? '}'
;

SequenceExp returns ocl::SequenceExp:
  {ocl::SequenceExp}
  'Sequence' '{' (elements+=OclExpression ("," elements+=OclExpression)*)? '}'
;

SetExp returns ocl::SetExp:
  {ocl::SetExp}'Set' '{' (elements+=OclExpression ("," elements+=OclExpression)*)? '}'
;

IntRangeCollection returns ocl::IntRangeCollectionExp:
  {ocl::IntRangeCollectionExp} 'Set' '{' from=OclExpression ".." to=OclExpression '}'
;

TupleExp returns ocl::TupleExp:
  {ocl::TupleExp}
  'Tuple' '{' (tuplePart+=TuplePart ("," tuplePart+=TuplePart)*)? '}'
;

TuplePart returns ocl::TuplePart:
  varName=SINGLE_QUOTED_STRING (':' type=DataType)? '=' initExpression=OclExpression
;

MapExp returns ocl::MapExp:
  {ocl::MapExp}
  'Map' '{' (elements+=MapElement (',' elements+=MapElement)*)? '}'
;

MapElement returns ocl::MapElement:
  '(' key=OclExpression ',' value=OclExpression ')'
;

EnumLiteralExp returns ocl::EnumLiteralExp:
  '!!' litValue=[value::LiteralExpression|QUALIFIED_UIDENT]
;

QUALIFIED_UIDENT:
  UIDENT ('::' UIDENT)*
;

OclUndefinedExp returns ocl::OclUndefinedExp:
  {ocl::OclUndefinedExp}
  'OclUndefined'
;

LetExp returns ocl::LetExp:
  'let' variable=LocalVariable 'in' in_=OclExpression
;

IfExp returns ocl::IfExp:
  'if' condition=OclExpression 'then' thenExpression=OclExpression
  'else' elseExpression=OclExpression 'endif'
;

BraceExp returns ocl::BraceExp:
  '(' exp=OclExpression ')'
;

BooleanExp returns ocl::BooleanExp:
```

```
  booleanSymbol= 'true'|booleanSymbol= 'false'
;

PropertyCall returns ocl::PropertyCall:
  OperationCall|NavigationOrAttributeCall
  |IterateExp|IteratorExp|CollectionOperationCall
;

OperationCall returns ocl::OperationCall:
  '.' (operationName=SINGLE_QUOTED_STRING|operationName=LIDENT) '('
    (arguments+=OclExpression (',' arguments+=OclExpression)*)?
  ')'
;

NavigationOrAttributeCall returns ocl::NavigationOrAttributeCall:
  '.' (name=SINGLE_QUOTED_STRING|name=LIDENT)
;

IterateExp returns ocl::IterateExp:
  '->' 'iterate' '(' iterators+=Iterator (',' iterators+=Iterator)* ';'
  result=LocalVariable '|' body=OclExpression ')'
;

Iterator returns ocl::Iterator:
  name=LIDENT (':' type=[dt::DataType|UIDENT])?
;

IteratorExp returns ocl::IteratorExp:
  '->' name=LIDENT '(' iterators+=Iterator (',' iterators+=Iterator)*
  '|' body=OclExpression ')'
;

CollectionOperationCall returns ocl::CollectionOperationCall:
  '->' operationName=LIDENT '(' (arguments+=OclExpression
    (',' arguments+=OclExpression)*)?
  ')'
;

LocalVariable returns ocl::LocalVariable:
  name=LIDENT (':' type=[dt::DataType|UIDENT])? '=' initExpression=OclExpression
;

OclType returns dt::DataType:
  CollectionType
  | TPrimitive
  | OclAnyType
  | TupleType
  | OclModelElement
  | MapType
  | OclType_abstractContents
;

CollectionType returns dt::TArray:
  BagType
  | OrderedSetType
  | SequenceType
  | SetType
;

BagType returns dt::TArray:
  'Bag' '(' baseType=[dt::TPrimitive|UIDENT] ')'
;

OrderedSetType returns dt::TArray:
  'OrderedSet' '(' baseType=[dt::TPrimitive|UIDENT] ')'
;

SequenceType returns dt::TArray:
  'Sequence' '(' baseType=[dt::TPrimitive|UIDENT] ')'
;

SetType returns dt::TArray:
  'Set' '(' baseType=[dt::TPrimitive|UIDENT] ')'
;
```

```
OclAnyType returns dt::DataType:
  {ocl::OclAnyType}'OclAny'
;

OclType_abstractContents returns ocl::OclType :
  {ocl::OclType} 'OclType'
;

TupleType returns dt::DataType:
  {ocl::TupleType} ('TupleType' | 'Tuple') '('
    (attributes+=TupleTypeAttribute (',' attributes+=TupleTypeAttribute)*)?
  ')'
;

TupleTypeAttribute returns ocl::TupleTypeAttribute:
  name=LIDENT ':' type=DataType;

OclModelElement returns ocl::OclModelElement:
  elem=[Element|LIDENT];

MapType returns ocl::MapType:
  'Map' '(' keyType=DataType ',' valueType=DataType ')'
;
```

C

BAL grammar

We provide in the following the XTEXT grammar for the BAL action language embedded in the BLOCKLIBRARY specification language for the specification of the blocks operational semantics.

```
ActionBlock returns action::ActionBlock:
  (elements+=ActionBlockElement)+
;

ActionBlockElement returns action::ActionBlockElement:
  (ghost?='ghost')?
  ((localVariable=ActionLocalVariable ';')
  | (
    (expression=ActionExpression ';')
    | expression=ActionPrimaryComplexExpression
    | expression=AssertExpression
    ))
;

ExpressionActionBlock returns action::ActionBlock:
  elements+=ActionBlockElement
;

ActionLocalVariable returns action::LocalVariable:
  'var' name=LIDENT '=' init=ActionBoolOpCallExp
;

ActionExpression returns action::ActionExpression:
   ActionAssignVariable
;

ActionAssignVariable returns action::ActionExpression:
  ActionBoolOpCallExp
  (({action::VariableAssignmentExp.assignedVariable=current}
    '=') exp=ActionBoolOpCallExp)?
;
ACTIONASSIGNOP    : '=';

ActionBoolOpCallExp returns action::OperatorCallExp :
  ActionRelOpCallExp (
    ({action::BoolOpCallExp.source=current} operationName=ACTIONBOOLOP)
    argument=ActionRelOpCallExp
  )*
;
ACTIONBOOLOP      : '&&'|'||'|'=='|'!='|'->';

ActionRelOpCallExp returns action::OperatorCallExp :
  ActionAddOpCallExp (
    ({action::RelOpCallExp.source=current} operationName=ACTIONRELOP)
    argument=ActionAddOpCallExp
```

```
  )*
;
ACTIONRELOP     : '>'|'<'|'>='|'<='|'>.'|'<.'|'>=.'|'<=.' ;

ActionAddOpCallExp returns action::OperatorCallExp :
  ActionMulOpCallExp (
    ({action::AddOpCallExp.source=current} operationName=ACTIONADDOP)
    argument=ActionMulOpCallExp
  )*
;
ACTIONADDOP : '-'|'+'|'-.'|'+.';

ActionMulOpCallExp returns action::OperatorCallExp :
  ActionNotOpCallExp (
    ({action::MulOpCallExp.source=current} operationName=ACTIONMULOP)
    argument=ActionNotOpCallExp
  )*
;
ACTIONMULOP : '*'|'/'|'*.'|'/.';

ActionNotOpCallExp returns action::OperatorCallExp :
  ({action::NotOpCallExp} operationName=ACTIONUNARYOP source=ActionNotOpCallExp)
  | ActionPropertyCallExp
;
ACTIONUNARYOP: ACTIONNOTOP | '-';
ACTIONNOTOP: '!';

ActionPropertyCallExp returns action::PropertyCallExp:
  source=ActionPrimaryExpression (calls+=ActionPropertyCall)*
;

ActionPropertyCall returns action::PropertyCall:
  ActionAttributeCall
  | ActionSquareBracketCall
;

ActionAttributeCall returns action::AttributeCall:
  '.' name=LIDENT
;

ActionSquareBracketCall returns action::SquareBracketCall:
  '[' exp=ActionExpression ']'
;

ActionPrimaryExpression returns action::ActionExpression:
  ActionOperationCall
  | ActionPrimaryComplexExpression
  | ActionBLModelElement
  | ActionLiteralExpression
;

ActionOperationCall returns action::OperationCall:
  operationName=LIDENT '(' (
    arguments+=ActionExpression
    (',' arguments+=ActionExpression)*
  )? ')'
;

ActionPrimaryComplexExpression returns action::ActionExpression:
  ActionITEExpression
  | ActionForExpression
  | ActionWhileExpression
;

ActionITEExpression returns action::ITEExpression:
  'if' '(' condition=ActionExpression ')'
  (
    then=ExpressionActionBlock
    |  ('{'then=ActionBlock'}')
  )
  (=>
    'else'
    (
      else=ExpressionActionBlock
```

```
      | ('{' else=ActionBlock '}')
    )
  )?
;

ActionForExpression returns action::ForExpression:
  'for' '('
    iter=ActionLocalVariable ';'
    condition=ActionExpression ';'
    update=ActionExpression ')' '{'
      (spec+=SimpleAnnotationExpression)*
      block=ActionBlock
    '}'
;

ActionWhileExpression returns action::WhileExpression:
  'while' '(' condition=ActionExpression ')' '{'
    (spec+=SimpleAnnotationExpression)*
    block=ActionBlock
  '}'
;

AssertExpression returns action::AnnotationExpression:
  AssertActionExpression
  | AssertOclExpression
;

AssertActionExpression returns action::AnnotationExpression:
  {action::AssertExpression} kind=ASSERTKINDBAL
  '{' balExpression=ActionExpression '}'
;

AssertOclExpression returns action::AnnotationExpression:
  {action::AssertExpression} kind=ASSERTKINDOCL
  '{' oclExpression=OclExpression '}'
;
ASSERTKINDBAL : 'bal_assert';
ASSERTKINDOCL : 'ocl_assert';

SimpleAnnotationExpression returns action::AnnotationExpression:
  SimpleAnnotationActionExpression
  | SimpleAnnotationOclExpression
;

SimpleAnnotationActionExpression returns action::AnnotationExpression:
  {action::SimpleAnnotationExpression} kind=SIMPLEANNOTKINDBAL
  '{' balExpression=ActionExpression '}'
;

SimpleAnnotationOclExpression returns action::AnnotationExpression:
  {action::SimpleAnnotationExpression} kind=SIMPLEANNOTKINDOCL
  '{' oclExpression=OclExpression '}'
;

SIMPLEANNOTKINDBAL : 'bal_variant' | 'bal_invariant';
SIMPLEANNOTKINDOCL : 'ocl_variant' | 'ocl_invariant';

ActionBLModelElement returns action::BLModelElementExp:
  elem=[Element|UIDENT]
;

ActionLiteralExpression returns action::ActionExpression:
  ActionVariableReferenceExp
  | ActionParenthesis
  | ActionStringExpression
  | ActionRealExpression
  | ActionIntExpression
  | ActionBoolExpression
  | ActionEnumLiteralExp
;

ActionVariableReferenceExp returns action::VariableExp:
  referredVariable=[pivot::VariableReferencePivot|LIDENT]
;
```

```
ActionParenthesis returns action::ParenthesisExp:
  '(' exp=ActionExpression ')'
;

ActionStringExpression returns action::StringExp:
  stringSymbol=SINGLE_QUOTED_STRING
;

ActionRealExpression returns action::RealExp:
  realSymbol=DOUBLE
;

ActionIntExpression returns action::IntegerExp:
  integerSymbol=(INT|DIGIT)
;

ActionBoolExpression returns action::BooleanExp:
  booleanSymbol= 'true'|booleanSymbol= 'false'
;

ActionEnumLiteralExp returns action::EnumLiteralExp:
  '!!' litValue=[value::LiteralExpression|QUALIFIED_UIDENT]
;
```

D

Why3 libraries

D.1 Why3 data types theories

D.1.1 Numeric data types definition theories

```
theory Boolean
  use import bool.Bool
  type boolean_type = Bool.bool
end
theory RealSingle
  use import real.RealInfix
  type tRealSingle = real
end
theory RealDouble
  use import real.RealInfix
  type tRealDouble = real
end
theory RealFloatingPoint
  use import floating_point.DoubleFull
  type tRealFloatingPoint = DoubleFull.double
end
```

Listing D.1: Scalar data types theories re-definition from standard Why3 theories

```
theory RealInteger
  use import bool.Bool
  use import int.Int

  type tRealInteger

  constant nBits : int
  constant signed : bool
  constant max_RealInteger : int
end
```

Listing D.2: General Integer data type theories in Why3

```
theory SignedInt8
  use import int.Int
  use import bool.Bool

  constant nBits_signed_8 : int = 8
  constant signed_signed_8 : bool = True
  constant max_RealInteger_signed_8 : int = 128

  type tRealSignedInt8 = int

  clone export RealInteger with
    constant nBits = nBits_signed_8,
    constant signed = signed_signed_8,
    constant max_RealInteger = max_RealInteger_signed_8,
    type tRealInteger = tRealSignedInt8
  predicate limit_tRealSignedInt8 (x : tRealSignedInt8) =
    (-max_RealInteger_signed_8) <= x <= (max_RealInteger_signed_8 - 1)
end
theory SignedInt16
  use import int.Int
  use import bool.Bool

  constant nBits_signed_16 : int = 16
  constant signed_signed_16 : bool = True
  constant max_RealInteger_signed_16 : int = 32768

  type tRealSignedInt16 = int

  clone export RealInteger with
    constant nBits = nBits_signed_16,
    constant signed = signed_signed_16,
    constant max_RealInteger = max_RealInteger_signed_16,
    type tRealInteger = tRealSignedInt16
  predicate limit_tRealSignedInt16 (x : tRealSignedInt16) =
    (-max_RealInteger_signed_16) <= x <= (max_RealInteger_signed_16 - 1)

end
```

Listing D.3: Signed integers data types theories definition in WHY3

```
theory SignedInt32
  use import int.Int
  use import bool.Bool

  constant nBits_signed_32 : int = 32
  constant signed_signed_32 : bool = True
  constant max_RealInteger_signed_32 : int = 2147483648

  type tRealSignedInt32 = int

  clone export RealInteger with
    constant nBits = nBits_signed_32,
    constant signed = signed_signed_32,
    constant max_RealInteger = max_RealInteger_signed_32,
    type tRealInteger = tRealSignedInt32
  predicate limit_tRealSignedInt32 (x : tRealSignedInt32) =
    (-max_RealInteger_signed_32) <= x <= (max_RealInteger_signed_32 - 1)
end
theory SignedInt64
  use import int.Int
  use import bool.Bool

  constant nBits_signed_64 : int = 64
  constant signed_signed_64 : bool = True
  constant max_RealInteger_signed_64 : int = 9223372036854775808

  type tRealSignedInt64 = int

  clone export RealInteger with
    constant nBits = nBits_signed_64,
    constant signed = signed_signed_64,
    constant max_RealInteger = max_RealInteger_signed_64,
    type tRealInteger = tRealSignedInt64
  predicate limit_tRealSignedInt64 (x : tRealSignedInt64) =
    (-max_RealInteger_signed_64) <= x <= (max_RealInteger_signed_64 - 1)
end
```

Listing D.4: Signed integers data types theories definition in WHY3

```
theory UnsignedInt8
  use import int.Int
  use import bool.Bool

  constant nBits_unsigned_8 : int = 8
  constant signed_unsigned_8 : bool = True
  constant max_RealInteger_unsigned_8 : int = 256

  type tRealUnSignedInt8 = int

  clone export RealInteger with
    constant nBits = nBits_unsigned_8,
    constant signed = signed_unsigned_8,
    constant max_RealInteger = max_RealInteger_unsigned_8,
    type tRealInteger = tRealUnSignedInt8
  predicate limit_tRealUnsignedInt8 (x : tRealUnSignedInt8) =
    0 <= x <= (max_RealInteger_unsigned_8 - 1)
end
theory UnsignedInt16
  use import int.Int
  use import bool.Bool

  constant nBits_unsigned_16 : int = 16
  constant signed_unsigned_16 : bool = True
  constant max_RealInteger_unsigned_16 : int = 65536

  type tRealUnSignedInt16 = int

  clone export RealInteger with
    constant nBits = nBits_unsigned_16,
    constant signed = signed_unsigned_16,
    constant max_RealInteger = max_RealInteger_unsigned_16,
    type tRealInteger = tRealUnSignedInt16
  predicate limit_tRealUnsignedInt16 (x : tRealUnSignedInt16) =
    0 <= x <= (max_RealInteger_unsigned_16 - 1)
end
```

Listing D.5: Unsigned integers data types theories definition in WHY3

```
theory UnsignedInt32
  use import int.Int
  use import bool.Bool

  constant nBits_unsigned_32 : int = 32
  constant signed_unsigned_32 : bool = True
  constant max_RealInteger_unsigned_32 : int = 4294967296

  type tRealUnSignedInt32 = int

  clone export RealInteger with
    constant nBits = nBits_unsigned_32,
    constant signed = signed_unsigned_32,
    constant max_RealInteger = max_RealInteger_unsigned_32,
    type tRealInteger = tRealUnSignedInt32
  predicate limit_tRealUnsignedInt32 (x : tRealUnSignedInt32) =
    0 <= x <= (max_RealInteger_unsigned_32 - 1)
end
theory UnsignedInt64
  use import int.Int
  use import bool.Bool

  constant nBits_unsigned_64 : int = 64
  constant signed_unsigned_64 : bool = True
  constant max_RealInteger_unsigned_64 : int = 18446744073709551616

  type tRealUnSignedInt64 = int

  clone export RealInteger with
    constant nBits = nBits_unsigned_64,
    constant signed = signed_unsigned_64,
    constant max_RealInteger = max_RealInteger_unsigned_64,
    type tRealInteger = tRealUnSignedInt64
  predicate limit_tRealUnsignedInt64 (x : tRealUnSignedInt64) =
    0 <= x <= (max_RealInteger_unsigned_64 - 1)
end
```

Listing D.6: Unsigned integers data types theories definition in WHY3

D.1.2 Common functions definition theories

In order to simplify the writing of expressions we introduce in the *CommonFunctions* theory a getter function as a shortcut notation for the use of the *nth* standard Why3 function. In this theory we add two lemmas on the *mem* standard function. These two lemmas have been defined in order to ease the verification of more complex lemmas detailed in Sections D.1.3 and D.4.

```
theory CommonFunctions
  use import list.List
  use import list.Mem
  use import list.NthNoOpt

  (* === Getter on lists === *)
  function ([]) (a: list 'a) (i: int) : 'a = nth i a

  (* mem x l *)
  lemma mem_nil: forall b: 'a.
    mem b Nil = false

  lemma mem_cons: forall l: list 'a, b,c: 'a.
    mem b (Cons c l) -> b = c \/ mem b l
end
```

Listing D.7: CommonFunctions theory definition using Why3

D.1.3 String data types definition theories

The *String* data type is used as a standard data type in many formalisms such as OCL, Ecore or Simulink that we use in this PhD. In order to express formal properties using this data type, it was mandatory to provide a formalisation for this data type. This is what we provide in this section. We start this with the formalisation for characters in the *Char* theory as we decided to represent a *String* as a set of characters. We finally we provide the formalisation for the *String* data type based on the *Char* theory.

It is worth noting that this formalisation is highly inspired from the module provided in the Why3 standard library[1]. As we needed to express properties and to verify them using Why3, we needed to convert this module to a theory.

In this formalisation of characters, a character is a record type with one field that is an integer value. Its formalisation is provided in Listing D.8.

The complete specification for the Why *String* theory is provided in Listings D.9 and D.10. We provided it again here in the interest of clarity. The formal verification report of the related *String* type lemmas has been generated with the Why3 toolset and is provided in Figure D.11.

[1] http://why3.lri.fr/stdlib-0.83/

```
theory Char
  (* see module string.Char *)
  use import int.Int

  type tChar = { code: int }

  predicate code_value_limit (c: tChar) =
    c.code >= 32 /\ c.code <= 126

  function toLower_char (c: tChar) : tChar

  axiom ToLower_char_ident : forall c: tChar.
      (32 <= c.code < 65 \/ 90 < c.code <= 126) -> toLower_char c = c

  axiom ToLower_char_change : forall c: tChar.
      (65 <= c.code <= 90) -> toLower_char c = {code=c.code + 32}

  function toUpper_char (c: tChar) : tChar

  axiom ToUpper_char_ident : forall c: tChar.
      (32 <= c.code < 97 \/ 122 < c.code <= 126) -> toUpper_char c = c

  axiom ToUpper_char_change : forall c: tChar.
      (97 <= c.code <= 122) -> toUpper_char c = {code=c.code - 32}
end
```

Listing D.8: Char theory definition using Why3

Proof obligations	Alt-Ergo-Pro (1.0.0)
lemma concat_length	0.01
lemma concat_l_cons	0.01
lemma concat_r_cons	0.03
lemma concat_l_nil	0.01
lemma concat_r_nil	0.00
lemma concat_l_mem	0.01
lemma concat_r_mem	0.01
lemma subString_nil	0.00
lemma subString_length_nil	0.01
lemma subString_0_0	0.01
lemma subString_length_0_0	0.02
lemma subString_0_x	0.03
lemma subString_length_0_x	0.02
lemma subString_x_y	0.02
lemma subString_length_x_y	0.02
lemma subString_OutOfBound	0.01
lemma length_one	0.01

Figure D.11: String theory lemmas verification with WHY3 and SMT solvers

```
theory String
  (* see module string.String *)
  use import int.Int
  use import Char
  use import list.Length
  use import list.Append
  use import list.List
  use import list.Mem
  use import list.NthNoOpt
  use import blocklibrary_common.CommonFunctions

  type string_type = list tChar

  function concat (s1 s2: string_type) : string_type = s1 ++ s2

  lemma concat_length: forall s1, s2: string_type.
    length (concat s1 s2) = length s1 + length s2

  lemma concat_l_cons: forall s1, s2: string_type, c1: tChar.
    concat (Cons c1 s1) s2 = Cons c1 (concat s1 s2)

  lemma concat_r_cons: forall s1, s2: string_type, c1: tChar.
    concat s1 (Cons c1 s2) = concat (concat s1 (Cons c1 Nil)) s2

  lemma concat_l_nil: forall s1, s2: string_type.
    (s1 = Nil -> concat s1 s2 = s2)

  lemma concat_r_nil: forall s1, s2: string_type.
    (s2 = Nil -> concat s1 s2 = s1)

  lemma concat_l_mem: forall s1, s2: string_type, c1: tChar.
    mem c1 s1 -> mem c1 (concat s1 s2)

  lemma concat_r_mem: forall s1, s2: string_type, c1: tChar.
    mem c1 s2 -> mem c1 (concat s1 s2)

  function toLower (s1: string_type) : string_type

  axiom toLower_content: forall s1: string_type, i: int.
    0 <= i < length s1 -> nth i (toLower s1) = toLower_char (nth i s1)

  function toUpper (s1: string_type) : string_type

  axiom toUpper_content: forall s1: string_type, i: int.
    0 <= i < length s1 -> nth i (toUpper s1) = toUpper_char (nth i s1)
```

Listing D.9: String theory definition using Why3 (I)

```
  function subString (s: string_type) (lo up:int) : string_type =
    if lo >= length s \/ lo < 0 \/ up < 0 \/
       up >= length s \/ up < lo then Nil
    else match s with
         | Nil -> Nil
         | Cons hd tl ->
             if lo = 0 then
               if up = 0 then
                 Cons hd Nil
               else Cons hd (subString tl 0 (up-1))
             else subString tl (lo-1) (up-1)
         end

  lemma subString_nil: forall x,y: int.
    subString Nil x y = Nil

  lemma subString_length_nil: forall x,y: int.
    length (subString Nil x y) = 0

  lemma subString_0_0: forall s: string_type, c: tChar.
    subString (Cons c s) 0 0 = Cons c Nil

  lemma subString_length_0_0: forall s: string_type, c: tChar.
    length (subString (Cons c s) 0 0) = 1

  lemma subString_0_x: forall s: string_type, c: tChar, x: int.
    (0 <= x < length s) ->
    subString (Cons c s) 0 x = Cons c (subString s 0 (x-1))

  lemma subString_length_0_x: forall s: string_type, c: tChar, x: int.
    (0 <= x < length s) ->
    length (subString (Cons c s) 0 x) = 1 + length (subString s 0 (x-1))

  lemma subString_x_y: forall s: string_type, c: tChar, x,y: int.
    (0 < x <= y < length s) ->
    subString (Cons c s) x y = subString s (x-1) (y-1)

  lemma subString_length_x_y: forall s: string_type, c: tChar, x,y: int.
    (0 < x <= y < length s) ->
    length (subString (Cons c s) x y) = length (subString s (x-1) (y-1))

  lemma subString_OutOfBound:  forall l: string_type, lo up: int.
    (lo >= length l -> (subString l lo up) = Nil) /\
    (lo < 0 -> (subString l lo up) = Nil) /\
    (up < 0 -> (subString l lo up) = Nil) /\
    (up >= length l -> (subString l lo up) = Nil) /\
    (up < lo -> (subString l lo up) = Nil)

  lemma length_one: forall l: list 'a, e: 'a.
    length (Cons e l) = 1 + length l
end

theory StringRich
  use export Char
  use export UTF8Table
  use export String
end
```

Listing D.10: String theory definition using Why3 (II)

D.2 BlockLibrary StructuralFeature definition theory

We provide here the definitions for the BlockLibrary `StructuralFeature`. Input port `StructuralFeature` is equiped with the *size_inpg* function (Listing D.12) returning for any input port data structure its size. This function implementation is verified with some lemmas discharged using the Why3 toolset. Verification results are provided in Figure D.13.

```
theory InPortGroup
  use import blocklibrary_string.String
  use import blocklibrary_scalar.Boolean
  use import int.Int

  type tInPortGroup 'a

  function name_inpg (tInPortGroup 'a) : string_type
  function min_size_inpg (tInPortGroup 'a) : int
  function max_size_inpg (tInPortGroup 'a) : int
  function isDimensionalizable_inpg (tInPortGroup 'a) : boolean_type
  function isVirtual_inpg (tInPortGroup 'a) : boolean_type
  function value_inpg (tInPortGroup 'a) : 'a

  axiom tInPortGroup_min_max_one: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = one -> pg.min_size_inpg = one

  axiom tInPortGroup_min_max_value: forall pg: tInPortGroup 'a.
    pg.min_size_inpg >= zero /\ pg.max_size_inpg >= zero

  axiom tInPortGroup_min_max_size: forall pg: tInPortGroup 'a.
    pg.min_size_inpg <= pg.max_size_inpg \/
    pg.max_size_inpg = zero

  function size_inpg (pg: tInPortGroup 'a) : int =
    if pg.max_size_inpg = zero then zero else
    if pg.max_size_inpg = one then one else
    pg.max_size_inpg - pg.min_size_inpg

  lemma size_inpg_max_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = zero -> size_inpg pg = zero

  lemma size_inpg_min_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg <> zero /\ pg.min_size_inpg = zero ->
    size_inpg pg = pg.max_size_inpg

  lemma size_inpg_max_one: forall pg: tInPortGroup 'a.
    pg.max_size_inpg = one -> size_inpg pg = one

  lemma size_inpg_min_non_zero: forall pg: tInPortGroup 'a.
    pg.max_size_inpg > one /\ pg.min_size_inpg <> zero ->
    size_inpg pg = pg.max_size_inpg - pg.min_size_inpg
end
```

Listing D.12: Input port group theory definition using Why3

Proof obligations	Alt-Ergo-Pro (1.0.0)	CVC4 (1.3)	Simplify (1.5.4)	Spass (3.7)	Z3 (4.3.1)
lemma size_inpg_max_zero	0.02				
lemma size_inpg_min_zero	0.02				
lemma size_inpg_max_one	0.01				
lemma size_inpg_min_non_zero	0.03	(2s)	(2s)	0.08	(2s)

Figure D.13: InPortGroup theory lemmas verification with WHY3 and SMT solvers

```
theory OutPortGroup
  use import blocklibrary_string.String
  use import blocklibrary_scalar.Boolean
  use import int.Int

  type tOutPortGroup 'a

  function name_outpg (tOutPortGroup 'a) : string_type
  function min_size_outpg (tOutPortGroup 'a) : int
  function max_size_outpg (tOutPortGroup 'a) : int
  function isDimensionalizable_outpg (tOutPortGroup 'a) : boolean_type
  function isVirtual_outpg (tOutPortGroup 'a) : boolean_type
  function value_outpg (tOutPortGroup 'a) : 'a

  axiom tOutPortGroup_min_max_one: forall pg: tOutPortGroup 'a.
    pg.max_size_outpg = one /\ pg.min_size_outpg = one
end
```

Listing D.14: Output port group theory definition using WHY3

```
theory Parameter
  use import blocklibrary_string.String
  use import blocklibrary_scalar.Boolean
  use import list.List
  use import list.Length

  type tParameter 'a = {
    name_pt : string_type ;
    isMandatory_pt : boolean_type ;
    isDimensionalizable_pt : boolean_type ;
    value_pt : 'a
  }
end
```

Listing D.15: Parameter theory definition using WHY3

```
theory MemoryVariable
  use import blocklibrary_string.String
  use import int.Int

  type tMemoryVariable 'a = {
    name_mv : string_type ;
    value_mv : 'a
  }
end
```

Listing D.16: MemoryVariable theory definition using WHY3

D.3 Generic functions definitions and general purpose lemmas

In order to simplify the code generated with our code generation, we decided to introduce the list getter function. This function has been defined on the Why3 arrays standard library module. We have decided to define it here on a theory in order to use it in the OCL predicates and functions definitions.

```
theory CommonFunctions

  use import int.Int
  use import list.List
  use import list.Mem
  use import list.NthNoOpt

  (* === Getter on lists === *)
  function ([]) (a: list 'a) (i: int) : 'a = nth i a
```

Listing D.17: Getter function definition

In order to ease both manual and automatic proof of OCL functions lemmas, we introduced some additional lemmas on some Why3 standard library functions. We provide here their formalisation. The following lemmas are verified by relying on the Alt-Ergo theorem prover (version 1.0.0) if not said otherwise. The report provided by the Why3 tool is provided in Figure D.22.

```
  (* predicate mem (x: 'a) (l: list 'a) *)
  lemma mem_nil: forall b: 'a.
    mem b Nil = false

  lemma mem_cons: forall l: list 'a, b,c: 'a.
    mem b (Cons c l) = (b = c \/ mem b l)

  lemma mem_cons_mem: forall l: list 'a, b: 'a.
    mem b (Cons b l)

  lemma mem_present_not_nil: forall l: list 'a, b: 'a.
    mem b l -> l <> Nil

  lemma mem_absent_cons: forall l: list 'a, a,b: 'a.
    not (mem a (Cons b l)) -> a <> b /\ not (mem a l)

  lemma mem_decidable_presence: forall l: list 'a, a: 'a.
    mem a l \/ not (mem a l)

  lemma mem_list_not_mem_and_not_eq: forall l: list 'a, e,f: 'a.
    (not (mem e l) /\ e <> f) -> not (mem e (Cons f l))
```

Listing D.18: *Mem* predicate additional lemma

```
  (* function length (l: list 'a) : int *)
  use import list.Length

  lemma length_cons: forall l: list 'a, b: 'a.
    length (Cons b l) = length l + 1

  lemma length_gt_0: forall l: list 'a.
    length l > 0 -> l <> Nil

  lemma length_positive: forall l: list 'a.
    length l >= 0

  lemma length_nil: forall l: list 'a.
    l = Nil -> length l = 0

  (* Proof done using Z3 v4.3.1 *)
  lemma length_destruct: forall l: list 'a.
    length l > 0 -> exists l2: list 'a, b: 'a. l = Cons b l2
```

Listing D.19: *List* additional lemmas

```
  (* type option = Some 'a | None *)
  use import option.Option

  lemma option_decidable: forall e: 'a.
    Some e <> None
```

Listing D.20: *Option* type additional lemma

```
  use import HighOrd as HO

  lemma ho_decidable_prop: forall p: HO.pred 'a, a: 'a.
    (p a = true) \/ (p a = false)
end
```

Listing D.21: High order predicate additional lemma

Proof obligations	Alt-Ergo-Pro (1.0.0)	Z3 (4.3.1)
lemma mem_nil	0.00	
lemma mem_cons	0.00	
lemma mem_cons_mem	0.00	
lemma mem_present_not_nil	0.00	
lemma mem_absent_cons	0.00	
lemma mem_decidable_presence	0.00	
lemma mem_list_not_mem_and_not_eq	0.00	
lemma length_cons	0.00	
lemma length_gt_0	0.00	
lemma length_positive	0.00	
lemma length_nil	0.00	
lemma length_destruct	0.03	0.31
lemma option_decidable	0.00	
lemma ho_decidable_prop	0.00	

Figure D.22: CommonFunctions theory lemmas verification with WHY3 and SMT solvers

D.4 OCL LANGUAGE OPERATIONS DEFINITIONS

We have defined an abstract type named *oclType*. This type is meant to gather the general properties to be defined on any type handled in our implementation of the OCL. We decided to grant the *oclType* inhabitants with an *excluding middle* axiom that was necessary in order to prove some of the lemmas expressed on our implementation of the OCL predicates and functions. In addition to the *excluding middle* axiom we add the commutativity of the difference operator for any inhabitant of the *oclType*. This lemma is proven by relying on the ALT-ERGO SMT solver. A report has been generated with the WHY3 tool containing the name of the prover used for a successful proof of each one of the following lemmas. This report is provided in Figure D.24.

```
theory OclType
  type oclType

  axiom oclType_decidable_equality: forall a,b: oclType.
    a = b \/ a <> b

  lemma oclType_diff_comm: forall a,b: oclType.
    a <> b <-> b <> a
end
```

Listing D.23: *oclType* definition and related lemma

Proof obligations	Alt-Ergo-Pro (1.0.0)
lemma oclType_diff_comm	0.00

Figure D.24: *OclType* theory lemma verification with WHY3 and SMT solvers

In the following we provide both implementation and lemmas for each OCL operation handled in our transformation from the BlockLibrary language to the Why language. Each lemma is verified once again by relying on the Why3 toolset. For some lemmas it was necessary to provide a Coq proof as SMT solvers were not able to discharge the generated proof obligations. The Why3 report on the verification is provided in Figures D.34, D.37, and D.47.

```
theory OCLCollectionOperation
  use import int.Int
  use import bool.Bool
  use import real.RealInfix
  use import list.List
  use import list.Length
  use import list.Mem
  use import list.NthNoOpt
  use import list.HdTlNoOpt
  use import option.Option
  use import HighOrd as HO
  use import blocklibrary_common.CommonFunctions
```

Listing D.25: OCL operations theory imports section

```
(* count l a
  Freely inspired from standard List.NumOcc.
  Returns the number of occurrences of a in l. *)

  function count (l: list oclType) (b: oclType) : int =
    match l with
    | Nil -> 0
    | Cons hd tl -> (if hd = b then 1 else 0) + count tl b
    end

  lemma count_nil: forall l: list oclType. forall b: oclType.
    l = Nil -> count l b = 0

  lemma count_cons_one: forall l: list oclType, b: oclType.
    count (Cons b l) b = 1 + count l b

  lemma count_cons_first: forall l: list oclType, b,c: oclType.
    count (Cons b l) c = (if b = c then 1 else 0) + count l c
```

Listing D.26: *count* function definition

```
(* list_mem l2 l1
  Returns wether all l2 elements are part of l *)

  predicate list_mem (l2: list oclType) (l: list oclType) =
    forall i: int. 0 <= i < length l2 /\ mem l2[i] l

  lemma list_mem_cons_one: forall l: list oclType, b: oclType.
    list_mem (Cons b Nil) l -> mem b l

  lemma list_mem_true: forall l1 l2: list oclType.
    list_mem l1 l2 -> (forall i: int. 0 <= i < length l1 -> mem l1[i] l2)

  lemma list_mem_false: forall l1 l2: list oclType.
    (exists i: int. 0 <= i < length l1 /\ not (mem l1[i] l2)) ->
    not (list_mem l1 l2)
```

Listing D.27: *list_mem* predicate definition

```
(* list_not_mem l2 l1
   Returns wether none of l2 elements are part of l1 *)

  predicate list_not_mem (l2: list oclType) (l: list oclType) =
    forall i: int. 0 <= i < length l2 /\ not (mem l2[i] l)

  lemma list_not_mem_true: forall l1 l2: list oclType.
    list_not_mem l1 l2 ->
    (forall i: int. 0 <= i < length l1 -> not (mem l1[i] l2))

  lemma list_not_mem_false: forall l1 l2: list oclType.
    (exists i: int. 0 <= i < length l1 /\ mem l1[i] l2) ->
```

```
    not (list_not_mem l1 l2)
```

Listing D.28: *list_not_mem* predicate definition

```
(* append l1 l2
   Returns the result of appending l2 at the end of l1.
   Freely inspired from standard List.Append *)

  function append (l1 l2: list oclType) : list oclType = l1 ++ l2

  lemma append_assoc: forall l1 l2 l3: list oclType.
    append l1 (append l2 l3) = append (append l1 l2) l3

  lemma append_cons: forall l1 l2: list oclType, b: oclType.
    Cons b (append l1 l2) = append (Cons b l1) l2

  lemma append_l_nil: forall l: list oclType.
    append l Nil = l

  lemma append_r_nil: forall l: list oclType.
    append Nil l = l

  lemma append_length: forall l1 l2: list oclType.
    length (append l1 l2) = length l1 + length l2

  lemma append_mem: forall x: oclType, l1 l2: list oclType.
    mem x (append l1 l2) <-> (mem x l1 \/ mem x l2)
```

Listing D.29: *append* function definition

```
(* Exclude element from a list *)
  function excluding (l: list oclType) (e: oclType) : list oclType =
    match l with
    | Nil -> Nil
    | Cons hd tl -> if (hd = e) then excluding tl e
                                else Cons hd (excluding tl e)
    end

  lemma excluding_nil: forall elem: oclType.
    excluding Nil elem = Nil

  lemma excluding_cons_one: forall e,f: oclType.
    excluding (Cons f Nil) e = if (e=f) then Nil else Cons f Nil

  lemma excluding_cons_present: forall l: list oclType, e: oclType.
    excluding (Cons e l) e = excluding l e

  lemma excluding_cons_absent: forall l: list oclType, e,f: oclType.
    e <> f -> excluding (Cons f l) e = Cons f (excluding l e)

  lemma excluding_result: forall l: list oclType, elem: oclType.
    not ( mem elem (excluding l elem))

(* Coq proof script
intros. elim l. rewrite excluding_nil. apply mem_nil. intros.
cut ((elem=a) \/ (elem<>a)). intro. elim H0.
intro. rewrite <- H1. rewrite excluding_cons_present. exact H.
intro. rewrite excluding_cons_absent. rewrite mem_cons. intro. elim H2.
intro. absurd (elem = a). exact H1. exact H3. exact H. exact H1.
apply oclType_decidable_equality.
*)
```

Listing D.30: *excluding* function definition

```
(* IndexOf l e
   Return the index of e in l.
   In OCL specification, it is not possible that elem is not in the list,
here we specify an error code (indexOf returns zero) *)

  function indexOf (l: list oclType) (e: oclType) : int =
    if (not mem e l) then 0 else
    match l with
    | Nil -> 0
    | Cons hd tl -> if e = hd then 1 else 1 + indexOf tl e
```

```
  end

lemma indexOf_not_present: forall l: list oclType, elem: oclType.
  not mem elem l -> indexOf l elem = 0

lemma indexOf_first: forall l: list oclType, elem: oclType.
  (indexOf (Cons elem l) elem) = 1

lemma indexOf_nth: forall l: list oclType, e1 e2: oclType.
  e1 <> e2 /\ mem e2 l -> (indexOf (Cons e1 l) e2 = (1 + indexOf l e2))
```

Listing D.31: *indexOf* function definition

```
(* insertAt l e i
   Returns l with element e inserted at index i *)

function insertAt (l: list oclType) (e: oclType) (i: int) : list oclType =
  if (i < 0) then (Cons e l) else
  if (i >= length l) then append l (Cons e Nil) else
  match l with
  | Nil -> Cons e Nil
  | Cons hd tl -> if i = 0 then Cons e l
                  else Cons hd (insertAt tl e (i-1))
  end

lemma insertAt_negative: forall l: list oclType, a: oclType, i: int.
  i < 0 -> insertAt l a i = (Cons a l)

lemma insertAt_nil: forall a: oclType, i: int.
  insertAt Nil a i = Cons a Nil

lemma insertAt_cons_0: forall l: list oclType, a,b: oclType.
  insertAt (Cons a l) b 0 = Cons b (Cons a l)

lemma insertAt_0: forall l: list oclType, a: oclType.
  insertAt l a 0 = Cons a l

lemma insertAt_cons_n: forall l: list oclType, a,b: oclType, i: int.
  i > 0 -> insertAt (Cons a l) b i = Cons a (insertAt l b (i-1))

lemma insertAt_outofbound: forall l: list oclType, elem: oclType, i: int.
  i >= length l -> insertAt l elem i = append l (Cons elem Nil)

lemma insertAt_outofbound_same: forall l: list oclType, a: oclType, i: int.
  i > length l -> (insertAt l a i = insertAt l a (i-1))
```

Listing D.32: *insertAt* function definition

```
(* setAt l e i
   Returns l with element at index i replaced by e *)

function setAt (l: list oclType) (e: oclType) (i: int) : list oclType =
  if (i < 0) then l else
  if (i >= length l) then l else
  match l with
  | Nil -> Nil
  | Cons hd tl -> if i=0 then Cons e tl
                  else Cons hd (setAt tl e (i-1))
  end

lemma setAt_negative: forall l: list oclType, e: oclType, i: int.
  i < 0 -> setAt l e i = l

lemma setAt_nil: forall e: oclType, i: int.
  setAt Nil e i = Nil

lemma setAt_cons_0: forall l: list oclType, e,b: oclType.
  setAt (Cons b l) e 0 = Cons e l

lemma setAt_outofbound: forall l: list oclType, e: oclType, i: int.
  i >= length l -> setAt l e i = l

lemma setAt_cons_n: forall l: list oclType, e,b: oclType, i: int.
```

```
    0 < i < length l -> setAt (Cons b l) e i = Cons b (setAt l e (i-1))
```

Listing D.33: *setAt* function definition

```
(* intersection l1 l2
   Returns the intersection of l1 and l2 (elements in both l1 and l2) *)

  function intersection (l1 l2: list oclType) : list oclType =
    match l1,l2 with
    | Nil , _ -> Nil
    | _ , Nil -> Nil
    | (Cons hd tl), _ -> if mem hd l2 then Cons hd (intersection tl l2)
                                      else intersection tl l2
    end

  lemma intersection_l_nil: forall l: list oclType.
    intersection l Nil = Nil

  lemma intersection_r_nil: forall l: list oclType.
    intersection Nil l = Nil

  lemma intersection_cons_nil_cons: forall l: list oclType, e: oclType.
    intersection (Cons e Nil) (Cons e l) = Cons e Nil

  lemma intersection_cons_cons_nil: forall l: list oclType, e: oclType.
    intersection (Cons e l) (Cons e Nil) = Cons e (intersection l (Cons e Nil))

  lemma intersection_cons_nil_cons_nil_same: forall e: oclType.
    intersection (Cons e Nil) (Cons e Nil) = Cons e Nil

  lemma intersection_cons_nil_cons_nil_diff: forall e,f: oclType.
    e <> f -> intersection (Cons e Nil) (Cons f Nil) = Nil

  lemma intersection_cons_cons_nil_diff: forall l: list oclType, e,f: oclType.
    e <> f -> intersection (Cons e l) (Cons f Nil) = intersection l (Cons f Nil)

  lemma intersection_cons_nil_cons_diff: forall l: list oclType, e,f: oclType.
    e <> f -> intersection (Cons e Nil) (Cons f l) = intersection (Cons e Nil) l

  lemma intersection_cons_nil_cons_l_diff: forall l: list oclType, e,f: oclType.
    e <> f /\ not (mem e l) -> intersection (Cons e Nil) (Cons f l) = Nil

  lemma intersection_cons_nil_l_not_mem: forall l: list oclType, e: oclType.
    not (mem e l) -> intersection (Cons e Nil) l = Nil

  lemma intersection_l_cons_nil_not_mem: forall l: list oclType, e: oclType.
    not (mem e l) -> intersection l (Cons e Nil) = Nil

(* Coq proof script
intros l e h1.
induction l. rewrite intersection_r_nil. trivial.
cut (a=e \/ a<>e).
intro H1. elim H1. intro H2. rewrite H2 in h1. contradiction h1. apply mem_cons_mem.
intro. rewrite intersection_cons_cons_nil_diff. rewrite IHl. trivial.
intro. contradiction h1. rewrite mem_cons. right. apply H0. apply H.
apply oclType_decidable_equality.
*)

  lemma intersection_cons_0_r_diff: forall l: list oclType, e,f: oclType.
    (e <> f /\ not (mem f l)) -> intersection (Cons e l) (Cons f Nil) = Nil

  lemma intersection_mem_cons_0_l: forall l: list oclType, e: oclType.
    mem e l -> intersection (Cons e Nil) l = Cons e Nil

(* Coq proof script
intros l e h1.
induction l. contradiction. cut (a=e \/ a <> e). intro. elim H. intro.
rewrite H0. rewrite intersection_cons_nil_cons. trivial.
intro. rewrite intersection_cons_nil_cons_diff. rewrite IHl. trivial.
rewrite mem_cons in h1. destruct h1. rewrite H1 in H0. contradiction H0. trivial. exact H1.
apply oclType_diff_comm. exact H0. apply oclType_decidable_equality.
*)
```

Proof obligations	Alt-Ergo-Pro (1.0.0)	CVC4 (1.3)	Coq (8.4pl3)	Simplify (1.5.4)	Spass (3.7)	Z3 (4.3.1)
lemma count_nil	0.00					
lemma count_cons_one	0.00					
lemma count_cons_first	0.01					
lemma list_mem_cons_one	0.01					
lemma list_mem_true	0.00					
lemma list_mem_false	0.01					
lemma list_not_mem_true	0.01					
lemma list_not_mem_false	0.01					
lemma append_assoc	0.01					
lemma append_cons	0.01					
lemma append_l_nil	0.00					
lemma append_r_nil	0.01					
lemma append_length	0.01					
lemma append_mem	0.01					
lemma excluding_nil	0.01					
lemma excluding_cons_one	0.01					
lemma excluding_cons_present	0.02					
lemma excluding_cons_absent	0.01					
lemma excluding_result			1.99			
lemma indexOf_not_present	0.01					
lemma indexOf_first	0.01					
lemma indexOf_nth	0.02					
lemma insertAt_negative	0.01					
lemma insertAt_nil	0.01					
lemma insertAt_cons_0	0.01					
lemma insertAt_0	0.03		1.98			
lemma insertAt_cons_n	0.01					
lemma insertAt_outofbound	0.02					
lemma insertAt_outofbound_same	0.02					
lemma setAt_negative	0.02					
lemma setAt_nil	0.01					
lemma setAt_cons_0	0.02					
lemma setAt_outofbound	0.01					
lemma setAt_cons_n	0.03					

Figure D.34: OCL operations theory lemmas verification with WHY3 and SMT solvers

```
  lemma intersection_l_mem_cons_0: forall l: list oclType, e: oclType.
    mem e l -> mem e (intersection l (Cons e Nil))

(* Coq proof script
intros l e h1.
induction l. apply mem_nil in h1. contradiction.
cut (a=e \/ a<>e). intro H0. elim H0.
intro. rewrite H. rewrite intersection_cons_cons_nil. apply mem_cons_mem.
intro. rewrite intersection_cons_cons_nil_diff. apply IHl.
rewrite mem_cons in h1. elim h1. intro. apply oclType_diff_comm in H. contradiction. trivial.
exact H. apply oclType_decidable_equality.
*)

  lemma intersection_cons_cons_same: forall l1,l2: list oclType, e: oclType.
    intersection (Cons e l1) (Cons e l2) = Cons e (intersection l1 (Cons e l2))

  lemma intersection_cons_cons_diff_mem_l: forall l1,l2: list oclType, e,f: oclType.
    (mem e l2 /\ e <> f) ->
    intersection (Cons e l1) (Cons f l2) = Cons e (intersection l1 (Cons f l2))

  lemma intersection_cons_cons_diff_not_mem_l: forall l1,l2: list oclType, e,f: oclType.
    (not (mem e l2) /\ e <> f) ->
    intersection (Cons e l1) (Cons f l2) = intersection l1 (Cons f l2)

  lemma intersection_mem_l: forall l,l2: list oclType, e: oclType.
    mem e l -> mem e (intersection (Cons e l2) l)

(* Coq proof script
intros l l2 e h1.
induction l. apply mem_nil in h1. contradiction.
cut (a=e \/ a<>e). intro H0. elim H0. intro H1.
rewrite H1. rewrite intersection_cons_cons_same. apply mem_cons_mem.
intro. rewrite mem_cons in h1. elim h1.
intro. rewrite oclType_diff_comm in H. contradiction.
intro. rewrite oclType_diff_comm in H. rewrite intersection_cons_cons_diff_mem_l.
apply mem_cons_mem. split. exact H1. exact H.
apply oclType_decidable_equality.
*)

  lemma intersection_mem_r: forall l,l2: list oclType, e: oclType.
    mem e l -> mem e (intersection l (Cons e l2))

(* Coq proof script
intros l l2 e h1.
induction l. apply mem_nil in h1. contradiction.
cut (a=e \/ a<>e). intro H0. elim H0. intro.
rewrite H. rewrite intersection_cons_cons_same. apply mem_cons_mem.
intro. rewrite mem_cons in h1. elim h1.
intro. rewrite <- H1. rewrite intersection_cons_cons_same. apply mem_cons_mem.
intro. cut (Mem.mem a l2 \/ (not (Mem.mem a l2))). intro H2. elim H2. intro.
rewrite intersection_cons_cons_diff_mem_l. rewrite mem_cons. right.
apply IHl. exact H1.
split. exact H3. exact H.
intro. rewrite intersection_cons_cons_diff_not_mem_l. apply IHl. exact H1.
split. exact H3. exact H.
apply mem_decidable_presence. apply oclType_decidable_equality.
*)

  lemma intersection_cons_not_mem_l: forall l1,l2: list oclType, e: oclType.
    not (mem e l2) -> (intersection (Cons e l1) l2 = intersection l1 l2)

(* Coq proof script
intros l1 l2 e h1.
induction l2. rewrite intersection_l_nil. rewrite intersection_l_nil. trivial.
apply mem_absent_cons in h1.
destruct h1. rewrite intersection_cons_cons_diff_not_mem_l. trivial.
split. exact H0. exact H.
*)

  lemma intersection_cons_mem_l: forall l1,l2: list oclType, e: oclType.
    mem e l2 -> (intersection (Cons e l1) l2 = Cons e (intersection l1 l2))

(* Coq proof script
intros l1 l2 e h1.
```

```
induction l2. apply mem_nil in h1. contradiction.
rewrite mem_cons in h1. destruct h1. rewrite H.
rewrite intersection_cons_cons_same. trivial.
cut (e=a \/ e<>a). intro H0. destruct H0. rewrite H0.
rewrite intersection_cons_cons_same. trivial.
rewrite intersection_cons_cons_diff_mem_l. trivial. split. exact H. exact H0.
apply oclType_decidable_equality.
*)

  lemma intersection_l_cons_not_mem: forall l1,l2: list oclType, e: oclType.
    not (mem e l1) -> (intersection l1 (Cons e l2) = intersection l1 l2)

(*
intros l1 l2 e h1.
induction l1. rewrite intersection_r_nil. rewrite intersection_r_nil. trivial.
apply mem_absent_cons in h1.
cut (Mem.mem a l2 \/ not (Mem.mem a l2)). intro H0. elim H0. intro.
destruct h1. rewrite intersection_cons_cons_diff_mem_l. apply IHl1 in H2.
rewrite H2. rewrite intersection_cons_mem_l. trivial. exact H. split. exact H.
apply oclType_diff_comm in H1. exact H1.
intro. destruct h1. rewrite intersection_cons_cons_diff_not_mem_l.
rewrite intersection_cons_not_mem_l. apply IHl1 in H2. rewrite <- H2. trivial.
exact H. split. exact H. rewrite oclType_diff_comm. exact H1. apply mem_decidable_presence.
*)

  lemma intersection_cons_l_reduc: forall l1, l2: list oclType, e,f: oclType.
    e <> f -> (mem e (intersection (Cons f l1) l2) -> mem e (intersection l1 l2))

  lemma intersection_cons_l_reduc_op: forall l1, l2: list oclType, e,f: oclType.
    e <> f -> (mem e (intersection l1 l2) -> mem e (intersection (Cons f l1) l2))

  lemma intersection_mem_r_op: forall l,l2: list oclType, e: oclType.
    mem e (intersection l (Cons e l2)) -> mem e l

(* Coq proof script
intros l l2 e h1.
induction l. rewrite intersection_r_nil in h1. exact h1.
cut (a=e \/ a<>e). intro H0. elim H0.
intro H1. rewrite H1. apply mem_cons_mem.
intro H1. rewrite mem_cons. right. apply IHl. apply intersection_cons_l_reduc in h1. exact h1.
apply oclType_diff_comm. exact H1. apply oclType_decidable_equality.
*)

  lemma intersection_not_mem_r: forall l,l2: list oclType, e: oclType.
    not (mem e l) -> not (mem e (intersection l (Cons e l2)))

  lemma intersection_cons_r_reduc: forall l1, l2: list oclType, e,f: oclType.
    e <> f -> (mem e (intersection l1 (Cons f l2)) -> mem e (intersection l1 l2))

(* Coq proof script
intros l1 l2 e f h1 h2.
induction l1. rewrite intersection_r_nil in h2. apply mem_nil in h2. contradiction.
cut (a=e \/ a<>e). intro H0. elim H0. intro. rewrite H.
cut (Mem.mem e l2 \/ not (Mem.mem e l2)). intro H1. elim H1.
intro. apply intersection_mem_l. exact H2.
intro. rewrite H in h2. rewrite intersection_cons_cons_diff_not_mem_l in h2. apply IHl1 in h2.
rewrite intersection_cons_not_mem_l. exact h2. exact H2.
split. exact H2. exact h1. apply mem_decidable_presence.
intro. apply intersection_cons_l_reduc_op. apply oclType_diff_comm. exact H. apply IHl1.
cut (Mem.mem f l1 \/ not (Mem.mem f l1)). intro H1. elim H1. intro H2.
apply oclType_diff_comm in H. apply intersection_cons_l_reduc in h2. exact h2. exact H.
intro. apply intersection_cons_l_reduc in h2. exact h2. apply oclType_diff_comm. exact H.
apply mem_decidable_presence. apply oclType_decidable_equality.
*)

  lemma intersection_mem_l_op: forall l,l2: list oclType, e: oclType.
    mem e (intersection (Cons e l2) l) -> mem e l

(* Coq proof script
intros l l2 e h1.
induction l. rewrite intersection_l_nil in h1. exact h1.
cut (a=e \/ a<>e). elim H1. intro H2. rewrite H2. apply mem_cons_mem.
intro. rewrite mem_cons. right.
apply IHl. apply (intersection_cons_r_reduc _ _ _ a).
```

```
apply oclType_diff_comm. exact H. exact h1.
apply oclType_decidable_equality.
*)

  lemma intersection_cons_r_reduc_op: forall l1, l2: list oclType, e,f: oclType.
    e <> f -> (mem e (intersection l1 l2) -> mem e (intersection l1 (Cons f l2)))

(* Coq proof script
intros l1 l2 e f h1 h2.
induction l1. rewrite intersection_r_nil in h2. apply mem_nil in h2. contradiction.
cut (e=a \/ e<>a). intro H1. elim H1. intro H2.
rewrite <- H2. cut (Mem.mem e l2 \/ not (Mem.mem e l2)). intro H3. elim H3. intro H4.
rewrite intersection_cons_cons_diff_mem_l. apply mem_cons_mem. split. exact H4. exact h1.
intro H4. rewrite <- H2 in h2. apply intersection_mem_l_op in h2. contradiction.
apply mem_decidable_presence.
intro. apply intersection_cons_l_reduc_op. exact H. apply IHl1.
apply intersection_cons_l_reduc in h2. exact h2. exact H.
apply oclType_decidable_equality.
*)

  lemma intersection_presence: forall l1 l2: list oclType, e: oclType.
    mem e l1 /\ mem e l2 <-> mem e (intersection l1 l2)

(* Coq proof script
intros l1 l2 e.
split. intro. destruct H.
induction l1. apply mem_nil in H. contradiction.
rewrite mem_cons in H. elim H.
intro H1. rewrite <- H1. apply intersection_mem_l. exact H0.
intro H1. cut (a=e \/ a<>e). intro H2. elim H2. intro.
rewrite H3. apply intersection_mem_l. exact H0.
intro. cut (Mem.mem a l2 \/ (not (Mem.mem a l2))).
intro H4. elim H4. intro.
apply intersection_cons_l_reduc_op. apply oclType_diff_comm. exact H3. apply IHl1. exact H1.
intro. apply intersection_cons_l_reduc_op. apply oclType_diff_comm.
exact H3. apply IHl1. exact H1.
apply mem_decidable_presence. apply oclType_decidable_equality.
intro. split. induction l1. rewrite intersection_r_nil in H. exact H.
cut (a=e \/ a<>e). intro H0. elim H0. intro. rewrite H1. apply mem_cons_mem.
intro. rewrite mem_cons. right.
apply oclType_diff_comm in H1.
apply IHl1.
apply (intersection_cons_l_reduc _ _ _ _ H1) in H. exact H.
apply oclType_decidable_equality. induction l2. rewrite intersection_l_nil in H. exact H.
cut (a=e \/ a<>e). intro H0. elim H0. intro. rewrite H1. apply mem_cons_mem.
intro. rewrite mem_cons. right.
apply oclType_diff_comm in H1.
apply intersection_cons_r_reduc in H. apply IHl2 in H. exact H. exact H1.
apply oclType_decidable_equality.
*)
```

Listing D.35: *intersection* function definition

```
(* union l1 l2
  Returns the union of two lists *)

  function union (l1 l2: list oclType) : list oclType = l1 ++ l2

  lemma union_presence: forall l1 l2: list oclType, e: oclType.
    mem e l1 \/ mem e l2 <-> mem e (union l1 l2)

  lemma union_empty: forall l1 l2: list oclType.
    (union l1 l2) = Nil <-> l1 = Nil /\ l2 = Nil
```

Listing D.36: *union* function definition

```
(* subList l lo up
   Returns the subset of elements of l between index lo and index up *)

  function subList (l: list oclType) (lo up:int) : list oclType =
    if lo >= length l \/ lo < 0 \/ up < 0 \/
       up >= length l \/ up < lo then Nil
    else match l with
```

Proof obligations	Alt-Ergo-Pro (1.0.0)	CVC4 (1.3)	Coq (8.4pl3)	Simplify (1.5.4)	Spass (3.7)	Z3 (4.3.1)
lemma intersection_l_nil					0.14	
lemma intersection_r_nil					0.14	
lemma intersection_cons_nil_cons						0.05
lemma intersection_cons_cons_nil	0.02					
lemma intersection_cons_nil_cons_nil_same	0.02					
lemma intersection_cons_nil_cons_nil_diff	0.06					
lemma intersection_cons_cons_nil_diff	0.07					
lemma intersection_cons_nil_cons_diff		0.35				
lemma intersection_cons_nil_cons_l_diff	0.04					
lemma intersection_cons_nil_l_not_mem					0.10	
lemma intersection_l_cons_nil_not_mem			2.18			
lemma intersection_cons_0_r_diff	0.03					
lemma intersection_mem_cons_0_l			2.20			
lemma intersection_l_mem_cons_0			1.87			
lemma intersection_cons_cons_same	0.02					
lemma intersection_cons_cons_diff_mem_l	0.05					
lemma intersection_cons_cons_diff_not_mem_l	0.06					
lemma intersection_mem_l			2.31			
lemma intersection_mem_r			1.94			
lemma intersection_cons_not_mem_l			2.25			
lemma intersection_cons_mem_l			2.22			
lemma intersection_l_cons_not_mem			2.21			
lemma intersection_cons_l_reduc	0.09					
lemma intersection_cons_l_reduc_op	0.05					
lemma intersection_mem_r_op			2.22			
lemma intersection_not_mem_r	0.02					
lemma intersection_cons_r_reduc			2.00			
lemma intersection_mem_l_op			2.26			
lemma intersection_cons_r_reduc_op			2.22			
lemma intersection_presence			2.38			
lemma union_presence	0.02					
lemma union_empty	0.03				0.26	

Figure D.37: OCL operations theory lemmas verification with WHY3 and SMT solvers (II)

```
        | Nil -> Nil
        | Cons hd tl ->
            if lo = 0 then
              if up = 0 then
                Cons hd Nil
              else Cons hd (subList tl 0 (up-1))
            else subList tl (lo-1) (up-1)
        end

  lemma subList_nil: forall l: list oclType, x,y: int.
    l = Nil -> (subList l x y = Nil)

  lemma subList_length_nil: forall l: list oclType, x,y: int.
    l = Nil -> length (subList l x y) = 0

  lemma subList_0_0: forall l: list oclType, c: oclType.
    subList (Cons c l) 0 0 = Cons c Nil

  lemma subList_length_0_0: forall l: list oclType, c: oclType.
    length (subList (Cons c l) 0 0) = 1

  lemma subList_0_x: forall l: list oclType, c: oclType, x: int.
    (0 <= x < length l) ->
    subList (Cons c l) 0 x = Cons c (subList l 0 (x-1))

  lemma subList_length_0_x: forall l: list oclType, c: oclType, x: int.
    (0 <= x < length l) ->
    length (subList (Cons c l) 0 x) = 1 + length (subList l 0 (x-1))

  lemma subList_x_y: forall l: list oclType, c: oclType, x,y: int.
    (0 < x <= y < length l) ->
    subList (Cons c l) x y = subList l (x-1) (y-1)

  lemma subList_length_x_y: forall l: list oclType, c: oclType, x,y: int.
    (0 < x <= y < length l) ->
    length (subList (Cons c l) x y) = length (subList l (x-1) (y-1))

  lemma subList_outofbound:  forall l: list oclType, lo up: int.
    (lo >= length l -> (subList l lo up) = Nil) /\
    (lo < 0 -> (subList l lo up) = Nil) /\
    (up < 0 -> (subList l lo up) = Nil) /\
    (up >= length l -> (subList l lo up) = Nil) /\
    (up < lo -> (subList l lo up) = Nil)
```

Listing D.38: *subList* function definition

```
(* sumReal l
   Returns the sum of the reals contained in l *)

  function sumReal (l: list real) : real =
    match l with
    | Nil -> 0.0
    | Cons hd tl -> hd +. sumReal tl
    end

  lemma sumReal_Value: forall l1: list real, e: real.
    sumReal (Cons e l1) = e +. sumReal l1

  lemma sumReal_Empty:
    sumReal Nil = 0.0
```

Listing D.39: *sumReal* function definition

```
(* sumInt l
   Returns the sum of the integers contained in l *)

  function sumInt (l: list int) : int =
    match l with
    | Nil -> 0
    | Cons hd tl -> hd + sumInt tl
    end

  lemma sumInt_Value: forall l1: list int, e: int.
    sumInt (Cons e l1) = e + sumInt l1
```

```
lemma sumInt_Empty:
  sumInt Nil = 0
```

Listing D.40: *sumInt* function definition

D.5 OCL ITERATION OPERATIONS DEFINITIONS

```
(* isUnique l f
   Returns true iff the image through f of every element of l is the same *)

predicate isUnique (l: list oclType) (f: HO.func oclType 'b) =
  forall i,j: int. (0 <= i < length l /\ 0 <= j < length l) -> f l[i] = f l[j]

lemma isUnique_res: forall l: list oclType, f: HO.func oclType 'b.
  isUnique l f -> (forall i j: int. (0 <= i < length l /\ 0 <= j < length l) ->
                                    f l[i] = f l[j])

lemma isUnique_nil: forall f: HO.func oclType 'b.
  isUnique Nil f
```

Listing D.41: *isUnique* predicate definition

```
(* anyAs l p
   Returns one element of l verifying p *)

function anyAs (l: list oclType) (p: HO.pred oclType) : option oclType =
  match l with
  | Nil -> None
  | Cons hd tl -> if p hd then Some hd else anyAs tl p
  end

lemma anyAs_nil: forall p: HO.pred oclType.
  anyAs Nil p = None

lemma anyAs_cons_hd: forall l: list oclType, p: HO.pred oclType, e: oclType.
  p e -> anyAs (Cons e l) p = Some e

lemma anyAs_cons_tl: forall l: list oclType, p: HO.pred oclType, e: oclType.
  not (p e) -> anyAs (Cons e l) p = anyAs l p
```

Listing D.42: *anyAs* predicate definition

```
(* select l p
   Returns all elements of l verifying p *)

function select (l: list oclType) (p: HO.pred oclType) : list oclType =
  match l with
  | Nil -> Nil
  | Cons hd tl -> if p hd then Cons hd (select tl p)
                          else select tl p
  end

lemma select_nil: forall p: HO.pred oclType.
  select Nil p = Nil

lemma select_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
  p e -> select (Cons e Nil) p = Cons e Nil

lemma select_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
  not (p e) -> select (Cons e Nil) p = Nil

lemma select_cons_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
  p e -> select (Cons e l) p = Cons e (select l p)

lemma select_cons_not_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
  not (p e) -> select (Cons e l) p = select l p

lemma select_mem_reduc: forall l: list oclType, b: oclType, p: HO.pred oclType.
  mem b (select l p) -> mem b l
```

```
(* Coq proof script
intros l b p h1.
induction l. rewrite select_nil in h1. exact h1.
rewrite mem_cons. cut (b=a \/ b<>a). intro H1. destruct H1.
left. exact H. right. apply IHl.
cut ((infix_at p a = true) \/ (infix_at p a = false)). intro H1. destruct H1.
rewrite select_cons_verified in h1. rewrite mem_cons in h1. destruct h1. contradiction.
exact H1. exact H0. rewrite select_cons_not_verified in h1. exact h1. rewrite H0. discriminate.
apply ho_decidable_prop. apply oclType_decidable_equality.
*)

  lemma select_mem: forall l: list oclType, b: oclType, p: HO.pred oclType.
    (mem b l /\ p b) -> mem b (select l p)

(* Coq proof script
intros l b p (h1,h2).
induction l. apply mem_nil in h1. contradiction.
cut ((infix_at p a = true) \/ (infix_at p a = false)). intro H0. destruct H0.
rewrite select_cons_verified. rewrite mem_cons. rewrite mem_cons in h1. destruct h1.
left. exact H0. right. apply IHl. exact H0. exact H.
rewrite select_cons_not_verified. rewrite mem_cons in h1. destruct h1. rewrite <- H0 in H.
rewrite H in h2. discriminate h2. apply IHl. exact H0. rewrite H. discriminate.
apply ho_decidable_prop.
*)

  lemma select_not_mem: forall l: list oclType, b: oclType, p: HO.pred oclType.
    (mem b l /\ not (p b)) -> not (mem b (select l p))

(* Coq proof script
intros l b p (h1,h2).
intro. induction l. apply mem_nil in h1. contradiction.
apply IHl. rewrite mem_cons in h1. destruct h1. rewrite <- H0 in H.
rewrite select_cons_not_verified in H.
apply select_mem_reduc in H. exact H. exact h2. exact H0.
rewrite mem_cons in h1. destruct h1. rewrite <- H0 in H.
rewrite select_cons_not_verified in H. exact H. exact h2.
cut ((infix_at p a = true)\/(infix_at p a = false)). intro H1. destruct H1.
rewrite select_cons_verified in H. rewrite mem_cons in H.
destruct H. rewrite H in h2. contradiction. exact H. exact H1.
rewrite select_cons_not_verified in H. exact H. intro. rewrite H2 in H1. discriminate.
apply ho_decidable_prop.
*)
```

Listing D.43: *select* predicate definition

```
(* reject l p
   Returns all elements of l not verifying p *)

  function reject (l: list oclType) (p: HO.pred oclType) : list oclType =
    match l with
    | Nil -> Nil
    | Cons hd tl -> if p hd then reject tl p
                            else Cons hd (reject tl p)
    end

  lemma reject_nil: forall p: HO.pred oclType.
    reject Nil p = Nil

  lemma reject_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
    p e -> reject (Cons e Nil) p = Nil

  lemma reject_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
    not (p e) -> reject (Cons e Nil) p = Cons e Nil

  lemma reject_cons_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
    p e -> reject (Cons e l) p = reject l p

  lemma reject_cons_not_verified: forall e: oclType, l: list oclType, p: HO.pred oclType.
    not (p e) -> reject (Cons e l) p = Cons e (reject l p)

  lemma reject_mem_reduc: forall l: list oclType, b: oclType, p: HO.pred oclType.
    mem b (reject l p) -> mem b l

(*Coq proof script
```

```
intros l b p h1.
induction l. rewrite reject_nil in h1. exact h1.
rewrite mem_cons. cut (b=a \/ b<>a). intro H1. destruct H1.
left. exact H. right. apply IHl.
cut ((infix_at p a = true) \/ (infix_at p a = false)). intro H1. destruct H1.
rewrite reject_cons_verified in h1. exact h1. exact H0.
rewrite reject_cons_not_verified in h1. rewrite mem_cons in h1. destruct h1. contradiction.
exact H1. rewrite H0. discriminate. apply ho_decidable_prop. apply oclType_decidable_equality.
*)

  lemma reject_verified_diff: forall l: list oclType, a,b: oclType, p: HO.pred oclType.
    a <> b -> mem a (reject (Cons b l) p) = mem a (reject l p)

  lemma reject_verified: forall l: list oclType, a: oclType, p: HO.pred oclType.
    p a -> not (mem a (reject l p))

(* Coq proof script
intros l a p h1.
induction l. intro. rewrite reject_nil in H. apply mem_nil in H. contradiction.
intro. cut (a0=a \/ a0<>a). intro H1. destruct H1. rewrite H0 in H.
rewrite reject_cons_verified in H. contradiction. exact h1.
rewrite reject_cons_not_verified in H. rewrite mem_cons in H. destruct H.
rewrite H in H0. auto. contradiction.
apply (reject_verified_diff l a a0) in H. contradiction.
intuition. apply oclType_decidable_equality.
*)

  lemma reject_mem: forall l: list oclType, b: oclType, p: HO.pred oclType.
    (mem b l /\ p b) -> not (mem b (reject l p))
```

Listing D.44: *reject* predicate definition

```
(* oneFrom l p
   Returns true iff only one element of l verifies p *)

  predicate oneFrom (l: list oclType) (p: HO.pred oclType) =
    match l with
    | Nil -> false
    | _ -> length (select l p) = 1
    end

  lemma oneFrom_nil: forall p: HO.pred oclType.
    not (oneFrom Nil p)

  lemma oneFrom_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
    p e <-> oneFrom (Cons e Nil) p

  lemma oneFrom_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
    not (p e) <-> not (oneFrom (Cons e Nil) p)

  lemma oneFrom_cons_destruct_verified: forall l: list oclType, e: oclType,
                                                p: HO.pred oclType.
    oneFrom (Cons e l) p -> (oneFrom (Cons e Nil) p \/ oneFrom l p)

  lemma oneFrom_cons_first_not_verified: forall l: list oclType, e: oclType,
                                                 p: HO.pred oclType.
    (not (p e) /\ oneFrom (Cons e l) p) -> oneFrom l p
```

Listing D.45: *oneFrom* predicate definition

```
(* collect l f
   Returns the image through f of all elements of l *)

  function collect (l: list oclType) (f: HO.func oclType 'b) : list 'b =
    match l with
    | Nil -> Nil
    | Cons hd tl -> Cons (f hd) (collect tl f)
    end

  lemma collect_nil: forall f: HO.func oclType oclType.
    collect Nil f = Nil

  lemma collect_cons_nil: forall e: oclType, f: HO.func oclType oclType.
    collect (Cons e Nil) f = Cons (f e) Nil
```

```
  lemma collect_cons: forall l: list oclType, e: oclType, f: HO.func oclType oclType.
    collect (Cons e l) f = Cons (f e) (collect l f)
end
```

Listing D.46: *collect* predicate definition

Proof obligations	Alt-Ergo-Pro (1.0.0)	CVC4 (1.3)	Coq (8.4pl3)	Simplify (1.5.4)	Spass (3.7)	Z3 (4.3.1)
lemma subList_nil		0.07			0.09	0.02
lemma subList_length_nil		0.07			0.07	0.02
lemma subList_0_0		0.29			0.15	
lemma subList_length_0_0		0.10			0.07	0.05
lemma subList_0_x		0.12				
lemma subList_length_0_x		0.51			0.11	
lemma subList_x_y		0.11				
lemma subList_length_x_y		0.08			0.11	
lemma subList_outofbound		0.11			0.11	0.03
lemma sumReal_Value	0.03	0.09			0.08	0.03
lemma sumReal_Empty	0.03	0.04			0.08	0.00
lemma sumInt_Value	0.04	0.09			0.08	0.03
lemma sumInt_Empty	0.04	0.05			0.08	0.00
lemma isUnique_res	0.03	0.09			0.13	0.02
lemma isUnique_nil	0.04	0.08			0.14	0.04
lemma anyAs_nil	0.03	0.08			0.07	0.03
lemma anyAs_cons_hd	0.03					
lemma anyAs_cons_tl	0.03					
lemma select_nil	0.03					
lemma select_cons_nil_verified	0.04					
lemma select_cons_nil_not_verified	0.05					
lemma select_cons_verified	0.03					
lemma select_cons_not_verified	0.04					
lemma select_mem_reduc			2.40			
lemma select_mem			2.31			
lemma select_not_mem			2.01			
lemma oneFrom_nil	0.04					
lemma oneFrom_cons_nil_verified	0.04					
lemma oneFrom_cons_nil_not_verified	0.04					
lemma oneFrom_cons_destruct_verified	0.09					
lemma oneFrom_cons_first_not_verified	0.05					
lemma reject_nil	0.04					
lemma reject_cons_nil_verified	0.04					
lemma reject_cons_nil_not_verified	0.04					
lemma reject_cons_verified	0.05					
lemma reject_cons_not_verified	0.04					
lemma reject_mem_reduc			2.43			
lemma reject_verified_diff	0.10					
lemma reject_verified			2.12			
lemma reject_mem	0.04					
lemma collect_nil	0.05					
lemma collect_cons_nil	0.04					
lemma collect_cons	0.05					

Figure D.47: OCL operations theory lemmas verification with WHY3 and SMT solvers (III)

E

ACSL verification using FRAMA-C

We provide in this appendix the C code generated by the GENEAUTO code generator from the *Counter* system with a synchronous observer as presented in Chapter 8.

```
#ifndef __GATypes__
#define __GATypes__
/* Named Constants */
#define FALSE 0
#define TRUE 1

/* Type declarations */
typedef char INT8;
typedef unsigned char UINT8;
typedef short INT16;
typedef unsigned short UINT16;
typedef int INT32;
typedef unsigned int UINT32;
typedef double REAL;
typedef float SINGLE;
typedef unsigned char BOOL;

/* Function-like macros */
#define TO_BOOL(X) ((X) ? 1 : 0)
#endif
```

Listing E.1: GATypes.h: GENEAUTO generated generic data types definition

```
#ifndef __Counter_types__
#define __Counter_types__
/* Includes */
#include "GATypes.h"

/* Type declarations */
typedef struct {
    BOOL reset;
    BOOL active;
} t_Counter_io;
typedef struct {
    BOOL UD1_memory;
    BOOL UD_memory;
} t_Counter_state;


/* Ghost type declarations */
/*@ ghost typedef struct {
    BOOL reset;
    BOOL active;
    BOOL safe;
} t_counter_spec_io;
```

```
*/
/*@ ghost typedef struct {
    UINT8 zero;
    UINT8 two;
    BOOL cptEq2;
    BOOL cptEq;
    BOOL orReset;
    UINT8 counter_spec_Switch;
    UINT8 counter_spec_Unit_Delay;
    UINT8 one;
    UINT8 Sum;
    UINT8 three;
    BOOL cptEqActive;
    UINT8 Unit_Delay_memory;
} t_counter_spec_loc;
*/
#endif
```

Listing E.2: Counter_types.h: GeneAuto generated model specific data types definition

```
#ifndef __Counter__
#define __Counter__
/* Includes */
#include "Counter_main.h"
#include "GATypes.h"
#include "Counter_types.h"

/* Variable Declarations */
extern BOOL Counter_LO1;
extern BOOL Counter_LO3;
extern BOOL Counter_UD;
extern BOOL Counter_UD1;

/* Function prototypes */
/*@ requires \valid(_state_);
    assigns _state_->UD1_memory,  _state_->UD_memory;
*/
extern void Counter_init(t_Counter_state *_state_);

/*@ requires \valid(_io_) && \valid(_state_);
    requires \separated(_io_, _state_);
    assigns _io_->active, _state_->UD_memory, _state_->UD1_memory,
            Counter_LO1, Counter_LO3, Counter_UD, Counter_UD1;
*/
extern void Counter_compute(t_Counter_io *_io_, t_Counter_state *_state_);
#endif
```

Listing E.3: Counter.h: GeneAuto generated function prototypes

```
/*  Counter.c
    Generated by Gene-Auto toolset ver 2.4.10
    (launcher GALauncherObservers)
    Generated on: 27/10/2014 11:27:16.873
    source model: Counter
    model version: 7.2 */
/* Includes */
#include "Counter.h"

/* Variable definitions */
BOOL Counter_LO1 = FALSE;
BOOL Counter_LO3 = FALSE;
BOOL Counter_UD = FALSE;
BOOL Counter_UD1 = FALSE;

/* Function definitions */
void Counter_init(t_Counter_state *_state_) {
    /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
   /*@ ensures _state_->UD1_memory == FALSE;
        assigns _state_->UD1_memory; */
    {
      _state_->UD1_memory = FALSE;
    }
    /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
    /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
```

```
   /*@ ensures _state_->UD_memory == FALSE;
       assigns _state_->UD_memory; */
   {
      _state_->UD_memory = FALSE;
   }
    /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
}

void Counter_compute(t_Counter_io *_io_, t_Counter_state *_state_) {
    BOOL reset;
    BOOL LO;
    BOOL LO2;
    BOOL LO4;
    /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
    /*@ ensures Counter_UD1 == _state_->UD1_memory;
       assigns Counter_UD1; */
   {
      Counter_UD1 = _state_->UD1_memory;
   }
    /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD1>  */
    /*  START Block: <SystemBlock: name=Counter>/<SourceBlock: name=reset>  */
   /*@ ensures reset == _io_->reset;
       assigns reset; */
   {
      reset = _io_->reset;
    }
    /*  END Block: <SystemBlock: name=Counter>/<SourceBlock: name=reset>  */
    /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO>  */
   /*@ ensures LO == !reset;
       assigns LO; */
   {
      LO = !reset;
    }
    /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO>  */
    /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO1>  */
   /*@ ensures Counter_LO1 == (Counter_UD1 && LO);
       assigns Counter_LO1; */
   {
      Counter_LO1 = Counter_UD1 && LO;
    }
    /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO1>  */
    /*  START Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
   /*@ ensures Counter_UD == _state_->UD_memory;
       assigns Counter_UD; */
   {
      Counter_UD = _state_->UD_memory;
    }
    /*  END Block: <SystemBlock: name=Counter>/<SequentialBlock: name=UD>  */
    /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO2>  */
   /*@ ensures LO2 == !Counter_UD;
       assigns LO2; */
   {
      LO2 = !Counter_UD;
    }
    /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO2>  */
    /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO3>  */
   /*@ ensures Counter_LO3 == (LO && LO2);
       assigns Counter_LO3; */
   {
      Counter_LO3 = LO && LO2;
    }
    /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO3>  */
    /*  START Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO4>  */
   /*@ ensures LO4 == (Counter_LO3 && Counter_LO1);
       assigns LO4; */
   {
      LO4 = Counter_LO3 && Counter_LO1;
    }
    /*  END Block: <SystemBlock: name=Counter>/<CombinatorialBlock: name=LO4>  */
    /*  START Block: <SystemBlock: name=Counter>/<SinkBlock: name=active>  */
   /*@ ensures _io_->active == LO4;
       assigns _io_->active; */
   {
      _io_->active = LO4;
```

```
    }
    /*  END Block: <SystemBlock: name=Counter>/<SinkBlock: name=active>  */
    /*  START Block memory write: <SystemBlock: name=Counter>/
                                  <SequentialBlock: name=UD>  */
   /*@ ensures _state_->UD_memory == Counter_LO1;
       assigns _state_->UD_memory; */
   {
      _state_->UD_memory = Counter_LO1;
    }
    /*  END Block memory write: <SystemBlock: name=Counter>/
                                <SequentialBlock: name=UD>  */
    /*  START Block memory write: <SystemBlock: name=Counter>/
                                  <SequentialBlock: name=UD1>  */
   /*@ ensures _state_->UD1_memory == Counter_LO3;
       assigns _state_->UD1_memory; */
   {
      _state_->UD1_memory = Counter_LO3;
    }
    /*  END Block memory write: <SystemBlock: name=Counter>/
                                <SequentialBlock: name=UD1>  */
}
```

Listing E.4: Counter.c: GeneAuto generated function for the Counter model

```
#ifndef __Counter_main__
#define __Counter_main__

/* Includes */
#include "Counter.h"
#include "Counter_types.h"
#include "GATypes.h"

/* Function prototypes */
void main(t_Counter_state *_state_, t_Counter_io *_io_);
#endif
```

Listing E.5: Counter_main.h: GeneAuto generated main function prototype

```
/*  Counter_main.c
    Generated by Gene-Auto toolset ver 2.4.10
    (launcher GALauncherObservers)
    Generated on: 27/10/2014 11:27:16.891
    source model: Counter
    model version: 7.2 */
/* Includes */
#include "Counter_main.h"

/* Ghost declarations variables */
//@ ghost t_counter_spec_io * _counter_spec_input;
//@ ghost t_counter_spec_loc * _counter_spec_state;

/* Observers Predicates definitions */
/*  START Block: <SystemBlock: name=Counter>/
                 <SystemBlock: name=counter_spec>  */
/*@ predicate counter_spec_init (t_counter_spec_loc *obsState) =
      obsState->Unit_Delay_memory == 0;
*/
/*@ predicate counter_spec_compute (t_counter_spec_io *obsInput,
                                    t_counter_spec_loc *obsState) =
    ((3 == obsState->Unit_Delay_memory) || obsInput->reset) ==>
    ((0 == 2) == obsInput->active) &&
    (!((3 == obsState->Unit_Delay_memory) || obsInput->reset)) ==>
    (((obsState->Unit_Delay_memory + 1) == 2) == obsInput->active);
*/
/*@ predicate counter_spec_update (t_counter_spec_io *obsInput,
                                   t_counter_spec_loc *obsState) =
      ((3 == obsState->Unit_Delay_memory) || obsInput->reset) ==>
      obsState->Unit_Delay_memory == 0
      &&
      (!((3 == obsState->Unit_Delay_memory) || obsInput->reset)) ==>
      obsState->Unit_Delay_memory == obsState->Unit_Delay_memory + 1 ;
*/
/*  END Block: <SystemBlock: name=Counter>/
               <SystemBlock: name=counter_spec>  */
```

```
/* Function definitions */
/*@ requires \valid(_state_) && \valid(_io_);
    requires \separated(_state_, _io_, _counter_spec_input, _counter_spec_state);
    requires \valid(_counter_spec_state) && \valid(_counter_spec_input); */
void main(t_Counter_state *_state_, t_Counter_io *_io_){
    Counter_init(_state_);
   //@ ghost _counter_spec_state->Unit_Delay_memory = 0;
   //@ assert counter_spec_init (_counter_spec_state);
    /*@ loop assigns _counter_spec_state->Unit_Delay_memory, _io_->active;
        loop assigns _state_->UD_memory, _state_->UD1_memory;
    */
    while (TRUE) {
        Counter_compute(_io_, _state_);
        //@ ghost _counter_spec_input->reset = _io_->reset;
        //@ ghost _counter_spec_input->active = _io_->active;
        /*@ ghost
          if ((3 == _counter_spec_state->Unit_Delay_memory) ||
              _counter_spec_input->reset)
            _counter_spec_state->Unit_Delay_memory = 0 ;
          else
            _counter_spec_state->Unit_Delay_memory =
              _counter_spec_state->Unit_Delay_memory + 1 ;
        */
        //@ assert counter_spec_compute (_counter_spec_input, _counter_spec_state);
        //@ assert counter_spec_update (_counter_spec_input, _counter_spec_state);
    }
}
```

Listing E.6: Counter_main.c: GeneAuto generated main function

```
------------------------------------------------------------------------------
--- Properties of Function 'Counter_init'
------------------------------------------------------------------------------

[  Valid  ] Post-condition (file Counter.c, line 28) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 35) at block
            by Wp.typed.
[  Valid  ] Pre-condition (file Counter.h, line 19)
            by Call Preconditions.
[  Valid  ] Assigns (file Counter.c, line 29) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 36) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.h, line 20)
            by Wp.typed.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior
            by Frama-C kernel.

------------------------------------------------------------------------------
--- Properties of Function 'Counter_compute'
------------------------------------------------------------------------------

[  Valid  ] Post-condition (file Counter.c, line 49) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 56) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 63) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 70) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 77) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 84) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 91) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 98) at block
            by Wp.typed.
```

```
[  Valid  ] Post-condition (file Counter.c, line 105) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 112) at block
            by Wp.typed.
[  Valid  ] Post-condition (file Counter.c, line 119) at block
            by Wp.typed.
[  Valid  ] Pre-condition (file Counter.h, line 24)
            by Call Preconditions.
[  Valid  ] Pre-condition (file Counter.h, line 25)
            by Call Preconditions.
[  Valid  ] Assigns (file Counter.c, line 50) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 57) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 64) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 71) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 78) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 85) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 92) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 99) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 106) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 113) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.c, line 120) at block
            by Wp.typed.
[  Valid  ] Assigns (file Counter.h, line 26)
            by Wp.typed.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior at block
            by Frama-C kernel.
[  Valid  ] Default behavior
            by Frama-C kernel.

------------------------------------------------------------------------------
--- Properties of Function 'main'
------------------------------------------------------------------------------

[    -    ] Pre-condition (file Counter_main.c, line 45)
            tried with Wp.typed.
[    -    ] Pre-condition (file Counter_main.c, line 46)
            tried with Wp.typed.
[    -    ] Pre-condition (file Counter_main.c, line 47)
            tried with Wp.typed.
[  Valid  ] Loop assigns (file Counter_main.c, line 58)
            by Wp.typed.
[  Valid  ] Assertion (file Counter_main.c, line 52)
            by Wp.typed.
[  Valid  ] Assertion (file Counter_main.c, line 71)
```

```
            by Wp.typed.
[  Valid  ] Assertion (file Counter_main.c, line 72)
            by Wp.typed.
[  Valid  ] Default behavior
            by Frama-C kernel.
[  Valid  ] Counter_compute_pre: Pre-condition (file Counter.h, line 24)
            at call 'Counter_compute' (file Counter_main.c, line 62)
            by Wp.typed.
[  Valid  ] Counter_compute_pre_2: Pre-condition (file Counter.h, line 25)
            at call 'Counter_compute' (file Counter_main.c, line 62)
            by Wp.typed.
[  Valid  ] Counter_init_pre: Pre-condition (file Counter.h, line 19)
            at call 'Counter_init' (file Counter_main.c, line 50)
            by Wp.typed.

------------------------------------------------------------------------------
--- Status Report Summary
------------------------------------------------------------------------------
    54 Completely validated
     3 To be validated
    57 Total
------------------------------------------------------------------------------
```

Listing E.7: FRAMA-C WP verification output.

Bibliography

[1] ACSL: ANSI/ISO C Specification Language. http://frama-c.com/download/acsl.pdf.

[2] Blocklibrary repository. http://block-library.enseeiht.fr/html.

[3] Why3 website at LRI. http://www.why3.lri.fr.

[4] Foundational subset for executable uml models (fuml) specification, v1.1. http://www.omg.org/spec/FUML/1.1/.

[5] Uml xmi serialisation format specification, v2.4.1. http://www.omg.org/spec/UML/2.4.1/.

[6] DO-331 model-based development and verification supplement to do-178c and do-278a. Technical report, RTCA & EUROCAE, December 2011.

[7] DO-332 object-oriented technology and related techniques supplement to do-178c and do-278a. Technical report, RTCA & EUROCAE, December 2011.

[8] DO-333 formal methods supplement to do-178c and do-278a. Technical report, RTCA & EUROCAE, December 2011.

[9] Mathieu Acher, Raphaël Michel, Patrick Heymans, Philippe Collet, and Philippe Lahire. Languages and tools for managing feature models. In Julia Rubin, Goetz Botterweck, Andreas Pleuss, and David M. Weiss, editors, *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2012, Zurich, Switzerland, June 4, 2012*, pages 25–28. ACM, 2012. ISBN 978-1-4673-1751-1. URL http://dl.acm.org/citation.cfm?id=2666071.

[10] Achilleas Achilleos, Nektarios Georgalas, and Kun Yang. An open source domain-specific tools framework to support model driven development of oss. In DavidH. Akehurst, Régis Vogel, and RichardF. Paige, editors, *Model Driven Architecture- Foundations and Applications*, volume 4530 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-72900-6. doi: 10.1007/978-3-540-72901-3_1. URL http://dx.doi.org/10.1007/978-3-540-72901-3_1.

[11] Duane Albert Adams. *A Computation Model with Data Flow Sequencing*. PhD thesis, Stanford, CA, USA, 1969. AAI6913919.

[12] A.E.Rugina and J.-C.Dalbin. Experiences with the gene-auto code generator in the aerospace industry. In *ERTS*, 2010.

[13] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, Lecture Notes in Computer Science. Springer-Verlag, 2014. To appear.

[14] Daniel Balasubramanian, Corina S. Păsăreanu, Michael W. Whalen, Gábor Karsai, and Michael Lowry. Polyglot: Modeling and analysis for multiple statechart formalisms. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 45–55, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001427. URL http://doi.acm.org/10.1145/2001420.2001427.

[15] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61937-6. doi: 10.1007/BFb0031808. URL http://dx.doi.org/10.1007/BFb0031808.

[16] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at www.SMT-LIB.org.

[17] M. Encarnación Beato, Manuel Barrio-Solórzano, Carlos E. Cuesta, and Pablo de la Fuente. Uml automatic verification tool with formal methods. *Electron. Notes Theor. Comput. Sci.*, 127(4):3–16, April 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.10.024. URL http://dx.doi.org/10.1016/j.entcs.2004.10.024.

[18] Michael Beine, Rainer Otterbach, and Michael Jungmann. Development of safety-critical software using automatic code generation. Technical report, SAE Technical Paper, 2004.

[19] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26095-0. doi: 10.1007/11431855_34. URL http://dx.doi.org/10.1007/11431855_34.

[20] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. ISSN 0306-4379. doi: http://dx.doi.org/10.1016/j.is.2010.01.001.

[21] Julien Bertrane, Patrick Cousot, Radhia Cousot, Laurent Mauborgne Jérôme Feret, Antoine Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*, number AIAA-2010-3385, pages 1–38. American Institue of Aeronautics and Astronautics, April 2010.

[22] Loïc Besnard, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Compilation of polychronous data flow equations. In Sandeep K. Shukla and Jean-Pierre Talpin, editors, *Synthesis of Embedded Software*, pages 1–40. Springer US, 2010. ISBN 978-1-4419-6399-4. doi: 10.1007/978-1-4419-6400-7_1. URL http://dx.doi.org/10.1007/978-1-4419-6400-7_1.

[23] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation of synchronous data-flow languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tuscon, Arizona, June 2008.

[24] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 43(7):121–130, June 2008. ISSN 0362-1340. doi: 10.1145/1379023.1375674. URL http://doi.acm.org/10.1145/1379023.1375674.

[25] NikolajS. Bjørner, MarkE. Stickel, and TomásE. Uribe. A practical integration of first-order reasoning and decision procedures. In William McCune, editor, *Automated Deduction—CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 101–115. Springer Berlin Heidelberg, 1997.

ISBN 978-3-540-63104-0. doi: 10.1007/3-540-63104-6_13. URL http://dx.doi.org/10.1007/3-540-63104-6_13.

[26] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *In Proceedings of Formal Methods, 2006 (FM 2006)*, volume 4085/2006, pages 460–475. Springer-Verlag, 2006.

[27] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. URL http://proval.lri.fr/publications/boogie11final.pdf.

[28] Matteo Bordin, Tonu Naks, Andres Toom, and Marc Pantel. Compilation of heterogeneous models: Motivations and challenges. In *ERTS*, page (electronic medium), http://www.sia.fr, 2012. Société des Ingénieurs de l'Automobile.

[29] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. *SIGPLAN Not.*, 24(2):14–24, November 1988. ISSN 0362-1340. doi: 10.1145/64140.65005. URL http://doi.acm.org/10.1145/64140.65005.

[30] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic adaptation for models of computations. In Benoît Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *Proceedings of the 11th International Conference on Application of Concurrency to System Design*, pages 153–162. IEEE Computer Society, 2011. ISBN 978-0-7695-4387-1. doi: http://dx.doi.org/10.1109/ACSD.2011.17. URL /software/downloads/ModHelX/2011SemAdaptACSD.pdf.

[31] A Braganca and R.J. Machado. Extending uml 2.0 metamodel for complementary usages of the /spl lt/extend/spl gt/ relationship within use case variability specification. In *Software Product Line Conference, 2006 10th International*, pages 5 pp.–130, 2006. doi: 10.1109/SPLINE.2006.1691584.

[32] Daniel Calegari and Nora Szasz. Verification of model transformations: a survey of the state-of-the-art. *Electronic Notes In Theoretical Computer Science*, 292:5–25, 2013.

[33] Dominique Cansell and Dominique Méry. Foundations of the b method. *Computing and informatics*, 22(3-4):221–256, 2012.

[34] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41641. URL http://doi.acm.org/10.1145/41625.41641.

[35] Paul Caspi. Clocks in dataflow languages. *Theor. Comput. Sci.*, 94(1):125–140, March 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90326-B. URL http://dx.doi.org/10.1016/0304-3975(92)90326-B.

[36] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[37] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time simulink to lustre. In Rajeev Alur and Insup Lee, editors, *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2003. ISBN 3-540-20223-4. doi: 10.1007/978-3-540-45212-6_7. URL http://dx.doi.org/10.1007/978-3-540-45212-6_7.

[38] Paul Caspi, Grégoire Hamon, and Marc Pouzet. *Real-Time Systems: Models and verification —Theory and tools*, chapter Synchronous Functional Programming with Lucid Synchrone. ISTE, 2007.

[39] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Germán Puebla, Balthasar Weitzel, and Peter Y. H. Wong. Hats - a formal software product line engineering methodology. In *SPLC Workshops*, pages 121–128, 2010.

[40] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996. ISSN 0360-0300.

[41] EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982. ISBN 978-3-540-11212-9. doi: 10.1007/BFb0025774. URL http://dx.doi.org/10.1007/BFb0025774.

[42] Martin Clochard, Claude Marché, and Andrei Paskevich. Verified programs with binders. In *Programming Languages meets Program Verification (PLPV)*. ACM Press, 2014.

[43] Darren Cofer and Steven Miller. Do-333 certification case studies. In JuliaM. Badger and KristinYvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 1–15. Springer International Publishing, 2014. ISBN 978-3-319-06199-3. doi: 10.1007/978-3-319-06200-6_1. URL http://dx.doi.org/10.1007/978-3-319-06200-6_1.

[44] Darren D. Cofer, John Hatcliff, Michaela Huhn, and Mark Lawford. Software certification: Methods and tools (dagstuhl seminar 13051). *Dagstuhl Reports*, 3(1):111–148, 2013. doi: 10.4230/DagRep.3.1.111. URL http://dx.doi.org/10.4230/DagRep.3.1.111.

[45] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[46] MichaelA. Colón, Sriram Sankaranarayanan, and HennyB. Sipma. Linear invariant generation using non-linear constraint solving. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40524-5. doi: 10.1007/978-3-540-45069-6_39. URL http://dx.doi.org/10.1007/978-3-540-45069-6_39.

[47] Benoît Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoît Baudry. Bridging the chasm between executable metamodeling and models of computation. In *Proceedings of the 5th International Conference on Software Language Engineering*, 2012. URL /software/downloads/ModHelX/2012BridgingTheChasm.pdf.

[48] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[49] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[50] Krzysztof Czarnecki and UlrichW. Eisenecker. Components and generative programming. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66538-0. doi: 10.1007/3-540-48166-4_2. URL http://dx.doi.org/10.1007/3-540-48166-4_2.

[51] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005. ACM, ACM. URL http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

[52] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In Mehdi Jazayeri, RüdigerG.K. Loos, and DavidR. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41090-4. doi: 10.1007/3-540-39953-4_3. URL http://dx.doi.org/10.1007/3-540-39953-4_3.

[53] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44284-4. doi: 10.1007/3-540-45821-2_10. URL http://dx.doi.org/10.1007/3-540-45821-2_10.

[54] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[55] Maulik A. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, 2003.

[56] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3): 201–215, July 1960. ISSN 0004-5411. doi: 10.1145/321033.321034. URL http://doi.acm.org/10.1145/321033.321034.

[57] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi: 10.1145/368273.368557. URL http://doi.acm.org/10.1145/368273.368557.

[58] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The BWare project: Building a proof platform for the automated verification of B proof obligations. pages 290–293.

[59] E. Denney, B. Fischer, and J. Schumann. Adding assurance to automatically generated code. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 297–299, March 2004. doi: 10.1109/HASE.2004.1281768.

[60] JackB. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 1974. ISBN 978-3-540-06859-4. doi: 10.1007/3-540-06859-7_145. URL http://dx.doi.org/10.1007/3-540-06859-7_145.

[61] Arnaud Dieumegard, Pierre-Loïc Garoche, Temesghen Kahsai, Alice Taillar, and Xavier Thirioux. Compilation of synchronous observers as code contracts. In *SAC '15*, 2015.

[62] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL http://doi.acm.org/10.1145/360933.360975.

[63] David Déharbe, Silvio Ranise, and Jorgiano Vidal. A prototype implementation of a distributed satisfiability modulo theories solver in the toolbus framework. *Journal of the Brazilian Computer Society*, 14(1):71–86, 2008. ISSN 0104-6500. doi: 10.1007/BF03192553. URL http://dx.doi.org/10.1007/BF03192553.

[64] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 112–121, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371419. URL http://doi.acm.org/10.1145/2371401.2371419.

[65] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. pages 15–29, 2004. URL http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz.

[66] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. pages 173–177, 2007. URL http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf.

[67] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The Spirit of Ghost Code. In *CAV 2014, Computer Aided Verification - 26th International Conference*, Vienna Summer Logic 2014, Austria, July 2014. URL https://hal.inria.fr/hal-00873187.

[68] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[69] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

[70] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *PPES*, volume 18 of *OASICS*, pages 59–68. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. ISBN 978-3-939897-28-6.

[71] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[72] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from dataflow programs. In Jan Maluszyński and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 207–218. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-54444-9. doi: 10.1007/3-540-54444-5_100. URL http://dx.doi.org/10.1007/3-540-54444-5_100.

[73] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Springer Verlag, 1993.

[74] André Heuer, Vanessa Stricker, Christof J. Budnik, Sascha Konrad, Kim Lauenroth, and Klaus Pohl. Defining variability in activity diagrams and petri nets. *Sci. Comput. Program.*, 78(12):2414–2432, December 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2012.06.003. URL http://dx.doi.org/10.1016/j.scico.2012.06.003.

[75] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–583, 1967.

[76] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *ERTS*, page (electronic medium), http://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[77] Nassima Izerrouken, Marc Pantel, and Xavier Thirioux. Machine-checked sequencer for critical embedded code generator. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2009. ISBN 978-3-642-10372-8.

[78] Nassima Izerrouken, Marc Pantel, Xavier Thirioux, and Olivier Ssi Yan Kai. Integrated formal approach for qualified critical embedded code generator. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 199–201. Springer, 2009. ISBN 978-3-642-04569-1.

[79] Jean-Marc Jézéquel, David Mendez, Thomas Degueule, Benoit Combemale, and Olivier Barais. When Systems Engineering Meets Software Language Engineering. In *CSD&M'14 - Complex Systems Design & Management*, Paris, France, November 2014. Springer. URL http://hal.inria.fr/hal-01024166.

[80] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209. URL http://doi.acm.org/10.1145/1013208.1013209.

[81] Edson A. Oliveira Junior, Itana M. S. Gimenes, and José C. Maldonado. Systematic management of variability in uml-based software product lines. *Journal of Universal Computer Science*, 16(17): 2374–2393, sep 2010. http://www.jucs.org/jucs_16_17/systematic_management_of_variability.

[82] Gilles Kahn. A Preliminary Theory for Parallel Programs. Rapport de recherche R0006, 1973. URL http://hal.inria.fr/inria-00177890. Rapport IRIA.

[83] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[84] KyoC. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. ISSN 1022-7091. doi: 10.1023/A:1018980625587. URL http://dx.doi.org/10.1023/A3A1018980625587.

[85] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *CoRR*, abs/1409.2378, 2014. URL http://arxiv.org/abs/1409.2378.

[86] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, pages 168–173, Budapest, Hungary, April 2008.

[87] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc. ISBN 0-471-93608-1. URL http://dl.acm.org/citation.cfm?id=145448.146725.

[88] John L. Kelly, Carol Lochbaum, and V.A Vyssotsky. A block diagram compiler. *Bell System Technical Journal, The*, 40(3):669–678, May 1961. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1961.tb03236.x.

[89] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. doi: 10.1007/BFb0053381. URL http://dx.doi.org/10.1007/BFb0053381.

[90] Stephen Cole Kleene. *Introduction to metamathematics*. Bibl. Matematica. North-Holland, Amsterdam, 1952.

[91] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A Domain Specific Language For Source Code Analysis And Manipulation. In A. Walenstein and S. Schuppe, editors, *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation 2009*. IEEE, 2009. URL http://oai.cwi.nl/oai/asset/15097/15097A.pdf.

[92] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The epsilon object language (eol). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35909-8. doi: 10.1007/11787044_11. URL http://dx.doi.org/10.1007/11787044_11.

[93] Paul R. Kosinski. A data flow language for operating systems programming. *SIGPLAN Not.*, 8(9): 89–94, January 1973. ISSN 0362-1340. doi: 10.1145/390014.808289. URL http://doi.acm.org/10.1145/390014.808289.

[94] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In RichardF. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69823-4. doi: 10.1007/978-3-540-69824-1_17. URL http://dx.doi.org/10.1007/978-3-540-69824-1_17.

[95] Y. Fang L. D. Zuck, A. Pnueli and B. Goldberg. VOC: A translation validator for optimizing compilers. ENTCS, Elsevier Science, 2002.

[96] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336546. URL http://doi.acm.org/10.1145/336512.336546.

[97] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In Fabrice Kordon and Yvon Kermarrec, editors, *Reliable Software Technologies – Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01923-4. doi: 10.1007/978-3-642-01924-1_17. URL http://dx.doi.org/10.1007/978-3-642-01924-1_17.

[98] E. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446.

[99] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 114–123, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289949. URL http://doi.acm.org/10.1145/1289927.1289949.

[100] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. ISBN 1-59593-027-2.

[101] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[102] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[103] Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1-2):99–106, 2010. ISSN 1614-5046. doi: 10.1007/s11334-009-0109-0. URL http://dx.doi.org/10.1007/s11334-009-0109-0.

[104] Claude Marché. Verification of the functional behavior of a floating-point program: an industrial case study. *Science of Computer Programming*, 96(3):279–296, March 2014. doi: 10.1016/j.scico.2014.04.003.

[105] Minsky Marvin. Matter, mind and models. *Semantic information processing*, pages 425–432, 1968.

[106] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In Steve Reeves and Elvinia Riccobene, editors, *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 238–251, Pisa, Italy, June 2012. Springer. http://hal.inria.fr/hal-00681781/en/.

[107] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892. URL http://doi.acm.org/10.1145/1118890.1118892.

[108] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0880-9. doi: 10.1145/2069216.2069252. URL http://doi.acm.org/10.1145/2069216.2069252.

[109] Robin Milner and R. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, (7), 1972.

[110] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

[111] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Software*, 30(3):50–57, 2013. doi: 10.1109/MS.2013.43. URL http://doi.ieeecomputersociety.org/10.1109/MS.2013.43.

[112] Mirabelle Nebut. An overview of the signal clock calculus. *Electron. Notes Theor. Comput. Sci.*, 88:39–54, October 2004. ISSN 1571-0661. doi: 10.1016/j.entcs.2003.05.005. URL http://dx.doi.org/10.1016/j.entcs.2003.05.005.

[113] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. *SIGOPS Operating Systems Review*, 30:229–244, 1996.

[114] Special C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[115] Special C. of RTCA. DO-330, software tool qualification considerations, 2011.

[116] C. O'Halloran. Issues for the automatic generation of safety critical software. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 0:277, 2000. ISSN 1527-1366. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2000.873677.

[117] OMG. Mof specification. http://www.omg.org/spec/MOF/2.4.2/PDF, .

[118] OMG. OCL specification. http://www.omg.org/spec/OCL/,.

[119] OMG. Qvt specification. http://www.omg.org/spec/QVT/1.1/PDF/,.

[120] OMG. Uml specification. http://www.omg.org/spec/UML/2.5/Beta2/PDF,.

[121] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. CSL, 1995. URL citeseer.ist.psu.edu/owre93user.html.

[122] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3): 307–338, 2011. URL http://hal.inria.fr/inria-00638936.

[123] L-C. Paulson. The Isabelle reference manual. Technical Report 283, 1993. URL citeseer.ist.psu.edu/paulson95isabelle.html.

[124] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Tools and Algorithms for Construction and Analysis of Systems, TACAS 98*, 1384:151–166, 1998.

[125] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998. ISBN 3-540-64356-7.

[126] Frédéric Pothon. Do-330/ed-215 benefits of the new tool qualification document. Technical report, January 2013. URL http://www.open-do.org/?p=2150&preview=true.

[127] CorinaS. Păsăreanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21314-7. doi: 10.1007/978-3-540-24732-6_13. URL http://dx.doi.org/10.1007/978-3-540-24732-6_13.

[128] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982. ISBN 978-3-540-11494-9. doi: 10.1007/3-540-11494-7_22. URL http://dx.doi.org/10.1007/3-540-11494-7_22.

[129] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *Compiler Construction (CC 2010)*, volume 6011 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 2010.

[130] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA*, volume 50, 2002.

[131] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[132] Michael Ryabtsev and Ofer Strichman. Translation validation: From simulink to c. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 696–701. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_57. URL http://dx.doi.org/10.1007/978-3-642-02658-4_57.

[133] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using größner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 318–329, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964028. URL http://doi.acm.org/10.1145/964001.964028.

[134] Bastian Schlich and Stefan Kowalewski. Model checking c source code for embedded systems. *Int. J. Softw. Tools Technol. Transf.*, 11(3):187–202, June 2009. ISSN 1433-2779. doi: 10.1007/s10009-009-0106-5. URL http://dx.doi.org/10.1007/s10009-009-0106-5.

[135] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, February 2007. ISSN 1389-1286.

[136] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7. URL http://dl.acm.org/citation.cfm?id=244522.244560.

[137] Alexander Smith, Andreas Veneris, M Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(10):1606–1621, 2005.

[138] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In Sang Lyul Min and Wang Yi, editors, *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, pages 21–33. ACM, 2006. ISBN 1-59593-542-8. doi: 10.1145/1176887.1176892. URL http://doi.acm.org/10.1145/1176887.1176892.

[139] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 244–257, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02776-5. doi: 10.1007/978-3-642-02777-2_24. URL http://dx.doi.org/10.1007/978-3-642-02777-2_24.

[140] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and DennisR. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_34. URL http://dx.doi.org/10.1007/978-3-642-05089-3_34.

[141] Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001. ISSN 0164-1212. doi: 10.1016/S0164-1212(00)00089-3.

[142] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083192. URL http://doi.acm.org/10.1145/1082983.1083192.

[143] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981. ISSN 0001-0782. doi: 10.1145/358746.358755. URL http://doi.acm.org/10.1145/358746.358755.

[144] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, August 1976. ISSN 0001-0782. doi: 10.1145/360303.360308. URL http://doi.acm.org/10.1145/360303.360308.

[145] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.

[146] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Computer Science*, pages 308–319. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44190-8. doi: 10.1007/3-540-45757-7_26. URL http://dx.doi.org/10.1007/3-540-45757-7_26.

[147] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-41865-8. doi: 10.1007/3-540-45319-9_9. URL http://dx.doi.org/10.1007/3-540-45319-9_9.

[148] Andres Toom, Tonu Naks, Marc Pantel, Marcel Gandriau, and Indra Wati. Gene-Auto - an Automatic Code Generator for a safe subset of Simulink-Stateflow and Scicos. In *ERTS*, page (electronic medium), http://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[149] Andres Toom, Nassima Izerrouken, Tonu Naks, Marc Pantel, and Olivier Ssi-Yan-Kai. Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset. In *ERTS*, page (electronic medium), http://www.sia.fr, 2010. Société des Ingénieurs de l'Automobile.

[150] Andres Toom, Arnaud Dieumegard, and M.Pantel. Specifying and verifying model transformations for certified systems using transformation models. In *Embedded Real-Time Software and Systems, ERTS2*, 2014. URL http://www.erts2014.org/Site/0R4UXE94/Fichier/erts2014_8D1.pdf.

[151] S. Vestal. Assuring the correctness of automatically generated software. In *Digital Avionics Systems Conference, 1994. 13th DASC., AIAA/IEEE*, pages 111–118, Oct 1994. doi: 10.1109/DASC.1994.369494.

[152] Valentino Vranic and Jan Snirc. Integrating feature modeling into UML. In *Conference Proceedings NODe 2006, GSEM 2006, Erfurt, Germany, September 18-20, 2006*, pages 3–15, 2006. URL http://subs.emis.de/LNI/Proceedings/Proceedings88/article4672.html.

[153] H. R. Walters. *On Equal Terms — Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[154] Timothy Wang, Romain Jobredeaux, Heber Herencia-Zapana, Pierre-Loïc Garoche, Arnaud Dieumegard, Eric Feron, and Marc Pantel. From design to implementation: an automated, credible autocoding chain for control systems. *CoRR*, abs/1307.2641, 2013. URL http://arxiv.org/abs/1307.2641.

[155] M.W. Whalen and M.P.E. Heimdahl. An approach to automatic code generation for safety-critical systems. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 315–318, Oct 1999. doi: 10.1109/ASE.1999.802346.

[156] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica to *56*. Cambridge University Press, second edition, 1997. ISBN 9780511623585. URL http://dx.doi.org/10.1017/CBO9780511623585. Cambridge Books Online.

[157] Professor Glynn Winskel. Lecture notes on denotational semantics for part ii of the computer science tripos. 2005.

[158] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. A concurrent portfolio approach to smt solving. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 715–720, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_60. URL http://dx.doi.org/10.1007/978-3-642-02658-4_60.

[159] Anna Zaks and Amir Pnueli. Program analysis for compiler validation. In Shriram Krishnamurthi and Michal Young, editors, *PASTE*, pages 1–7. ACM, 2008. ISBN 978-1-60558-382-2.

[160] Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal verification integration approach for dsml. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 336–351. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41532-6. doi: 10.1007/978-3-642-41533-3_21. URL http://dx.doi.org/10.1007/978-3-642-41533-3_21.

[161] Hantao Zhang. Sato: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pages 272–275, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63104-6. URL http://dl.acm.org/citation.cfm?id=648233.753307.

[162] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the UML: deriving products. In *Software Product Lines - Research Issues in Engineering and Management*, pages 557–588. 2006. doi: 10.1007/978-3-540-33253-4_15. URL http://dx.doi.org/10.1007/978-3-540-33253-4_15.

[163] Paul Ziemann and Martin Gogolla. An extension of ocl with temporal logic. In *Critical Systems Development with UML*, pages 53–62, 2002.