A MARTE-Based Reactive Model for Data-Parallel Intensive Processing: Transformation Toward the Synchronous Model

Yu Huafeng

▶ **To cite this version:**

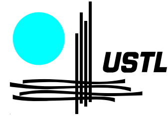
Yu Huafeng. A MARTE-Based Reactive Model for Data-Parallel Intensive Processing: Transformation Toward the Synchronous Model. Modeling and Simulation. Université des Sciences et Technologie de Lille - Lille I, 2008. English. NNT : . tel-00497248

HAL Id: tel-00497248
https://theses.hal.science/tel-00497248

Submitted on 2 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Numéro d'ordre : XXXXXXXXX

Université des Sciences et Technologies de Lille

Thèse

présentée pour obtenir le titre de
DOCTEUR spécialité Informatique

par

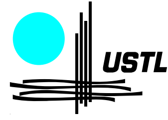
HUAFENG YU

Un Modèle Réactif Basé sur MARTE Dédié au Calcul Intensif à Parallélisme de Données : Transformation vers le Modèle Synchrone

Thèse soutenue le 27 novembre 2008,
devant la commission d'examen formée de :

François Terrier	Professeur INSTN CEA	**Président**
Françoise Simonot-Lion ..	Professeur INPL, ÉNSMN ..	**Rapporteur**
Charles André	Professeur UNSA	**Rapporteur**
Abdoulaye Gamatié	CR CNRS	**Examinateur**
Jean-Luc Dekeyser	Professeur USTL	**Directeur**
Éric Rutten	CR INRIA	**Co-directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
U.S.T.L., Cité Scientifique, 59655, Villeneuve d'Ascq, France



Number : XXXXXXX

A MARTE-Based Reactive Model for Data-Parallel Intensive Processing: Transformation Toward the Synchronous Model

By

HUAFENG YU

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Sciences

in the

École Doctorale Sciences pour l'Ingénieur

in the

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Committee in charge:

François Terrier	Professor at INSTN CEA	**President**
Françoise Simonot-Lion ..	Professor at INPL, ÉNSMN .	**Rapporteur**
Charles André	Professor at UNSA	**Rapporteur**
Abdoulaye Gamatié	CNRS Research scientist	**Examinateur**
Jean-Luc Dekeyser	Professor at USTL	**Directeur**
Éric Rutten	INRIA Research scientist ...	**Co-directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
U.S.T.L., Cité Scientifique, 59655, Villeneuve d'Ascq Cédex, France

Acknowledgements

First, I am grateful to the members of my dissertation committee. I wish to thank Prof. François Terrier for his acceptance to be the president of committee, Prof. Françoise Simonot-Lion and Prof. Charle André for their technical understanding, insight, and love of discovery and investigation for this thesis.

This thesis would not have been possible without the guidance of my thesis advisors, Prof. Jean-Luc Dekeyser, Dr. Éric Rutten and Dr. Abdoulaye Gamatié. Their continuous supervision, encouragement, and of course constructive criticism have been great help and support in my research work. Moreover, I would not forget the humour of Dr. Rutten and luminous discussions in the office and metro with Dr. Gamatié.

I also need to thank the members of INRIA DaRT team and POP ART team, Adolf, Alain, Anne, Antoine, Arnaud, Ashish, Calin, César, Éric P., Gwenaël, Imran, Julien S., Julien T., Karine, Lossan, Luc, Myriam, Ouassila, Pierre, Rabie, Sébastien, Vincent, Vlad, for sharing the good ambiance during my stay at INRIA. In addition, I appreciate the review of this thesis by Imran, César and Lingbo. I also want to express my additional gratitude to Éric P. for sharing accommodation with me, Imran for sharing experiences of outgoings and photography.

Special thanks go to my family, I could not accomplish my study in France without your support and encouragement for these five years.

Finally, thanks for the support and sponsorship given by the INRIA Lille Nord Europe, region of Nord Pas De Calais, USTL, and LIFL.

Contents

Introduction **1**

I State of the art **7**

1 System on Chip **9**
1.1 Introduction 9
1.2 Application domains 11
1.3 SoC design 11
1.3.1 SoC codesign 11
1.3.2 Productivity issue 13
1.3.3 Validation in SoC 14
1.4 Conclusions 14

2 Data-intensive processing and the Array-OL formalism **15**
2.1 Intensive digital signal processing 15
2.2 A high-level data dependency model: Array-OL 17
2.2.1 Basic characteristics 18
2.2.2 Task parallelism 19
2.2.3 Data parallelism 19
2.2.4 Compilation and optimization of the Array-OL language 22
2.2.5 Inter-repetition dependency 23
2.3 The need for design environment and methodology 23
2.4 Conclusions 24

3 Model-Driven Engineering and Gaspard2 **25**
3.1 Model-Driven Engineering 26
3.1.1 Model and modeling 26
3.1.2 Metamodel and metamodeling 28
3.1.3 Model transformations 30
3.1.4 MDE in practice 33
3.1.5 Conclusions 38
3.2 An MDE-based environment for SoC co-design: Gaspard2 39
3.2.1 High-level co-modeling for SoC design 40
3.2.2 Gaspard2 and MDE 43
3.2.3 Modeling of high-level control in Gaspard2 44
3.3 Conclusions 45

4 Control and validation in Gaspard2 and reactive systems **47**
4.1 Control and validation requirements in Gaspard2 47
4.1.1 Gaspard2 control requirements 48
4.1.2 Validation issue of Gaspard2 49
4.1.3 Conclusions 49
4.2 Reactive systems and the synchronous approach 50
4.2.1 Introduction 50
4.2.2 The synchronous approach 51
4.2.3 Synchronous languages 52
4.2.4 Using synchronous languages for Gaspard2 control modeling and validation 65
4.3 Conclusions 65

II Synchronous modeling and reactive control of DIP applications **69**

5 Synchronous modeling of DIP applications **71**
5.1 General modeling approach 72
5.1.1 Step 1: space refactoring 73
5.1.2 Step 2: space-time analysis and mapping 73
5.1.3 Step 3: synchronous modeling 76
5.1.4 Step 4: code generation 76
5.2 General synchronous modeling propositions 76
5.2.1 Array data structure 76
5.2.2 Parallelism modeling 77
5.2.3 Hierarchy and composition 78
5.3 Gaspard2 and synchronous representations 79
5.3.1 A Gaspard2 abstract syntax 79
5.3.2 Synchronous equations abstract syntax 80
5.4 The translation between the two representations 81
5.4.1 Structural translation 81
5.4.2 Translation of behavioral aspects 81
5.4.3 Correctness of the translation 83
5.4.4 Translation with serialization and partition semantics 85
5.5 Conclusions 88

6 Reactive control extension of Gaspard2 **89**
6.1 Introduction 90
6.1.1 Control modeling in DIP applications 90
6.1.2 Basic ideas of Gaspard2 control 92
6.1.3 Previous control proposition in Gaspard2 96
6.2 An extension proposal for control 96
6.2.1 Mode switch task and modes 96
6.2.2 State graphs and state graph tasks 97
6.2.3 Task compositions of SGT and MST in Gaspard2 101
6.3 Reactive control extension in Gaspard2 105
6.3.1 Gaspard2 control in a dataflow context 105

6.3.2 Reactive automata based control mechanism in Gaspard2 106
6.4 Typical examples 107
6.4.1 A typical example of a counter 107
6.4.2 A control example for cell phone video effect 109
6.5 Conclusions 110

III Integration into an MDE framework and case study 113

7 Transformation from Gaspard2 to synchronous languages 115
7.1 The Gaspard metamodel and profile 116
7.2 Synchronous equational metamodel 117
7.2.1 Common aspects 117
7.2.2 From abstract syntax to synchronous metamodel 118
7.2.3 Signal 118
7.2.4 Equation 119
7.2.5 Node 120
7.2.6 Module 121
7.2.7 IP deployment 121
7.3 The transformation rules 122
7.3.1 From Gaspard2 models to synchronous models 122
7.3.2 Transformation tools 128
7.3.3 Template-based code generation and code generators 128
7.3.4 The synchronous transformation chain 128
7.4 Conclusions 130

8 Transformation of the control extension of Gaspard2 131
8.1 MARTE-compatible control for Gaspard2 132
8.1.1 Mixed structure-behavioral modeling 132
8.1.2 System behavior modeling with the help of UML 132
8.1.3 Using UML state machines 133
8.1.4 Using UML Collaborations in mode task component 138
8.1.5 A complete Gaspard2 control structure 139
8.2 Control extension to Gaspard2 metamodel 139
8.2.1 The metamodel of state graphs 140
8.2.2 The metamodel of events 140
8.3 Extended synchronous metamodel 140
8.3.1 StateMachine 142
8.3.2 BooleanExpression 142
8.4 Transformations 142
8.4.1 From a UML model to a Gaspard2 model 144
8.4.2 From a Gaspard2 model to a synchronous mixed-style model 146
8.4.3 From a mixed-style model to an equational model 147
8.4.4 From a mixed-style model to an automaton model 151
8.5 Conclusions 154

9 A case study on multimedia cell phone 155
9.1 Introduction 155
9.2 Modeling of the example in Gaspard2 156
9.2.1 A global view of the example 156
9.2.2 A macro structure for the video effect processing 159
9.2.3 Repetitive modeling of video effect processing 159
9.2.4 A complete modeling of the phone example. 161
9.2.5 Requirements of formal application verification. 165
9.3 Application validation and analysis 168
9.3.1 Functional validation and analysis 168
9.3.2 Validation considering non-functional aspects 172
9.4 Discrete controller synthesis 173
9.5 Related works 175
9.6 Conclusions 175

Conclusions 177

Bibliography 181

Appendix A The Gaspard2 metamodel 193

Appendix B Code examples 199

Nomenclature 213

Résumé/Abstract 216

Introduction

Context: data-parallel intensive processing on SoC

Since its appearance, computer has a great influence on people. The evolution of electronics, particularly integrated circuits (ICs) plays a significant role in this progress. More powerful, reliable and complex, but less energy-consumptive, expensive and smaller summarize the trend of this evolution. Compared to personal computer systems, which have been rapidly developed in the past thirty years or so, embedded systems are taking over. Embedded systems are generally specific-purpose computer system, which are assembled as a part of electronic, mechanical or physical devices. They have also computing units, memory, I/O devices, etc. In general, embedded systems are constrained with size, power consumption, cost, etc., as a consequence of their built-in nature and mass production. They are used everywhere from daily life to industry, for instance, telecommunication, transport, aeronautics, automobile and power plant.

Behind these superficial development of computer/embedded systems, the development of IC technology, for instance, the photolithography (45 nm process now) enlarges the circuit scale very quickly (more than 20 million gates at the moment). System-On-a-Chip or System On Chip (SoC or SOC) is proposed as a new methodology for embedded system design, which emerged around 2000. In SoC, the computing units, memory, I/O devices, etc., are all integrated into a single chip. Thanks to the great-scale integration technology, a single chip can accomplish complex tasks, which is even equivalent to some computer systems. Moreover, multiple processors can be integrated into one chip (Multi-processor System on Chip, MPSoC), in which communications can be achieved through Networks on Chip (NoC).

Following the fast development of electronics, particularly the increment in computing power, embedded systems become less specific than ever. Reconfigurable computing in embedded system has emerged too. Field-Programmable Gate Arrays (FPGAs) contains programmable logic components and interconnects. These programmable parts make FPGAs more flexible and fast to be adapted to new applications and new domains. General-purpose computing units, such as processors, can be integrated into a chip, which offer even more flexibility. Various flexible and dynamic applications can be implemented with them. Moreover, the emergence of middle-ware or real-time operating systems on embedded systems make system adaptivity possible according to the constraints of environment, hardware, platform, etc. For instance, adaptive systems help to manage the balance between performance and energy.

In the wake of fast development in hardware, new SoC design methodologies/languages have emerged in succession. Computer-aided design tools lead to more interests in Hard-

ware/Software (HW/SW) codesign, which signifies the design of software and hardware are carried out concurrently to accelerate system design. This methodology benefits from the advantages offered by both hardware and software, i.e., hardware provides *high performance*, whereas software enables *flexibility*. In spite of the dominant position of hardware design of ICs in the past decades, the current trend shows that software design is becoming increasingly important, because software makes the products distinct from others, which is critical in the SoC market. Meanwhile, hardware design is prone to being similar to software design, for instance, programming languages (Hardware Description Language–HDL–, SystemC) are used for hardware design. Moreover, high-level system design is possibly carried out using a same programming language or modeling language.

Data processing becomes more important in contrast to classical embedded controllers in SoC applications. Signal processing, particularly multimedia processing, is one of the most important applications of data processing. Nowadays multimedia mobile devices, such as Personal Digital Assistant (PDA), multimedia cell phones and mobile multimedia players, are ubiquitous. These devices provide many multimedia functionalities, such as mp3 playback, camera, video and mobile TV. These functionalities greatly contribute to making profit in the market. These features, together with their small size and long power supply make them irreplaceable in the market.

The previously mentioned multimedia data processing is considered as data-parallel intensive processing (DIP), which also includes high-definition TV and radar/sonar signal processing. Parallel processing is a key feature of these applications. Unlike general parallel applications, which focus on code parallelization and their communications, DIP applications concentrate on regular data partition, distribution and their access. The data manipulated in these applications are generally in multidimensional data structure, such as multidimensional arrays.

Issues: complexity of design

According to the observation of Gordon E. Moore, the co-founder of Intel, the number of transistors in an IC is doubled every two years. As the computing power is increased, more functionalities are expected to be integrated into the system. As a result, more complex software and hardware applications are integrated. This leads to the *system complexity* issue. For instance, a multimedia cellular phone integrates different modules of telecommunication, music/video playback, mobile TV and Global Positioning System (GPS). These different modules require different expertise, development methodology and tools of their domains. Each of these modules may have complex implementation to meet the requirements of end users, such as reliability and user-friendliness. System complexity is currently a big constraint of SoC developments.

As a result of the overall complexity of SoCs, system design (particularly software design) is not expected to increase at the same pace of hardware, because of limited development budget, staff, tools, etc. In addition, product life cycles shrink while design time seems to increase. The system complexity makes the design more difficult, not only in analysis, but also in development, integration, validation, etc. This evolution, which is out of balance between production and design, has become an issue after 2000, and it finally leads to the *productivity gap*.

In general, SoC designs are expected to be reliable, particularly in safety-critical embed-

ded systems, such as aeronautics and power plant. SoC reliability can be ensured by safe design approaches or the validation step in the design flow. Averagely, two third of the time in the development cycle is dedicated to validation, which makes it a key factor in the design. Validation can be achieved by formal verification, simulation, test, etc. However, system size and complexity has a significant influence on validation. For instance, formal verification could not be carried out on a complete SoC because of the state space explosion problem. Hence, only certain critical parts of an SoC are verified by this technology, such as communication protocols and some parts of digital circuits. Simulation and test can be carried out on the whole system, but they cannot guarantee the correctness of system. Their efficiency generally relies on the level of abstraction, at which they are carried out, e.g., high-level simulation takes less time to accomplish, but it is less precise than the lower one.

Motivations: safe design via formal languages

Much research has been carried out to search solutions in relation to the previous mentioned issues. Raising levels of abstraction has been widely studied to reduce the system design complexity, e.g., the usage of high-level hardware description languages for circuit design. System-level design with software and hardware partition also reduces the complexity, thus accelerates the design process. Design and validation at a high level make development more efficient and fast because of a more concise design description. Software and hardware reuse helps to avoid "reinvent the wheels", hence enables to leverage the cost of design. These reusable software or hardware blocks, called Intellectual Properties (IPs), may come from third parties or in-house designs.

In recent years, using the same programming/modeling languages for both hardware and software design (e.g., SystemC) attracts much research interests. High-level modeling based system design has also been developed rapidly, e.g. Model-Driven Engineering (MDE). MDE enables high-level and system-level modeling with the possibilities of integration of heterogeneous components in the system. Model transformations can be carried out to generate executable models (or executable code) from high-level models. Transformations can be divided into several modular sub-transformations, each of which addresses some specific concerns. MDE is also supported by large number of standards and tools. They help the spread of MDE in the SoC design.

Gaspard2 [65] is an MDE-based methodology and development environment dedicated to high-level system design of DIP applications on chip. Benefiting from the advantages offered by the MDE approach, particularly fast design and code generation, it currently provides high-level system modeling and model transformations towards different languages, including SystemC, VHDL, OpenMP Fortran/C and synchronous languages. These languages are used for different purpose, e.g., simulation, synthesis, execution and validation.

Safe design is one of the main concerns of Gaspard2. Design considering desired properties and formal validation contribute to ensure design correctness. The first one helps to reduce the cost of validation in the design. High-level specification and its transition to low-level description, which are correct according to safe properties, enable to avoid the validation, fault detection and correction iterations in the design. As safe properties are still limited with regard to design correctness, formal validation is also encouraging.

In the embedded system design, formal models of computations (MoCs) help to describe unambiguous behavior of system. For instance, synchronous languages for reactive systems.

These MoCs help to build systems with rigorous semantics, on which system behavior can be reasoned about so that the correctness of system can be ensured. Using these MoCs in the design helps to verify the design correctness. Moreover, formal validation can also be carried out at this level with the help of MoCs.

Integration of high-level safe control in DIP applications is another objective in Gaspard2. This control contributes to endow Gaspard2 with dynamic behavior. This requirement of safe control integration is due to the rapid development in software and hardware, where applications are expected to be increasingly flexible and adaptive. In order to answer the safe design requirements, this control mechanism is intended to exhibit unambiguous behavior and clear semantics, so that the control can be ensured to be safe. Modeling this control with MoCs is a promising approach, which can meet the previous requirements. Moreover, validation tools are also expected to be used to verify the correctness of the introduced control.

Synchronous languages, considered as a MoC for embedded systems, enable rigorous system design. They have been successfully applied in several critical domains, such as avionics, automotive and nuclear power plants. Moreover, great number of tools are provided for these languages for the purpose of analysis and validation, which is considered as another advantage of using these languages.

Based on these observations, the work presented in this dissertation is involved in using synchronous languages as a MoC for the safe design of DIP applications, which provide, on one hand, safe operators proved by the MoC to ensure expected properties, on the other hand, formal validation to check the correctness of Gaspard2 applications.

Contributions: synchronous modeling, control extension and transformations in Gaspard2

The contributions of the work presented in this dissertation is within the context of SoC productivity issue. High-level validation and safe control extension, which is based on the study of synchronous languages, are two main objectives.

The synchronous modeling bridges the gap between DIP applications specified in Gaspard2 and synchronous data-flow languages. This modeling is based on the core formalism of data and task parallelism of Gaspard2. It is intended to capture the high-level properties of Gaspard2, such as parallelism and data dependency. The resulting model, which is based on common aspects of synchronous languages, enables the code generation for Lustre, Signal, Lucid synchrone, etc. High-level validation is then carried out with the code in order to check the correctness of the corresponding Gaspard2 specifications.

Gaspard2 control has also been extended and improved to have formal semantics and complex composition operators. This extension and improvement are based on the reactive control. Reactive control exhibits unambiguous semantics and verifiability, which help to check the correctness of the control. Moreover, some properties inbuilt in the reactive control help to reduce fault occurrences. This control extension mainly concerns state graphs and their semantics and compositions. As this control is introduced in Gaspard2 without consideration of any execution model, it remains generic at a high level of abstraction. Consequently, it can be translated into or projected onto different execution models, e.g., the synchronous model, which is presented here.

Implementations of modeling and transformation have been carried out in the framework of Model-Driven Engineering (MDE). DIP applications are first specified with UML

models in graphical form. With the help of the intermediate synchronous model, code in Lustre and Signal can then be generated automatically from Gaspard2 models. Transformation rules are also given for the control extension, whose implementation is a perspective. The generated code can be checked for the correctness of design. A case study of the multimedia processing of cellular phones is finally illustrated, with emphasis on the high-level modeling through Gaspard2 and the formal validation and controller synthesis of the application through the tools associated with synchronous languages.

Outline

This dissertation has three Parts. Part I presents the state of the art. It has four chapters. Chapter 1 and Chapter 2 discuss SoC and DIP applications, which are the context of this dissertation. Chapter 3 concerns MDE and the Gaspard2 development environment dedicated to DIP applications. Chapter 4 is related to control concerns, and concentrates on reactive control in synchronous languages.

Part II exhibits the contributions of modeling, which has two chapters. Chapter 5 presents the synchronous modeling of DIP applications, and Chapter 6 focuses on the reactive control integrated in Gaspard2.

Part III contains two chapters about implementations and one chapter about case study. Model transformations in the framework of MDE are presented in Chapter 7, while Gaspard2 control transformation is discussed in Chapter 8. Chapter 9 presents a case study, which concerns the multimedia functionality of a modern cellular phone. This case study illustrates the implementation of the proposed reactive control of Gaspard2, and formal validation carried out on this control.

Part I

State of the art

Chapter 1

System on Chip

1.1 Introduction **9**
1.2 Application domains **11**
1.3 SoC design **11**
1.3.1 SoC codesign 11
1.3.2 Productivity issue 13
1.3.3 Validation in SoC 14
1.4 Conclusions **14**

1.1 Introduction

Nowadays, SoC is becoming one of the principle solutions for embedded systems. As the name implies, all the needed electrical components in the system are directly integrated into one single chip. The components to be integrated can be any of, but not restricted to, the following components:

- processors (microprocessors, Digital Signal Processors (DSPs), etc.);
- memory blocks (RAM, ROM, flash, etc.);
- inter-component connections (bus, crossbar, etc.);
- external interfaces (USB, FireWire, Ethernet, etc.);
- analog/digital converters;
- timing sources (oscillators, phase-locked loops, etc.).

Compared to general purpose computer systems, SoCs are dedicated to applications that need to be deployed in other systems. As the whole system is integrated into one single chip, its size is restricted by the chip. Moreover, as the integration level increases, the chip size is becoming smaller, e.g., TSMC started to produce 32 nanometer node 0.183 square micrometer six-transistor SRAM cell from 2005. The small size is beneficial to the development of

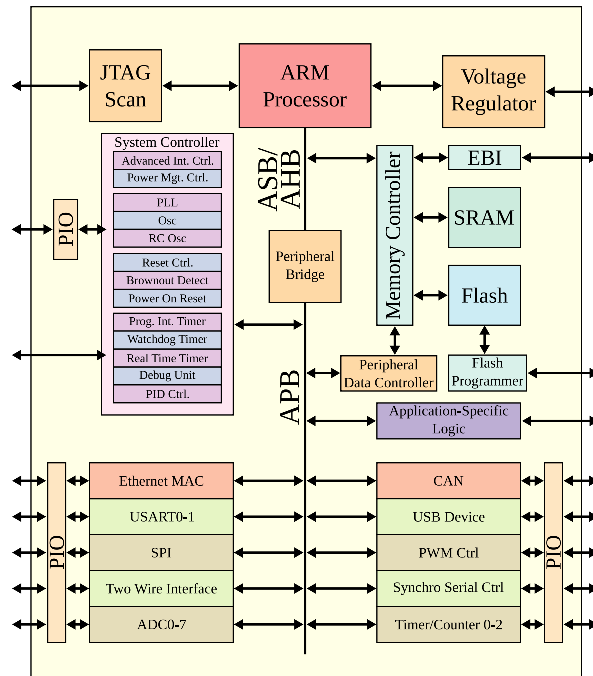


Figure 1.1: An example of SoC from Wikipedia: a microcontroller-based SoC.

mobile electronic devices, such as PDAs and smart phones, which persist with miniaturization. Hence, small size is a significant feature of SoCs. SoCs are generally required to have low-energy consumption, as they are always embedded in other systems, or implemented as mobile systems, which do not always provide sufficient energy. In comparison with computing power growth, the energy technology demonstrates tardy progress. In consequence, the low-energy consumption will be a remedy for the mobile devices, as the on-die signal delivery in SoCs helps to consume less energy. Computer systems are designed as self-contained systems, which perform general purpose computing tasks. Whereas SoCs can integrate dedicated hardware, such as DSPs and accelerators, to boost performance for some specific applications (e.g., signal/image/video processing). Furthermore, multi-core processors in SoCs make it possible to profit from parallel processing.

The previously mentioned characteristics make SoCs well adapted to the modern mobile electronic products for great public (smart phones, PDAs, set-top boxes, DVD players, etc).

However, the design and manufacturing of SoCs require big investments. For instance, the fabrication of SoCs requires *masks*, which are considered to be costly. Moreover, the photolithography requires mandatory ultra-clean workspaces. Hence, manufacturing of SoCs is always involved in mass production, which helps to reduce the overall cost including non-recurring engineering (NRE).

Multiprocessor SoC. Following the strong requirements of the mobile multimedia computing, and considering constraints from time-to-market, cost, energy consumption, etc., for the system design, SoCs are expected to be specialized to answer the previously mentioned needs, as well as the needs of flexibility, high-performance, etc. Multiprocessors SoCs (MPSoC) [67] are emerging, where multiple homogeneous or heterogeneous processing ele-

ments are integrated on the chip, together with on-chip interconnection (e.g. interconnection Networks on-Chip (NoC) [11], hierarchical memory, I/O components, etc. MPSoC is expected to satisfy the requirements of both high-performance and low-energy consumption demanded by mobile multimedia applications.

1.2 Application domains

Benefiting from the advantages of small size, low-energy consumption and powerful computing capacity, embedded systems are omnipresent nowadays. For instance, compared to 260 millions processors that were sold in the year of 2004, 14 billions embedded processors (such as microprocessor, microcontroller, DSP, etc.) have been sold. They are widely used in diverse domains, such as telecommunication, aeronautics, automobile, domestic appliance products, medical equipments, mobile electronic products, etc. Embedded applications cover both critical systems and simple systems. Among all these applications, SoCs become increasingly important, as SoCs can be used in more complex systems, such as video processing, Internet connectivity, electronic commerce, etc.

High-performance computing (HPC). HPC applications indicate a significant application domain of SoCs, such as embedded multimedia devices, radar/sonar signal processing devices, physical image equipments, etc., which have become quickly widespread over recent years. The applications on these devices are always involved in signal (data)-intensive parallel computing, which are generally regular for reasons of high performance. Moreover, some specific building blocks on SoCs are dedicated to handle large amount of data parallel processing with performance. These blocks include DSPs, hardware accelerators, etc.

1.3 SoC design

The chip design becomes system design as the chip itself is a system. SoC system-level design try to take the advantages of existing technologies to address SoC complexity issue [26], such as system-level architecting and architectural verification, hardware/software codesign, and high-level modeling.

1.3.1 SoC codesign

Gajski and Kuhn [44] presented the *Y chart* (Figure 1.2) for the system-level design of embedded systems. No matter how complex a SoC design is, it supposes that the design can be considered from three basic viewpoints, each of which concentrates on different properties of the system. The Y Chart has three axes to represent behavioral specification, architectural structure and physical design respectively. The Y Chart also depicts different levels of abstraction in the design. Four levels are illustrated through circles from outside to the center in Figure 1.2, which include: system level, processor level, logic and circuit level. These levels denote different time granularities and precisions in the design.

Inspired by the Y chart, the Y schema is usually adopted to represent the SoC design approaches. Its three axes represent functional behavior, hardware architecture and implementation in specific technologies (e.g., circuit, programming languages, etc.). The central

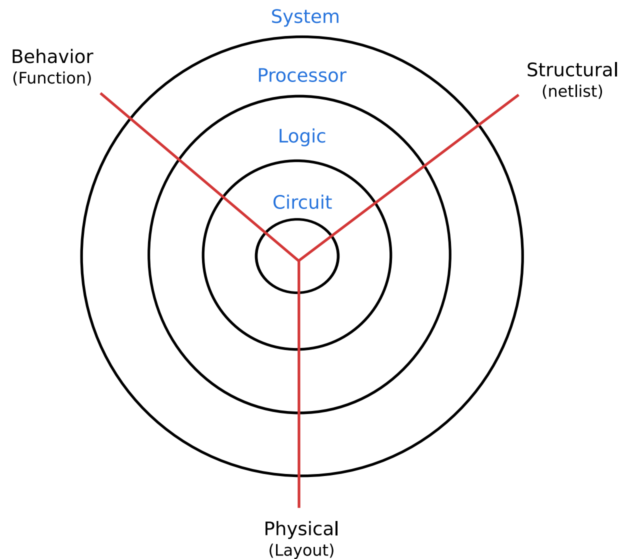


Figure 1.2: Y chart for the system-level design.

point of these three axes denotes the allocation of the resources provided by hardware architecture to functional behavior. At the same time, elementary concepts in software and hardware can be deployed with IPs implemented by some specific technologies.

SoCs are generally designed for some specific applications, dissimilar to computer systems. Given sufficient computing capacity, the SoC design obtains flexibility since applications can be partitioned into hardware and software designs. Figure 1.3 shows SoC hardware/software codesign, which is similar to the Y Chart. The partition of hardware and software is a trade-off between the performance provided by hardware and the flexibility given by software. Application behavior (software) and architecture (hardware) can be developed by different teams in order to benefit from their different experiences in the corresponding domains. Moreover, the possible concurrent design of hardware and software by different teams assists in shortening the design time. For instance, the software teams do not need to wait for the configuration of architecture accomplished by hardware teams, to start software development. The application behavior is then mapped onto the hardware architecture, on which analyses can be carried. Analysis results can be used for the modification of the original design at the modeling stage and at the mapping stage for different purposes. If the mapping result is approved by the analysis, it can be utilized for implementation purpose.

In the hardware design, IP building blocks have been widely used in the SoC design for re-usability and time-to-market reasons. SoC IPs involve processors, memory, standard cell, I/O, etc. Software can also have IPs, such as some elementary functions in the multimedia processing, which include Fast Fourier Transform (FFT), filters, encoding/decoding algorithms, etc. IPs help SoC designers to avoid re-inventing the wheels for some existing designs, which they do not necessarily have sufficient experience. IP technology turns out to be one of the most encouraging aspects in the SoC approach.

In the past years, RTL-level languages such as Verilog HDL [107] or VHDL [5] are used in the SoC design. This has been a great step in EDA (Electronic Design Automation) as it releases designers from low-level logic design. Besides, behavioral aspects of a design can be specified with C and C++, which are used to generate RTL-level languages through

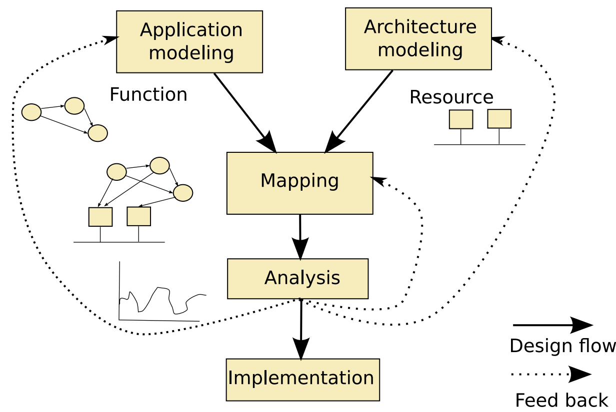


Figure 1.3: Design flow incorporating mapping and analysis [26].

behavioral synthesis. As systems become more complexe, system-level design languages attract more attension, which include SystemC [63] and SystemVerilog [124]. Currently, UML are also taken into account for the SoC design [87]. UML profile for SoC [105], the Systems Modeling Language (SysML) [105], as well as the UML profile for Modeling and Analysis of Real-Time and embedded Systems (MARTE) [98], etc, are proposed as OMG standards.

Using formal models of computation (MoCs) in software design makes it possible to carry out fast validation and efficient synthesis. The MoCs [36] include finite state machines (FSMs) models, data-flow models, synchronous models, etc. These MoCs are expected to be composed together in a hierarchical way so that complex systems can be modeled. As these models have finite states, consequently, they enable efficient synthesis. Their formal semantics, together with their owing finite states make the formal analysis feasible. The current trend shows that software design plays an increasing role in the SoC design, since software helps the products of one company to be distinct from those of others, which is significant in the market [132, 50]. However, the complexity of the software, such as multimedia applications on mobile devices is becoming the next bottleneck in SoC design [26].

1.3.2 Productivity issue

Current technology allows to integrate more than twenty million gates in a single chip, which makes it possible: the number of gates available in a chip become more than the requirement of software applications. As software design does not advance at the same rhythm as that of hardware. In consequence, a gap has been emerging between hardware manufacture and software design. Many partial solutions to address this issue are proposed, which include IP reuse, behavioral synthesis, softwarization (memory, processor), system-level hardware/software codesign, high-level abstraction, etc.

However, neither of these approaches can provide a complete solution. Hence, many of these approaches are mixed to obtain the maximum efficiency in the design. For instance, IP reuse, hardware/software codesign, high-level abstraction can be proposed in a framework in order to benefit from the advantages provided by the three approaches: IP reuse helps to separate concerns so that unacquainted work can be accomplished by certain experts of that domain; hardware/software codesign enables concurrent design, moreover, hardware design becomes similar to software design, for instance, software programming languages are

extended for the hardware design (e.g., VHDL, SystemC, etc.), which reduce the complexity by using the same programming languages; high-level abstraction contributes to design a system without too many implementation details.

1.3.3 Validation in SoC

Design correctness is an important issue in the SoC design, since it has a great impact on user's confidence on the products, time-to-market, cost, etc. For critical systems, ignorance of design correctness may invoke a disaster which can cause a lost of human lifes. But design correctness of SoCs always remains one of the critical challenges, although SoCs are turning to be more complex. Validation accounts for nearly two-thirds of the overall chip design cost, even so, design teams always deliver chips late and miss projected tape-out deadlines due to verification problems.

Simulation and test offers a compromise between the verification quality and cost, which is a main solution to this problem nowadays. However, formal methods are becoming interesting because the formal models of computation (MoCs) are increasingly accepted to be used for the specification of embedded systems. Using MoCs enables to specify the system with unambiguous semantics, which offers the possibility of either correct-by-construction or formal validation with associated tools. High-level validation is also encouraged as it aids to find design bugs at the early design stage, which can avoid late bug discovery in the design cycle.

1.4 Conclusions

As mentioned previously, SoCs exhibit interesting advantages over classical computer systems, which include smaller physical size, less energy consumption, improved performance in certain situations, etc. Hence, SoCs are frequently found in embedded systems. However, SoCs provide less flexible functionality with regard to computer systems, particularly when new applications are to be integrated into SoCs. Hardware and software codesign approaches are usually adopted in SoCs. The SoC validation is carried out through repetitive simulation and verification, until the systems are completely considered to be validated.

In spite of the widespread usage, SoCs also face several challenges. The main challenge is the productivity issue, which is caused by the balanceless advancement of hardware and software. Another challenge is the validation issue, which is the most time-consuming stage in the SoC design. Despite these challenges, new methodologies and technologies are continuously proposed to meet the requirements of SoC design. Particularly, using formal models for the specification of SoCs and proposition of unified frameworks, which integrate different approaches for fast and efficient design, are encouraged. In the context of this thesis, the synchronous languages and an MDE framework are taken into account, which are presented in the following chapters.

Chapter 2

Data-intensive processing and the Array-OL formalism

2.1 Intensive digital signal processing **15**
2.2 A high-level data dependency model: Array-OL **17**
2.2.1 Basic characteristics 18
2.2.2 Task parallelism 19
2.2.3 Data parallelism 19
2.2.4 Compilation and optimization of the Array-OL language 22
2.2.5 Inter-repetition dependency 23
2.3 The need for design environment and methodology **23**
2.4 Conclusions **24**

Signal processing is one of the most important SoC applications. It concerns the interpretation, analysis, storage and manipulation of signals, which can be signals of sound, image, video, radar, etc. A signal is the carrier of the information of interest. According to the different signals to be processed, these applications can be classified into: *analog signal processing* (signals are captured by sensors, which are not yet digitized) and *digital signal processing* (digitized signals that can be processed by SoCs or computers directly). Only digital signal processing is involved here, which includes filtering, removal of noise, information extraction, compression and decompression, etc.

2.1 Intensive digital signal processing

Among various types of digital signal processing, we are interested in *intensive signal processing* (ISP), which is always decomposed into two steps: *systematic signal processing* (SSP) for the first step and *intensive data processing* (IDP) for the second. SSP involves regular processing of large amounts of signal, which is independent of signal values. Whereas, IDP is considered as *irregular* processing because the processing results rely on signal values. Figure 2.1 shows the relation between these classifications of digital signal processing.

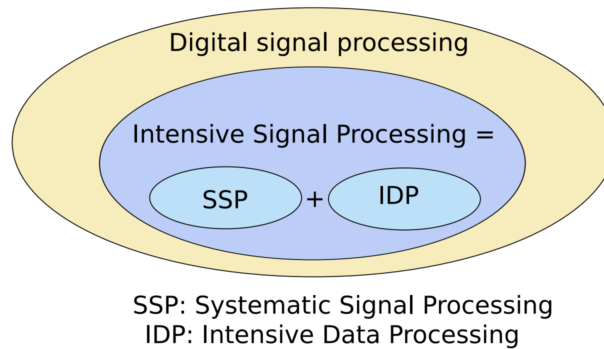


Figure 2.1: Relations of different signal processing

Some application examples. Some typical examples of intensive signal processing are presented here, which include:

- sonar signal processing: a submarine is equipped with several hydrophones around it, which are used for listening to and recording underwater sounds. A classical sonar signal processing chain is composed of several stages. The first stage involves systematic signal processing, which includes FFT. FFT adds a frequency dimension to the processed signals. The results are used in the following stages for communication or object detection purpose.

- Image encoding/decoding: JPEG 2000 is a wavelet-based image compression standard. The encoder [2] can also be divided into several stages. The first ones are considered to be systematic processing, which include *Color components transformation* and *Tiling*. The following stages involve irregular processing, such as *wavelet transform*, *quantization* and *coding*. The decoder works in an inverse way: irregular phases are followed by systematic phases.

- Aspect ratio converting: the conversion of a high-definition video format (16:9) to a standard-definition (4:3) [80] can also be divided into two stages: the first one consists of line processing of the original 16:9 video signals in order to create pixels through interpolation, the results are then processed by removing some lines so that the final ration is 4:3.

These examples show how the signal processing can be divided into stages, such as SSP and IDP. As SSP is independent from signal contents, it is possible to use certain *generic* models for the processing specification. However, IDP involves the processing of signal contents, which may vary from one to another according to the signal contents. Hence it is not appropriate to use some generic models for the computing specification.

Multidimensional array. *Multidimensional array* are often used as the main data structure in SSP applications. As signal contents are not involved in the processing, it is appropriate to abstract them, which facilitates the modeling of SSP. Consequently, array *type* and array *shape* are sufficient in the SSP modeling.

The previous examples also exhibit various semantics of signal dimensions (two dimensions of an image, temporal dimension, frequency dimension, etc.). For instance, the temporal dimension can be represented by an infinite dimension. The dimension number of a

signal can also be changed (increased or decreased) in the processing. Moreover, some applications can have signals with *toric* dimensions, i.e., data stored in these dimensions are processed in a *modulo* way.

Some Languages for signal processing. There already exist some languages for the specification of signal processing applications.

- StreamIt [129] and Synchronous Data Flow (SDF) [78] are stream processing languages, but they are not considered to be multidimensional languages for signal processing. StreamIt is an object-oriented imperative language that is intended to allow maximum optimization for the specification of synchronous dataflow at a high level of abstraction. The extension of SDF, MultiDimensional SDF [95] is a multidimensional language, whose applications are described using oriented acyclic graph. The nodes, called actors, in the graph consume and produce data, called tokens.
- The Alpha language [88] is a functional language, whose application are composed of systems of recurrent equations. Alpha is based on polyhedral model, which is extensively used for automatic parallelization and the generation of systolic arrays. Alpha is a multidimensional language with single assignment specification.
- High-performance Fortran(HPF) [62] is a language dedicated to scientific parallel computing. It takes high level of abstraction into account. HPF uses multidimensional arrays in parallel loops, where operations are carried out on sub-arrays. It also enables regular data distributions.
- Synchronous dataflow languages also defines arrays in order to deal with specific algorithms and architectures. For instance, in Lustre, array has been introduced in order to design and simulate systolic algorithms [56]. This work leads to the implementation of their results on circuits [114]. More recently, an efficient compilation of array in Lustre programs has been proposed [94]. It is similar to the Signal language. In contrast, array of processes [16] has been introduced in Signal, which is adapted to model systolic algorithms.
- Array-OL [19, 20] is also a multidimensional language for the specification of intensive signal processing, which will be presented in the next section.

2.2 A high-level data dependency model: Array-OL

Array Oriented Language (Array-OL) is first proposed by Alain Demeure ([31] in French and [30] in English) at THALES Underwater System (TUS) in 1995. It is dedicated to the specification of intensive signal processing where large number of signals are regularly processed by a set of repeated tasks. Its typical applications include radar/sonar signal processing and multimedia (image, audio and video) processing.

Array-OL is not a programming language, thus no execution concerns are involved in the language. It focuses on the full parallelism specification at a high level, hence it is not associated with an execution model. Instead of specifying certain specific scheduling of parallel tasks, only data dependencies between these tasks are specified. Some tasks that achieve some computing functionalities, such as filters and FFT, are referred to as *elementary*

tasks, which are considered as *black boxes* provided with interfaces. All these features make it possible to reduce the complexity of application design.

2.2.1 Basic characteristics

Some basic characteristics of this domain-specific language are presented here [19, 20]:

- *multidimensional array*: data manipulated in Array-OL are in the form of multidimensional array, which has at most one possible *infinite dimension*. These arrays can be specified with certain type specification, such as array shape. Nevertheless, data types, e.g., *Integer* and *Boolean*, are unnecessary, because data values stored in the array are not handled. Consequently data values are concealed. These features imply that only array spatial manipulations are involved in the language. Moreover, these arrays can be *toroidal*. This characteristic enables to model some spatial dimensions that represent some physical tori (e.g. hydrophones around a submarine). Other examples are some frequency domains obtained by FFTs.

- *granularity*: as data are represented as multidimensional arrays, which are repeatedly processed, processing granularity is therefore flexible through re-organizing the repeated processing and these arrays.

- *data dependency expressions*: Array-OL expresses true data dependency in order to describe maximum parallelism in the application. In such a way, except for the minimal partial order, which results from the specified data dependencies, no other order is *a priori* assumed;

- *functionally-deterministic specifications*: any execution schedule that respects the data dependencies specified in Array-OL necessarily leads to the same results;

- *single assignment*: the language handles values, not variables, so a value is produced only once, hence Array-OL is a single assignment language;

- *pure spatial specification*: there are no time or order specifications defined in the language, full parallelism is enabled by the pure spatial specification. In order to obtain some specific implementations, such as data-flow implementations, a specific space-time mapping should be defined. While the mapping is unnecessary to be defined in the Array-OL specification.

An Array-OL application can be specified at two levels: *global level* and *local level*. A global-level specification, which is called a *global model*, is dedicated to describe the data dependencies between tasks. A task in Array-OL represents a functionality, which can be either a hierarchical task or an elementary task. The data dependencies are expressed through arrays. The local-level specification, which is called a *local model*, is dedicated to describe how arrays are processed regularly and repeatedly through ODT (Opérateurs de Distribution de Tableaux, array distribution operators in English) in Array-OL tasks.

The parallelism specified in Array-OL is classified into two families: *task parallelism* and *data parallelism*. Task parallelism is expressed in the *global model*. On the contrary, data parallelism is described in the *local model*.

2.2.2 Task parallelism

A global-level specification is composed of a directed acyclic graph, where vertices represent tasks and directed edges represent data dependencies between tasks: arrays. Figure 2.2 shows an example of a global model. There are neither specific restrictions on the array number that tasks consume and produce, nor restrictions on the number of dimensions that an array possesses. From another point of view, the term *global* signifies that the arrays processed in this level are always complete arrays with regard to the arrays processed in a finner level of granularity.

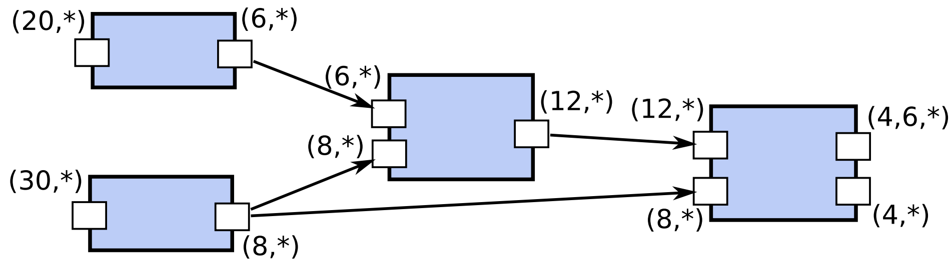


Figure 2.2: An example of a global-level Array-OL specification. The gray boxes represent vertices in the graph, i.e., tasks, while the small white boxes on gray boxes represent data required and produced by tasks. These data have shapes, which are specified in parentheses. Data dependencies are specified by directed edges, i.e., directed arrows in the figure.

Task parallelism is expressed at this level. The graph can be considered as a network of tasks, which run in parallel. Note that the directed graph merely expresses data dependencies, i.e., it does not provide any specific scheduling solution. However from the dependency specification, a minimum execution order can be deduced. The graph should be acyclic, otherwise it results in self-dependency, which is not allowed in Array-OL.

2.2.3 Data parallelism

A local-level Array-OL specification is expressed by a *repetition context task* (RCT), which defines a repetition context for certain *repetitive tasks* (RTs), i.e., the RTs are repeated in the RCT. ODTs connect an RCT with its RTs, which define how the input/output arrays of the RCT are regularly accessed by the RTs. Repetitions (or instances according to different context) of an RT are supposed to be independent from one to another in general. The number of repetitions of an RT is determined by the *repetition space* associated to this RT. The repetition space is also defined as a multidimensional array. The product of all the elements in the repetition space denotes the number of repetitions of this RT. From the point of view of some languages that have loop operators, each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space defines the bounds of the loop indexes of the nested parallel loops [19].

Data parallelism [109] is expressed in the local model. Data parallelism is considered as a kind of explicit parallelism for Single Instruction Multiple Data (SIMD) machines and vector machines. Data parallelism is expressed in data parallel languages, where operations are specified to run in parallel over collections of data. The parallelism is conceptually simple and deterministic, compared to task parallelism, where asynchronous accesses of some share memory may result in *non-determinism*. As data can be expressed in a uniform way, data

parallel languages are well adapted in large scale computing applications. This is the case of Array-OL, where data are in the form of multidimensional array, and operations are stateless tasks.

An RCT and any of its RTs do not have the same interfaces, therefore a special operator, called *tiler*, is used to describe how to bridge between the two interfaces. A tiler describes how an array can be cut into subarrays with the same shape in a regular way or how some subarrays are used to build an array. These subarrays can be also multidimensional arrays, which are inputs/outputs of the RT. Whereas, the whole arrays are inputs/outputs of the RCT. In order to distinguish the different usages of these subarrays, they are called *tiles* in the case that they are a part of an array that belongs to an RCT, in contrast to *patterns*, which are taken as inputs/outputs of an RT. A tiler contains the following information, from which the subarrays can be taken (resp. stored) from (resp. in) an array, i.e., the correspondence between coordinate indexes of the array and their subarrays.

- F: a *fitting* matrix (how array elements fill the tiles).
- $\mathbf{o}$: the *origin* of the *reference tile* (for the *reference repetition*).
- P: a *paving* matrix (how the tiles cover arrays).

Fitting. In order to determine a tile in an array, a *reference element* should be given to indicate the origin point, from which the positions of all its other elements in the tile can be found. The *fitting* matrix is used in the computing to determine these elements in the tile. Their coordinates, denoted by $\mathbf{e_i}$, are computed as in the following equation:

$$\forall\, \mathbf{i}, [\mathbf{0}, ..., \mathbf{0}] \le \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = (\text{ref} + F \times \mathbf{i}) \bmod \mathbf{s}_{\text{array}} \tag{2.1}$$

where $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern (the same shape as the tile), $\mathbf{s}_{\text{array}}$ is the shape of the array, F is the fitting matrix and $[\mathbf{0}, ..., \mathbf{0}]$ denotes a matrix filled with 0, whose shape is identical to $\mathbf{s}_{\text{pattern}}$.

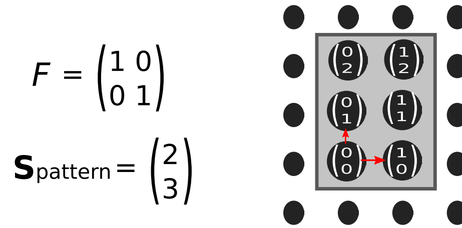


Figure 2.3: An example of fitting. The shape of the pattern is determined by $\mathbf{s}_{shape}$, thus this pattern has 6 elements, which are indexed by vectors. The pattern can be mapped onto an array, which is illustrated by the right-hand figure.

Figure 2.3 shows an simple example of how fitting works. Here, there are 6 elements in this tile since the shape of the pattern is $\binom{2}{3}$. The reference element is represented by vector $\binom{0}{0}$. The indexes of the remaining elements are thus $\binom{0}{1}, \binom{0}{2}, \binom{1}{0}, \binom{1}{1}$, and $\binom{1}{2}$. The positions of these elements in the tile relative to the reference point are determined as follows:

$$F \times \binom{0}{0} = \binom{0}{0}, F \times \binom{0}{1} = \binom{0}{1}, F \times \binom{0}{2} = \binom{0}{2}, F \times \binom{1}{0} = \binom{1}{0}, F \times \binom{1}{1} = \binom{1}{1}, F \times \binom{1}{2} = \binom{1}{2}.$$

For each tile, its own reference element should be specified.

Paving. A similar mechanism to fitting is used to determine all the reference elements of each tile in the same array, which is called *paving*. The reference element for paving is given by the *origin* vector, referred to as **o**. Each reference element of other tiles is built relatively to this one, similar to *fitting*. The following equations illustrates the computing:

$$\forall\, \mathbf{r}, [\mathbf{0}, ..., \mathbf{0}] \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \text{ref}_{\mathbf{r}} = (\mathbf{o} + P \times \mathbf{r}) \bmod \mathbf{s}_{\text{array}} \tag{2.2}$$

where $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and $\mathbf{s}_{\text{array}}$ the shape of the array. Figure 2.4 illustrates an example of paving for a [9 × 8]-array, which is composed of 6 tiles.

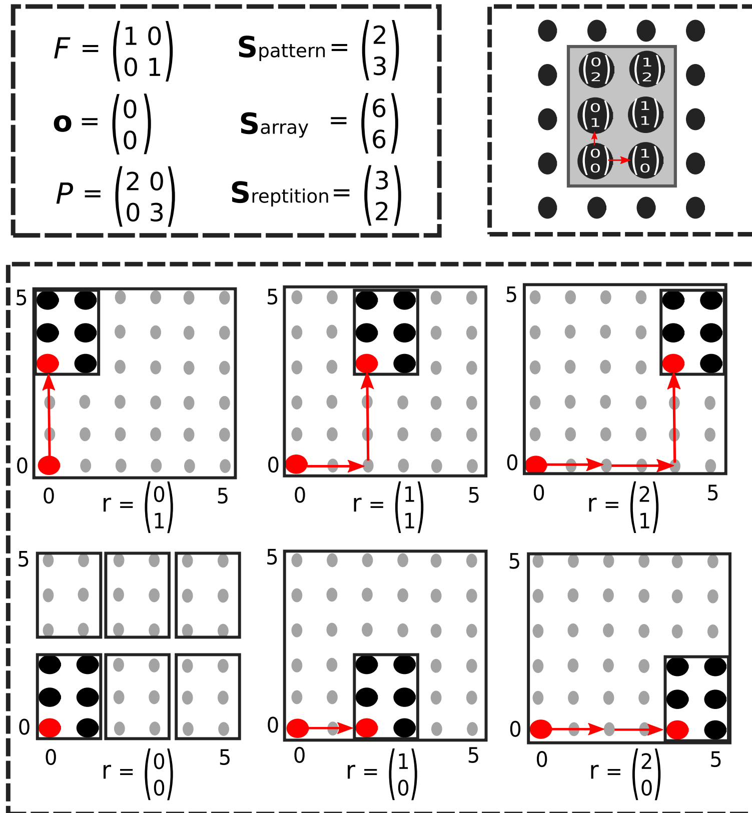


Figure 2.4: Paving example: a 2-dimension pattern tiling perfectly a 2-dimension array. The figure at the bottom shows how the array is tiled six times respectively accroding to **r**. The arrows in these figures illustrate how each reference element of the tiles are calculated from the *origin* of the array.

An example of repetitive context task. A simple example (Figure 2.5) of matrix multiplication is illustrated by using the Array-OL language. The matrix $A1$ is with the size [5, 3] and $A2$ is with the size [2, 5]. The matrix $A3$ is the matrix product of $A1$ and $A2$: $A1 \cdot A2 = A3$. The matrix product can be decomposed into *scalar products* of each line vector of $A1$ and every column vector of $A2$. Because there are no dependencies between these scalar products, they can execute in parallel. Figure 2.5 shows the Array-OL specifications of this matrix multiplication. The elementary task is scalar product, which takes two vectors as inputs

and produces one vector as output. The vectors here are tiles taken from (stored in) the corresponding arrays according to the associated tiler information. The repetition space associated to the scalar product task is [2, 3], which signifies:

- the scalar product task is repeated 2×3 (i.e. 6) times.
- the tiles are processed by a repetition of the task at the same point in the repetition space, i.e., the repetition space defines the correspondence between input tiles and output tiles.

The red dotted arrows and green arrows in the Figure 2.5 illustrate how tiles are obtained from array according to the repetition space.

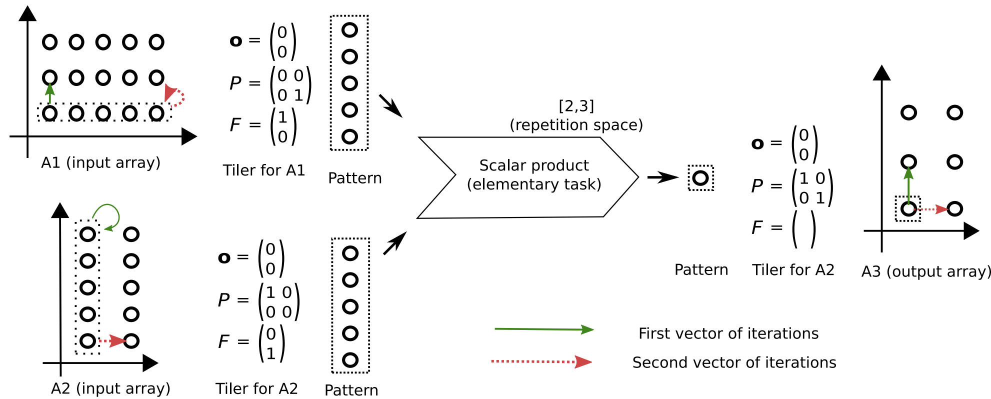


Figure 2.5: An example of scalar product illustrated by an original graphical formalism of Array-OL, which is different from the one of [19]. This graphical formalism is adopted here in order to explicitly illustrate the function of tiler.

2.2.4 Compilation and optimization of the Array-OL language

As a specification language, Array-OL is independent of execution platforms. It is not associated with an execution model. The advantage is obvious: the designer is liberated from low-level execution issues. However, when an Array-OL specification is implemented, execution issues should be taken into account. Implementation of an Array-OL specification can be considered as a language compilation. However, this compilation is difficult to accomplish in only one step because of the distinct differences between a programming language and a specification language. In order to simplify the implementation, a re-factoring step is introduced, at which an Array-OL specification is transformed to be implementable on an execution platform [121, 32]. The re-factoring can be served as an optimization with regard to a specific execution platform, for the purpose of better performance [19]. Some re-factoring functions have been proposed in the theses of Julien Soula [121] and Philippe Dumont [32]. These re-factoring functions allow to create or remove hierarchies, to manipulate repetitions, etc. The second step of compilation concerns the implementation of the re-factored Array-OL on specific execution platforms.

2.2.5 Inter-repetition dependency

In the first proposition of Array-OL, repetitions of an RT are independent from one to the other, hence there are no data dependencies between these repetitions. This modeling contributes to obtain maximum parallelism in the specification, but also adds a limit on its application, since no dependency can be specified between these repetitions. The extension of Array-OL with *Inter-repetition dependency* (IRD) [19] addresses this problem. It allows to model uniform dependencies between repetitions in the form of patterns. An inter-repetition dependency connects one of the outputs of an RT with one of its inputs in the condition that the type of this input and output must be identical. From the point of view of the RT itself, this inter-repetition dependency makes it self-dependent. However, in the repetition context of the RT, a *dependency vector* associated to the inter-repetition dependency is used to manifest one repetition of the RT relies on another one (or some repetitions rely on some other ones), i.e., it expresses the dependency relation in terms of vector. If the depended repetition is not defined in the repetition space, a default value is then chosen.

2.3 The need for design environment and methodology

The first version Gaspard [128] (Gaspard classic) was a specification environment dedicated to signal processing applications. It extends the Array-OL language. A basic compilation strategy of Array-OL was defined but it does not allow to generate efficient code from any Array-OL specification, especially those who define infinite arrays. So transformations have been proposed to re-factor such Array-OL specifications to certain hierarchical ones that can be compiled to produce efficient code on parallel (and even sequential) architectures. But the Array-OL compiler provided by Gaspard remains limited with regard to the following aspects:

- time: the development of a compiler through classical programming is time-consuming, and the validation of the compiler shares the same feature.
- maintainability: reading and modification of the source code of a compiler for the purpose of its maintenance always demand considerable effort.
- flexibility: the extension of a classical hard-coded compiler is not obvious, even if all the interfaces of used functions are well defined.
- re-usability: one compiler is designed for one specific language in general, hence the re-usability is a problem when some parts of the compiler are expected to be reused.
- documentation: hard code is not a good way to be used as documentation, even if it is provided with comments.

Moreover, the complexity of integration of different heterogeneous components, such as software and hardware components, on a single chip with functionalities specified in Array-OL is another issue. As these components are heterogeneous, they usually have different formalisms as well as different tools and platforms with regard to analysis, validation, simulation, etc. These difficulties call for new technologies, methodologies and platforms for the specification, modeling, code generation, simulation, validation and synthesis of intensive signal processing.

On the contrary, the *model*-based approach has been widely adopted not only in the industry but in the academia. Model-based developments are becoming interesting in relation to the constraints mentioned previously. In the next chapter, MDE and an MDE-based version of Gaspard, i.e., Gaspard2, will be presented.

2.4 Conclusions

The Array-OL formalism dedicated to systematic intensive signal processing is presented in this chapter. Nowadays, it is not limited for the specification of signal processing, other similar processing, e.g., data-intensive processing, is also its application domain. Hence, we call all these kinds of processing *data-parallel intensive processing* (DIP), which defines the application domain of Array-OL and the context of this thesis.

Array-OL utilizes the multidimensional array data structure for the specification of intensive data, which benefits from several advantages: toric array can be specified for some special applications, such as sonar signal processing and frequency processing; temporal and spatial dimensions are processed in the same way, hence a maximum parallelism is specified, which can be re-factored according to an architecture when the application is mapped onto the architecture; ODT operators allow a high-level data dependency specification (such as patterns) with regard to the manipulation of array indexes, as patterns are also array. This kind of dependencies enables the specification of multi-granularity degrees, which makes the application specification flexible. Array-OL only specifies data dependency, and it is independent from any execution model, which contributes to a fast application specification. Properties, e.g., single assignment, are defined in Array-OL to guarantee the correctness of specification.

The previous characteristics make Array-OL distinct from other languages in the same application domain, such as SDF, Alpha, StreamIt, synchronous languages and HPF. However, as Array-OL is a specification language, it is possible to project it onto the execution models provided by the previously mentioned languages [4, 33, 125, 133]. The following chapters concentrate on the projection of Array-OL languages onto synchronous languages, for application validation issues. Synchronous languages also contributes to the integration of control with formal semantics into DIP. All these works have been achieved, however, in the Gaspard2 environment, which takes Array-OL as core formalism of DIP.

Chapter 3

Model-Driven Engineering and Gaspard2

3.1 Model-Driven Engineering **26**
3.1.1 Model and modeling 26
3.1.2 Metamodel and metamodeling 28
3.1.3 Model transformations 30
3.1.4 MDE in practice 33
3.1.5 Conclusions 38
3.2 An MDE-based environment for SoC co-design: Gaspard2 . . . . **39**
3.2.1 High-level co-modeling for SoC design 40
3.2.2 Gaspard2 and MDE 43
3.2.3 Modeling of high-level control in Gaspard2 44
3.3 Conclusions **45**

In Chapter 1, the advance of hardware, particularly in SoC, has been presented. On one hand, the computing capability provided by a processing unit increases very quickly, on the other hand, the parallel architectures play more important role. This leads to the gap between software development and the hardware computing capability, as the former does not benefit the same advancement rhythm as the latter. Moreover, following the computing capability enhancement, computer systems or SoC are apt to be more complex, particularly, when heterogeneous components are integrated together in the system. Classical programming languages are becoming difficult to be adapted in current system designs. The Array-OL language presented in Chapter 2 exhibits these problems. Dedicated to DIP, it is difficult to fix its position in current signal processing applications implemented as SoC or even MPSoC.

However, new software design and development methodologies, which are proposed according to current system designs, emerge continuously. Among intensive research activities that are dedicated to address fast and efficient software design issues, MDE [116, 89] is one of the most promising approaches that are unavoidable to be referred to. In Section 3.1, MDE is briefly discussed, with emphasis on its principles. Then, Gaspard2, as a result of the evolution of Array-OL adapted to MDE, is presented in Section 3.2.

3.1 Model-Driven Engineering

According to Wikipedia [1]: "MDE pertains to software development, which refers to a range of development approaches that are based on the use of software modeling as a primary form of expression." With the fast development of MDE, it has become a promising approach of software engineering, which attracts much attention in industry and academia. In this dissertation, MDE plays a very important role, which contributes to modeling, automatic code generation and a bridge between different technologies.

The key element in MDE is *model*. Two core relations around model in MDE is *representation* (model is a representation of a system) and *conformance* (model is conform to a meta-model) [18]. These two relations are separately presented in Section 3.1.1 and Section 3.1.2. As another key notion of MDE, *model transformation* is also discussed in Section 3.1.3. The advantages of MDE are exhibited progressively following the introduction of the previous concepts. MDE in practice is also considered in this section.

3.1.1 Model and modeling

A model refers to a representation of system with an accepted level of abstraction, i.e., all unnecessary details of the system are neglected for the sake of simplicity, formality, comprehensibility, etc. However, the model concept is not a novelty. According to Favre [40], the notion of model casts back to more than five thousand years. The alphabet of Ugaritic cuneiform (3400 B.C.) already introduced the same notion by defining a set of abstract representations (characters) and rules (pronunciations) that allowed the expression of some reality (sentences). More recently, in the information technology, programming languages, relational data bases, semantic web, etc., are all based on the same idea, where a set of predefined and linked (or concatenated) concepts represent some reality once they are given certain interpretations.

Intuitively, according to different granularity degrees of abstraction, there may exist several levels of abstraction. But accepted levels of abstraction may not be unique, which is determined up to certain specific requirements. However the choice of a good level of abstraction does not implies a simple and easy work. First, the evolution of abstraction level in software design is briefly discussed, which partly explains the model notion in MDE.

3.1.1.1 Raising levels of abstraction in software design

The history of software evolution can be considered as a history of raising abstraction levels. At the very beginning, machine code (first generation languages) helped people to escape from direct manipulations of physical elements in a machine, but it is still not easy because programming with large number of "1" and "0" does not imply an interesting work. The later assembly languages (second generation languages) help to do this tedious work through substituting the numbers by some literal instructions, but it still remains difficult for programmers because they need to know hardware instructions very well. It turns out to be a big obstacle for those who do not know anything about hardware. Efforts were made so that programming languages become independent from specific machines. High-level languages (third generation languages), such as FORTRAN, LISP and C, make it possible. Developers can put their focus on the functionalities, which appear more interesting to them. Some of

[1] http://en.wikipedia.org/wiki/Model-driven_engineering

these languages, e.g., APL (A Programming Language), C, PROLOG, and ML (ML stands for metalanguage) are major language paradigms still in use nowadays.

However, the increasing software complexity results in the software crisis, which involves the development time, cost, etc. Object-Oriented Analysis and Design (OOAD) is developed to partly address this problem. The basic principle is that a system is composed of a set of interacting objects, which are independent from one another in the sense that their local states are private and can be only accessed by some provided operations. The independence of objects makes the re-usability possible. OOAD also involves the definitions of object, class and their relations. These classes are independent from implementation concerns. They share several features, such as *encapsulation*, *modularity*, *polymorphism* and *inheritance*. At run time, objects are then created dynamically according to their class definitions. The notions, such as *class* and *object*, imply a clear separation between *specification* and *implementation*. However, the design of *classes* is still restrained by object-oriented languages and the virtual machines on which their implementing *objects* execute.

OOAD helps to address the software complexity problem to some extent. Based on the practices of object-oriented development, recent software research focuses on software/hardware modeling, domain-specific modeling, heterogeneous system integration, high-level abstraction, etc., where OOAD is not well adapted.

3.1.1.2 Modeling with machine-recognizable model

Modeling approaches also play an important role in software evolution, particularly in system analysis and modeling. Several modeling approaches that should be cited include: Merise [127] (1970), Structured Systems Analysis and Design Methodology (SSADM) [38] (1980) and Unified Modeling Language (UML) [104] (1995). These approaches help to understand the current MDE model concept. Each of them proposes certain concepts and a notation to describe the system to be designed. In general, in each stage of the system life cycle, a set of documents composed of some diagrams that allow designers, developers, users, etc., to share their system designs.

These approaches have some important advantages. The abstraction conduced by modeling allows to emphasize the overall system structure, while bypassing implementation details and specific implementation technologies. This contributes to handle big and complex systems rapidly and efficiently. These approaches enable to represent a system or a part of the system with different point of views, which allows to separate the system by aspects and to understand it with specific domain views. These approaches can also be found in the Aspect-Oriented Programming (AOP) [69] and Domain-Specific Language (DSL) [91].

However, these approaches were criticized [40] for the heaviness and lack of flexibility in a rapid software design and development. The resulting models of these approaches, which is called *contemplative* models by Favre et al., are essentially used for communication and apprehension. It remains passive with regard to production, although the first concern of information technology is to produce the artifacts interpretable by machines [40]. Hence, in order to be productive so as to accelerate system design and implementation, machine-recognizable models, which are not only human-recognizable, become indispensable and critical.

3.1.1.3 Model and MDE

MDE [116, 89] emerges to partly satisfy the requirements of two communities, i.e., programming language community (particularly OOAD community) and system analysis and modeling community. *Model* is the key concept in MDE, which is systematically utilized throughout the whole system life cycle. Unlike classes and objects in OOAD, models in MDE are flexible, as they do not require to take implementation aspects into account. The modeling method proposed by MDE also makes up for the deficiency of traditional modeling approaches through the proposition of machine-recognizable model. MDE is intended to provide a development framework, where models transit from a contemplative state to a productive state. Thus, models become the first class elements in the software development process, which aims at improving its portability and maintainability through separation of concepts, particularly separation of concepts of specific domains or technologies in order to boost productivity and quality, and also through improving its re-usability, which is always proved by the tools and design patterns.

Transition from solution space to problem space Instead of focusing on the solution space, which is a feature of programming languages, MDE helps to concentrate on the overall behavioral and structural modeling of a system without distraction due to some specific domains of computing technologies. This analysis and modeling are always carried out by using the concepts of application domains that forms the problem space, i.e., analysis and modeling of the problem itself, not a solution to the problem.

There are several direct advantages of this transition: capability of representing large-scale systems and capability of handling heterogeneous systems. First, one can focus on the problem itself without knowing too many specific computing technologies, as a result, the accessible system complexity can be raised.

Secondly, it is possible to separate different aspects of a system with different views. For instance, it allows to express a system from a global or a simplified point of view of a system. It also offers the possibility to separate the system model into different parts according to the relations between the concepts in the model. The separation of views permits equally to develop the system according to the domain aspects.

Thirdly, a model may be, although not evident, a composition of several models, each of which is adapted to a specific formalism that is appropriate to a specific domain. The composition is possible because all the models can be expressed in a common uniform language. Hence different development teams from different application domains can cooperate on the same heterogeneous system with their specific domain concepts, where implementations of different computing technologies are not involved.

3.1.2 Metamodel and metamodeling

In order to be interpretable by a machine, the expression, with which a model is represented is pre-defined formally. This is achieved by a *metamodel*. In MDE, a metamodel allows the definition of a model in a model specification language, which defines the concepts and their relations that are available during designing the model, i.e., it defines the syntax of models.

A model that is designed according to a given metamodel is considered to *conform* to this metamodel at a higher level. This relation is analogous to a text and its language grammar. As the name implies, a metamodel is also a model, i.e., it conforms to another metamodel. In

order to define a model, it is not convenient to define an infinite succession of metamodels, each of which conforms to another one at a higher level. One formal solution to this issue is the definition of a metamodel, which conforms to itself, i.e., it can be expressed only using the concepts it defines. Currently, widely used metamodels, such as Ecore [34] and MOF [99], are examples of such kind of metamodels or metametamodels.

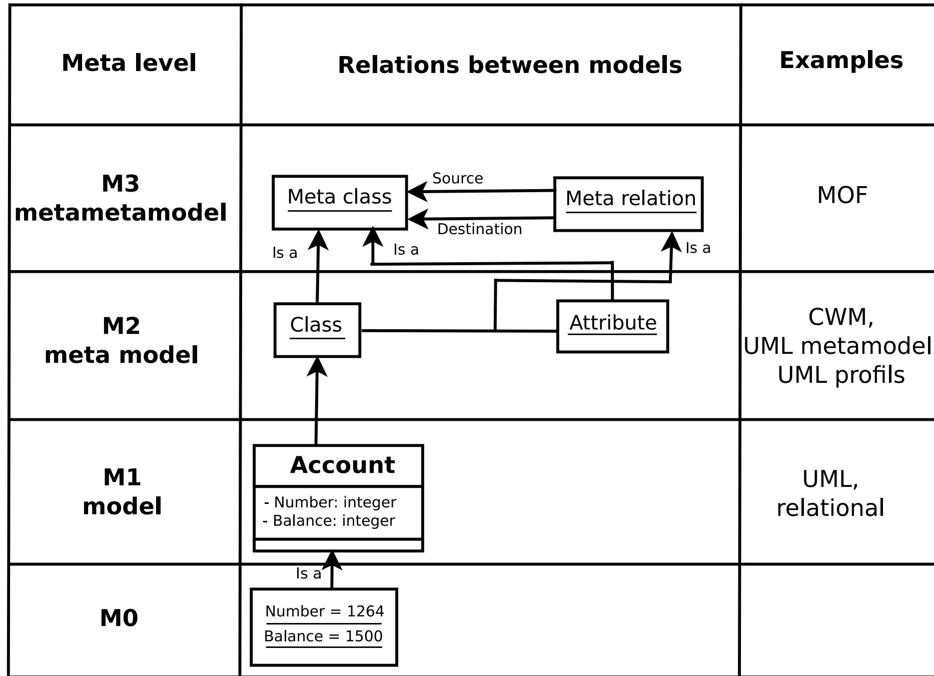


Figure 3.1: Different levels of modeling.

Figure 3.1 illustrates the relation between models and metamodels. The **M0** level is the representation of some reality (a computer program). In this example, several variables (*Number* and *Balance*) take values that are assigned to them. The **M1** level is the lowest level of abstraction, where the concepts can be manipulated by developers. In this example, declarations are found for the variables used at the **M0** level and the notion of *Account*, which contains these variables. The model at the **M1** level conforms to the metamodel at the level of **M2**. The concepts manipulated by developers at **M1** are defined and situated at this level. One of the best known metamodel is UML metamodel. *Account* is a *Class*, whereas variable declarations are *Attributes* enclosed in the *Class*. Finally, a metamodel at the **M2** level conforms to a metametamodel (at the level of **M3**). The latter conforms to itself. In the example, the concepts, such as *Class* and *Attribute*, are metaclasses, whereas the containing relation is a metarelation. The metametamodel can describe itself, e.g., metaclass and metarelation are still metaclasses and relations, such as source and destination, are metarelations.

If the highest-level metamodel is defined so as to conform to itself in a formal way, and the syntax and semantics of this metamodel are described explicitly, then the models that conform to this metamodel can be interpreted by a computer. Once significations of the concepts in this metamodel are programed, a computer will be capable to read any model that conforms to this recursive metamodel directly or indirectly. However, a metamodel is only composed of structural information in relation to its models, no semantics are involved

formally. A model makes sense with the help of its interpretation, either by users through a provided specification, which includes the concepts of the metamodel, or by a machine during the transformation of the model.

3.1.3 Model transformations

Model transformation is another key concept in MDE, which is always considered as a routine work in contrast to the *intellectual* modeling. Models in MDE are used for not only understanding and communication, but generating concrete results in the development, such as source code. With the help of metamodel, to which they conform, models can be recognized by machines. As a result, they can be processed, i.e., a model is taken as input (source) and then some models (target) are generated. This process is called model transformation on condition that both source and target models conform to their explicitly specified metamodel respectively.

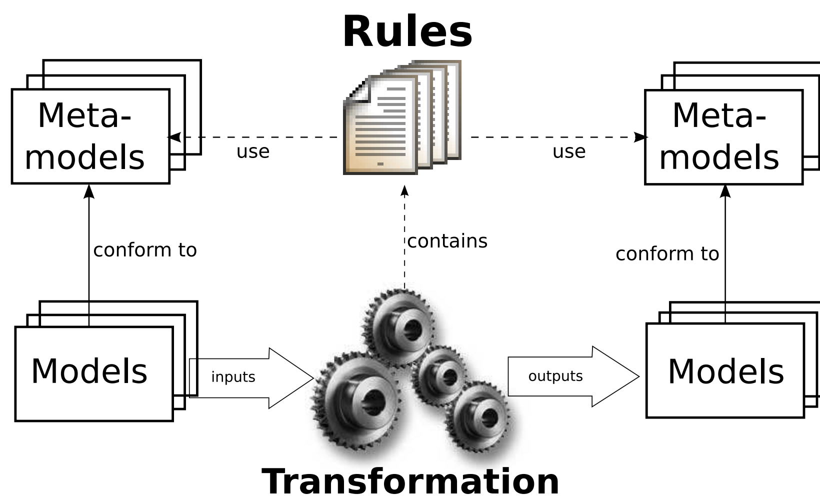


Figure 3.2: A model transformation allows to transform source models into target models in consideration of a set of rules. These rules are defined by using the concepts of the metamodels, to which the source/target models conform.

3.1.3.1 Model transformation classifications

Transformations can be classified according to different point of views. In the next, several proposed classifications are briefly presented. According to the homogeneity and heterogeneity of the models in the two sides of transformation, [90] proposes two kinds of transformations: *exogenous* and *endogenous*. An endogenous transformation only considers one metamodel, i.e., the same metamodel for the source model and the target model. An exogenous transformation uses different source metamodel and target metamodel. According to the abstraction level of source and target models, a transformation can be considered as a vertical one, when two levels are different, or a horizontal one when the models are situated at the same level of abstraction.

In addition to the unidirectional transformation, whose direction is implied by *source* and *target*, a transformation can also be bidirectional. In the unidirectional transformation case,

only source mode can be modified by users, then target model is re-generated accordingly. However, in the bi-direction case, the target model can be also modified, which requires the source model to be modified in a synchronized way. Consequently, bidirectional transformation always leads to a model synchronization issue. [122] presents a survey on bidirectional transformations.

3.1.3.2 Transformation rules

Model transformation is always implemented by an engine, which executes a set of (transformations) rules. The rules can be declarative (which outputs are obtained from some given inputs) or imperative (how to transform). For instance, declarative rules are, in general, expressed in three parts: two *patterns* and a *rule body*. The two patterns are source pattern and target pattern respectively in a unidirectional transformation or both source/target patterns in a bidirectional transformation. A source pattern is composed of some necessary information about part of the source metamodel, according to which a segment of source model can be found and transformed. Correspondingly, a target pattern consists of some necessary information about part of the target metamodel, according to which a segment of target model can be generated. The link between these two patterns is the rule body (or a logical part according to [29]), which defines the relation between the source pattern and the target pattern. Figure 3.3 shows an example of a transformation rule in TrML [37]. In this example, the source pattern is *Tiler*, the target pattern is *Node* and the rule body is called *GTiler2SNode*. Moreover, external functions can be declared and associated with the rule body, which is illustrated with the *action* annotation.

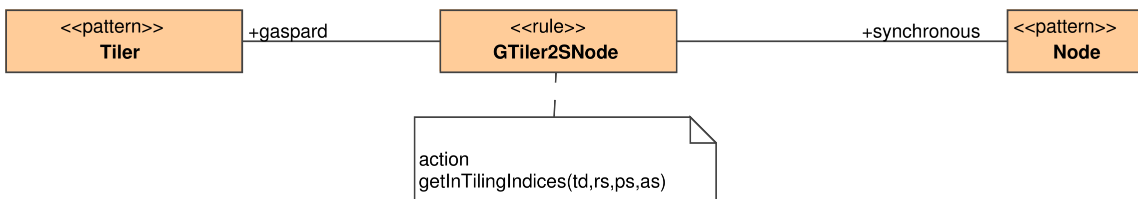


Figure 3.3: A transformation rule expressed with TrML.

Declarative rules can be composed in a sequential way or a hierarchical way. Thus, flexibility and re-usability of transformation can be obtained. In the sequential case, all the rules can be executed one by one, hence, all the source patterns of these rules cover the source model and all the target patterns cover the target model. In the hierarchical case, the root rule can have sub-rules. Its source/target patterns cover directly the whole source/target models. Transformation rules are in general mixed-style rules so that complex transformation can be implemented.

3.1.3.3 A multi-level approach in modeling and transformation

Between the abstraction levels of a model and its resulting code, intermediate levels can be created. At each level, a model and its metamodel are defined, hence a complete model transformation turns into a transformation chain, which consists of successive transformations. Apparently, these intermediate models are not added in order to increase the work-

load. On the contrary, they are added when it is difficult to bridge the gap between two models directly. For instance, the models at the two levels are too different, which leads to the requirements of supplementary information. A typical example is the Platform-Specific Model (PSM) defined in Model-Driven Architecture (MDA) that is situated between Platform-Independent Model (PIM) and the resulting code. This multi-level approach contributes to reduce the complexity of transformation. For instance, the information needed to transform a high-level model to a low-level one is divided into several portions, each of which is included in a transformation. Hence it makes the transformations modular, i.e., modifications of one transformation will have no effect on other transformations if the intermediate models are not changed. Another advantage is that the development of a chain of transformations can be concurrent once intermediate models are defined.

3.1.3.4 Traceability

In some cases, model transformation information is expected to be logged for certain usage. For instance, relations between the elements of source and target model and modifications or debug information of the target model, are needed to be logged in order to backtrack and find the corresponding elements in the source model. Traceability in the model transformation consists in finding the transformation relation between the elements in source/target pattern. For instance, a *trace* can be observed and saved in the execution of a transformation [3], which enables the traceability. However, traceability is still not well supported in the transformation tools currently.

3.1.3.5 Productivity issue

The modeling approach proposed in MDE and its corresponding model transformation help to address the productivity issue. As mentioned in Section 3.1.1, high-level modeling reduces the complexity of system design, hence it contributes to improve productivity. Moreover, one of the distinct features of MDE from other modeling approaches is: models can be directly used to generate implementation-level results (e.g., executable source code) from high-level models. This production is achieved by automatic model transformations.

3.1.3.6 Transformation tools

Meta-Object Facility (MOF) Query/View/Transformation (QVT) [101] is an Object Management Group (OMG) standard on model query and transformation. Several transformation languages and tools, such as ATLAS Transformation Language (ATL) [64] and Kermeta [66] already exist. ATL is a model transformation language (a mixed style of declarative and imperative constructions) designed according to QVT. Kermeta is a metaprogramming environment based on an object-oriented Domain Specific Language. Eclipse Modeling Framework Technology (EMFT) project was initiated to develop new technologies that extend or complement Eclipse Modeling Framework (EMF). Its query component offers capabilities to specify and execute queries against EMF model elements and their contents. EMF Java Emitter Templates (JET) [35] is a generic template engine for the purpose of code generation. The JET templates are specified by using a JSP-like (JavaServer Pages) syntax and are used to generate Java implementation classes. Finally, these classes can be invoked to generate source code, such as Structured Query Language (SQL), eXtensible Markup Language

(XML), Java source code.

3.1.4 MDE in practice

MDE is still not completely well-rounded, and there still exist propositions and initiatives. In particular, we can find them with various rules, under many different names, such as Model-driven Architecture (MDA), Model-Driven Development (MDD), Model Integrated Computing (MIC) and Model-Driven Software development (MDSD). We insist on the essence of all proposals that form a foundation of principles and concepts rather than the subtle nuances implied by these different names.

3.1.4.1 MDA

One of the best known MDE initiative is MDA [97, 92], which is proposed by OMG [96]. In MDA, two types of models are distinguished according to the abstraction level: PIM and PSM. The former generally expresses functional requirements of a system, while PSM generally involves implementation concerns, e.g., a PSM can be an executable model itself, or be used to generate certain source code. Transformation specifications are also proposed by OMG to bridge these two types of model, such as MOF QVT [101].

3.1.4.2 UML

UML is taken as one of the main unified visual modeling language in MDE. The UML metamodel [104] was standardized in 1997 by OMG. As a standard, UML has been widely adopted in industry and academia. It is proposed to answer the requirements of modeling specification, communication, documentation, etc. It takes the advantages of component re-use, unified modeling of heterogeneous system, different facet modeling of a system, etc. The proposition of UML is based on several languages, such as OMT, Booch and OOSE, which had a great influence on the object-based modeling approach. Consequently, UML is very similar to object-based languages. As UML is wide spread in industry and academia for modeling purpose, large number of tools[2] have been developed to support it.

UML distinguishes structural and behavioral modeling. The fist one concentrates on the static structure of a system, which involves the constructs, such as *class*, *component* and *deployment*. The second one focuses on behavioral aspects of the system, which can be expressed by *activities*, *interactions*, *state machines*, etc.

However, its expressivity and precision are not always well defined in certain cases for the specification of some specific systems. UML can be extended through *profiles*, which are composed of *stereotypes*, tagged values, etc. The latter allow to specialize UML classes and to be attached with *tags*, which make it possible to add attributes to these classes.

There are also discussions on the semantics of UML. Some believe its semantics is not well defined. In particular, the semantics of UML behavioral modeling brings certain ambiguities [43]. This problem cannot be addressed by Object Constraint Language (OCL), which is dedicated to the specification of static syntactic constraints on UML constructs. From this point of view, the validation of UML applications cannot be achieved in a very precise way. Related works [118, 51] have been carried out to give a clear and formal semantics to UML.

[2]http://en.wikipedia.org/wiki/List_of_UML_tools

UML components

A UML component is a self-contained, modular and reusable unit that is considered as an autonomous unit in a system or a subsystem. It is also replaceable in its environment at design time and run-time if its fungible component has compatible interfaces. A component is provided only with its specified interfaces and the functionality that it provides. Its implementation is concealed, and its behavior is defined in terms of its interfaces.

As a subtype of *class*, a UML component has an external view (or *black-box* view) through its publicly visible properties and operations. Moreover, a *behavior*, such as state machines, can be associated with the component in order to express a more precise external view. A component also has an internal view (*white-box* view) via its private properties and realizing classifiers. This view shows how the external behavior is realized internally [103].

State Machines

UML state machines are an object-based variant of *State charts* [58]. UML StateMachine package defines similar concepts to *State charts* that can be used for either complex discrete behavior modeling, or the expression of the usage protocol of a part of system. The former is called *behavioral state machines* and the latter is called *protocol state machines*, which allow the specification of a life-cycle of some objects or invocation order of its operations. *Protocol state machines* are not involved in implementations; in contrast, they enforce legal object usage scenarios.

Figure 3.4 illustrates an extract of the metamodel of UML state machines, which includes the main concepts of UML state machines and their relations. Next, main concepts of UML state machines are briefly presented:

StateMachine : a *StateMachine* is a concept used to exhibit the behavior of a part of system. This behavior can be expressed by execution traces of the state machine, obtained when transiting between states. The transitions are fired by events. During this execution, a series of activities associated with the elements of the state machine can be carried out. A *StateMachine* can be a sub-machine, i.e., it refines a state in another state machine. A state machine may have *Regions* and *Pseudostates*.

Region : a *Region* is an orthogonal part of either a composite state or a state machine [103]. Simply, a region is introduced as an intermediate elements in order to describe the relation between state machines and other concepts used in them (e.g., states and transitions). A region may contain vertices (states and pseudostates) and transitions.

Vertex : a *Vertex* is similar to a node if state machines are considered as node-edge graphs. But a node does not necessarily imply a state, i.e., a vertex can be also a *Pseudostate*, which is not a state but conveys some information about some states or the state machines.

State : A *State* in a state machine can be any of the following kinds: *simple state*, *composite state* and *sub-machine* state. Composite states and sub-machine states make it possible to define state machines in a hierarchical way.

- A *simple state* indicates the state is not refined, hence, it is a normal state. *FinalState* is a special state, which indicates the termination of the system.

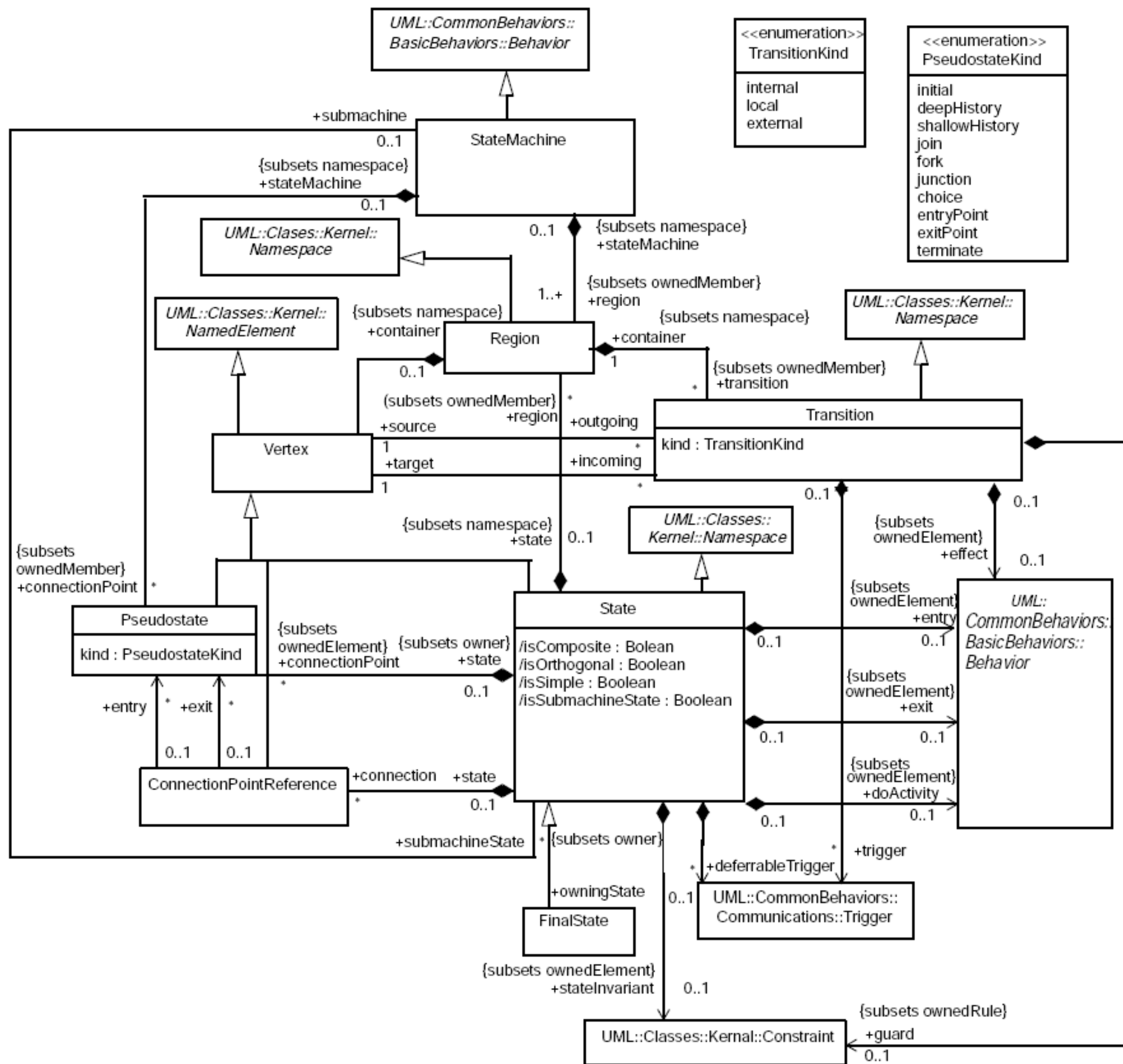


Figure 3.4: An extract of the metamodel of UML state machines [103].

- A *composite state* implies the state is refined by some states, transitions, etc., which are defined explicitly in the composite state. Figure 3.5 shows an example, where the state *EffectOn* is refined. The refinement is explicitly defined by a set of interconnected state and transitions in the state *EffectOn*.

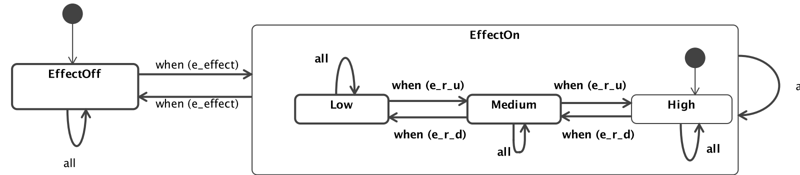


Figure 3.5: An example of composite state. This state machine represents a video effect switch. User can switch the effect on, where one of the three effects can then be selected: Low, Medium and High. User selection events can trigger the transitions, which is described in **Transitions** section.

- A *sub-machine state* suggests that the state is refined by another state machine, which is considered as a sub-machine. This enables re-usable state machines, which can be referred to in some states of other state machines. Compared to a composite state, whose refinement is directly defined in the state, the refinement of a sub-machine state can be defined in another StateMachine diagram. Figure 3.6 illustrates two state machines, which are defined in two different diagrams. The right-hand one, which is called *EffectOnMachine*, is referred to as a sub-machine of the left-hand one through the state *EffectOn*.

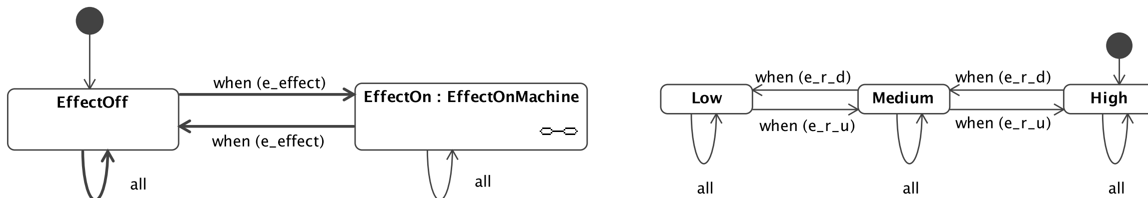


Figure 3.6: An example of a sub-machine state, which is semantically equal to the one in Figure 3.5.

Composite states and sub-machine states contribute to construct hierarchical state machines. Previously mentioned composition of states is considered as mutual exclusive composition, i.e., the states are active in an exclusive way, which are called *OR* states. States can also be composed together in a parallel way, i.e., they can be active at the same time. These states are called *AND* states. Figure 3.7 shows an example that contains both *AND* states and *OR* states.

Pseudostate : pseudostates can be classified into several families: *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, etc. These pseudostates convey the corresponding information of the states or the state machine they are connected to.

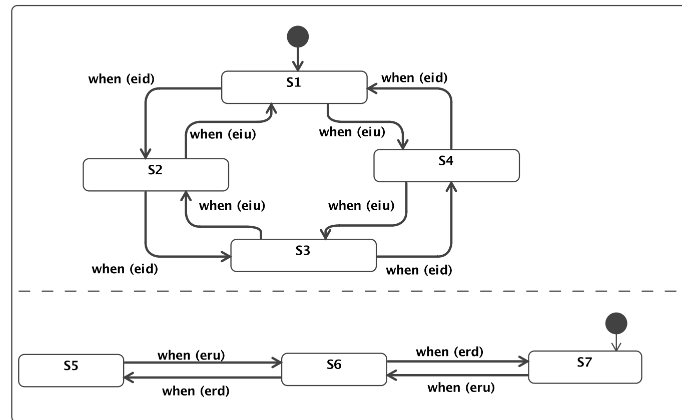


Figure 3.7: An example of the parallel composition of state machines.

- *initial* is connected to one of the states in a state machine (more precisely, in a region), which is the initial state of the region.
- *deepHistory* represents the most recent active states in a composite state that contains this pseudostate. This pseudostate is connected to a state if there is no most recent active states (e.g., the composite state is entered for the first time). Otherwise, it indicates which states should be reactivated upon entering the composite state. Only one deep history pseudostate is allowed in a composite state, which is inconsistently specified in UML [43].
- *shallowHistory* is similar to a deep history pseudostate, except that it indicates a direct sub-state of the composite state (unlike a deepHistory, which memorizes all the states to re-enter in the hierarchy). The sub-state denoted by shallow history pseudostate is actually the initial state for the composite state (inconsistency in UML [43]).

Transitions : a transition is a directed connection between a source vertex and a target vertex (state or pseudostate). A transition can have several triggers, any satisfaction of these triggers can fire the transition. In the previously illustrated examples, the prefix *when* on a transition signifies a trigger associated with *ChangeEvents*. The prefix *all* indicates a self transition when no triggers are satisfied.

Collaboration

A collaboration specifies the relation between collaborating elements from a point of view of the functionality that they cooperate to accomplish. However, a collaboration is not intended to define an overall structure of these elements. These elements in a collaboration are called *roles*, whose properties or identification can be ignored, i.e., only their useful properties and types are referenced in the collaboration.

Figure 3.8 illustrates an example of collaborations, which are defined in a composite structure diagram with components. The two dashed ellipses express two collaborations, which

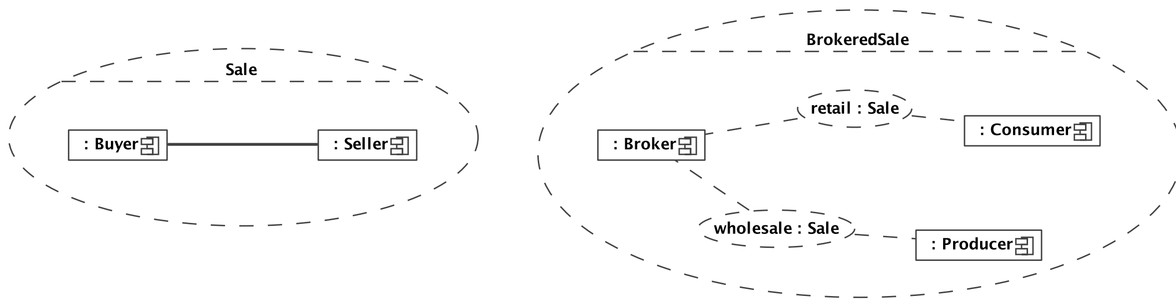


Figure 3.8: An example of collaborations taken from [103].

are called *Sale* and *BrokeredSale* respectively. *Sale* describes the collaborating relation between *Seller* and *Buyer*. *Sale* is then used in the *BrokeredSale* to depict a more complex collaborating relation between *Broker*, *Producer* and *Consumer*.

3.1.4.3 Profiles for real-time and embedded system design

Systems Modeling Language (SYSML) [102] is an OMG standard that aims at describing complex systems. However, SYSML does not put the accent on embedded system design. This is not the case of the UML profile for Modeling and Analysis of Real-Time and embedded Systems [98] (MARTE for short). As the name suggests, MARTE is intended to provide sufficient concepts for the modeling and analysis of real-time and embedded systems. It, on one hand, reuses some concepts (e.g., stereotypes) proposed in SYSML, on the other, defines concepts compatible with SysML. However, MARTE provides precise concepts for the detailed modeling of hardware architecture in relation to SoC. MARTE is considered as a replacement and refinement of UML Profile for Schedulability, Performance and Time [100] (SPT). With regard to SPT, MARTE refines their time concepts in order to model three types of time: logical, discrete, and dense.

The repetitive structure modeling (RSM) package in MARTE offers the possibility of describing DIP applications. It enables to specify a regular repetition mechanism that can be applied to both software and regular hardware architecture (e.g. vector processors and cell processors). This package, which is partly compatible with Gaspard2, is proposed by the DaRT team.

3.1.5 Conclusions

As previously mentioned, MDE has several advantages: the possibility of platform-independent modeling without involvement of implementation details; re-usability and productivity of models; modeling and specification of different facets of a system from different points of view; rapid automatic model transformations. Without doubt, these advantages offered by MDE are highly attractive for the improvement of Gaspard. In next section, the integration of Gaspard into MDE is presented.

3.2 An MDE-based environment for SoC co-design: Gaspard2

The new methodology of MDE has been widely spread in the research because of the advantages mentioned in Section 3.1. It also attracts the attention of the DaRT team. The compiler of Gaspard has been integrated within the MDE framework. It leads to the emergence of Gaspard2 [65], a new design methodology and development environment dedicated to DIP applications for SoC. Gaspard2 enables fast design and code generation with the help of UML graphical tools (e.g., MagicDraw UML[3] and Papyrus[4]) and Eclipse EMF[5].

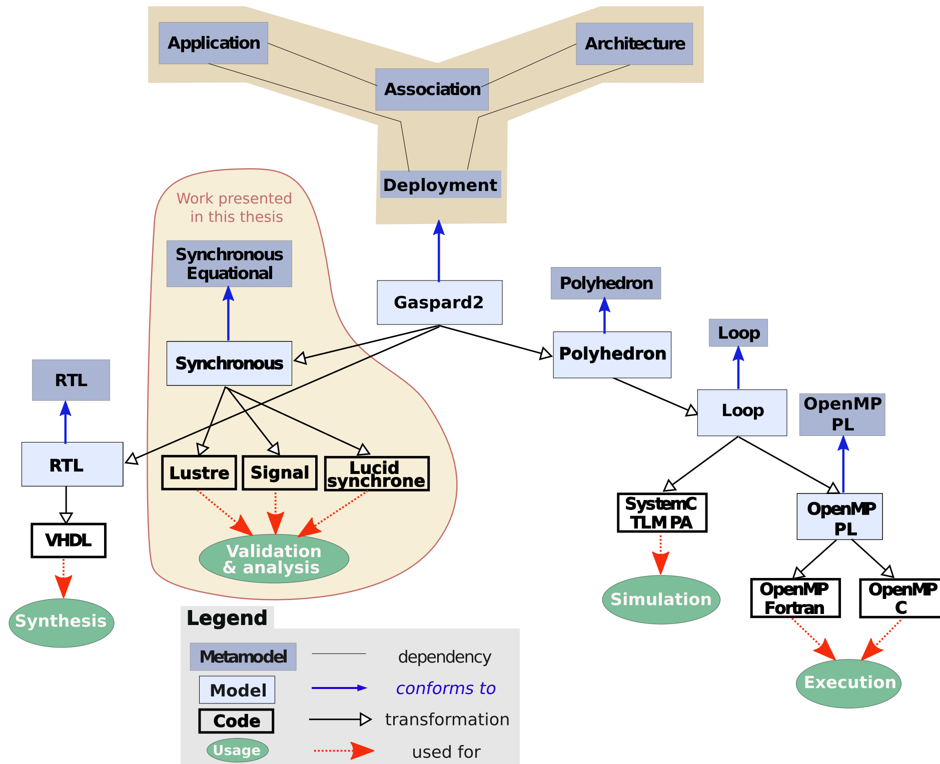


Figure 3.9: A global view of the Gaspard2 environment for SoC design.

Figure 3.9 shows a global Gaspard2 environment. The main features of Gaspard2 are classified into three families:

high-level co-modeling: it enables system design using the Gaspard2 profile [10] at a high level of abstraction, which provides various packages of the concepts of software, hardware, deployment, etc.

model transformation: transformation chains are developed to generate target code at the implementation level.

Targeting different platforms: different target code can be used for different purpose, such as validation, simulation, synthesis and execution.

These features are detailed in the following sections with the help of Figure 3.9.

[3]www.magicdraw.com/
[4]www.papyrusuml.org/
[5]www.eclipse.org/emf/

3.2.1 High-level co-modeling for SoC design

One of the most important features of Gaspard2 is its software/hardware co-modeling at a high level of abstraction. More precisely, it enables to model *software applications*, *hardware architectures*, their *association* and IP *deployment* separately, but in a unique modeling environment. This concept is partially based on the Y-chart (Figure. 3.9 and [65]). Models of software application and hardware architecture can be defined separately, independently and concurrently. Then, software application can be mapped onto hardware architecture. Moreover, supplementary information of certain software and hardware implementations (IPs, such as filters and FFT for software and processor and DSP for hardware) are described in the model, which is called *deployment*. The whole model is platform-independent, i.e., they are not associated with an execution, simulation, validation or synthesis technology generally.

3.2.1.1 Gaspard2 metamodel

The Gaspard2 metamodel is originated from the work presented in [28]. The proposition was inspired by several OMG standards, including SPT [100] for the aspect of hardware architecture, SysML [102] for the aspect of the association of hardware and software, and UML for the aspects of component. The resulting modeling also takes the Array-OL language [31, 19] into account for the specification of DIP applications.

The main features of the Gaspard2 metamodel is briefly presented here. More details about the metamodel can be found in Appendix A.

- **Component.** Gaspard2 adopts a *component*-based approach. Gaspard2 components are based on UML component constructs [103], which has been briefly presented in Section 3.1.4.2. Figure 3.10 illustrates an example of a simple component *ColorFilter*, which is stereotyped by *ApplicationComponent*. Gaspard2 components benefit from the advantages of UML components, including modular and reusable definition (and usage) both in the specification context and in the execution context, hierarchical composition once being provided compatible interfaces, encapsulation and autonomy. These features make it possible to construct complex and large-scale systems.

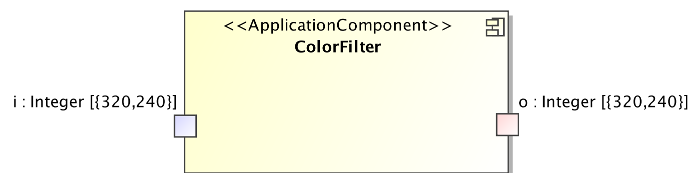


Figure 3.10: An example of Gaspard2 *Component*.

- **Factorization** is one of the most interesting feature provided by Gaspard2 metamodel. With the help of the concepts defined in this package, Gaspard2 metamodel is capable to express data parallelism in a compact manner. Factorization is inspired by Array-OL, for the specification of regular repetitive structures present in software, hardware and association. Several concepts defined in this package are described in the next:

  Shape can be specified for an instance or a port of a component. It is specified through the *multiplicity* property of the instance or the port, which implies a collection of the

corresponding elements. This collection is defined in the form of *multidimensional array*, whose elements are positive integers indicating the maximum number of elements stored in the corresponding array dimensions. For instance, a shape, [40, 30], defined for an instance in a repetition context indicates that the instance is repeated 40 × 30 times. This shape is also called *repetition space* of this instance. This concepts is inspired by the bound notion of the parallelly nested loops, which are present in high-performance computing languages [19]. The same shape [40, 30] on a port indicates an [40, 30]-array that is processed by the component owning the port. A shape is extended to have a special dimension, i.e., infinite dimension, which is the result of mapping some discrete time computing (or dataflow) onto a space model.

Tiler in Gaspard2 represents a *special* connector, used in a repetition context, that is associated with some topological information for array processing. A Gaspard2 tiler is equivalent to an Array-OL tiler in semantics. A tiler defines: a *fitting* matrix describing how array elements fill patterns; an *origin* of the *reference pattern*; a *paving* matrix describing how patterns cover arrays. Figure 3.11 shows an example of Gaspard2 Tiler. Two tilers are used to connect *MonochromeMode* components and its internal repetitive component *mono*.

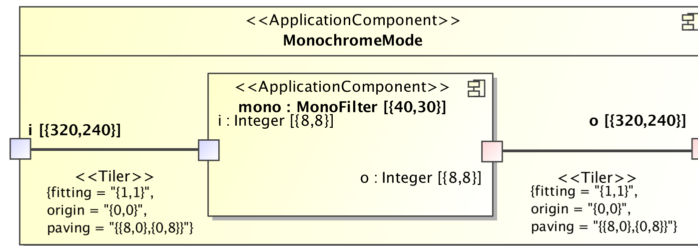


Figure 3.11: An example of Gaspard2 *Tiler*.

Reshape connects two arrays, which can be considered to be a transformation of array between these two arrays. The values stored in the array remains unchanged after the transformation. A reshape has two tilers at each end, which explain how to *displace* a tile from the source array to the target array. A reshape represents run-time links between the source and target array.

InterRepetition is used to specify an acyclic IRD among the repetitions of the same instance, compared to a tiler, which describe the dependency between the repeated instance and its owning component. Particularly, an IRD specification leads to the sequential repetition execution of an instance. A *DefaultLink* provides a default value for repetitions whose dependency for the input is absent.

- **Application** focuses on the description of data parallelism and data dependencies between application components. These components and dependencies completely describe the functional behavior in terms of *tasks*. Application components mainly manipulate multidimensional arrays. ODTs, such as *tiler*, offer the opportunity of expressing data parallelism. Task parallelism can also specified through task-dependency graphs. The tasks are either atomic computations (elementary task) on arrays or composite tasks (hierarchical task).

- **Architecture** specifies the hardware architecture at a high abstraction level. It enables

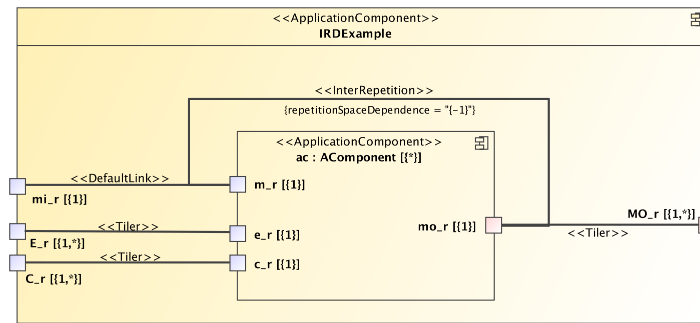


Figure 3.12: An example of Gaspard2 *InterRepetition*.

to dimension hardware resources. A mechanism, similar to the one used in application, enables to specify repetitive architecture in a compact way to cope with the increasing popularity of these kinds of regular parallel computation units in hardware.

- **Association** allows one to express how the application is projected on the hardware architecture, i.e., which hardware component executes which functionality. One particularity of this metamodel is to consider the mapping as well as the parallelism both in the application and architecture.

- **Deployment** enables to add supplementary information to elementary tasks, which concerns implementation information about the compilation and the intercommunications of their implementations. These implementations are always software or hardware IPs.

3.2.1.2 Gaspard2 profile

The Gaspard2 profile [10] is very similar to Gaspard2 metamodel. As it is replaced by the MARTE profile, it will not be detailed in this thesis. It provides a refinement of UML *Components* with regard to DIP and SoC context. The functionalities (tasks) in the software application are stereotyped by *ApplicationComponents*. As a result, a software application in Gaspard2 can be considered as a set of component dependency graphs (see the concrete example in Figure 3.13), which are often defined hierarchically. These components are connected through ports stereotyped as *In*, *Out*, etc. Each port is defined with a *multiplicity*, which indicates the shape of the data (array) that across the port.

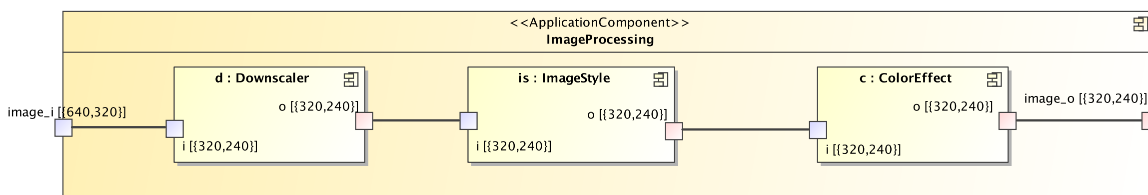


Figure 3.13: An example dependency graph of Gaspard2 *Components*.

3.2.1.3 Domain-specific metamodels in Gaspard2

In addition to Gaspard2 metamodel previously presented, the Gaspard2 environment provides several other domain-specific metamodels, which bridge between Gaspard2 notions and some specific technologies. These metamodels include: Polyhedron, Loop, OpenMP PL, RTL and Synchronous metamodel.

- Polyhedron metamodel is intended to implicit the association in Gaspard2 through polyhedral technique, which enables the representation of a spacial allocation of computing resources (processors) to task repetitions through parametrized polyhedral concepts.
- Loop metamodel has been proposed to refine the polyhedron metamodel for code generation. Loop statements are described in this metamodel, compared to polyhedra information in the polyhedron metamodel. Loop statements indicate how certain repetitions of a task are executed by processors. They are therefore parametrized by the processor indexes. SoC simulation and high-performance computing is the targeted application domain.
- OpenMP PL metamodel allows to represent the essential part of some procedural languages, e.g., Fortran and C, accompanied with OpenMP statements [106]. This metamodel is used for the code generation of high-performance computing on shared memory computers without communication between processors.
- RTL metamodel has been proposed to describe hardware accelerators, particularly the FPGA accelerator, at RTL level (Register Transfer Level). It is independent from HDL languages (e.g., VHDL and Verilog).
- Synchronous metamodel is one of the contributions of this thesis, which will be presented in the next parts.

3.2.2 Gaspard2 and MDE

Due to the points mentioned in Section 2.3 MDE has been adopted in Gaspard2. Gaspard2 benefits from several advantages provided by MDE: modeling at different levels of abstraction, which reduces the complexity in the modeling and model transformation through the intermediate-level models; modeling in a uniform language helps to reduce the complexity in the modeling and also in the integration of heterogeneous systems, technologies, etc. UML-based profiles for specific systems, such as MARTE, are available as standards; reusability helps to build complex systems at a reduced cost; increasing tools of modeling and transformation are developed with/for MDE, which offer a wide range of choice for the development.

3.2.2.1 Model transformations

For the purpose of automatic code generation, Gaspard2 adopts MDE model transformations towards different languages, such as OPENMP FORTRAN/C, SYSTEMC, VHDL and synchronous languages. Model transformations are organized as several transformation chains for these languages, as illustrated in Figure 3.9.

From Deployed models to Loop model. Two successive transformations are defined in order to transform a deployed Gaspard2 model (a Gaspard2 model, in which elementary tasks are deployed with IP models) into a Loop model (a model, which conforms to the Loop metamodel). The first one involves the transformation of a Gaspard2 model into a Polyhedron model, where repetitions are expressed by *polyhedrons*, data arrays are mapped on the certain memory, etc. The second transformation generates a model of loop expressions from the Polyhedron model, which conforms to the Loop metamodel.

From Loop model to SystemC/PA. This transformation enables the code generation for SystemC at the TLM-PA level, where data access are based on patterns (instead of bytes). The latter helps to speed up the simulation. The transformation generates the simulation of hardware and software application components. The hardware components are transformed into SystemC modules with their ports connected. A part of application that executes on processors, is generated as sets of dynamically scheduled and synchronized activities. The execution semantics of this part of application complies with the execution model defined for the Gaspard2 MPSoC applications.

From Loop model to OpenMP Fortran/C. OpenMP Fortran/C code is also generated from loop models. Two steps are involved in this transformation: (1). generation of an OpenMP PL model, where task scheduling, variable declarations and synchronisation barrier are addressed; (2). generation OpenMP Fortran/C code from the OpenMP PL model.

From Deployed model to RTL model This transformation enables the generation of RTL models from deployed Gaspard2 models. VHDL code can be generated from RTL models, which can be easily synthesized onto FPGA.

3.2.2.2 Gaspard2 and MARTE

The UML profile for MARTE has been standardized by OMG in July 2007. The Repetitive Structure Modeling package (RSM) in MARTE has been proposed according to Gaspard2. *Shape* and *LinkTopology* are two main concepts in RSM. The first one enables the shape specification of an repetitive element through *multiplicity*, which is defined as a multidimensional array. *Shape* implies a possible collection of link ends associated with the repetitive elements [98]. *LinkTopology* consists in adding some topological information to data dependencies associated with certain repetitive elements, which enables the repetitions to determine their input/output data at run-time.

3.2.3 Modeling of high-level control in Gaspard2

The Gaspard2 framework is particularly adapted for the design of DIP applications that compute large amounts of data in a regular manner. However, compared to the structural modeling, it lacks features that enable behavioral specification, which can be caused by functional requirements or adaptivity requirements of the environment. As dynamic behavior becomes increasingly common in current DIP applications, e.g., mobile multimedia systems, due to the requirements of market, QoS, etc., suitable behavioral modeling concepts are highly encouraged.

However, dynamic behavior is always considered to be a damage to the regularity in Gaspard2 applications, which benefits high-performance computing. Consequently, behavioral specification in Gaspard2 is expected to comply with the regularity to some extent. The first proposition of control that enables dynamic behavior [72] is based on *mode* or *configuration* concepts, which is inspired from mode automata [84]. However, the trace semantics of mode automata is incompatible with the parallel semantics in Gaspard2, hence the resulting automata-based control combines sequential automaton transitions and parallel repetitions, through the inter-repetition dependency defined in Gaspard2. Some main features of this proposition are given here:

- state-based control: the control is inspired from the synchronous mode automata, which adopts state-based control of dataflows. It has several advantages: the control design is simple and clean; state-based analysis and verification can be carried out in a direct way.
- control-computation separation: unlike mode automata, the control and data computation is specified in a separate way. As a result, data computation can be specified independently of control.
- multigranularity for the control: array multigranularity is used to keep the control in accordance with data computation in case that they are not synchronized.
- inter-repetition dependency is used to handle the problem of the parallelism specified in Gaspard2 and sequential execution of automata.
- a UML control profile and metamodel is proposed according to the previous features. A simplified version of this metamodel is considered for the integration of control features in VHDL code.

Extensions to this control have been proposed in [45] recently with regard to hierarchical and parallel composition and formal semantics based on mode automata and the Array-OL language. The extension of Gaspard2 with control features helps to strengthen its expressivity, which allows the specification of more complex behaviors in the applications. A formally defined syntax and semantics also enable to benefit from formal validation techniques, such as model-checking and discrete controller synthesis, to check the design correctness.

3.3 Conclusions

Based on an MDE approach, Gaspard2 has the advantages on high-level co-modeling for the design of on-chip DIP applications through standard modeling language, fast model transformation for the generation of low-level code in different languages. The Gaspard2 environment provides the Gaspard2 profile and metamodel, as well as several domain-specific metamodels, such as OpenMP Fortran/C, RTL, etc. Model transformations have also been developed for the generation of low-level executable code. The generated code can be used for the purpose of execution, simulation, synthesis, etc. These profiles, metamodels and model transformations allow the code generation from a high-level graphical application specification.

However, design correctness is one of main concerns of Gaspard2. Here, we are interested in high-level validation issues. Design problems found in Gaspard2 can be: bad usage of UML notations; violation of expected properties of Gaspard2, for instance *safe array assignment*, *acyclic data dependency*, etc.; violation of functional properties, e.g., *Safe control* and *correct implementation*; incompatibility with the environment of the system, hardware architecture, etc. For instance, the synchronizability between components in relation to environment constraints, execution time, etc.

Currently, no tools are provided to guarantee the correctness of Gaspard2 specifications with regard to the above problems. To solve this problem, using correct modeling concepts and existing but sophisticated formal validation technologies and tools is one of the main objectives of this thesis, which will be detailed in the following chapters.

As MARTE is becoming increasingly adopted, Gaspard2 is apt to be compatible with MARTE, particularly, MARTE models are adopted in Gaspard2 for the specification of systematic signal processing. However, only MARTE RSM package is completely compatible with Gaspard2, i.e., other Gaspard2 concepts in the application, hardware, association, deployment and control packages are needed to be translated with MARTE concepts. The control modeling, which is compatible with MARTE, as another contribution of this thesis, will be presented in Chapter 6.

Chapter 4

Control and validation in Gaspard2 and reactive systems

4.1 Control and validation requirements in Gaspard2 . . . . **47**
4.1.1 Gaspard2 control requirements 48
4.1.2 Validation issue of Gaspard2 49
4.1.3 Conclusions 49
4.2 Reactive systems and the synchronous approach **50**
4.2.1 Introduction 50
4.2.2 The synchronous approach 51
4.2.3 Synchronous languages 52
4.2.4 Using synchronous languages for Gaspard2 control modeling and validation 65
4.3 Conclusions **65**

4.1 Control and validation requirements in Gaspard2

As mentioned in Chapter 1, the SoC hardware design/production advancement has greatly changed our modern life, in particular, the mobile computing become progressively popular. The electronic hardware provides enough computing capacity for more complex software applications while reducing power consumption. As a result, applications on chips become increasingly more complex following the trend of the integration of more and more various functionalities into one single chip. For instance, various multimedia features, such as video/mp3 playback, TV broadcast and camera/photo, are developed and integrated into one multimedia cellular phone. These applications are not just a complement to the basic communication functionality provided by the cellular phone, on the contrary, they are becoming *killer applications*. At the same time, end users are apt to purchase the cellular phones that provide enough multimedia functionalities to obtain the mobile computing capability at a minimum price.

Furthermore, a multimedia functionality, e.g., the video playback, may have different *modes* due to the following possible requirements:

- end users: they may expect to change the video effect mode while watching a video clip, which includes black and white mode, negative mode, etc. These modes provide more choices for end users, which help to increase user satisfaction of the application;
- hardware/platform: in some cases, the hardware/platform may need to change application modes owing to the resource change. For instance, while the processor load is high, the application can be switched to a less processor load mode. Or inversely, while the processor load is low, the application is changed so as to have a better service quality;
- environment: communication quality can be considered as an environmental constraint. Thus, a low communication quality leads to a low video quality mode, e.g., the low resolution mode, so that the service is not interrupted.
- Quality of Service (QoS): so as to be friendly to end users, the passage from one mode to another should be carried out in comfortable way, i.e., a progressive change.

The mode change mechanism confers *flexibility* onto the software application design: first, dynamic behavior is enabled in the application from the a functional point of view; second, the application has the ability to be adaptive to nonfunctional constraints. Consequently, it provides a better QoS to end users.

4.1.1 Gaspard2 control requirements

Based on *data dependencies* and the *repetition* concepts in Array-OL, Gaspard2 allows a concise specification for DIP applications, e.g., the previously mentioned mobile multimedia applications. Consequently, it benefits from the design efficiency and rapid system performance evaluation. However, Gaspard2 does not have any operators that allow the specification of *control behavior*. Hence the application domain of Gaspard2 is limited. In particular, design of more complex and flexible applications does not limited by hardware resource any more. Consequently, the integration of control modeling in Gaspard2 is encouraging.

Control behavior can be modeled in different forms, at different levels, etc., in the application. In Gaspard2, control behavior is introduced through a control model, which is separated from the data computation. Thus the data computation and the control model is independently reusable. Furthermore, the control model can act as an interface between the application and its environment, platform or hardware, which enables application adaptivity. The expected control mechanism is based on the following characteristics:

High-level control. Gaspard2 is dedicated to high-level application specifications. Hence, a corresponding high-level control mechanism is expected. This high-level control should be concise but expressive and adequate enough to describe the expected application behavior.

State-based control. State machines have been widely adopted in academia and industry, which enable to specify control-oriented reactive systems. Its concurrent and hierarchical specifications make it possible to describe complex system too.

Control with safe concern. Gaspard2 takes safe design concerns into account. For instance, certain properties that avoid faults in the design can be integrated into this control.

Therefore it contributes to the construction of correct systems apriori and it reduces the overall validation cost and time.

Verifiable control. Apart from conception with safe concerns, the verifiability of the proposed control is another distinct feature, which enabless to ensure design correctness by using aposteriori formal validation tools.

4.1.2 Validation issue of Gaspard2

DIP applications can be specified at a high level in the Gaspard2 framework, but there are no tools that can guarantee the correctness of the specifications. However, design correctness is one of the main properties expected in Gaspard2, which is also one of the most important issues in the application design. Design with some safe concerns can only partly ensure the correctness with regard to certain properties. Hence application validation through some *existing* tools is necessary, which can offer a solution at a low cost. However, we are only interested in high-level validation issues in Gaspard2, which are classified into four families:

- UML related problems (good usage of Gaspard2 UML notations) can be partially verified by OCL, UML graphical tools, model transformations, etc. However, as it is a problem related to UML validation, this kind of validation is not involved in the current work;
- Gaspard2 related problems concern the violation of the Gaspard2 properties, for instance *safe array assignment*, *acyclic data dependency*, etc.;
- application functional problems denote the violation of expected functional properties. Based on the verifiable control, correctness verification of an application is involved;
- non-functional problems are related to the incompatibility with the environment of the system, hardware architecture, etc. For instance, the *synchronizability* between components in consideration of environment constraints, execution time, etc. These problems are expected to be verified at a high abstraction level to some extent in order to reduce the design-validation loop.

4.1.3 Conclusions

Based on the previous analysis on requirements of control modeling and application validation, our contributions for satisfying these requirements are mainly inspired by synchronous dataflow languages, which are detailed in the next Section 4.2. In particular, *Mode automata* [84] is well adapted in the Gaspard2 context, which is based on dataflow languages. First it enables to specify state-based control. Second, it provides the possibility to specify dataflow processing. Mode automata are also designed in consideration of safety concerns. The modes are exclusive so that, on one hand the exclusivity of modes makes a clear separation of application functionality, which reduces possible fault risks, and on the other hand the faults in one mode will not take effect in other modes. Moreover, we propose to utilize the existing and sophisticated formal validation tools associated with synchronous languages to address validation problems. The main advantage of using synchronous languages is to provide a set of solutions to current Gaspard2 requirements in the same synchronous dataflow framework. Consequently, it avoids integrating heterogeneous technolo-

gies, each of which meet different requirements of Gaspard2, so as to reduce the system complexity.

4.2 Reactive systems and the synchronous approach

4.2.1 Introduction

Embedded systems and real-time systems are two general terms that have been widely adopted in the computer and electronic societies. But sometimes they are too general to identify some specific systems, such as a system that reacts with its environment in a way the latter cannot wait. Signal processing and critical control systems are this kind of systems, which is called *reactive system* [61] in comparison with two other kinds of systems [53]:

- *transformational systems* that take some input values and transform them into output values. These systems terminate after accomplishing these transformations. Compilers are this kind of systems.
- *interactive systems* that their environments need to wait for the computing results from the systems.

4.2.1.1 Reactive systems

More precisely, a reactive system reacts continuously to the external stimuli received from its environment, then carries out the computing and triggers events that can be observed in the system, and sends back computing results with the satisfaction of the response time required by the environment.

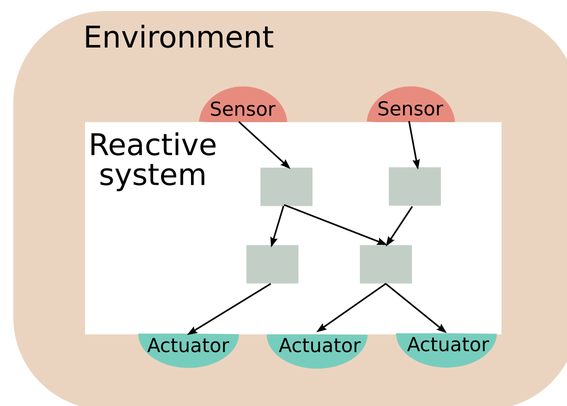


Figure 4.1: A reactive system example.

A reactive system (Figure. 4.1) can be seen as a network of concurrent computing units, which can communicate with each other. The system communicates with its environment through two kinds of devices: *sensor* and *actuator*. Sensors are used to capture messages (events and data) from the environment in the way that these messages can be processed by the reactive system. Actuators send messages (computing results) to the environment recognizable by the environment.

Reactive systems have several features [53]: they are generally *concurrent* systems, which require concurrency specification. *Determinism* is an expected property for reactive systems. *Time constraints* should be imperatively satisfied, which include input frequency and system response time with regard to the environment. Reactive systems are sometimes *safety critical* systems, where design faults can cause unacceptable loose of money, market, even human lives. This requires the system specification is *correct*.

However, classical languages are proved not appropriate to specify reactive systems. For instance, concurrent programming languages, such as Ada or classical languages with concurrent constructs, are considered as asynchronous languages as their concurrent processes execute independently from others in general. The communications among the processes are asynchronous or synchronous, but without time limits. Hence synchronous languages are proposed to specify reactive systems. Basic principles of synchronous languages are subsequently presented in Section 4.2.2, and some synchronous languages are discussed in Section 4.2.3.

4.2.2 The synchronous approach

Despite of the complexity in specifying concurrent processes and event occurrences in reactive systems, which leads to the system undependability, the synchronous approach *tries to simplify* the system design by proposing a simply but mathematical sound model, which is inspired by the synchronous process algebras introduced by Robin Milner [93]. This model considers a system that continuously reacts to external stimuli instantaneously, i.e., its execution is composed of infinite reactions and the execution time of each reaction is negligible with respect to the response delays of its external environment. From this point of view, the time behavior of a system is abstracted in a very simple but elegant way.

Signal. Signals or events are broadcast in the system and can be considered to be received and emitted simultaneously in respect of the instantaneous reaction. The modeling of event occurrences are therefore simply: the occurrences are considered to be simultaneous.

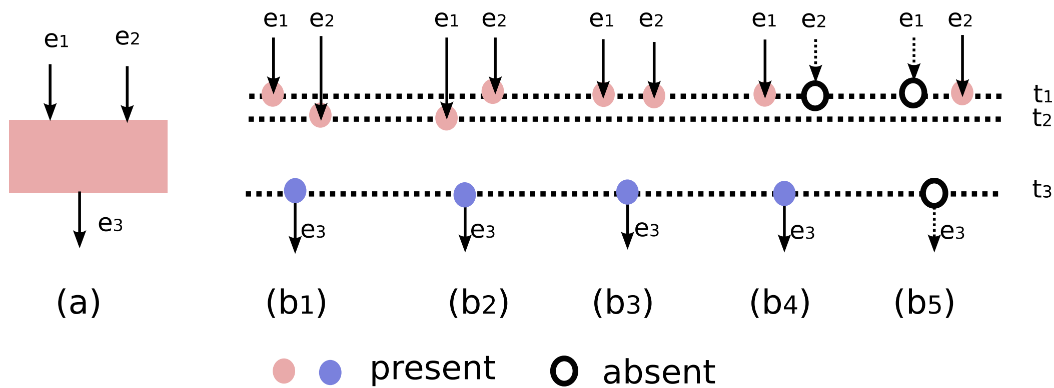


Figure 4.2: Illustration of possible event occurrences.

Consider the example in Figure. 4.2: (a) is a simple reactive system. e_1 and e_2 are two events that are captured from the environment, and will be processed by the system. e_3 is the event that the system sends back to the environment. (b_1), ..., (b_5) illustrate some possible event occurrences in each reaction. For instance, in (b_1), e_1 arrives at time t_1, and e_2 arrives

at time t_2. Thus, there is a time difference between t_1 and t_2. e_3 is produced after the reaction at time t_3. Whereas in (b_2), the occurrences of e_1 and e_2 are inverse. In (b_3), the two events arrive at the same time. In (b_4) and (b_5), events maybe be absent, which is illustrated by a hollow circle. Note that the time t_1 denotes the occurrence time of the first event between e_1 and e_2, t_2 denotes the occurrence time of last event between e_1 and e_2 and t_3 denotes the moment that e_3 appears.

From the point of view of the synchronous approach, the synchronous modeling of event occurrence in reactive systems can be illustrated in Figure 4.3. The concrete physical time of the event occurrences and of the system execution in (b_1), (b_2) and (b_3) is abstracted as an instant in the logical time, where event occurrences and computing appear simultaneously and instantaneously, as illustrated in c. c illustrates only one reaction. The continuous system execution, which consists in infinite reactions in total order, can then be mapped onto a sequence of instants in a discrete logical time.

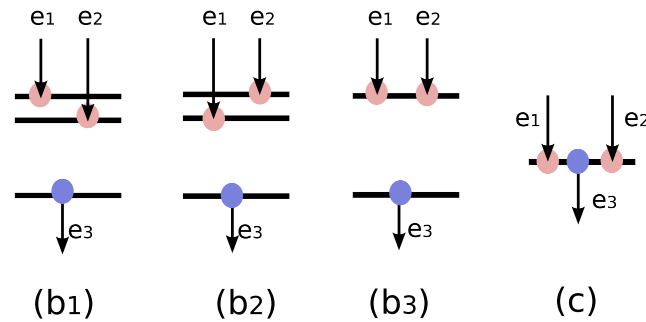


Figure 4.3: Illustration of the synchronous modeling of event occurrences.

Reaction. The execution, referred to R, of a reactive system is composed of a set of infinite non-overlapping reactions r_k, denoted as $R = \{ r_t \mid t \in \mathbb{N} \}$. As each reaction is considered instantaneous, the set of reactions can be mapped onto a discrete logical time, where each reaction corresponds to one instant. A reaction can be indexed by the instant number representing the time. Figure 4.4 illustrates an example of the reactions from the point of view of the logical time.

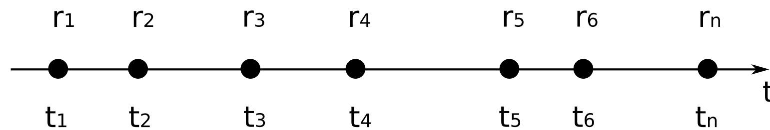


Figure 4.4: Reactions of a reactive system on a logical time.

4.2.3 Synchronous languages

Graphical languages. Based on the synchronous model, some languages for the specification of reactive systems have been proposed. The STATEMATE semantics of STATECHARTS has been proposed in [59], which is based on STATECHARTS [58]. STATECHARTS is a graphical language based on automata in consideration of hierarchical composition and concurrent communication for the specification of reactive systems. This language has been widely used

in different application domains, which leads to a large quantity of variations in syntax and semantics [130]. The semantics of STATECHARTS that is used here for illustration is the one that is adopted in STATEMATE environment. A Statecharts diagram consists of states and transitions. A state can be refined with some substates and internal transitions that define the state hierarchy. Two such refinements are available: *and* and *or* states. An *and* state contains substates that are activated concurrently , whereas an *or* state contains substates that are activated in an alternative way. When a state is left, each substate is also left. The substates of an *and* state may communicate by internal events which are broadcast all over the scope of the events. Substates of an and state may contain transitions which can be executed simultaneously.

Argos [83] is a graphical synchronous language, which is a variation of Statecharts. It is a subset of Statecharts with synchronous semantics. Argos is based on Boolean Mealy machines. It does not allow arrows that pass multi levels in the design. Compared to STATEMATE, it has different step semantics . In the STATEMATE semantics of Statecharts, it adopt delta-cycle microsteps, i.e., the effect that are caused by the transitions are taken into consideration only in the next step, whereas in Argos, the effects are taken into consideration immediately in the same reaction.

SyncCharts [6, 7] is a graphical model for the synchronous language Esterel (discussed in the following section), which inherites many features from Statecharts and Argos. It adopts synchronous semantics, which akins to Esterel, so it will be detailed in the following Esterel section. Safe State Machine (SSM) [8] is a commercial version of SyncCharts. SyncCharts is dedicated to control-dominated reactive systems, which are represented by a hierarchy of communicating and concurrent finite state machines. Determinim is a important feature of SyncCharts, which helps to build correct control systems.

Textual languages. Apart from the graphical languages, there are different styles of textual synchronous languages: textual imperative languages, such as Esterel [21]; and declarative data-flow languages, such as Lustre [54], Signal [76] and Lucid synchrone [112].

Esterel [21, 14] is a synchronous language dedicated to control-dominated reactive systems, such as control circuits, human-machine interface and communication protocols. Esterel has been developed since 1983 at CMA [1] (Applied Mathematics Center, Ecole des Mines de Paris), and INRIA in Sophia-Antipolis [2]. Esterel takes signals [3] as the main data to be processed. A signal is either a pure signal that denotes the presence or absence of this signal, or a signal carrying a value. Signals are synchronously broadcast to all the processes instantaneously.

Compared to the event-driven style Esterel, Lustre and Signal are data-flow languages. Inspired by the data-flow approach [68], Lustre and Signal concentrate on synchronous bounded-memory data-flow systems. All classic operators in these languages are extended to handle data-flows in a synchronous way. A program of Lustre and Signal is composed of a system of equations, which are used to transform data-flows. These languages are considered as high-level languages with regard to some automatic control and electronic circuits design, where their behaviors can be captured by dynamical equations.

[1] http://www.cma.ensmp.fr

[2] http://www-sop.inria.fr

[3] They are distinguished from the language Signal

Some applications. Despite of the different styles, all synchronous languages have strong mathematical foundations that help to verify the design correctness. Hence, they are adopted in the critical system design. For instance, Schneider Electric uses the Scade environment, which is based on Lustre, to develop the control software for nuclear plants. Aerospatiale uses the same environment for the development of Airbus flight control. Snecma uses Sildex, a Signal-based tool to develop airplane engines.

In the following part, several languages are detailed. Lustre is taken as an example for the introduction of some basic concepts in synchronous languages.

4.2.3.1 Lustre

Lustre [24, 54] has been defined at VÉRIMAG [4] in Grenoble, France at the end of 1980's. It is based on two approaches: the data-flow approach [68] and the synchronous model.

All the data manipulated in Lustre are data flows. A variable defined in Lustre signifies a pair of two elements: value and clock. Hence a variable is a infinite sequence of values associated with a logical clock on a discrete time, which denotes the presence of this variable. So a variable is considered as:

$$X = \{ X_t \mid t \in \mathbb{N} \}$$

where t is a natural number that represents an instance in its logical clock associated with X.

Lustre operators. Classical operators over basic data types are available in Lustre (arithmetic operators: $+$, $-$, $/$, $\times$, *div*, *mod*; binary operators: *and*, *or*, *not*; conditional operators: *if then else*). These operators are extended in order to process data flows that have the same clock. For instance:

$$X = (A + B) + 1;$$

actually means:

$$\forall\, t \in \mathbb{N} \;\; X_t = (A_t + B_t) + 1;$$

The conditional operator is different with regard to a classical one:

$$X = if\ C > 0\ then\ A\ else\ B;$$

has the same meaning as:

$$\forall\, t \in \mathbb{N} \;\; if\ C_t > 0\ then\ X_t = A_t\ else\ X_t = B_t;$$

Lustre proposes other temporal operators that can handle the value relations at different instants of the same clock or different clocks:

- *pre* operator can be used to get the variable value at the previous instant, i.e., the value is memorized to be used at the next instant. For instance, the result of $B = pre(A)$ leads to a new variable B that has the same clock as A, but its values is that of A at the previous instant (see the example in 4.5).

[4] http://www.verimag.fr

- **->** is an operator that operates two operands owning the same clock. It is used to set the first value of a variable to the first value of another variable (see the example in 4.5).
- *when* is an operator whose result can have a slower clock than the clock of the operands. It is similar to a sampling mechanism. The two operands of *when* share the same clock. The second operand should be a Boolean variable, so the sampling occurs at the instants when this variable is *true*. (see the example in 4.5)
- *current* is an operator that is inverse to *when*. Its result has a faster clock than the one of the operand. The clock of the operand is defined from other variables, from which the clock of the result of *current* is computed. The values of the result are the same as those of the operand when they have the same presences, otherwise, the result takes the value of previous presence of the operand (see the example in 4.5).

t	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	...
A	7	5	7	1	5	3	9	8	4	...
$pre(A)$	nil	7	5	7	1	5	3	9	8	...
3	3	3	3	3	3	3	3	3	3	...
3 -> A	3	5	7	1	5	3	9	8	4	...
B	F	T	T	F	T	T	F	F	T	...
A *when* B		5	7		5	3			4	...
current (A *when* B)	nil	5	7	7	5	3	3	3	4	...

Figure 4.5: Examples of some basic Lustre operators.

A Lustre syntax. Only a simplified version of Lustre syntax is given here. But it is expressive enough to understand the following work presented in this thesis. The complete Lustre syntax can be found in [54]. Some basic concepts are listed here:

```
node node_name ({input variable declaration list} )
  returns ( {output variable declaration list} )
  var ( {local variable declaration list} )
  let
          ( {system of equations} )
  tel
```

Figure 4.6: Skeleton of Lustre node.

- Module: a *module*, which can be considered as a package, is composed of a set of Lustre programs. Moreover, these programs are declared in a flat (or non hierarchical) way in its module;
- Node: a Lustre programs is a *Node*, which fulfills a certain functionality. A *node* is illustrated in the Figure 4.6. It is identified by its *node_name*. A node has interfaces, where input/output variables are declared. It contains a set of equations. These equations, which are called a system of equations, form a network of operators;

- Variable declaration: input/output variables are declared in the interfaces of a node, whereas local variables are declared locally in a *var* statement. The variable is declared as `variable_name : type_name`. An integer array variable with the shape [2, 2] can be declared as: `array_variable : integer^2^2`;
- Equation: an equation is declared as `identifier = expression;`, which is considered as an assignment;
- Expression: an expression is either a signal, or an operation on signals.

Compilation of Lustre programs The Lustre compiler allows the *static checking* at compiling time, which helps to enforce the reliability of the Lustre programs [54]. The static checking includes:

- type checking: the classical check on type consistency;
- definition checking: one and only one variable definition is allowed in Lustre, which is also called *single assignment*;
- absence of infinite recursive node invocation: it helps guarantee the reaction time limit and to obtain the executable code similar to automata;
- clock consistency: operands appeared in the equations should have consistent clocks. Classical data-flow languages allow operations on different flows, which may need unbounded memory to support these operations. Under the constraints of bounded memory, Lustre only provides the operations on clock consistent flows;
- absence of cyclic definition: cyclic definition appeared at the same clock instant, e.g., `X = Y; Y = X;` may result in: `X = X`, which is causal cycle in Lustre. One of the solutions is to use the `pre` operator to break this cycle: `X = pre(Y); Y = X;` .

Notwithstanding the concurrent specification, a Lustre program is compiled to the C code within an infinite sequential loop (Figure. 4.7).

```
<Initialize memories>
Loop
    <Read inputs>
    <Update memories>
    <Compute outputs>
Endloop
```

Figure 4.7: The generated loop from the Lustre program.

Program verification. The Lustre compiler provides the static syntactic verification, but obviously, it is not enough for the specification of critical systems. Formal verification is another technique that can check certain required properties specified for the programs. Lustre proposes *synchronous observers* [55] to express the safety properties. The observers are specified in normal Lustre language, which consider the inputs and outputs of the programs.

If the safety properties are satisfied for any inputs and outputs, then the observer always responses *OK*, otherwise, the program fails to pass this verification. Assertions can also be made to describe some environment hypotheses. Lustre is associated with some verification tools, such as LESAR [55].

Lustre Array. Array, as a basic data structure in Lustre, was first introduced in [56] for the purpose of systolic array design and simulation. [114] proposed a tool, POLLUX to generate synchronous circuits. In [94], the author proposes an efficient compilation scheme as well as optimizations for array iterators in Lustre. Recent work on Lustre array [57] involves the array content analysis through abstract interpretation.

A code example. A *node* (Figure 4.8) is considered as a basic functionality unit in Lustre. Each node produces the same results given the same inputs due to its determinism. Nodes have modular declarations that enable their reuse. Each node has an *interface* (input at line (l1) and output at (l2)), local variable declaration (l3), and *equation*s (line (l5) and (l6)). Variables are called *signal*s in Lustre. Equations are signal assignments. In these equations, there is a node invocation (l5) that is declared outside this node. In Lustre, modularity and hierarchy are inbuilt. The composition of these equations, denoted by ";", stands for their parallel execution with regard to the data dependencies between them. The node has the same signification independently of the equation order.

```
node node_name (A1:int^4)                  (l1)
 returns(A3:int^4);                        (l2)
 var A2:int^4;                             (l3)
 let                                       (l4)
    A2 = a_function(A1);                   (l5)
    A3 = A1+A2;                            (l6)
 tel                                       (l7)
```

Figure 4.8: An example of Lustre code. The node takes A1 as input and the output is A3, which is the sum of A1 and A2. A2 is a local signal, which is the transformation result of A1 through a function. A1, A2 and A3 are all array with size (4).

Some applications. Since about fifteen years, Lustre has been successfully transferred to industry, e.g., the SCADE environment[5], which takes Lustre as the core language. SCADE is used by Airbus, Schneider Electric, Eurocopter, etc.

4.2.3.2 Other synchronous languages

Signal

Similar to Lustre, Signal is also a dataflow language and specifies systems in a block diagram style. Certain *signals* (variables) are defined and related in a block. Each signal

[5] http://www.esterel-technologies.com

(e.g. `x`) represents an infinite typed sequence (e.g. $(x_\tau)_{\tau \in \mathbb{N}}$), which is mapped onto the *discrete time*. $\perp$, which represents the absence of the signal at certain instant on this *time*, expands the domain of the signal. The signal has an associated clock, which represents the set of instants where the signal is present. A *process* (or a *node*) is considered as a block or a program, which is composed of a system of equations over signals. Signal integrates the *multiclock* mechanism, which enables a process to be deactivated while other processes are activated. Hence, this process is considered to have its own and local clock.

Primitive constructs. Five primitive constructs of Signal are presented in the following table, where the first column indicates their literal names, the second column shows Signal equations and the last column provides their meanings in terms of values (indexed by $t \in \mathbb{N}$) in the sequence:

Relation	`Z := X op Y`	`X`, `Y` and `Z` have the same clock, and $Z_t = X_t\ op\ Y_t$, the operator *op* represents generic operators, such as addition and subtraction, which is applied point-wise on the sequences of `X`, `Y` and `Z`.

Delay	`Y := X $ 1 init C`	$\forall t\ X_t \neq \perp \Leftrightarrow Y_t \neq \perp\ \ and\ \forall t > 0$: $Y_t = X_k,\ where\ k = max\{t' \mid t' < t\ and\ X_{t'} \neq \perp\}, Y_0 = C$.
Undersampling	`Y := X when B`	This is a multiclock statement, which implies: $Y_t = X_t\ when\ B_t = true$, $otherwise\ Y_t = \perp$, where B is a Boolean signal. Hence Y oversamples X according to B.
Merge	`Z := X default Y`	This is also a multiclock statement, which signifies: $Z_t = X_t\ when\ X_t \neq \perp$ $,\ otherwise\ Z_t = Y_t$.
Composition	`P | Q`	It is the composition of P and Q.
Hiding	`P where x`	`x` is local to the process `P`.

From these constructs, other constructs can be derived, e.g., the `cell` operator. Figure 4.9 illustrates: `C = A cell B`, where the values of `C` are taken from `A`, but its clock is the *union* of the clock of `A` and of `when B`.

Compared to Lustre, Signal allows to specify the oversampling mechanism. For instance, for a given clock $c1$, a faster clock $c2$ can be built on $c1$ (see the following example in Figure 4.10).

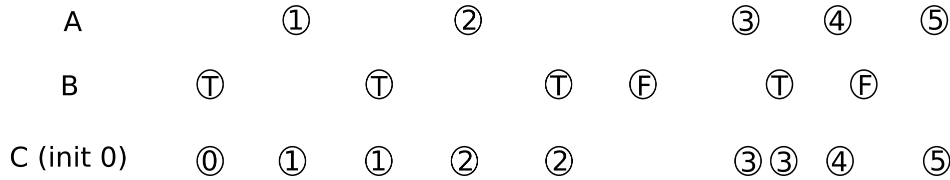
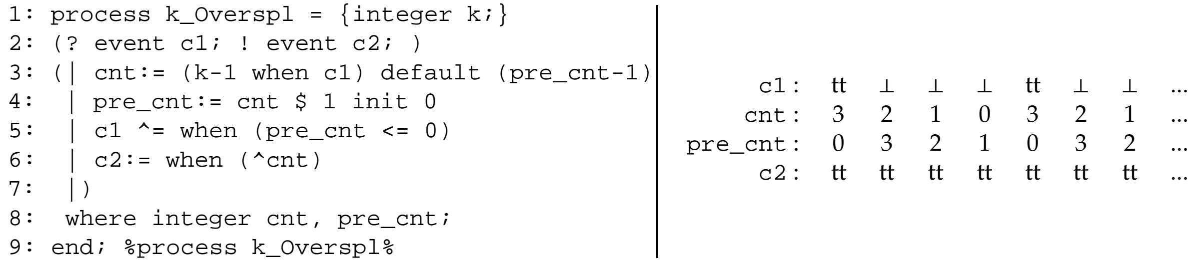


Figure 4.9: Illustration of C = A cell B.



```
1: process k_Overspl = {integer k;}
2: (? event c1; ! event c2; )
3: (| cnt:= (k-1 when c1) default (pre_cnt-1)
4:  | pre_cnt:= cnt $ 1 init 0
5:  | c1 ^= when (pre_cnt <= 0)
6:  | c2:= when (^cnt)
7:  |)
8:  where integer cnt, pre_cnt;
9: end; %process k_Overspl%
```

c1 :	tt	⊥	⊥	⊥	tt	⊥	⊥	...
cnt :	3	2	1	0	3	2	1	...
pre_cnt :	0	3	2	1	0	3	2	...
c2 :	tt	tt	tt	tt	tt	tt	tt	...

Figure 4.10: A clock oversampling process in Signal and its corresponding trace.

An extension of Signal clock system. Signal also addresses hardware/software codesign issue, e.g., the cospecification and cosimulation through the combination of Signal and Alpha languages [120]. Signal takes the advantages of behavioral specification, whereas Alpha [131] is dedicated to the specification on regular numeral computations. The former enables control-based analysis and validation technologies and tools, but falls short of manipulation and optimization of multidimensional data processing. The latter is designed for the algorithms of multidimensional data processing with the help of affine recurrence equations, from which optimal hardware and software implementations can be derived for specific architectures.

The refinement of signal clocks in Signal is carried out as a result of the refinement of Alpha program while it is implemented as specific architectures. This clock refinement is based on the *affine transformation* [119], which is built on the time indexes of signals. For instance, two Signal clocks: c and c_1 are considered synchronous, and their clock is denoted as $T = \{ t \mid t \in \mathbb{N}\}$. The refinements of the two clocks are represented by the affine transformations, which are defined as: $T_1 = \{ nt + \phi_1 \mid t \in \mathbb{N}\}$ and $T_2 = \{ dt + \phi_2 \mid t \in \mathbb{N}\}$ respectively, where $n,\ d,\ \phi_1,\ \phi_2 \in \mathbb{N}$. The clock c_1 is considered to be obtained through an $(n,\ \phi\ , d)$-affine transformation applied on the clock c, where $\phi = \phi_1 - \phi_2$ and $\phi \in \mathbb{Z}$. Then c and c_1 are considered to have the $(n,\ \phi\ , d)$-relation, which can then be used to analyze the synchronizability problem between several clocks undergoing the affine transitions. For instance, the clocks c and c_1 have the $(n_1,\ \phi_1\ , d_1)$-relation, and clocks c and c_2 have the $(n_2,\ \phi_2\ , d_2)$-relation. According to these two relations, the synchronization between c_1 and c_2 can be deduced.

Signal compiling. In the Signal compiling process, the compiler builds a dependency graph from the Signal code. This graph is then used for the static correctness analysis, e.g., causal cycles and temporal inconsistency in terms of Signal clock.

Model checking. Model checking of Signal programs can be carried out by the Sigali toolbox [17], which is associated with the Signal environment. Sigali adopts *polynomial dynamical*

systems (PDS) over $\mathbb{Z}/_{3\mathbb{Z}}$ = {-1, 0, 1}) as the formal model. Signal programs can be transformed into polynomial equations, which are manipulated by Sigali. The three states of a Boolean signal can be encoded as: $true \longmapsto 1$, $false \longmapsto -1$ and $\perp \longmapsto 0$. In case of non-Boolean signals, only their clocks are considered, i.e., their values are ignored.

PDS can seen as an implicit representation of an automaton. Each set of states can be represented by its associated polynomials. By manipulating polynomial equations, Sigali avoids the enumeration of the state space. In Sigali, a polynomial is represented by a Ternary Decision Diagram (TDD), which is an extension of Binary Decision Diagram (BDD).

Model checking can be carried out on PDS [86] with regard to the following properties:

Liveness: the system can always evolve. This property can be checked through the Sigali command: `alive(system)`. `system` is the system to be checked;

Invariance: a set of states is invariant when all the recursive reachable states are also in this set. This property can be checked through the Sigali command: `invariant(system, proposition)`. Each state in an automaton is associated with `propositions`. In previous command, `proposition` can be considered as a set of states whose `proposition` is true;

Reachability: iff the system has always a trajectory towards a state starting from the initial states, this state is considered reachable. This property can be checked through the command: `reachable(system, proposition)`.

Discrete controller synthesis. Sigali is also a tool that enables discrete controller synthesis [85]. A controller, which is specified with certain control objectives, is synthesized in a system through running the controller with the system in parallel. The resulting system is the desired system satisfying the control objectives. First, the PDS of a system can be described as:

$$S = \begin{cases} X' & = P(X, Y, U) \\ 0 & = Q(X, Y, U) \\ 0 & = Q_0(X) \end{cases}$$

where X, Y and U are sets of variables. Elements of X are *state variables*, which indicate the current states of the system and X' indicate their next states. Y and U are composed of *controllable event* and *uncontrollable event* variables respectively. The controllability of events is specified with regard to the controller to be synthesized in the system. Events from the environment of the system are always uncontrollable, whereas the internal events in the system are always controllable. The first equation implies state transitions. The second equation is considered as a constraint equation. The last one is the initialization equation.

The controller can be defined with two equations $C(X, Y, U) = 0$ and $C_0(X) = 0$, where the first equation indicates the synchronous control, and the second one concerns the initial states of the controller in consideration of the control objectives. Hence the resulting system integrated with the controller is:

$$S = \begin{cases} X' & = P(X, Y, U) \\ 0 & = Q(X, Y, U) \\ 0 & = C(X, Y, U) \\ 0 & = Q_0(X) \\ 0 & = C_0(X) \end{cases}$$

According to certain properties specified, the synthesized controller can ensure:

Invariance: the invariance of a set of states can be obtained by using: `S_Invariance(System, proposition);`

Reachability: the reachability of a set of states is ensured by using: `S_Reachability(system, proposition);`

Signal-Meta. Signal-Meta [22] is a metamodel dedicated to Signal. It is defined in Generic modeling environment (GME) [1], a model-integrated configurable toolkit that supports the creation of domain-specific models for large-scale systems. A tool is also developed to transform the graphical Signal-Meta specifications into the Signal code.

Signal GTi. Signal GTi [115] is an extension to Signal with constructs for hierarchical task preemption. Tasks are defined as the association of a data-flow process with a time interval on which it is executed. In Signal, a process is defined to react to the environment in an infinite way, however, no explicit constructs are defined to handle the termination, interruption or sequencing of processes, which implies limitations of process behaviors to a slice of certain reaction instants. Both dataflow and tasking paradigms are available within the same language-level framework.

Lucid synchrone

Lucid synchrone [111] is a language for the specification of reactive systems. It is based on Lustre language from the point of view of time model, and it also benefits from OBJECTIVE CAML, which is its host language. It is a strong typed and higher order functional language. The type system presented in Lucid synchrone helps to achieve type and clock inference, causality analysis.

Lucid synchrone distinguishes combinatorial functions and sequential functions. The former represents the functions that do not have states. The latter implies the functions whose results depend on its internal states. Lucid synchrone enables a multiclock system, e.g., it has the downsampling operator `when` and oversampling operator `merge`, similar to Lustre. Moreover, it oversampling mechanism makes it possible to define outputs that are faster than inputs. The *match/with* statement makes it possible to activate different equations according to the values of some variables. The following example shows a node containing this statement:

```
type modes = Up | Down

let node m i = o where
  match m with
```

```
                  Up   -> do o = i + 1 done
                | Down -> do o = i - 1 done
        end
```

Another interesting feature is the explicit state machine specification in Lucid synchrone, which is detailed in Section 4.2.3.3.

Esterel

Esterel [21, 14] adopts an event-driven style, compared to previously mentioned dataflow languages. Esterel is an imperative language, which is well situated for control-dominated reactive systems. Esterel programs is composed of a set of nested threads that run concurrently or sequentially. Communications between threads are accomplished by pure *signals* or valued *signals*, which are distinguished from the variable of other programming languages. Signals in Esterel are considered as some kind of *events*, even they carry values. Whereas a variable is similar to a unit of memory, whose value can be changed during execution.

Statements. Main statements of Esterel are based on the operations on signals:

- `emit X`: emit the signal `X`;
- `present X then s1 else s2 end`: monitor the signal `X`, if it appears, then execute the statement `s1`, otherwise, execute `s2`;
- `await X`: pause the execution until the next appearance of `X`.

Despite of the concurrent or sequential execution of the statements, Esterel has a *single* and *global* clock, where the signals are manipulated at each instant. Multiclock Esterel has also been studied recently in order to release initial single clock requirement [15]. The execution of threads at each instant is considered instantaneous, which starts from where it paused at the previous instant and pauses again or terminates when the execution (or reaction) in this instant accomplishes. The following statements are involved in the organization of statements:

- `pause`: pause the program until the next reaction;
- `s1 ; s2`: pass the control to `s2` immediately when `s1` finish;
- `s1 || s2`: start both statements `s1` and `s2` at the same time, and the control is passed to next statements only when both these two statements finishes;
- `loop s1 end`: repeat the statements `s1`.

Esterel enables the *preemption* specification too. `abort s1 when X` is the statement that has a strong preemption flavor. The statements `s1` will be killed immediately when `X` appears. By comparison, `weak abort s1 when X` allows `s1` to finish the execution of the current reaction. Previously mentioned statements are considered as basic statements of Esterel, more complex statements can be derived from these basic ones.

Constructive causality analysis Causality analysis is a very important static analysis in Esterel, which helps to find *deadlocks* in the specification. Compared to causality analysis in Lustre, Esterel is based on constructive causality [13], which allows a finer grain verification of certain causal statements by using constructive logic rules. As a result, it provides Esterel a better solution compared to the Lustre one, which rejects all causality cycles.

4.2.3.3 Mixed-style languages

Mode automata and Matou

In several tools or development environment dedicated to dataflow applications, the *multi-paradigm* has been proposed to integrate languages in different styles, i.e., dataflow and some imperative feature:

- SIMULINK and STATEFLOW[6] : the first one is used for the specification of block diagrams where some operators of STATEFLOW are used for the computation of some dataflow. The results of the computation serve to control the system.
- SCADE and SyncCharts[7]: in the SCADE environement, state machines of SyncCharts are embedded to activate dataflow processing specified in Lustre.
- PTOLEMY II [23]: state machines can be also mixed with dataflow equations.

Mode automata [82] derive from the concept of *combination of formalisms*, which is similar to previous mentioned multi-paradigm. Mode automata are proposed to extend the dataflow language Lustre with certain imperative style, without many modifications of language style and structure.

Mode automata are mainly composed of modes and transitions. In an automaton, each mode has the same interface. In each mode, equations can be specified. But only a subset of Lustre is used for modes. For instance, operators, such as initialization, sampling and oversampling, are not allowed to be used in modes. Transitions can be associated with conditions, which act as triggers. However, the `pre` operator cannot be used in conditions.

Mode automata can be composed together in a parallel way, which is called *parallel composition*. The composition result is the Cartesian product of the sets of modes of the automata to be composed. The composition also makes it possible that the automata communicate with each other in a way that one output of one automaton can be taken as an input of another automaton. In order to avoid the causality problem, *weak transition* is imposed in mode automata, which implies that new values of a variable can only be taken into account in the next instant.

Hierarchical composition is based on the refinement of certain modes in the automata. At each level, the variables in the states are considered *global*, as all the state at this level can see these variables. However, a variable cannot be multiply defined at different hierarchical levels.

[6]http://www.mathworks.com/products/simulink
[7]http://www.esterel-technologies.com

Matou: an implementation of mode automata based on Lustre. In Matou, mode automata can be specified with the *Targos* format, which is an explicit specification of automata. Then mode automata can be compiled into DC code, which is an intermediate format in the process of compiling Lustre to C. Consequently, analysis and verification tools associated with DC can be used. Mode automata can also be compiled into polynomial dynamical systems over $\mathbb{Z}/3\mathbb{Z}$, so that Sigali can be used for model checking and discrete controller synthesis.

State machines in Lucid synchrone

In Lucid synchrone, state machines can be directly specified [112]. Lucid synchrone state machines (LSM) can be composed in a parallel and hierarchical way. In each state of LSMs, equations can be specified in the same way as mode automata.

However, LSMs make some distinctions or improvements from mode automata:

- Strong transitions are allowed in LSMs, compared to mode automata, which only enable weak transitions. Strong transitions enable strong preemption so that the transition conditions (guards) are evaluated in the same reaction and the equations in the new states are evaluated immediately too;
- LSMs allow local variable definitions used in the states. However, variables used in the states of mode automata are global variables;
- Computation defined in the states of LSMs can be resumed when these states are re-entered. But it is not the same case as mode automata, where the computation is always reset;
- LSM states can be parametrized, similar to a function. It enables to initialize a state with some input values, which can be used in the equations in the state. In this manner, information can be passed between states, and it helps to reduce the state number.

Polychronous mode automata

Polychronous mode automata [126] aim at extending Signal with multiclocked mode automata. Polychronous mode automata also enable strong transitions, and at most two transitions (strong transition and/or weak transition) can be fired in one reaction, which are similar to Lucid synchrone state machines. However, polychronous mode automata allow to define local clocks, which is considered as an advantage for the specification of multiclock system. A meta model has been built for the polychronous mode automata, which enables automatic Signal code generation.

A mixed control and data for the distribution

[108] presents another method that combines control and data processing. This method is proposed to unify different specifications for the control and data processing for distributed embedded systems. The resulting specification is a conditioned data flow graph. The SynDEx software [74] is used to automatically generate a distributed and consistent implementation.

4.2.4 Using synchronous languages for Gaspard2 control modeling and validation

Synchronous languages are proposed to be used for the Gaspard2 control and validation purpose. On one hand, system behavior specified with synchronous languages is clear and deterministic. Their control mechanisms contribute to the construction of a safe control mechanism for Gaspard2. On the other hand, synchronous languages provide associated tools for the validation purpose, e.g., their compilers can check the causality, determinism and reactivity of the synchronous programs; their associated model checkers can verify program correctness according to specified properties. As a result, these languages have been successfully used in several critical domains (e.g. avionics, automotive and nuclear power plants.).

We therefore propose the transformation of Gaspard2 specifications into executable code in synchronous languages in order to benefit from the existing validation tools associated with these languages. This approach does not need intensive development of Gaspard2-specific validation tools. In addition, it enables to borrow ideas of safe control with formal semantics from synchronous languages.

4.3 Conclusions

This chapter first presented the control and validation requirements of Gaspard2. The expected control has the following characteristics: high-level and state-based. This helps to build a simple but efficient control mechanism. These characteristics meet the requirements for the high-level dynamic behavioral specification of DIP applications. Validation is another concern of Gaspard2. First, verification of Gaspard2 applications without control is expected. However, no tools are provided to support this verification. Second, the introduced control should be safe and verifiable so that the control can also be verified with ease.

The proposed approach. Synchronous languages define unambiguous system behavior and their associated tools enable formal analyses and verifications. The MDE framework provides the solutions of modeling and model transformations that bridge the gap between high-level specifications and low-level execution models. Hence it is natural to take the advantages of these two technologies for the modeling, model transformation and validation of Gaspard2 applications (Figure 4.11).

Figure 4.12 illustrates the global view of the proposed approach. The involved transformations are located in box $S1$, which start from the Gaspard2 model. MDE transformations (*Transf1* in the figure) are then carried out on this Gaspard2 model into the synchronous model. The obtained model serves to generate code in synchronous languages (*Code generation* in the figure).

Another work, which appears in box $S2$, involves the integration of control in term of automata in Gaspard2 and the previous synchronous modeling. The bridge between the two modeling is *Transf2* in the figure, which is an extension of *Transf1* with control aspects.

With the help of the code and tools provided by synchronous languages, the application analysis and validation are possible (box $S3$ in Figure 4.12). For example, The generated code in Lustre, Signal, Lucid synchrone and mode automata can be used for various purposes: synchronizability analysis, causality verification, simulation, model checking, discrete con-

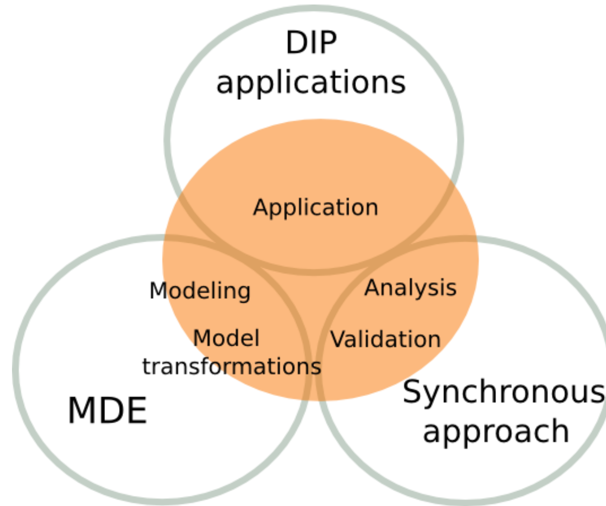


Figure 4.11: A combination of three domains.

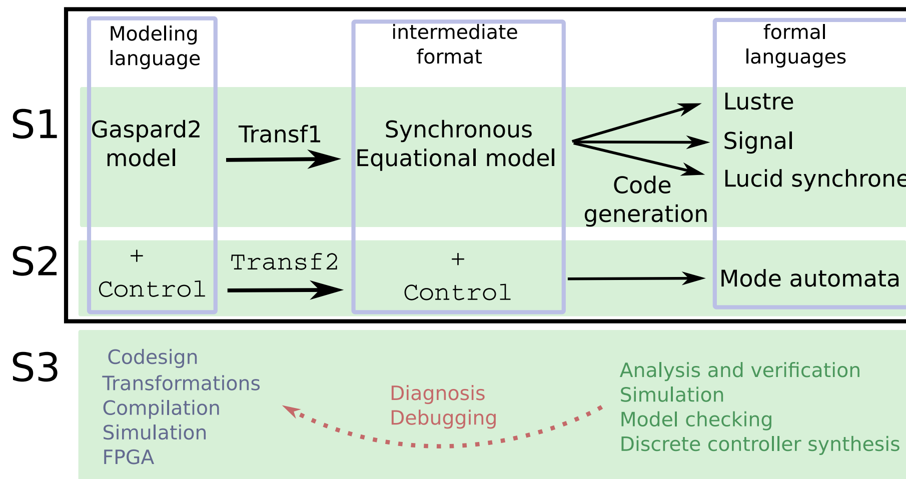


Figure 4.12: A global view of the proposed approach.

troller synthesis, etc. The results of the analysis and valication contribute to uncovering the corresponding problems that are present in the original Gaspard2 designs. The corrected designs can be transformed into other low-level implementations for the purpose of simulation, performance evaluation, compilation, etc.

The synchronous modeling without control concepts is presented first as a basis in this thesis in Chapter 5. This modeling involves the construction of a synchronous equational model, which is a common model to the three synchronous languages. A translation of a Gaspard2 model into the synchronous model is also presented with only their abstract syntax.

Then the model transformations between the two models, within the MDE framework, are also presented in Chapter 7. These transformations are based on two metamodels, i.e., Gaspard2 metamodel and synchronous metamodel, and a set of transformation rules. These transformations have been implemented with the Eclipse environment.

Control concepts, which are integrated in Gaspard2 model and the synchronous model are presented separately as well in Chapter 6. The control mechanism is inspired by mode automata and is based on control and data computation separation. Parallel and hierarchical composition, which contributes to build complex systems, is also described.

Extensions of the Gaspard2 and synchronous metamodels, in consideration of the introduced control concepts, are then presented in Chapter 8. Transformation between Gaspard2 models and synchronous models is also discussed. Pure equational models or mode automata model can be obtained as a result of the transformation.

A case study is presented in the last Chapter 9. It first illustrates the modeling, including DIP and control modeling, and formal analysis and validation carried out on the generated code with the help of tools associated with synchronous languages.

Part II

Synchronous modeling and reactive control of DIP applications

Chapter 5

Synchronous modeling of DIP applications

5.1 General modeling approach **72**
5.1.1 Step 1: space refactoring 73
5.1.2 Step 2: space-time analysis and mapping 73
5.1.3 Step 3: synchronous modeling 76
5.1.4 Step 4: code generation 76
5.2 General synchronous modeling propositions **76**
5.2.1 Array data structure 76
5.2.2 Parallelism modeling 77
5.2.3 Hierarchy and composition 78
5.3 Gaspard2 and synchronous representations **79**
5.3.1 A Gaspard2 abstract syntax 79
5.3.2 Synchronous equations abstract syntax 80
5.4 The translation between the two representations **81**
5.4.1 Structural translation 81
5.4.2 Translation of behavioral aspects 81
5.4.3 Correctness of the translation 83
5.4.4 Translation with serialization and partition semantics 85
5.5 Conclusions **88**

This chapter presents the synchronous modeling of DIP applications, in the form of Gaspard2 data-dependency model[1], so that a Gaspard2 model can be translated into an executable model of synchronous dataflow languages, in consideration of the constraints of these languages. This modeling bridges the gap between DIP and synchronous languages, and also contributes to the analysis of DIP so that safe-design concepts (e.g., safe operators and their formal semantics) can be integrated into DIP, and formal validation of design can be carried out with its clear semantics.

[1]Although only Gaspard2 model is mentioned in this chapter, the MARTE RSM model is similar to the Gaspard2 model, therefore, the synchronous modeling of the MARTE RSM model is the same as the Gaspard2 one.

5.1 General modeling approach

As a high-level specification language, Gaspard2 is not intended to be involved in implementation details that may mess up the specification. Furthermore, it provides the possibility to exploit the parallelism according to the hardware architecture on which it maps. These features make it not obvious to translate Gaspard2 specifications into executable implementations in certain programming languages, because this is not a direct structural translation. For instance, some features in Gaspard2, such as the *infinite* dimension and *tiler*, could not be translated directly due to implementation constraints of programming languages, platforms or hardware. These constraints involve memory space, computing power, execution time, etc. In spite of the distinctions between the specification and implementations, a translation is expected to preserve the properties of Gaspard2, such as parallelism and single assignment.

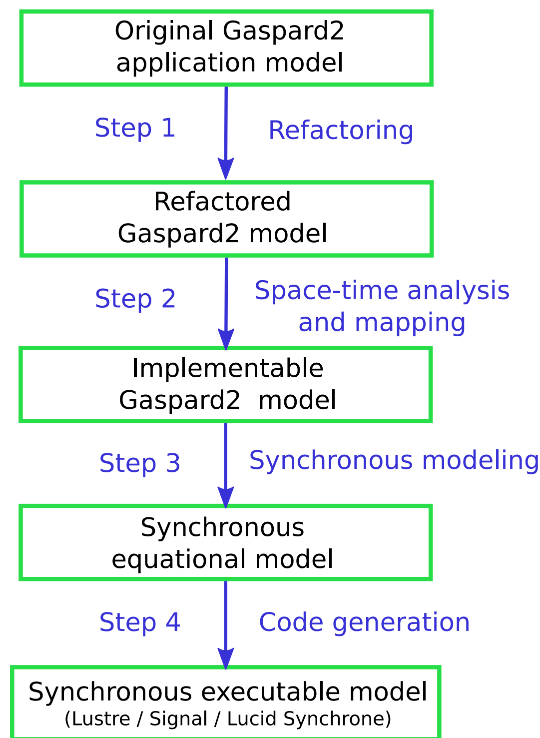


Figure 5.1: An executable synchronous model is obtained after four steps of analysis and modeling. This figure also illustrates the intermediate models in this process.

In order to obtain an eligible modeling, the previously mentioned problems are separately addressed according to the *separation of concerns* approach. As a result, the translation of a Gaspard2 model into synchronous languages can be divided into four steps: *refactoring*, *space-time analysis and mapping*, *synchronous modeling* and *code generation*. Each of these steps addresses a specific and independent problem, thus the overall complexity of translation can be managed at an acceptable level. This translation process is illustrated in Figure 5.1.

5.1.1 Step 1: space refactoring

The first step of modeling involves the refactoring of an original Gaspard2 model. A refactoring refers to a modification of original model without change of its functionality, e.g., change of granularity of tiles, increment or decrement of the array dimension number, change of component hierarchy, etc. It is carried out according to some requirements, which include optimization of applications, improvement of computing performance, satisfaction of certain constraints, etc. However, this step is not always necessary, particularly if the model is considered to satisfy the requirements. In Gaspard2, two kinds of analysis are supported:

- the first one involves some constraints that represent some non-functional aspects considering the platform, hardware and environment. For instance, execution rate, processor number, etc., are considered as constraints for refactoring. Hence, a high-level system analysis on performance can be carried out based on these factors. Currently, modeling of these constraints and analysis of performance is an ongoing work;

- the second one concerns the parallelism optimization and compilation, which is similar to *loop transformation* in some languages for high-performance computing. This work has been studied for Array-OL and Gaspard2 in [121, 32, 49]. The refactoring is composed of *fusion*, *change of paving*, *tiling*, *collapse* according to [49]. Implementation tools are also developed to support the refactoring. How these tools can be also used for the first kind of refactoring is still under study.

5.1.2 Step 2: space-time analysis and mapping

A refactored Gaspard2 model is considered as a pure-spatial parallel model, which can be equally an infinite spatial model. The spatial parallel model implies all the elements in this space may be processed in parallel, e.g., the repetition space in Gaspard2 defines one of this kind of models. The parallel model is based on a hypothesis on computing resources that are sufficient to fulfill the parallelism requirement. This characteristic leads to two problems while translating a Gaspard2 model into implementation models: the first one is that the refactored Gaspard2 model is a timeless model whereas implementation models, such as synchronous equational model (synchronous model for short) or FPGA model, are models with implicit time concepts (timed model for short). The second one is that a Gaspard2 model may be an infinite space model, whereas implementation models always suggest finite space models due to limited computing capacity. Hence, a space-time mapping is proposed to address these problems. This mapping (it will be detailed in Section 5.1.2.3) is intended to map some dimensions of space onto time, as a result, a finite-space timed model is obtained from an infinite-space timeless model. Herein, this mapping contributes to bridge the gap between two different models, i.e., specification model and implementation model. [52] presents a similar work on space-time mapping on parallel loops.

However, a space-time mapping is not necessary when a Gaspard2 model is a finite space model, i.e., this model can be translated into some executable models, such as OpenMP Fortran and SystemC directly, because the finite space of this model can be covered by the parallelism provided by these executable models.

5.1.2.1 Time dimensions in Gaspard2

A Gaspard2 time dimension is a special spatial dimension, which can be processed by implementations with time concept. For instance, the synchronous language Lustre has a basic clock, which is considered to be implicit. This clock is composed of an infinite sequence of instants, which are in total order and can be indexed by number. A Gaspard2 time dimension adopts the same point of view, i.e., all the elements in this dimension are considered to be in total order and can be processed in a sequential way. Compared to time dimension, spatial dimensions imply parallel processing.

In addition, some execution models, such as the synchronous model and the FPGA model, support *multiclock* system, which makes it possible to adopt a multidimentional-time mapping from a space model. From the same point of view, we can define some special multi-space-dimension, which corresponds to the multiclock mechanism in implementation models. These spatial dimensions are called *multi-time-dimension*. Other works on multidimentional time issue can be found in [41, 42].

5.1.2.2 Space-time analysis

The *space-time analysis* contributes to determine which space dimensions are mapped onto time dimensions. Initially, a Gaspard2 model has an infinite dimension, which is considered as a special dimension that can be mapped onto a time dimension [19]. But it is not the unique way, as multidimentional-time dimensions are introduced, the space-time relation may be complex.

Another contribution of space-time analysis is to find appropriate space-time mappings in a *coherent* manner when several mappings are considered at the same time. For instance, it is not restricted to choose different mappings for different spatial parallel models, for instance, the repetition spaces defined for different computing in a Gaspard2 application.

5.1.2.3 Space-time mapping

Once the previously mentioned analysis is obtained, a space-time mapping is determined correspondingly. In this case, *time tilers*[2] can be used to map an infinite-space timeless Gaspard2 model onto a finite-space timed model. In Figure 5.2, some examples of space-time mappings are illustrated, where the repetition space (A_1 with the shape[5,4,*]) has an infinite dimension (dimension Z). After the mappings, the infinite array A_1 becomes several flows of arrays. Each array in these flows corresponds to an array that can be processed at one instant of the discrete time. In the mapping m_1, the [5,4,*]-array is mapped onto a flow of arrays, where the infinite dimension of [5,4,*]-array is mapped onto a discrete time dimension t. However this is not a unique way in the mapping. Taking the granularity level into account is another important and interesting way. For instance, a [5,4,*]-array can be also mapped onto a flow f_2 of [5]-array where f_2 is four-time faster than f_1 (the mapping m_2). In an inverse way, the [5,4,*]-array A_1 can be mapped onto a flow f_3 of [5,4,2]-array where f_3 is two-time slower than f_1 (the mapping m_3).

When all the infinite dimensions (from different arrays) declared in a Gaspard2 model are mapped on a common discrete time axe through time tilers, a new model is obtained, in which a new hierarchy is added so that the infinite dimension (the time dimension) only

[2] A time tiler refers to a tiler that works on time dimensions in this thesis.

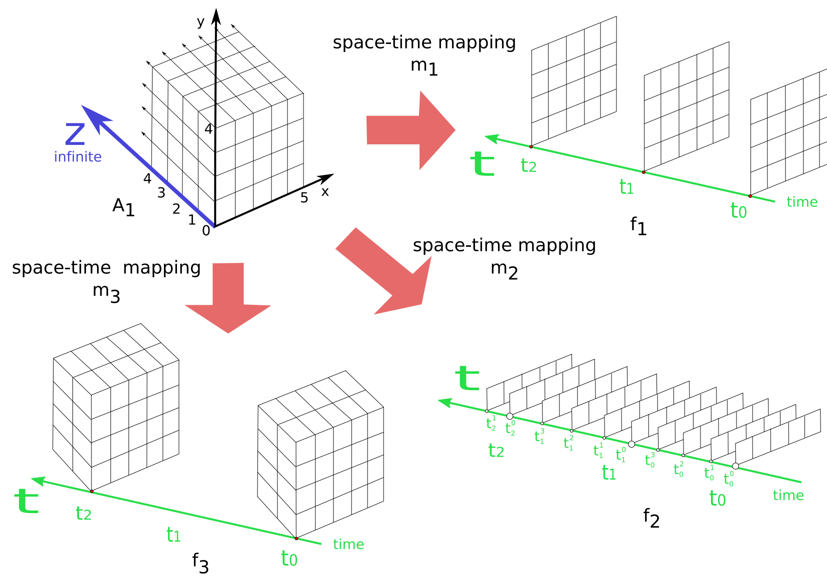


Figure 5.2: Illustration of space-time mapping of a [5,4,*]-array in consideration of different granularity levels.

appears at the highest level in the hierarchy of the Gaspard2 model. The space-time analysis and mapping results in a re-factorization of the original Gaspard2 model, which is similar to the one in the step 1. The difference is that, the refactoring at the first step mainly concentrates on space refactoring, however the refactoring in this step also consider the time issue.

5.1.2.4 Space-time mapping for synchronous modeling

According to different space-time mapping, many different results can be obtained. For instance, in the previous example in Figure 5.2, three different mappings lead to three different flows, i.e., f_1, f_2 and f_3, from the same original array A_1. Without any specified constraints, the basic mapping, i.e., m_1, illustrated in Figure 5.2 is chosen for the synchronous modeling in this thesis. This mapping represents a mapping such that the infinite dimension in a Gaspard2 model is considered as the time dimension, which is translated by the basic clock presented in synchronous languages. So in the synchronous model, there are no more arrays that have infinite dimensions. A Gaspard2 model is considered to be *uninterpretable* by a synchronous model (or an execution model) if there is still infinite dimensions after this space-time mapping, for instance, there is still infinite dimensions in different hierarchies in a Gaspard2 model.

This mapping has the following advantages. First, this mapping is general enough. For instance, f_2 and f_3 can be obtained from f_1 in a synchronous model by using the oversampling or downsampling mechanism of synchronous languages straightforwardly. However, it is not obvious to do it inversely. This mapping is also simple for the following model transformations, because time tilers that implement this mapping are less difficult to implement than other mappings.

5.1.3 Step 3: synchronous modeling

A mapped Gaspard2 model can then be translated into a synchronous equational model thanks to the synchronous modeling of Gaspard2 application specification. As the Gaspard2 model has been refactored and mapped on a discrete time, the main concern in this step is the structural and semantic translation of Gaspard2 models. This translation is detailed in the following sections.

5.1.4 Step 4: code generation

The synchronous equational model is translated into an executable model in this step. As these two models are similar in both structure and semantics, then this translation is a direct one, which is not detailed here.

5.2 General synchronous modeling propositions

The basic concepts of Gaspard2 (Chapter 3) and synchronous languages (Chapter 4) have been presented in Part I separately. Only some summary comparisons on the similarities and differences between these two languages are given here. More details will be given in the description of the synchronous modeling in the following subsections. These comparisons help the translation from one to another. These comparisons involve:

Gaspard2	synchronous languages
task parallelism	concurrency
data parallelism	concurrency
hierarchy	hierarchical invocation
elementary task	synchronous node
data dependency	causality
timelessness	synchronicity
array	dataflow array

5.2.1 Array data structure

In most programming languages, array is defined as a built-in data structure, which has the following characteristics:

- its elements have the same data type
- its elements occupy a contiguous area of storage
- its elements can be accessed by their regular contiguous indexes with regard to their addresses.

These characteristics make array a good candidate for regular computing: such as loop (e.g., *for* and *while* statements). Moreover, some programming languages support array operators (e.g., APL [39], HPF [62]), which enable to operate array directly without a loop statement. This simplification promotes the wide and easy use of array type in these languages.

The data structure manipulated in Gaspard2 is multidimensional array. As Gaspard2 is not a programming language, but a specification language, the data type of array is not important, so some common data types are used for the specification. However, during implementation stage, data types of IPs, which will be deployed on elementary tasks, will be taken into consideration. In contrast, the shape of arrays are important for the specification, which should be preserved.

In synchronous languages, array is also adopted as a basic data type, but in different manner. Lustre and Signal have array data type as a basic data type, whereas Lucid synchrone inherits the array data type from its host language OCAML. Array was first introduced in Lustre in [114], which helps the description of circuits. Later work on Lustre array in [94] favors a concise specification on array operations.

The synchronous modeling. The multidimensional arrays are translated into array-type signals in synchronous language in consideration of the following concerns: the data type of arrays in Gaspard2 are replaced by data types defined in synchronous languages (as a convention, they are compatible); the shape of an array declared in the original Gaspard2 may be changed due to the differences between these two languages. For instance, array shape differences caused by the space-time mapping.

5.2.2 Parallelism modeling

5.2.2.1 Task parallelism

All the tasks defined in the Gaspard2 dependency graph are considered as parallel tasks in spite of dependencies between these tasks. As synchronous languages allow to specify concurrency, Gaspard2 task parallelism can be translated as *node*/*process* concurrency.

5.2.2.2 Data parallelism

Inspired by the parallel nested loop definition in languages for high-performance computing, Gaspard2 uses a similar concept: *repetition space*. In addition to the repetition number that can be calculated from the repetition space, each element in this repetition space defines a repetition position for each repetition of the repetitive task. This repetition position also identifies the correspondences between input tiles, task repetitions, and output tiles. According to the data parallelism definition in Gaspard2, this repetition space does not force an order of the execution of the task repetitions.

The repetition space (Section 2.2.3) is however unfolded (as a repetition space is defined as a multidimensional array) in the synchronous modeling, i.e., there are the same number of task repetitions as the repetition number obtained by the product of all the elements in the shape of a repetition space. Repetition position of task repetitions is implicit and the correspondence between tiles and task repetitions is also pre-set.

The data that a repetitive task consumes and produces is defined by *tilers* over the input and output arrays. The implementation of a tiler can be divided into two steps: first, indexes of tiles are calculated according to the tiling information associated to a tiler, then tiles are built from the input/output array with the help of the indexes calculated from tilers. According to these two steps, two possible solutions are available for the tiler implementation:

- The tiler is implemented by external functions, i.e., external functions calculate all the indexes that are needed to build the tiles during translation. Hence, these indexes are considered to be pre-defined and can be used directly in the target model. In the current synchronous modeling, this solution is adopted. The indexes are calculated by some external functions during the translation so that the target synchronous model is free of tiler computation, hence will not be complicated.

- The tiler is implemented directly in target model, i.e., tiler computation functions are designed in the target model. The indexes are calculated at run time. This solution is obviously a dynamic one, because the indexes can be calculated dynamically at run time. However, the resulting model is complicated and the run-time tiler computation will increase execution time. For instace, when a Gaspard2 model is transformed into a OpenMP Fortran model, the computation of tiler indexes is implemented by Fortran language, and it is embedded in the target OpenMP Fortran model.

5.2.3 Hierarchy and composition

5.2.3.1 Gaspard2 hierarchy and composition

Hierarchy in Gaspard2. Gaspard2 hierarchical specification enables the design of complex applications. As presented in Chapter 3, Gaspard2 models can be considered to have a recursive two-level hierarchy. The high-level hierarchy represents the task parallelism, where it contains an acyclic graph of tasks. The tasks are connected to each other by their data dependencies. Each task in this graph can be refined by another graph of tasks, or by a task that expresses data parallelism. A task, which is situated at the lowest level, is considered as a functionality of certain regular computing that can be deployed by an IP.

Hierarchical composition in Gaspard2. The composition defined in Gaspard2 is similar to component composition or object composition, where the interfaces of components or objects are well defined in order to guarantee the correct composition, as well as some composition conventions. Only data dependencies are involved in the composition definition in Gaspard2, i.e., elements in a Gaspard2 model are composable if they have compatible data dependency.

5.2.3.2 Hierarchy and composition in synchronous languages

Hierarchy in synchronous languages. In Lustre and Lucid synchrone, *node* are declared in a non-hierarchical way or in a flat way. But a node can be invoked in another node. Hence these two languages prefer modular definition, rather than hierarchical definition. Whereas in Signal, one *process* are preferred to be defined in another process if it is invoked by that process.

Synchronous compositionality. The compositionality in synchronous languages has already been studied, for instance, [12] for data-flow languages and [81] for Argos. Details on synchronous compositionality will be given in the following translation.

5.2.3.3 Synchronous modeling

Hierarchy in Gaspard2 can be translated into, on one hand, the modularity of *nodes* in Lustre and Lucid synchrone, and their invocations in a hierarchical way, on the other hand, the hierarchical definition of *processes* in Signal. Parallel and synchronous composition operators in synchronous languages guarantee the same composition properties as those of Gaspard2.

5.3 Gaspard2 and synchronous representations

The translation of a Gaspard2 model into a synchronous model is presented through two abstract syntax of these languages, i.e., Lustre, Signal and Lucid synchrone. The presented abstract syntax aims to give a brief description of these two languages without involving too many technical details.

5.3.1 A Gaspard2 abstract syntax

$$
\begin{array}{lcll}
Application & ::= & \{Task\};\ \{Connection\};\{Deployment\} & (r0)\\
Task & ::= & taskname\ ;\ Interface\ ;\ Body & (r1)\\
Interface & ::= & i, o\ :\ \{Port\} & (r2)\\
Port & ::= & datatype\ ;\ shape & (r3)\\
Body & ::= & Structure^h \mid Structure^r \mid Structure^e & (r4)\\
Structure^e & ::= & null & (r5)\\
Structure^r & ::= & t_i : \{Tiler\}\ ;\ (\mathbf{r}, Task)\ ;\ t_o : \{Tiler\} & \\
 & & \mid t_i : \{Tiler\}\ ;\ (\mathbf{r}, Task)\ ;\ t_o : \{Tiler\}\ ;\ \{IRD\} & (r6)\\
Tiler & ::= & Connection\ ;\ (F; \mathbf{o}; P) & (r7)\\
Structure^h & ::= & Application & (r8)\\
Connection & ::= & p_i,\ p_o\ :\ Port & (r9)\\
IRD & ::= & Connection\ ;\ depvector; default & (r10)\\
Deployment & ::= & \{ipname\ ;\ taskname\ ;\ deploymentinformation\} & (r11)
\end{array}
$$

Figure 5.3: An abstract syntax of Gaspard2 concepts.

The Gaspard2 abstract syntax is illustrated in Figure 5.3. A Gaspard2 *Application* (rule ($r0$) where { } denotes a set) consists of a set of *Tasks*, *Deployment* and *Connections* (rule ($r9$)), which connect the tasks. These tasks share common features (rule ($r1$)): a *taskname*, an *interface* (rule ($r2$)) and a *Body* (rule ($r4$)). *Interface* specifies input/output *Ports* (typed by `i` or `o` in rule ($r2$), and *Port* is defined in rule ($r3$)) from which each task receives and sends multidimensional arrays. *Body* (rule ($r4$)), which depends on the type of task as follows:

- *Elementary task* (rule ($r5$)). It corresponds to an atomic computation block. Typically, it represents a function or an IP. Elementary task can be associated with an IP through *Deployment*, which contains deployment information (rule ($r11$), but only elementary task is involved in `Deployment`).
- *repetition context task(RCT)* (rule ($r6$)). It expresses data-parallelism in a task. The repetitions of the associated repetitive task are assumed to be independent and schedulable following any order, generally in parallel. The attribute **r** (in the rule ($r6$)) denotes

the *repetition space*, which indicates the number of repetitions. In addition, each task repetition consumes and produces sub-arrays with the same shape, which are defined via *tilers* (rule ($r7$)). *IRD* can be also specified in an RCT (rule ($r10$)), which describes the dependency between repetitions of the repetitive task. *depvector* specifies which repetition relies on which repetitions and *default* gives default value.

- *Hierarchical task* (rule ($r8$)). It is represented by a hierarchical acyclic task graph, in which each node consists of a task, and edges are labeled by arrays exchanged between task nodes.

5.3.2 Synchronous equations abstract syntax

This abstract syntax is constructed based on common aspects of synchronous languages, which is intended for the modeling of three synchronous dataflow languages. The syntax is illustrated in Figure 5.4:

$$
\begin{array}{lcll}
Module & ::= & \{Node\} & (s1) \\
Node & ::= & nodename \; ; \; Interface \; ; \; EqSystem & (s2) \\
Interface & ::= & Interface^i \; ; \; Interface^o & (s3) \\
Interface^i & ::= & \{SignalDeclaration\} & (s4) \\
Interface^o & ::= & \{SignalDeclaration\} & (s5) \\
SignalDeclaration & ::= & signal \; ; \; DataType & (s6) \\
DataType & ::= & type \; ; \; shape & (s7) \\
EqSystem & ::= & \{Equation\} \,|\, extnodelink & (s8) \\
Equation & ::= & EqLeft \; ; \; EqRight & (s9) \\
EqLeft & ::= & null \; | \; signal & (s10) \\
EqRight & ::= & signal \; |SignalDelay \; | \; Invocation & (s11) \\
Invocation & ::= & nodename \; ; \; \{signal\} & (s12) \\
SignalDelay & ::= & signal \; ; \; delayinstant \; ; \; \{defaultvalue\} & (s13)
\end{array}
$$

Figure 5.4: An abstract syntax of synchronous concepts.

As synchronous languages have been presented in a detailed way in Chapter 4, only a brief description of this syntax is given here. All the nodes are declared in a module (rule ($s1$)). A node is composed of $Interface$ and $EqSystem$ (rule ($s2$)). An $Interface$ is divided into two families: $Interface^i$ and $Interface^o$ (rule ($s3$)). In Signal, $Interface^i$ and $Interface^o$ can be absent for a node. An $EqSystem$ (rule ($s8$)) can have at least one *Equation*, or *extnodelink*, which indicates an external implementation of the node. An equation may be assignment of a *signal*, $SignalDelay$ or just an *Invocation* of another node (rule ($s9$), ($s10$) and ($s11$)). A $SignalDelay$ is similar to a *pre* operation, which takes the value of the signal at the previous *delayinstant* instant (rule ($s13$)). $\{defaultvalue\}$ are default values when no values are provided for the previous instants.

This syntax is also the basis of the metatmodel of synchronous dataflow languages presented in Chapter 7.

5.4 The translation between the two representations

The first step of the translation of Gaspard2 model into synchronous model is structural, and the second step involves the semantic explanations. Some proofs of translation correctness are then given after these two steps. In the following translation, Gaspard2 concepts are in italic font and synchronous concepts are in monospace font.

5.4.1 Structural translation

The synchronous `Module` (s1) is the container of all nodes. A Gaspard2 *Application* (r0) is first translated into a synchronous `Node` (s2): *Application* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `Node` (*A* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `B` denotes *A* is transformed into `B`). A *Task* (r1) is translated into a `Node`: *Task* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `Node`. An *Interface* (r2) can be translated into synchronous `Interfaces` (s3): *Interface* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `Interface`. A *Port* (r3) is a connection point of a task, it is translated into a synchronous `signal`: *Port* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `signal`.

A *Body* (r4) represents the internal structure of a task. It can be considered as an `Eqsystem` (s8): *Body* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `Eqsystem`. Three kinds of structures are involved in a body. A $Structure^e$ (r5) represents the structure of an elementary task, so no direct translation is carried out. However, deployment will be used for further translation of an elementary task. A $Structure^r$ (r6) is translated into a set of `equations` (s9) explained later. A $Structure^h$ (r8) is a structure that has a compound task. This compound task is an application that has been explained previously. *Deployment* (r10) is translated into an `extnodelink` (s8): *Deployment* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `extnodelink`.

IRD is translated by `SignalDelay`: *IRD* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `SignalDelay`, *depvector* is translated by `delayinstant`: *depvector* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `delayinstant`, and *default* is translated by `signal`: *default* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `signal`.

5.4.2 Translation of behavioral aspects

Except the structural translation, some special aspects in translation are presented here. These special aspects are usually caused by the different levels of the two languages from the point of view of programming languages. Synchronous languages involve some execution aspects in contrast to Gaspard2, so these execution aspects should be added in the translation.

5.4.2.1 Translation of tasks and their connections

Tasks, which are used in other tasks, are translated into `Invocations`: *Task* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `Invocation`, in addition to their translations into nodes. The *connections* between tasks are translated into `signals`: *Connection* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `signal`, which can be used in both `SignalDeclarations` and `Equations`.

Tiler (r7) is a special *connection* that has a tiling tag, which is identified by $(F; \mathbf{o}; P)$. In addition to being translated into `signal` (s6, s10 and s11): *Tiler* $\stackrel{\mathcal{T}}{\Longrightarrow}$ `signal`, a tiler node

should be created additionally, which achieves the tiling function (see Section 5.2.2.2): *Tiler* $\overset{\mathcal{T}}{\Longrightarrow}$ `Node`.

5.4.2.2 Repetition context task translation

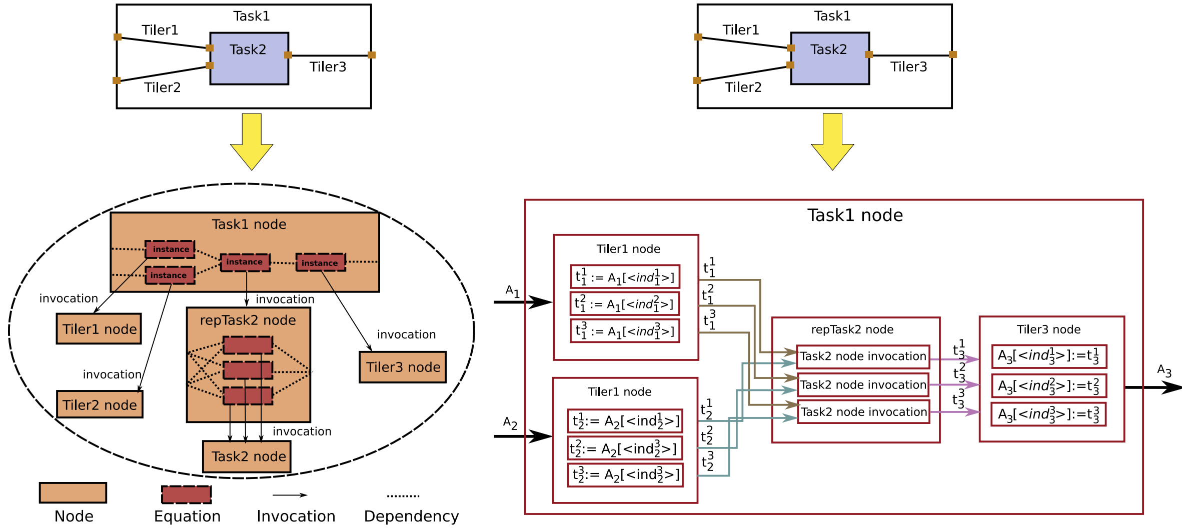


Figure 5.5: Synchronous modeling of tilers and a repetition context task. The left-side figure illustrates the structural translation, which focus on resulting nodes. The right-side figure exhibits data dependency and tile construction in resulting nodes. The relation between data dependency and tiles are built according to the repetition space specified for *Task2*.

A RCT can be translated by a set of nodes where one node invokes all others. Figure 5.5 shows an example of this translation. The two sub-figures illustrate the translation from different point of views. The left-hand one focuses on node invocations, whereas the right-hand one concentrates on construction of tiles. The Gaspard2 RCT *Task1* is translated into a set of nodes with invocation relations between them: `Task1 node` is the main node, in which there are four equations for invocations to others nodes: `Tiler1 node`, `Tiler2 node`, `Tiler3 node` and `repTask2 node`. `Tiler1 node`, `Tiler2 node` and `Tiler3 node` are nodes that achieve tiling function of *Tiler1*, *Tiler2* and *Tiler3*. `repTask2 node` is a node that regroups all the repetitions of `Task2 node`.

Repetitive task translation. A repetitive task (*Task2* in Figure 5.5) inside a RCT (*Task1*) is translated into two nodes: `Task2 node` and `repTask2 node`.

- `Task2 node` is the translation of the task *Task2* itself.
- The second node `repTask2 node` is the modeling of the repetition of the task *Task1*, in which `|r|` (three in the example) invocations to *Task2* form the equation system of `repTask2 node`. `repTask2 node` is then invoked in the RCT *Task1*.

5.4.3 Correctness of the translation

The correctness of the translation of the Gaspard2 model into the synchronous model can be checked at two levels: task level and task graph level. At the task level, elementary task and RCT are checked. At the task graph level, application or hierarchical task are checked.

5.4.3.1 Elementary task

Elementary task is considered as a *transformation* from inputs to outputs in Gaspard2, which can be implemented by IPs. This transformation has several properties:

- timelessness: the transformation time is not considered in Gaspard2, therefore it has no effect on task behavior.
- statelessness: there is no state memorized in the task, so the outputs only relies on inputs.
- determinism: the same outputs are produced if the inputs are the same. So this transformation is independent from the environment, its internal states, etc.
- synchronization between inputs and between outputs: according to the data dependency definition in Gaspard2, only the presence of all the inputs can fire the following computing task. A synchronous node adopts the same property, for instance, the *endochrone* respected by Lustre and Signal guarantees the same property in the translation.

An elementary task node in the synchronous model has the same properties exactly. Under the synchrony hypothesis, the reaction time of a node is negligible. No memory operators are introduced into the model, hence nodes are stateless. A synchronous node is determinist according to the synchronous language definition. Hence a synchronous node and a Gaspard2 task share the same properties.

5.4.3.2 Repetition context task

A Gaspard2 RCT has also the previous mentioned properties owned by an elementary task. In addition, its behavior of the transformation from inputs into outputs is specialized by its exposed internal structure. First, suppose that the internal repetitive task is an elementary task, the equivalence between an RCT and its translating node is illustrated hereinafter.

As illustrated in Figure 5.5, a single repetitive context task is translated into a set of nodes. The relation between these nodes are also shown in the same figure. Figure 5.6 illustrates the detail of these synchronous nodes with simplified equations.

First of all the four previously mentioned properties are checked here.

- timelessness: in spite of some tiling actions specified in a Gaspard2 RCT, an RCT is still considered to be timeless. The synchronous composition also guarantees the instantaneity of the results.
- statelessness: the tilers do not introduce state, so the RCT keeps stateless. The translated nodes presented in Figure 5.5 are all stateless nodes.

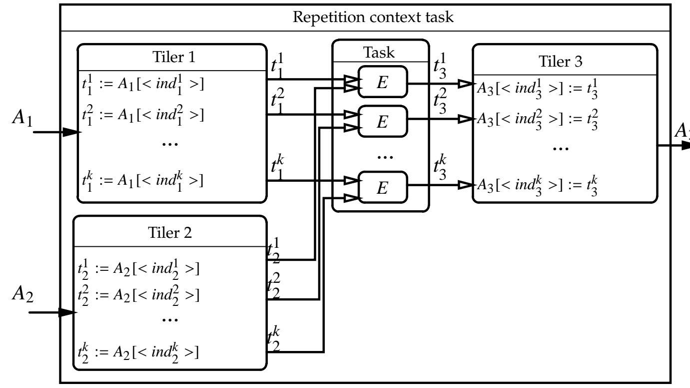


Figure 5.6: Synchronous modeling of tilers and a repetition context task.

- determinism: in the definition of a tiler node as presented in Figure 5.6, the output array is in fact a part of input array, and their indexes are pre-calculated, so the output relies only on the input. The result of composition of all the nodes from an RCT is still determinist because of the determinism, timelessness and statelessness of all these nodes.

- synchronization: the same as an elementary task.

The full parallelism specified in an RCT is completely preserved thanks to the synchronous composition operator, which allows a simultaneous, concurrent as well as non interleave composition of equations. It is true not only for the repetitive task, and also for the building of the tiles.

Finally, correctness of structural translation is checked here. Regardless of the multiple node declaration for the purpose of translation of an RCT, Figure 5.6 can be written in a textual equation system as in Equation 5.1. There is only a presentation form difference between Figure 5.6 and Equation 5.1.

$$\begin{array}{c} t_1^1 := A_1[< \mathrm{ind}_1^1 >] \;\; ; \;\; ... \;\; ; \;\; t_1^k := A_1[< \mathrm{ind}_1^k >]; \\ t_2^1 := A_2[< \mathrm{ind}_2^1 >] \;\; ; \;\; ... \;\; ; \;\; t_2^k := A_2[< \mathrm{ind}_2^k >]; \\ t_3^1 := E(t_1^1, t_2^1) \;\; ; \;\; ... \;\; ; \;\; t_3^k := E(t_1^k, t_2^k); \\ A_3[< \mathrm{ind}_3^1 >] := t_3^1 \;\; ; \;\; ... \;\; ; \;\; A_3[< \mathrm{ind}_3^k >] := t_3^k; \end{array} \tag{5.1}$$

where in the first two lines, patterns (t_1^1 , ... , t_1^k, t_2^1,...,t_2^k) are constructed from the two input arrays (A_1,A_2), in the third line, the k repetitions on E are illustrated, and the fourth line shows how patterns (t_3^1 , ... , t_3^k) are used to build the output array (A_3). $< \mathrm{ind}_i^j >$ denotes a set of index associated with the elements of a tile that corresponds to the point j in the repetition space; Only part of equations are given here for the sake of conciseness and clarity. If the tiler for an array A_i is $(F_i, \mathbf{o}_i, P_i)$, $\mathbf{s}_{\mathbf{p}_i}$ is the shape of the pattern associated with the task E tiles in array A_i and $\mathbf{s}_{\mathbf{a}_i}$ is the shape of array A_i, then $< \mathrm{ind}_i^j >$ denotes the following set of indexes (Equation 5.2):

$$\forall j \in \mathit{repetition\ space}, \;\; < \mathrm{ind}_i^j >= \{\mathbf{o}_i + jP_i + \mathbf{x}F_i \bmod \mathbf{s}_{\mathbf{a}_i}, \text{ where } \mathbf{0} \le \mathbf{x} < \mathbf{s}_{\mathbf{p}_i}\} \tag{5.2}$$

The equation system (5.1) is well structured, but is not a presentation with regard to its original Gaspard2 specification. Particularly, the repetitions are not clear. However, it can be resolved by considering the commutativity and associativity properties of the composition operator in synchronous languages. The operator helps to permute the equations in 5.1 so as to obtain the equations in 5.3.

$$\begin{array}{l} t_1^1 := A_1[< \mathrm{ind}_1^1 >] \;\; ; \;\; t_2^1 := A_2[< \mathrm{ind}_2^1 >] \;\; ; \;\; t_3^1 := E(t_1^1, t_2^1) \;\; ; \;\; A_3[< \mathrm{ind}_3^1 >] := t_3^1 \\ \ldots \\ t_1^k := A_1[< \mathrm{ind}_1^k >] \;\; ; \;\; t_2^k := A_2[< \mathrm{ind}_2^k >] \;\; ; \;\; t_3^k := E(t_1^k, t_2^k) \;\; ; \;\; A_3[< \mathrm{ind}_3^k >] := t_3^k \end{array} \tag{5.3}$$

where each line demonstrates one repetition j in the repetition space.

Each line of the equation system (5.3) consists of the introduction of a few intermediary local signals in the following equation (5.4), hence it can be trivially proved equivalent:

$$\forall j \in \textit{repetition space}, \;\; A_3[< \mathrm{ind}_3^j >] := E(A_1[< \mathrm{ind}_1^j >], A_2[< \mathrm{ind}_2^j >]) \tag{5.4}$$

However, it is costless to introduce intermediary signals, as they can be compiled away by the synchronous compilers and optimizers.

The modeling of a repeated computation then amounts to the synchronous composition of all the models of each point in the repetition space.

5.4.3.3 Hierarchical task

A hierarchical task is considered as a task that contains a task graph, whose internal tasks can be elementary tasks, RCTs or hierarchical tasks. A simple hierarchical task with only elementary tasks and RCTs will be considered first. As all the internal tasks in this hierarchical task are timeless, stateless and determinist, the same properties are also passed to the hierarchical task. Analogously, the nodes translated from the hierarchical task are all timeless, stateless, determinist nodes. A hierarchical task also expresses task parallelism, i.e., all the tasks in the hierarchical task run in parallel. Nodes obtained from the translation have the same property, i.e., all the nodes are concurrent.

A complex hierarchical task may have hierarchical tasks inside or it is an RCT whose internal repetitive task is still a hierarchical task, etc. Once a simple hierarchical task is proved to have the same properties, the same proof can be carried out recursively on the hierarchical task so that a complex hierarchical task is checked completely.

5.4.4 Translation with serialization and partition semantics

In addition to the translation presented previously, which is called translation with parallel semantics, other refined translations are discussed here, i.e., translation with serialization semantics and translation with partition semantics. These different semantics are distinguished according to the translation of the parallelism defined in an RCT, i.e., how the parallelism of its repetitions is handled in the translation. The translation with parallel semantics preserves the maximum parallelism of a Gaspard2 model, i.e., a full parallel model, new translations consider different constraints that result in fully or partly serialized models.

5.4.4.1 Translation with serialization semantics

While the parallel model (the result of the translation with parallel semantics) fully preserves the data-parallelism, a serialized model (the result of the translation with serialization semantics) offers the serialization of the parallelism, which makes sense when Gaspard2 tasks run on a single processor.

The difference between this translation and the previous translation with parallel semantics is the translation of the repetitive context task, in which the serialization is carried out, and of its internal elements, i.e., tilers and the repetitive task. In addition to computing tiles, an input tiler is also a flow generator, which sends the tiles in a dataflow manner. Inversely, an output tiler receives a flow of tiles and assembles them into arrays. The repetitive task has only one instance, which is repeatedly called in a sequential way so that flows of input tiles are processed in a dataflow way and the flow of output tiles are sent.

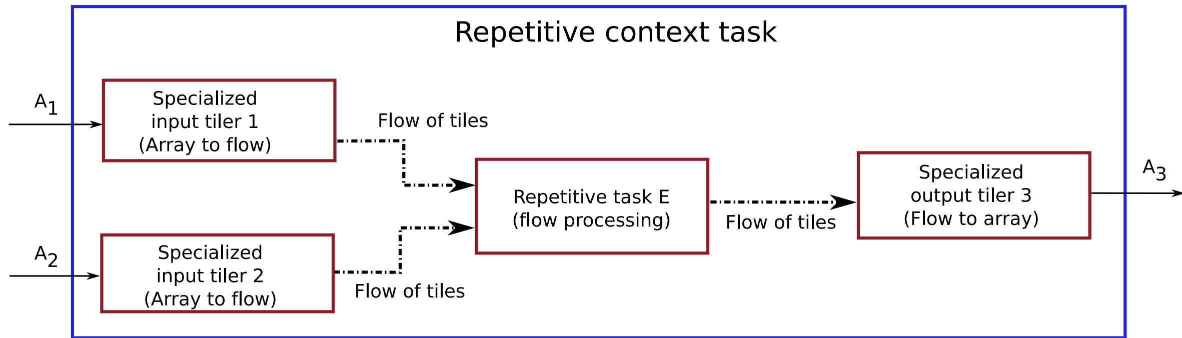


Figure 5.7: Result of a translation of the repetition context task in Figure 5.6, in which the repetitions of the repetitive task are serialized.

Tiler translation. Compared to the previous translation of tiler in a parallel model, the translation here should also take flows into account, which includes the following aspects: *1*) *clock*: tilers in this model have different clocks for the inputs and outputs, which have an oversampling or downsampling relation, in order to build flows from an array or inversely. For instance, the input tiler utilizes an oversampling mechanism so that the output is faster than the inputs, moreover, the outputs are serialized into a flow due the oversampling mechanism. *2*) *memorization*: the input and output clocks of a tiler are different, but the array (as input or output) should always be available for any clocks associated with the tile generation, which are faster than the clock of the array. Hence, the memorization of the array is necessary. *3*) *scheduling*: the flows produced by an input tiler should be coherent (particularly in order) with the flows taken by the output tilers, hence a common sequencer should be considered here, which is dedicate to scheduling of the tiles in a coherent way for both input and output tilers. [47] presents two examples of input and output tiler translation, which is encoded in Signal.

Repetition context task translation. A RCT is modeled by the composition of the three kinds of tasks, tilers and repetitive task. The former is described already, the latter is represented by a single invocation of the node representing the repeated body.

Conclusions. In spite of the different semantics of the parallel model and serialized model, they are strictly functional equivalence (or flow-equivalence [77]). The following analysis gives the proof:

- timelessness: the transformation of inputs into outputs of an RCT is always timeless, even if the tiles are serialized. Hence, the serialized time model does not change the behavior of an RCT.
- statelessness: memory are introduced in the model, but it is does not introduce execution states for the task, because it is always equivalent to inputs.
- determinism: a common sequencer is integrated for input and output tilers, however, the order of tiles decided by the sequencer does not change the values of tiles and output arrays. Hence a RCT is still determinist from the point of view of its interface.

The serialized model is more compact in comparison with the parallel model because it only defines one instance of the repetitive task in a RCT. It therefore significantly reduces the size problem, which is always caused by the enumeration of all repetitions of a repetitive task. More generally, Gaspard2 applications can be modeled by combining both parallel and serialized models, which extends the scalability of the synchronous modeling [47].

5.4.4.2 Translation with partition semantics

Mixed parallel and serialized model provides a refined modeling of a Gaspard2 model, this mixture can still be refined from a parallelism partition view, which enables partial parallelism specification, compared to the full parallelism specified by a parallel model. This partition modeling is reasonable from the point of view of the allocation of repetitive tasks on multi-processors, each of which executes certain serialized repetitions. Figure 5.8 shows an example of this modeling. The repetitive task E has six repetitions, which are partitioned into two groups. Each group is executed in a sub RCT, which executes three serialized repetitions. This modeling is based on the parallel and serialized modeling with an extension of repetition partition.

The partition is mainly carried out on the repetition space, as a result, some sub-spaces are defined. Each sub-space represents the repetitions that are serialized in the same sub RCT, similar to the serialized model. However, between these sub-spaces, they are considered to be processed in a parallel manner, similar to the parallel model. This partition of repetition space is predefined, where the translation can directly take the partition result into account.

The translation of a partitioned model is a little different from the previous translations of parallel and serialized model. The main difference is the introduction of sub-RCTs, each of which concentrates on the processing of a sub repetition space. However, the translation of a sub-RCT is similar to the one of the serialized model. The translation of tilers in an RCT is a little different from the one of the parallel model. According to the sub repetition space, the tiles are assembled into groups, which is very simple in the translation.

A serialized model is equivalent to a parallel model from a functional point of view. Due to the same reason, a partitioned model is equivalent to a parallel or a serialized model. As the partition is predefined and determinist, the partitioned model preserves the same properties as those of a parallel or serialized model. Moreover, a partitioned model is more suitable to model applications running on multi-processors.

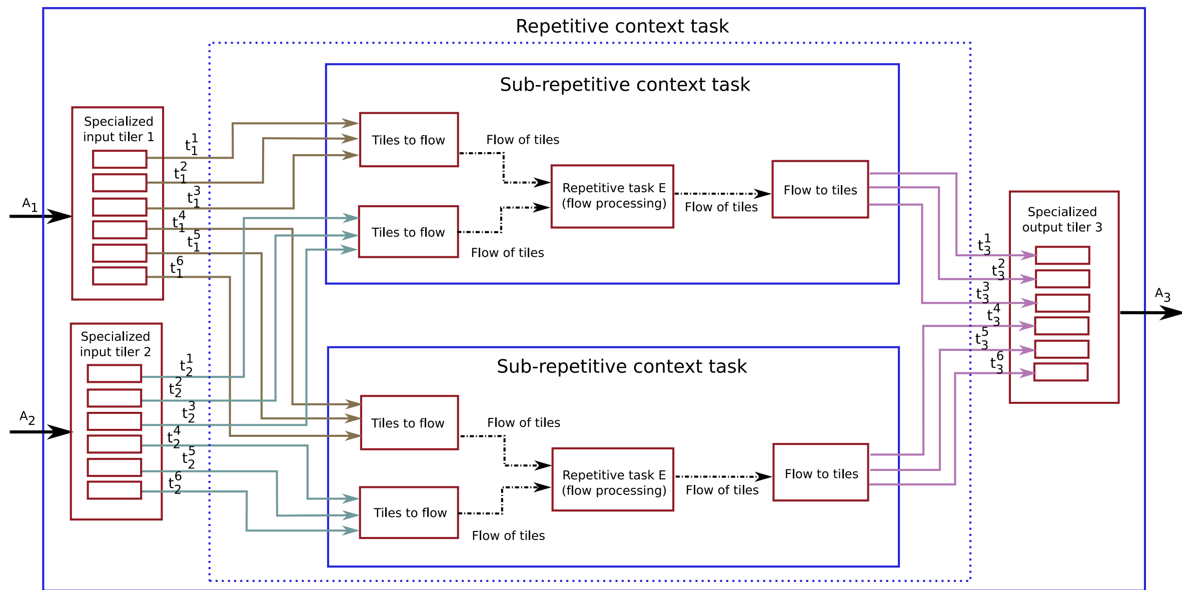


Figure 5.8: Result of a translation of the repetition context task in Figure 5.6, in which repetitions are partitioned for processing.

5.5 Conclusions

The previously presented synchronous modeling leads to an intermediate model between data-parallel applications (Chapter 3) and synchronous data-flow languages (Chapter 4). First, this model is generic enough so that it will not suffer from the complexity and particularity of target languages (Lustre, Signal and Lucid synchrone). It preserves only the common aspects of these languages. Hence, on one hand, it is possible to generate these three languages with only one model, on the other hand, this model is easy to be extended to adapt to anyone of these languages with its particular aspects. Moreover this synchronous model enables potential improvements, for instance, the integration of application control (Chapter 6).

Secondly, this model is intended to capture as many as possible the properties of the original Gaspard2 data-dependency model, such as the parallelism, determinism, data dependency and hierarchy. These properties are preserved in the resulting synchronous model with the help of similar properties in synchronous languages, such as concurrency, modularity, re-usability, causality and hierarchy. The property coherence between the two models contributes to guarantee that the verification carried out on the resulting synchronous code is also valid for the original Gaspard2 model.

Apart from the parallel model, serialized and partitioned models are also proposed for a wider adaption with non-functional constraints. These two models, which were briefly presented in this chapter, offer certain refined view of the basic parallel model at a high level.

Finally, this modeling provides the possibility to be implemented with the MDE approach. This implementation includes metamodeling, model transformations with the help of some MDE-based tools, which is detailed in Chapter 7.

Chapter 6

Reactive control extension of Gaspard2

6.1 Introduction **90**
6.1.1 Control modeling in DIP applications 90
6.1.2 Basic ideas of Gaspard2 control 92
6.1.3 Previous control proposition in Gaspard2 96
6.2 An extension proposal for control **96**
6.2.1 Mode switch task and modes 96
6.2.2 State graphs and state graph tasks 97
6.2.3 Task compositions of SGT and MST in Gaspard2 101
6.3 Reactive control extension in Gaspard2 . . . . **105**
6.3.1 Gaspard2 control in a dataflow context 105
6.3.2 Reactive automata based control mechanism in Gaspard2 106
6.4 Typical examples **107**
6.4.1 A typical example of a counter 107
6.4.2 A control example for cell phone video effect 109
6.5 Conclusions **110**

This chapter presents the study of control-related behavior specification for DIP applications, particularly in the Gaspard2 framework, and the integration of reactive and state-based control in Gaspard2 in accord with MARTE. This work is based on the previous proposition of Ouassila Labbani [71], which introduced a control mechanism in Gaspard2. The previously proposed control mechanism is intended to satisfy the requirement of behavioral specification in Gaspard2. However, some important features are absent in this proposition, which include parallel and hierarchical composition definition of control operators, formal semantics of the control model, etc. These features provide the possibility to build complex, but safe and unambiguous systems.

Therefore, the study presented here leads to the proposition of a refinement and extension of previous Gaspard2 control with new operators and refined semantics, which integrate the previously mentioned features. Moreover, the new control mechanism takes the following aspects into account:

- the control mechanism should be compatible with newly proposed UML MARTE profile, i.e., the control can be modeled by using the MARTE profile or MARTE compatible concepts;
- the control needs to consider *conciseness* and *clarity*, which will benefit, on one hand the documentation and communication, and on the other hand model transformation that enables the implementation level code generation;
- the control also needs to take *safe design* into account, particularly expected *properties* in the design, i.e., at the high-level design stage, which helps to reduce fault risks so as to reduce verification cost and time. Besides, it needs to provide the formal semantics to enhance verifiability, so that its behavior can be deduced or checked to be correct.

This extension is considered orthogonal to the synchronous modeling introduced in Chapter 5, i.e., a similar extension in the synchronous model is necessary. The control in the synchronous model should be compatible with the Gaspard2 one so that the translation of one into another is not difficult.

The proposed control extension does not aim at Array-OL, which is a pure data parallelism specification language. It is integrated into Gaspard2, which is subject to the refactoring and space-time mapping steps from the original Array-OL one. Thereby, the Gaspard2 specifications are optimized and executable in relation to some particular technologies or hardware architecture. Some primitive ideas of a state-based control mechanism in alignment with Array-OL and Gaspard2 are briefly presented in Section 6.1. New operators of the control extension are then presented in Section 6.2. The corresponding reactive control in Gaspard2 is presented in Section 6.3. Some examples are then given in the last Section 6.4.

6.1 Introduction

6.1.1 Control modeling in DIP applications

The core formalism of data parallelism in Gaspard2, namely Array-OL, allows a concise specification for DIP applications. It is based on *data dependency* descriptions and *repetition* operators to express data parallel applications. It allows to specify static applications, more precisely, it is not possible to change data processing at run-time. For instance, in the high-performance video and image processing, once filters are chosen, the application behavior is completely fixed statically. It is not possible to change the application configuration at run-time. Obviously, this kind of applications benefits from the simplicity in design and high performance in execution. Moreover, the system performance can be easily evaluated.

However they are considered as advantage only for the systems that do not require many resources for processing, such as memory, computing power, energy and communication. The disadvantages of these systems are evident: they are too specific and static. Even a small replacement or a modification of a functionality in the system leads to the reconstruction of the system. The absence of behavioral specification becomes a big constraint in current DIP applications, where abundant hardware resources provide the potentiality of application flexibility, in particular. For instance, the increasing processing capability of modern multimedia cellular phones permits to choose different video effects while playing back a video clip.

System behavior in systems can be specified in different forms, and at different levels, etc. In this thesis, we focuses on *control-related behavior*, which will only be called behavior afterwards for simplicity. According to the context of Gaspard2, the behavior modeling is based on the following characteristics:

High-level control. Behavior is present at several levels in a system with regard to Array-OL *tasks*: *a*) *intra-task* level, which is the finest level that makes a task dynamic, for instance, the if/then/else statements specified in a task result in dynamic changes during the execution of the task;

b) *inter-task* level (only atomic tasks are considered here), where dynamic application is specified, i.e., through the changes of tasks, the application is given a behavior; *c*) *application* level, where applications can be changed through the switches between applications. An application here refers to a hierarchical task that can accomplish a complete functionality. The overall behavior of the system is the composition of the control at these three levels. But we are only interested in the high level control in Gaspard2, including inter-task and application level control. The intra-task level control is too fine thus does not help to simplify the application specification, particularly, an atomic task (it does not have any hierarchy) is considered as an elementary task, whose behavior is not visible.

Data/control separation. The data control separation has been studied in the previous work of Ouassila Labbani [71]. Her study is based on SCADE[1], which integrates imperative features in the dataflow language Lustre. The imperative part can be specified with state machines, and the data computation part can be specified in Lustre.

State-based control. A state-based control mechanism, e.g., state machines, is preferred in Gaspard2, not only because of its wide adoption in academia and industry, but its verifiability supported by large quantity of formal verification tools. New extensions has been proposed based on the *mode automata* [84]. There are evident advantages: mode automata are a simplified version of Statecharts [58] in syntax, which have been adopted as a specification language for control oriented reactive systems. Mode automata has a clear and precise semantics, which makes the inference of system behavior possible and is supported by formal verification tools.

Safe control. The integrated control in Gaspard2 is intended to take safety design concerns into account, including safe properties and verifiability. The first one enables to build the system in a correct manner, which reduces the overall verification cost and time. The second one involves the verification of the resulting specification in a simple, fast and reliable way. Mode automata are designed in consideration of safety concerns. The modes are exclusive so that, on one hand the exclusivity of modes makes a clear separation of application functionality, which reduces possible fault risks, and on the other hand the faults in one mode will not take effect in others modes, which helps to avoid fault propagation.

Previous discussions exhibit the basic requirements of Gaspard2 with regard to the behavioral specification. The proposed control mechanism is based on these requirements.

[1] http://www.esterel-technologies.com

In the next section, these requirements are considered in combination with the Array-OL language, which is dedicated to the full parallelism specification that enables parallelism exploitations on different hardware architectures. In order to preserve the full parallelism advantage, the control extension is therefore carried out in Gaspard2, which integrates Array-OL main features in consideration of implementation aspects through certain techniques, such as *space-time mapping* and *re-factoring* of original Array-OL specifications. In our point of view, a Gaspard2 model is taken as a *machine interpretable* version of Array-OL for the purpose of implementation.

6.1.2 Basic ideas of control in accordance with the Gaspard2 core formalism

This subsection first presents the study of control notion in the context of Gaspard2 in a conceptual way without involvement of any execution model, i.e., a control mechanism that is compatible with the Array-OL specification model. Then the integration of this control mechanism in Gaspard2 is presented.

6.1.2.1 Array-OL-compatible behavior change

For the purpose of a better understanding, the expected behavior change of Gaspard2 is illustrated with an Array-OL graphical syntax, where no execution model is involved. First of all, how the control mechanism can work in a Gaspard2 RCT model is presented. Figure 6.1 shows an example of RCT, where four main types of elements in an Array-OL local-level specification are found: array, tiler, task and repetition space. Intuitively the behavior of a RCT can be achieved by changing any of these main elements or any combination of these elements. In the following examples, these possibilities are demonstrated in a functional point of view, which neglects when and how the control takes effect so as to make the RCT dynamic.

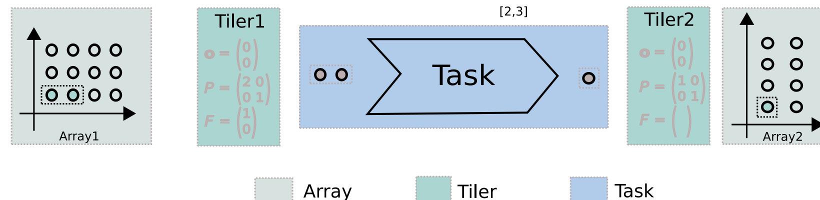


Figure 6.1: An example of Array-OL specification with regard to a task. A task, which is represented by the box in the center, is connected to its input array (left-hand box) and its output array (right-hand box) through tilers. The repetition space of the task is illustrated above the task box. The patterns taken by the task as input and output are also placed aside of the task in the task box. Only the tiles used by the task repetition at [0,0] in the repetition space are surrounded by a box with dash-lines.

- change of a task (Figure 6.2): a task can be changed in a local model in response to some change requirements from the environment, the platform or the application. For instance, the change of algorithms (image style, image quality, etc.) in a mobile multimedia device in order to answer user choices. In Figure 6.2, only one task, either *Task1*

or *Task2* is selected to run according to some control, which is not shown here. A prerequisite of this change is that the two tasks have the same interface so that they are compatible with other elements in the local model.

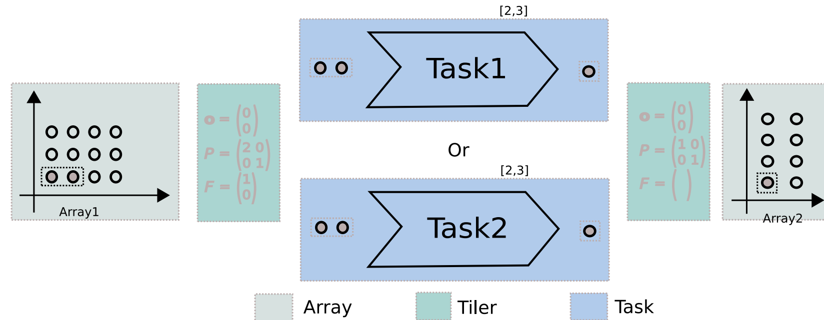


Figure 6.2: An example of task change in Array-OL specification.

- change of the *repetition space*: the *repetition space* associated to each task can also be changed, which leads to the coverage change of the input and output array processing. For instance, when watching a video clip, the window is changed to be bigger or smaller without the change the video display size, hence only the coverage of the video display through the window is changed. The video and its processing task are not changed.

- change of a tiler (Figure 6.3): a tiler in a local model can be changed under some conditions. A tiler makes a bridge between an array to be processed and a task. Furthermore, the corresponding interface of the task determines the pattern shape of the tiler. Thus, the change of a tiler without change of the corresponding task requires that the pattern shape of this tiler should not be changed. For instance, consider the zoom feature in a picture viewer: when the zoomed picture is larger than the display window, we need to move the display window so that we can see somewhere else in the picture. The original picture, the zoom algorithm (or the zoom task), and the size of display window are not changed, only the reference point in the picture for the display window is changed. This can be modeled as a tiler change, i.e., the changes of *original points* in the tiler.

- change of the input/output arrays (Figure 6.4): the input arrays or output arrays can be changed too. For instance, while watching a video clip on a cellular phone, its source can be changed: from an online library to local memory storage if they have the same video size.

- change of any combinations of task, tiler, array and repetition space: it is possible to change any combination of these four elements at the same time. In this case, the change can be seen as a change of the RCT directly, which is easier to model.

6.1.2.2 Array in representation of control

The changes of task, tiles or arrays in a local model should follow some control events. As all the data in Gaspard2 are modeled as array, it is natural to model the control events as arrays,

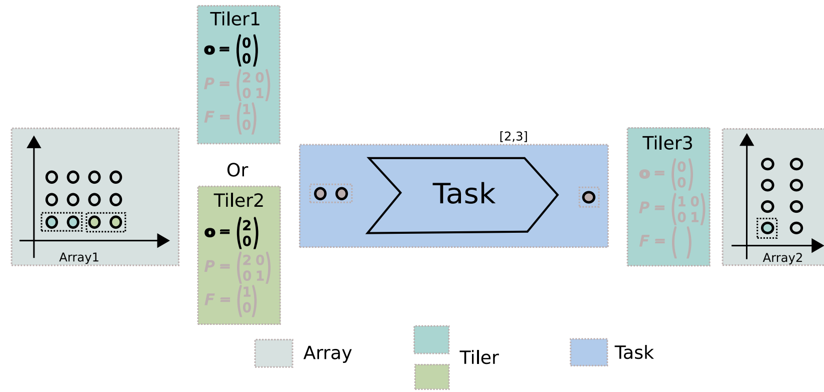


Figure 6.3: An example of tiler change in Array-OL specification.

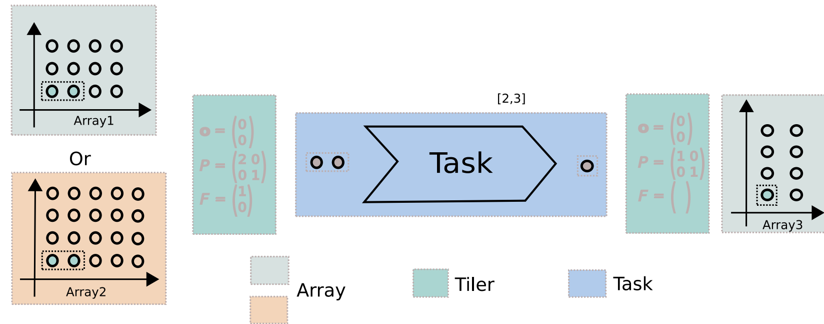


Figure 6.4: An example of array change inArray-OL specification.

e.g., an infinite flow of control events is modeled as an infinite array. However, Gaspard2 specifications do not take environment or platform aspects into account. Hence, the control event array is also modeled independently from environment constraints in the same way as data, e.g., what they are and when and how they are present. Control event arrays, similar to data arrays, are supposed to construct and exhibit *data dependencies* between tasks.

6.1.2.3 The correspondence between control and data array

In low-level implementation, according to the environment or platform, different dataflow may have different clocks, particularly dataflow and control flows, which leads to the synchronization problem in the composition of these flows. However, only data dependency is taken into account in Gaspard2, which is a high-level specification language, i.e., no execution model or non-functional constraints are involved, hence the synchronization between control and data computation or data computation granularity with regard to control [71] is not considered here.

As a dependency specification, data and control arrays are considered to be matched to each other, i.e., a good correspondence is required. The correspondence between control and data array indicates the control array can be regularly mapped onto the data array, which also implies the control scope of the data. This mapping can be considered as an *affine-transformation* that leads to two possible control scope levels: *task level* and *repetition level*. The first one indicates all the repetitions of a task are changed into those of other tasks.

Hence this is a complete change of task. The second one, repetition-level control enables to change the task at a finer level, which allows the switch between the repetitions of different tasks. As the control array can be mapped onto the data array, they can be also mapped onto the same concrete time.

Although at the specification level, the control array can be mapped onto the data array, this is not always true at the implementation level because of the non-functional constraints on control and data flows, e.g., a dataflow may have different clock from the clock of control events, which is not a determinist flow with regard to the dataflow. This issue will be discussed in Section 6.3 when the reactive execution model is introduced.

6.1.2.4 Synthesis and conclusions

Presented in the previous subsections, the Gaspard2 control can be summarized by the following characteristics: *a*) task level control enables to change arrays, tilers and tasks; *b*) control events are modeled as an array, which is considered similar to a data array; *c*) controls arrays only denote data dependency, hence they share the same time dimension with other arrays;

Figure 6.5 illustrates an example of the control mechanism for Gaspard2. In this example, an RCT *Task* has two input arrays and two output arrays. The *Task* can be divided into two parts: controller part and data computation part. The two input arrays are control array and data array, which are consumed by *controller task* and *repetitive tasks* (including *Task1* and *Task2*) respectively. The data part selects the right task (*Task1* and *Task2* are two exclusive tasks, which can be called a mode each) according to the computation results, called mode values, of the *controller task*. The whole RCT can be considered as a specialized Gaspard2 RCT, and it can be composed with other Gaspard2 RCTs.

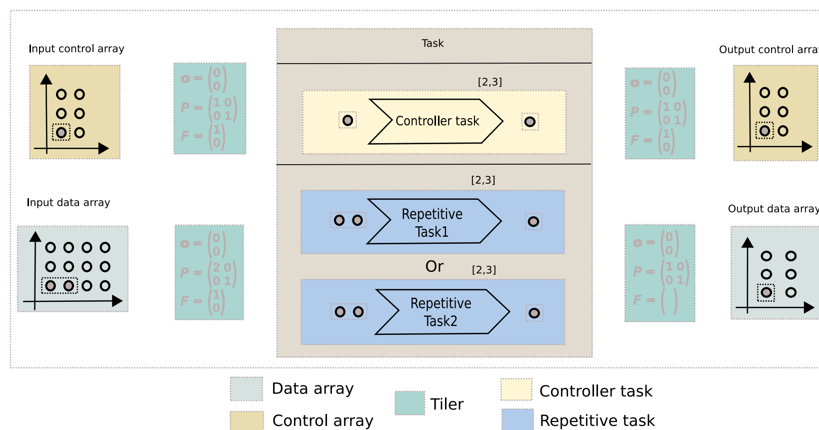


Figure 6.5: An example of the control analysis for Array-OL. The communication between the controller task and the repetitive tasks is not shown in this example.

In spite of the detailed analysis of Array-OL control, we do not aim at proposing new control concepts for Array-OL, because Array-OL is kept as a specification language. However this analysis helps to identify the appropriate control behavior for Gaspard2, and a control model for Gaspard2 is proposed accordingly.

6.1.3 Previous control proposition in Gaspard2

Based on almost the same analysis of control requirements for Array-OL, a control mechanism has been already proposed in Gaspard2 [71]. A brief description of this proposition is given in Section 3.2.3. This proposition builds basic Gaspard2 control concepts, however, it has several limits:

- parallel and hierarchical composition of automata are not defined although it is based on mode automata. These compositions are considered as basic mechanisms that enable to specify complex reactive control systems in Statecharts [58].

- the synchronization between control and computation is not well defined to guarantee a safe design. In mode automata, computations in each state share the same basic clock with events (or condition expressions) that fire the transitions. As a result, there is the coherence between the computations and events with regard to their clocks. Unlike mode automata, the independence between control and computation make it possible to have incompatible correspondence. The latter leads to an unsafe design.

- multigranularity is proposed for computations to match control. But it is difficult to handle this problem in the specification level. On one hand, only data dependency is specified in Gaspard2, thus the timing aspect of different dataflow is invisible. On the other hand, the appearances of control events are non-deterministic in general, which is not visible for the designer.

Hence, in Section 6.2, a refinement of this proposition is presented so as to satisfy the safe control requirement. Furthermore, it is extended in order to specify more complex system.

6.2 An extension proposal for control

Several basic control concepts, such as *mode switch task*, *state graphs* and *state graph task*, are presented first. Then a basic composition of these concepts, which builds the mode automata, is discussed.

6.2.1 Mode switch task and modes

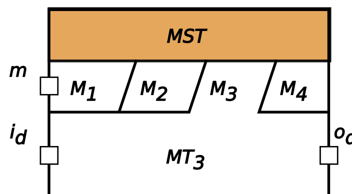


Figure 6.6: A mode switch task.

6.2.1.1 Mode switch tasks

A *mode switch task* (MST) has at least one mode and achieves a switch function that chooses one mode to execute from several alternative modes [72]. The MST in figure 6.6 (inspired from [71]) illustrates such a task through a *window* with multiple tabs and interfaces. For instance, it has m (mode value input) and i_d (data input) as inputs and o_d (data output) as output. Moreover, it is composed of several modes M_1, M_2, ..., M_n (tabs in the window). The switch between these modes is carried out according to the mode value received through m. All the inputs and outputs share the same time dimension, which ensures the correct one-to-one correspondence between all the inputs and outputs.

6.2.1.2 Modes

The modes, i.e., M_1, ..., M_n, in an MST are identified by the mode values: m_1, ..., m_n. Each mode $M_k(k \in \{1, ..., n\})$ is associated with a mode task MT_k, which can be a hierarchical task or an elementary computation task. The mode task transforms the input data i_d into the output data o_d. All the tasks $MT_k(k \in \{1, ..., n\})$ have the same interfaces (i.e., i_d and o_d). The activation of one certain mode task MT_k relies on the reception of mode value m_k by the MST through m. For any received mode value m_k, the modes run exclusively, i.e., whenever the MST executes, only the mode task MT_k associated with the received mode m_k is activated.

This modeling derives from the mode conception in mode automata [82]. The notion of exclusion among modes helps to separate different computing. As a consequence, programs are well structured and fault risk is reduced.

6.2.2 State graphs and state graph tasks

6.2.2.1 State graphs

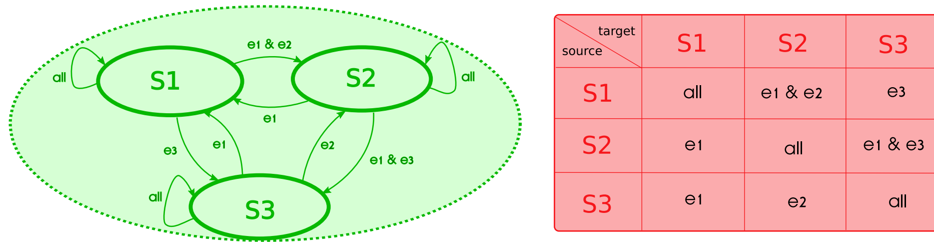
A state graph in Gaspard2 is similar to state charts [58], which is used to model the behavior using a state-based approach. It is also proposed to be a graphical representation of transition functions discussed in [45]. A state graph is composed of a set of vertices, which are called *states*. A state connects with other states through directed edges. These edges are called *Transitions*. Transitions can be conditioned by some events or Boolean expressions. A special label *all*, on a transition outgoing from state s, indicates any other events that do not satisfy the conditions on other outgoing transitions from s. Each state is associated with some mode value specifications, which provide mode values in this state.

A state graph can be specified with the help of state charts or state tables. The left-hand figure in Figure 6.7 shows a state graph similar to state charts. The right-hand figure in Figure 6.7 illustrates a state graph represented by a state-transition table, where the columns and lines represent source and target states.

Definition of state graphs

A Gaspard2 state graph is a sextuplet (Q, V_i, V_o, T, M, F) where:

- Q is the set of states defined in the state graph;
- V_i, V_o are the sets of input and output variables, respectively. V_i and V_o are disjoint, i.e., $V_i \cap V_o = \emptyset$. V is the set of all variables, i.e., $V_i \cup V_o$;



target / source	S1	S2	S3
S1	all	e1 & e2	e3
S2	e1	all	e1 & e3
S3	e1	e2	all

Figure 6.7: Different representations of a state graph.

- $T \subseteq Q \times C(V) \times Q$ is the set of transitions, labeled by *conditions* on the variables of V. The conditions C are Boolean expressions on V;
- M is the set of modes defined in the state graph;
- $F \subseteq Q \times M$ represent a set of surjective mappings between Q and M.

Compared to state charts and mode automata, initial states are not required in the state graph. In state charts or mode automata, transitions are carried out in an automatic way, i.e., the source state of one transition step is supposed to be identical to the target state of the previous transition. If a previous transition does not exist, the initial state is taken by default. Inversely a state graph only defines the transitions between states, which does not claim an automatic sequential execution. In this manner, sets of source states and events, in the form of array, can be provided to the state graph in order to get sets of target states and mode values also in the form of array. Consequently, a state graph is not an automaton. Figure 6.8 shows the input and output table of the example in Figure 6.7. This feature is proposed to adapt state graphs in the Gaspard2 repetition context. However, automata can still be built with the help of certain Gaspard2 operators, which are presented later.

Provided source state(s)	Provided event	Resulting target state(s)
S1	e1 & e2	S2
S2	e2	S2
S2	e1 & e3	S3
[S1, S2]	((e1 & e2), e1 & e3))	[S2, S3]

Figure 6.8: A table representing the *mapping* of a state graph. The last line shows "parallel transitions", where multiple source states are provided. Each of these states is associated with a trigger, and a target state is entered accordingly.

Parallel composition of state graphs

A set of state graphs (Figure 6.9) can be composed together. The composition of state graphs in this manner is considered as *parallel composition*. Figure 6.9 illustrates a parallel

composition example. The two state graphs are placed in the same ellipse, but separated by a dash line.

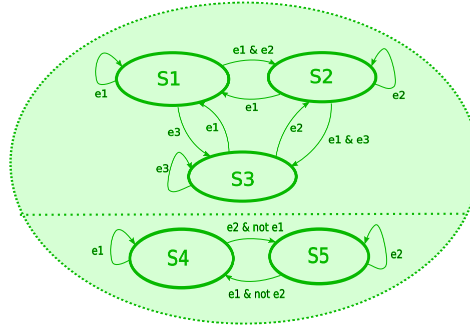


Figure 6.9: An example of the parallel composition of state graphs.

Two state graphs G^1: $= (Q^1, V_i^1, V_o^1, T^1, M^1, F^1)$ and G^2: $= (Q^2, V_i^2, V_o^2, T^2, M^2, F^2)$ are considered here to illustrate the composition. The Composition operator is noted as: ||. The parallel composition is defined as:

$$(Q^1, V_i^1, V_o^1, T^1, M^1, F^1) \parallel (Q^2, V_i^2, V_o^2, T^2, M^2, F^2)$$
$$= ((Q^1 \times Q^2), (V_i^1 \cup V_i^2) \backslash (V_o^1 \cup V_o^2), (V_o^1 \cup V_o^2), T, (M^1 \times M^2), F),$$

where

$$F = \{((q^1, q^2), (m^1, m^2)) \mid (q^1, m^1) \in F^1 \wedge (q^2, m^2) \in F^2 \},$$

and

$$T = \{((q^1, q^2), \mathrm{C}, (q^{1\prime}, q^{2\prime})) \mid (q^1, C^1, q^{1\prime}) \in T^1 \wedge (q^2, C^2, q^{2\prime}) \in T^2 \},$$

where C is a new expression on C^1 and C^2: $C = C^1$ *and* C^2, and $(V_i^1 \cup V_i^2) \backslash (V_o^1 \cup V_o^2)$ implies any output variable is not considered as an input variable in the composed graphs, hence it should be removed from the input variable set.

These state graphs in a parallel composition can be triggered to carry out transitions at the same time, upon the presence of the source states and events of both two state graphs. Moreover, the number of transitions fired is supposed to be always the same for these state graphs, i.e., the inputs of these state graphs have the same size.

Hierarchical composition of state graphs

Another interesting composition of state graphs is their hierarchical composition. A state in the state graph A can be refined by another state graph B, where B is considered as sub graph of A. Consider a state graph G: $= (Q, V_i, V_o, T, M, F)$ where $Q = \{q_0, q_1, ..., q_n\}$ and a corresponding set of refining automata $\{G^k\}_{k \in [0,n]}$, where $G_k = (Q^k, V_i^k, V_o^k, T^k, M^k, F^k)$. The composition can be defined (inspired from [84]) as:

$$G \triangleright \{G^k\}_{k \in [0,n]} = (Q', V_i', V_o', T', M', F')$$

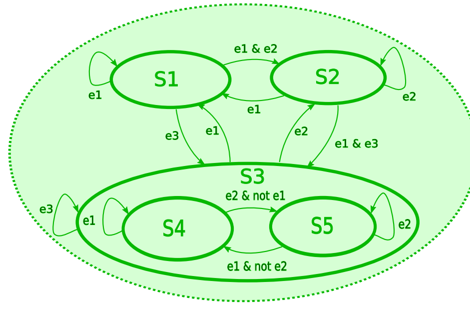


Figure 6.10: An example of hierarchical composition of state graphs.

where

$$Q' = Q \triangleright \{Q^k\}_{k\in[0,n]} = \bigcup_{k=0}^{n}\{q^k \triangleright q_j^k \mid j \in [o, n^k]\} \quad ,$$

$$V'_o = V_o \cup \bigcup_{k=0}^{n} V_o^k \quad ,$$

$$V'_i = (V_i \cup \bigcup_{k=0}^{n} V_i^k) \backslash V'_o \quad ,$$

$$T' = \{(q^k \triangleright q_{j_1}^k, C, q^{d\prime} \mid (q^k, C, q^d) \in T \wedge j_1 \in [0, n^k]\}$$
$$\cup\{(q^k \triangleright q_{j_1}^k, (C \wedge \neg \bigvee_{(q^k,C_m,q^d)\in T} C_m), q^k \triangleright q_{j_2}^k) \mid (q_{j_1}^k, C, q_{j_2}^k) \in T^k \wedge j_2 \in [0, n^k]\} \quad ,$$

$$M' = M \cup \bigcup_{k=0}^{n} M^k \quad ,$$

and

$$F' = \{((q, q^1, .., q^n), (m, m^1, ..., m^n)) \mid (q, m) \in F \wedge (q^1, m^1) \in F^1 \wedge ... \wedge (q^n, m^n) \in F^n \} \quad ,$$

$q^k \triangleright q_j^k$ denotes the state q^k is refined by the state q_j^k. T' has two kinds of transitions: $\{(q^k \triangleright q_{j_1}^k, C, q^d \mid (q^k, C, q^d) \in T \wedge j_1 \in [0, n^k]\}$ implies the transitions of G, and q^k and q^d denotes the source and target state respectively. As q^k is refined by states of G^k, $q^k \triangleright q_{j_1}^k$ is used instead of q^k. The target state is also a refined state of q^d, however, which state is entered is decided at run time. Hence, $q^{d\prime}$ is used instead. $\{(q^k \triangleright q_{j_1}^k, (C \wedge \neg \bigvee_{(q^k,C_m,q^d)\in T} C_m), q^k \triangleright q_{j_2}^k) \mid (q_{j_1}^k, C, q_{j_2}^k) \in T^k \wedge j_2 \in [0, n^k]\}$ denotes the transitions in G^k, i.e., these transitions are fired when no transitions of G^k are fired. This condition is expressed by: $\neg \bigvee_{(q^k,C_m,q^d)\in T} C_m$. $q_{j_1}^k$ and $q_{j_2}^k$ denote the source and target states of a transition in G^k. Figure 6.10 shows an example of the hierarchical composition of state graphs. In this example, the state $S3$ is refined by the state graph composed of states $S4$ and $S5$, denoted by $S3 \triangleright S4$ and $S3 \triangleright S5$.

Properties of state graphs

Determinism is required for the definition of state graphs, i.e., for each state in the graph, the input events can only satisfy the trigger of one transition. The determinism help to define unambiguous behavior, as a result, it reduces the fault risks.

Multi mode values can be defined in a single state if these mode values belong to different mode value sets, or they are used for different purpose, e.g., an SGT is used for the control of several MSTs, which uses different mode sets.

6.2.2.2 State graph tasks

A *state graph task* (SGT) (Figure 6.11) is a Gaspard2 task that is associated with state graphs. It achieves the functionality of mode value definition according to its associated state graphs. The mode values serve to activate different computations in MSTs. Hence, SGTs are ideal complements of MSTs, which are connected through mode values, i.e., An SGT acts as a controller and a MST plays a computation switch role according to the controller.

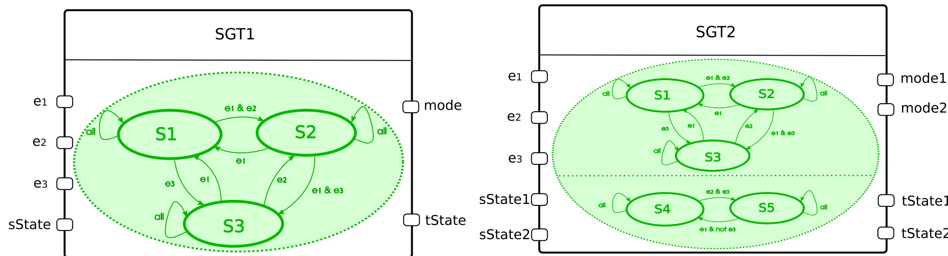


Figure 6.11: Examples of SGTs.

Similar to other Gaspard2 tasks, an SGT also defines its interfaces, which include the event inputs from its environment, source state inputs, target state outputs and mode outputs. The event inputs are used to trigger the transitions. The source state inputs indicate from which states transitions are triggered. Similarly, target state outputs are the target states of fired transitions. The mode outputs are associated with an MST so as to choose the mode to execute. In some cases, the output mode value could be different from the output state because of different mode definitions, e.g., the output state always indicates the target state of the fired transition, whereas the mode value can be that of the source or target state according to the mode definition.

6.2.3 Task compositions of SGT and MST in Gaspard2

Based on the previous presented SGTs and MSTs, different compositions of these Gaspard2 tasks are presented in this section.

6.2.3.1 A macro structure

Once the MST and the SGT are introduced, a *MACRO* can be used to compose them together. The macro task in Figure 6.12 illustrates one possible composition, which will be used as a basic composition. In this macro, the SGT produces a mode value (or a set of mode values) and sends it (them) to the mode task. The latter switches the modes accordingly.

6.2.3.2 Parallel composition of SGTs

In addition to the parallel composition of state graphs presented previously, the parallel composition can be also applied to a set of SGTs. An example is illustrated in Figure 6.13.

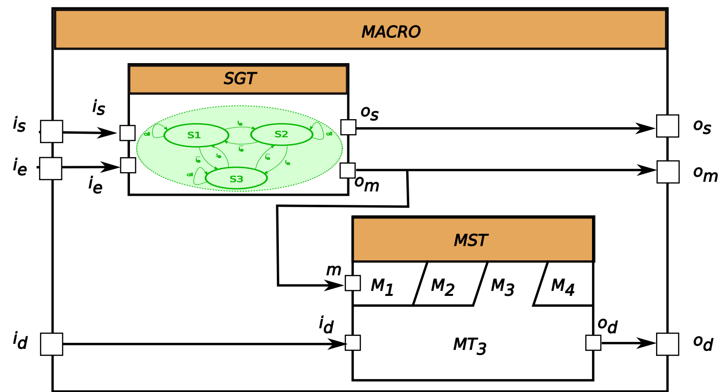


Figure 6.12: An example of a macro structure. The SGT is a controller that decides the right mode to run. According to SGT, MST selects the right mode task to run.

Because the SGTs are considered as normal Gaspard2 tasks, the resulting task (e.g., the *SGT* in the example) composed of several SGTs can be also considered as a normal Gaspard2 task. Thus it can be composed with other Gaspard2 tasks.

Parallel composition in different contexts

The interfaces of an SGT determine how many transitions should be fired for this SGT, e.g., eight inputs fired eight transitions, hence produce eight outputs. In this example, the repetition of transition is implicit. According to the number of transitions that are fired according to the interfaces of an SGT, several contexts are distinguished when several SGT are composed together: *compatible context* and *incompatible context*. The first one indicates all the SGTs share the same number of fired transitions. Otherwise, they are considered to be incompatible. The differences of the composition in these two contexts are:

- parallel composition in a compatible context makes it possible to have the same transition rate (or number), within one reaction of the task, for all the state graphs in the composition, thus they can be translated into the parallel composition of automata under certain conditions (e.g., inter-repetition dependency) presented in the next section. In this context, the *SGT* in the Figure 6.13 is semantically equal to a task that is associated with the state graphs in Figure 6.9.
- parallel composition in an incompatible context cannot be translated into parallel composition of automata because of their different transition rate. Although this composition is possible in STATEMATE semantics of state charts and SyncCharts, is not possible for mode automata. Consequently, this composition is only considered as a normal composition of Gaspard2 tasks, which can not be translated into automata.

Parallel composition with other Gaspard2 tasks

These SGTs can also be composed with standard Gaspard2 tasks, which can handle the control out of capacity of state graphs, e.g., some binary operations on events or conditions on numbers.

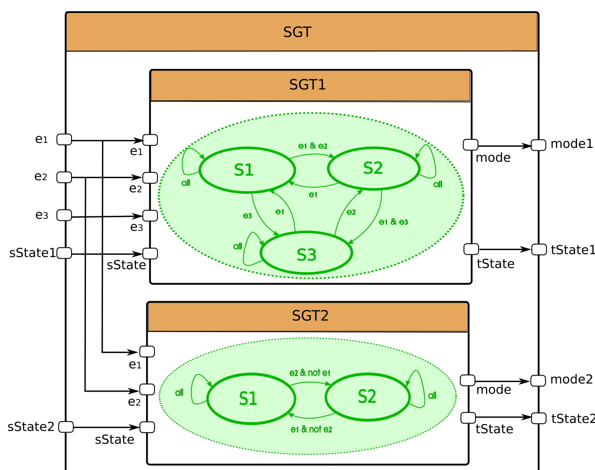


Figure 6.13: An example of parallel composition of SGTs.

6.2.3.3 Hierarchical composition

Hierarchical composition of state graphs using SGTs and MSTs can be also achieved. In this case, an SGT acts as a mode in an MST, which is only activated in this mode. The hierarchical composition presented in this section involves only the composition of SGTs via the MST. The MST defines the relation between an SGT and other SGTs, where state graphs in the latter act as refinements of state graph states in the former. This composition structure defines a two level hierarchy of SGTs with regard to their associated state graphs. Certainly, a nested definition can be applied in specification of the hierarchy.

An example is illustrated in Figure 6.15. The state graph associated with SGT_1 has three states: S_1, S_2 and S_3, which correspond to three modes M_1, M_2 and M_3 respectively. In the MST, the mode task of M_3 is SGT_2, i.e., SGT_2 is activated only when S_3 is active. Hence the state graph of SGT_2 is considered as a refinement (sub state graph) of S_3.

Similar to the parallel composition case, here we are interested in the composition in a compatible context. Moreover, the repetition context is also considered in the hierarchical composition in a finer way, i.e., two repetition contexts are taken into account: a *simple repetition context* and a *hierarchical repetition context*. The simple repetition context indicates the elements in this context are simply repeated with the same number. A repetition task defines a simple repetitive context with regard to its internal repeated tasks. The hierarchical repetition context defines repetitions in a nested way. For instance, a repetition context task T_1 (Figure 6.14) defines a repetition context C_1 with regard to its internal repeated task T_2. Besides, T_2 is itself a repetition context task that defines a repetition context C_2 with regard to its internal repeated task T_3. Thus T_1 and T_2 define a hierarchical repetition context $C_1 \times C_2$ to T_3. The right-hand figure in Figure 6.14 illustrates the repetition traces of T_1, T_2 and T_3 in case that C_1 and C_2 are equal to 2.

Hierarchical composition in a simple repetition context. A typical composition is an SGT and its associated MST are specified in the same repetition context, which leads to the fact that all the SGTs are specified in the same repetition context. It helps to ensure that the state graph share the same transition number for one reaction. The example in Figure 6.15

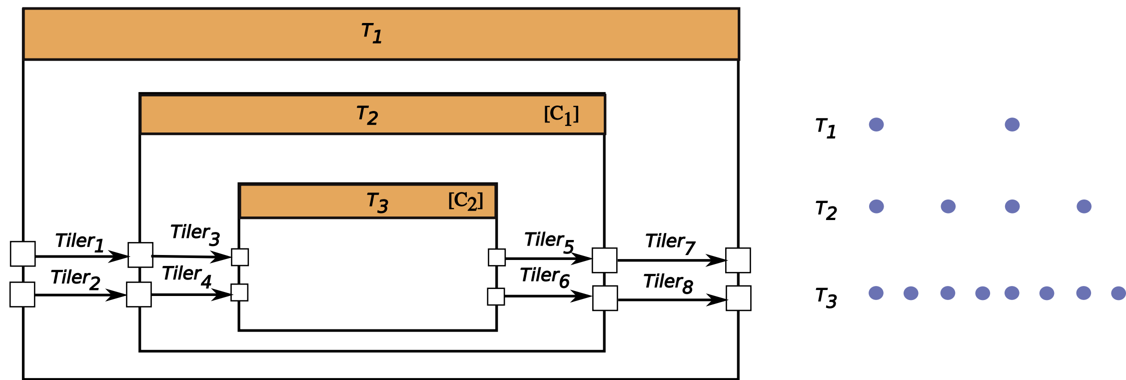


Figure 6.14: An example of hierarchical composition of SGTs in the same repetition context.

illustrates this kind of composition. The state graph of SGT_2 acts as a refinement of the state S_3. SGT_1 and SGT_2 share the same repetition context provided by RT. This example is similar to the example illustrated in Figure 6.10, except that the composition is carried out in a component composition way.

Hierarchical composition in a hierarchical repetition context. Apart from the previous definition in the simple repetition context, another interesting hierarchical composition is SGTs are defined in a hierarchical repetition context, i.e., the low level state graphs have a regularly more fast transition rate than the high level ones.

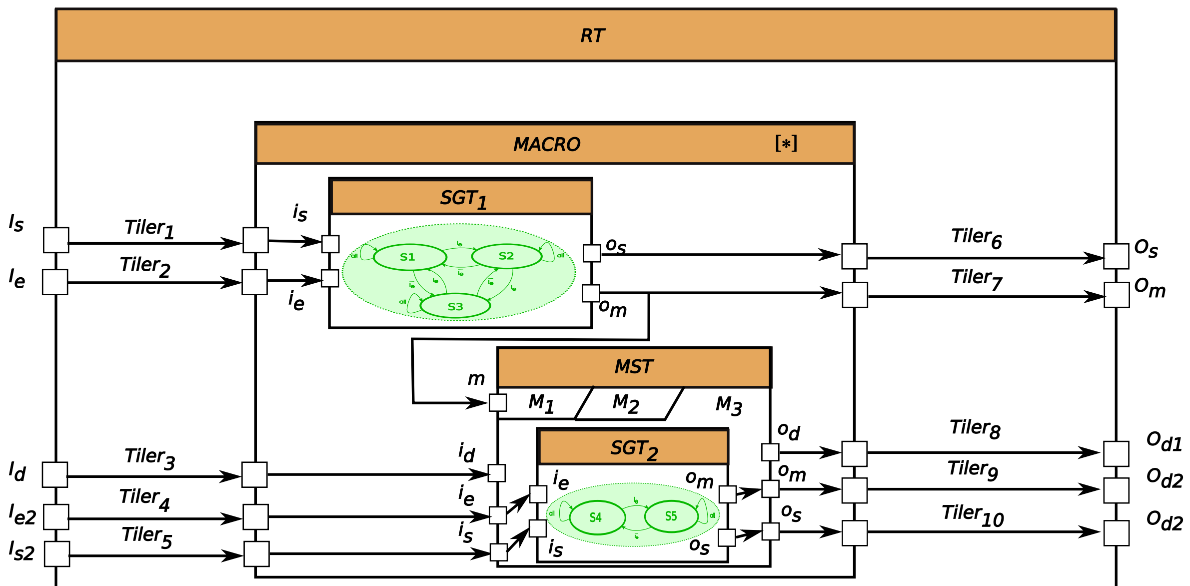


Figure 6.15: An example of hierarchical composition of SGTs in the same repetition context.

6.2.3.4 Conclusions

Inspired from mode automata and control-computation separation, we define the basic control mechanism based on state graph and mode switch. In order to be compatible and utilizable in Gaspard2, the SGT and MST are also defined. SGTs and MSTs can then be used as standard tasks in Gaspard2. These concepts provide the possibility for behavioral specification with necessary expressivity and compatibility with regard to Gaspard2. However, their semantics should be more precise for the safe design of DIP applications, with regard to their semantics and verifiability. The next Section 6.3 presents a dependable but not the unique usage of these control concepts in a synchronous reactive dataflow context, which enhances these concepts with formal semantics and also verifiability, which favor the safe design.

6.3 Reactive control extension in Gaspard2

Based on the previous mentioned control constructs, the reactive control is presented in this section. This control is based on mode automata [84], which aims at conferring safe design properties onto these control constructs. Section 6.3.1 discusses some issues of the Gaspard2 control used in a synchronous dataflow context. The synchronous reactive control is presented in Section 6.3.2.

6.3.1 Issues of Gaspard2 control specification in a dataflow context

The previous Section 6.1.2.1 illustrates the relation between the control and the data computation. In order to be compatible with Gaspard2 specifications, the control is modeled as an array (Section 6.1.2.2), which can be mapped onto the same time dimension as other data arrays do (Section 6.1.2.3), hence this time dimension is common to all arrays in the specification. However, an issue emerges when Gaspard2 specifications are mapped onto an execution model, e.g., a dataflow model. This issue involves the relation of control event and dataflow, which are always ambiguous in the specification. This issue does not come from the previous control proposition, but from the specification and implementation gap of Gaspard2 behavior.

When Gaspard2 specifications are mapped onto a dataflow model, the relation between control event and data specified in Section 6.1.2.3 implies that the two flows (event and data) have the same clock or they can be synchronized. However, this is only an ideal case, as Gaspard2 is not associated with any execution model. In an execution model, a control event may be uncorrelated with the repetitions of data computation with regard to their clocks. For instance, a user takes a remote control for a television. His commands of channel change do not correlate with the TV dataflow. This problem is caused by different specification styles of control and data computation, i.e., state-based control adopts event-driven style [58, 60] and data computation adopts dataflow style [19].

Although this problem is not discussed here, in order to avoid this problem in our synchronous model, we propose to use a classical synchronous dataflow control mechanism, which includes *synchronization* and *memory*. This control mechanism is illustrated in Figure 6.16. It can be achieved by using a clock mechanism, for instance, an equation in Signal as follows:

```
Df3 = (Df2 cell ^Df1) when ^Df1
```

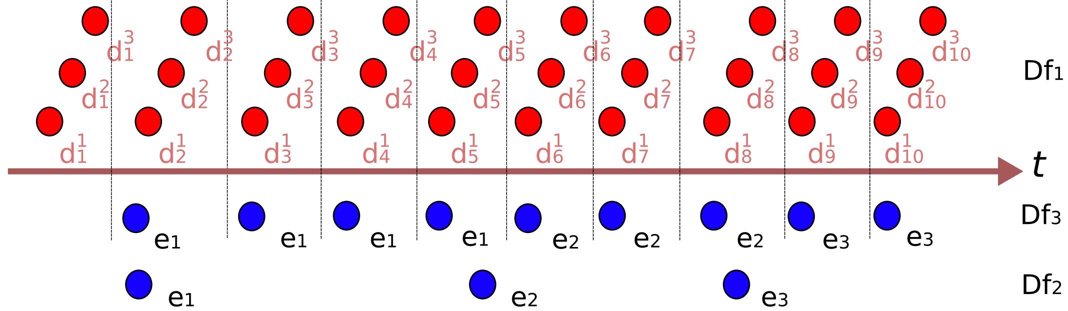


Figure 6.16: Memory is used for the control.

Based on this control mechanism, control event, from now on, is considered to be a flow synchronized to other dataflow in the synchronous model. Hence in Gaspard2, control and data computation can be specified to share the same time dimension. This result has been used for the illustration of Gaspard2 control in Section 6.1.2.

6.3.2 Reactive automata based control mechanism in Gaspard2

The basic concepts of the state graph based control in Gaspard2 have been presented in the previous section, but its semantics is not clear, we propose to integrate mode automata semantics in the Gaspard2 state graphs (GSG) under certain conditions, so as to enable safe properties and enhance its correctness verifiability. As mode automata are proposed in the synchronous reactive dataflow context, additional constructs (inter-repetition dependency and default link) are used in Gaspard2 to build Gaspard2 mode automata (GMA).

6.3.2.1 From Gaspard2 state graphs to Gaspard2 mode automata

The first problem of the construction of GMA is the incompatibility between Array-OL parallel semantics and automaton semantics in term of sequential traces. Even when an Array-OL specification is mapped on a time model, the problem cannot be completely addressed, because not always all space dimensions can be mapped onto the time dimension. Hence two kinds of execution are discriminated here: parallel execution and sequential execution according to the context:

Parallel execution. Acting as a controller, an SGT can be put in a repetition context, where the SGT is supposed to be repeated in parallel. Each repetition of the SGT is independent from other repetitions as the inputs required by each repetition of the SGT are provided at the same time (since they are in a parallel context). Hence, the SGT and its associated state graphs here serve as a simple `case` statements that has no memory of previous status. From the point of view of an automaton, these state graphs are automata that have only one transition step in their life cycles. These parallel automata are not addressed in this thesis.

Serialized execution. Compared to the parallel execution, an SGT also adopts the serialized execution, where a special dependency between the repetitions of the SGT. This dependency, called *inter-repetition dependency*, serializes the repetitions and conveys values between these repetitions (e.g., states). With the help of *inter-repetition dependency*, it is possible to establish GMA from GSGs, which requires two steps:

- Firstly, the structure of GMA is presented by a composition of a SGT and an MST, i.e., the $MACRO$ in Figure 6.12. The SGT in this macro structure acts as a state-based controller and the MST achieves the mode switch function. Compared to synchronous mode automata (SMA), where computations are set in the states, the computations in GMA is placed in the MST, i.e., outside of the GMA states.
- Secondly, IRD specifications should be specified for this macro structure when it is placed in a repetition context. The reasons are twofold: the macro structure represents only one map from source state on target state, whereas an SMA has continuous transitions that form an execution trace. Hence the macro should be repeated to have multiple transitions. An IRD forces the continuous sequential execution, which makes it similar to the execution of the SMA. Consequently, the repetition context is transformed into a serialized one. In this manner, the mode automata can be built and executable.

6.3.2.2 Synchronous mode automata modeling in Gaspard2

With the previous presented constructs, the modeling of synchronous mode automata in Gaspard2 is demonstrated here. The modeling result is the Gaspard2 mode automata, which can be eventually translated into synchronous mode automata. This modeling is illustrated with an example in Figure 6.17, where how to assemble these constructs are presented. The main idea is to place the $MACRO$ in a repetitive context, where each of its repetitions models one transition of mode automata. An IRD links the repetitions of $MACRO$ and conveys the current state (sends the target state of one repetition as the source state to the next repetition) between these repetitions. The states and transitions of the automata are capsulated in the SGT. The data computations inside the mode are set in the mode tasks. SGT and its mode tasks share the same repetition space, so they run at the same rate or clock. The detailed formal semantics of GMA can be found in [45].

6.4 Typical examples

Two examples are illustrated in this section. The first one is a typical example found in mode automata, which shows the construction of a complex mode automata, particular in the control part. The second example concerns the video color style processing on mobile multimedia devices. In this example, the construction of the Gaspard2 mode automata is shown.

6.4.1 A typical example of a counter

The example in Figure 6.18 is a typical example of mode automata, the computation result with regard to variable X is illustrated in the following table:

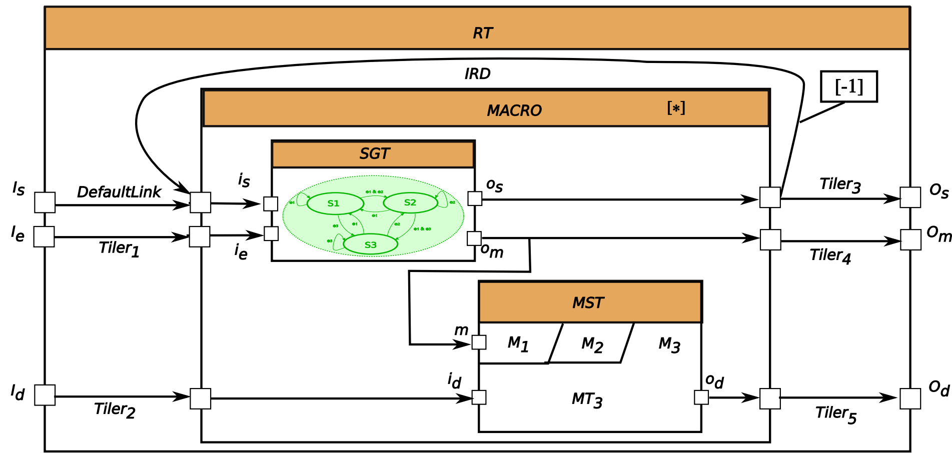


Figure 6.17: An example of the macro structure in a repetition context.

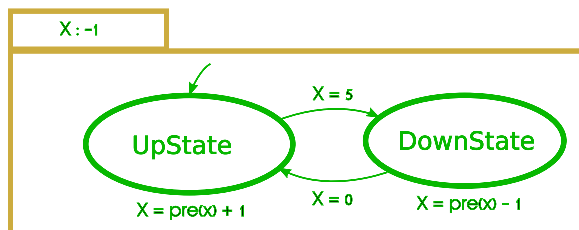


Figure 6.18: A typical example of mode automata.

instant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
X	0	1	2	3	4	5	4	3	2	1	0	1	2	3	4	5

The corresponding GMA to the example in Figure 6.18 is illustrated in Figure 6.19. Compared to previous examples, this example makes several distinctions:

- certain data calculated in the data computation, e.g., X, is used in the control part, which in turn selects the right mode for the computation. Apparently, there is a dependency cycle in the specification. However, this cycle is specified by the IRD_x, which avoids the causality problem. It is similar to the Lustre, where a *pre* operator helps to avoid the same problem.

- an IRD, i.e., IRD_x in the example, is also specified for the data computation. Whereas in the SMA case, a global variable, namely X, is used instead.

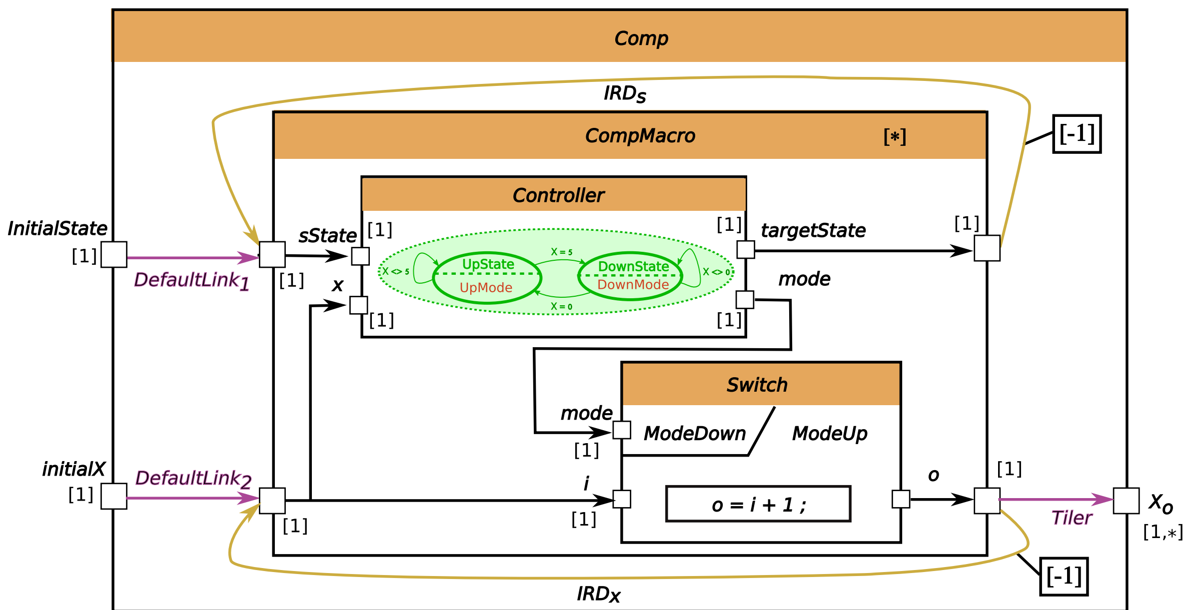


Figure 6.19: The Gaspard2 mode automata corresponding to the mode automata in Figure 6.18. This example is only used to illustrate how to use the control concepts without data-paralllel specifications.

6.4.2 A control example for cell phone video effect

Figure 6.20 illustrates an example of color style processing module (*ColorStyleProcessing*) used in a multimedia cellular phone. This module is used to manage the color style of a video clip. It provides two possible styles: *color* or *monochrome*, which are implemented by *MonochromeFilter* and *ColorFilter* respectively. These two filters are elementary tasks in this modeling, which should be deployed with IPs. The changes between these two filters are achieved by *ColorStyleSwitch* upon receiving mode values through its mode port *colorMode*. The mode values are determined by *ColorStyleController*, whose behavior is demonstrated by its associated state graph.

The *ColorStyleFilters* is a *MACRO*, which is composed of *ColorStyleController* and *ColorStyleSwitch*. *ColorStyleFilters* illustrates the processing of one frame of the video clip, which should be repeated. In the example *ColorStyleProcessing* provides the repetition context for *ColorStyleFilters*. The asterisk specified in *ColorStyleFilters* indicates that *ColorStyleFilters* will be repeated for infinite times, i.e., it can be seen as a temporal dimension. In addition, shapes of all the ports of *ColorStyleProcessing*, except *InitialState*, are added with a temporal dimension, indicated by an asterisk. An *IRD* is also defined, which connects the target states output (*targetState*) to the source state input (*sState*) of *ColorStyleController*. A vector associated with the *IRD* indicates the dependency relation between repetitions, e.g., *[-1]* in the example signifies the source state of one *ColorStyleController* repetition relies on the target state of its previous *ColorStyleController* repetition.

The whole *ColorStyleProcessing* can be translated by a synchronous mode automata finally.

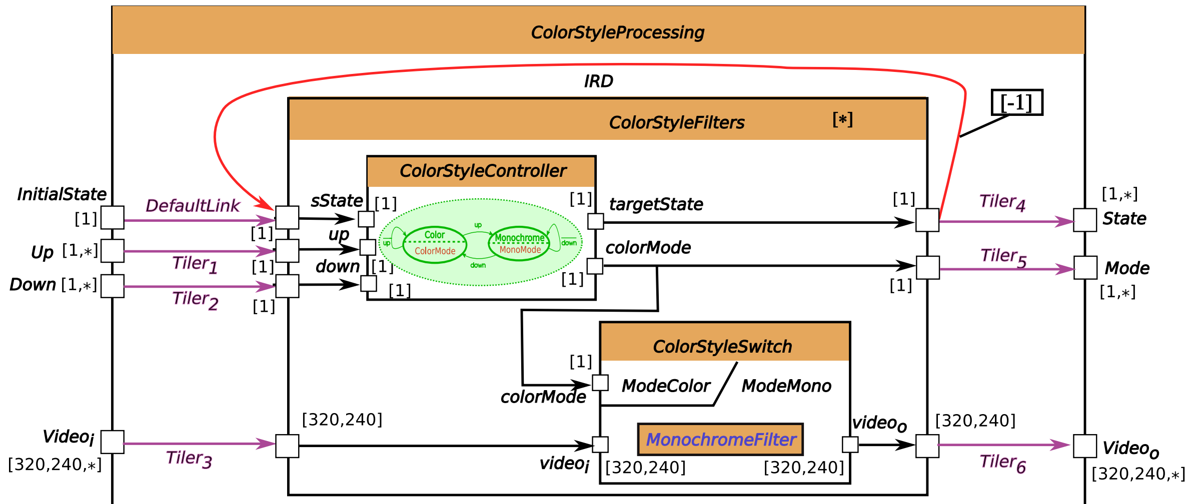


Figure 6.20: An example of color style filter in a multimedia phone modeled with the Gaspard2 mode automata. *MonochromeFilter* inside *ColorStyleSwitch* is considered to be an elementary task, which is deployed with an IP to fulfill the filter function.

6.5 Conclusions

This chapter presents the control extension of Gaspard2. This control mechanism is first discussed according to the requirements of control in DIP applications in terms of Gaspard2 model. This Gaspard2 model is considered to be a high-level model and independent from any execution models. The change of tiler, task, array and repetition space is discussed separately.

Based on the previous discussions, we present an extension proposal for the Gaspard2 control, which include state graphs, SGTs and MSTs. This extension has several advantages: formal semantics of state graphs is given (formal semantics of tasks is presented in [45]), parallel and hierarchical composition is formally defined. As the control is introduced in Gaspard2 without consideration of any execution model, it is generic and high-level. Consequently, it can be translated into or projected onto different execution models.

The extended control is then discussed in the context of synchronous reactive systems, where the synchronous execution model is involved. The problem of synchronization of control flow and dataflow is illustrated and discussed with examples. As the synchronous execution model is chosen, not all state graphs of Gaspard2 can be translated into synchronous automata, particularly the state graphs that enable parallel execution. Two typical examples of the control application, i.e., a counter and a color style filter, are shown finally.

This chapter concentrates on the modeling problems of the Gaspard2 control, it does not involve the transformation of the control into synchronous automata and the graphical implementation of the control. The transformation and graphical implementation of the control are presented in Chapter 8.

Part III

Integration into an MDE framework and case study

Chapter 7

Transformation from Gaspard2 to synchronous languages

7.1 The Gaspard metamodel and profile . . . 116
7.2 Synchronous equational metamodel . . . 117
7.2.1 Common aspects . . . 117
7.2.2 From abstract syntax to synchronous metamodel . . . 118
7.2.3 Signal . . . 118
7.2.4 Equation . . . 119
7.2.5 Node . . . 120
7.2.6 Module . . . 121
7.2.7 IP deployment . . . 121
7.3 The transformation rules . . . 122
7.3.1 From Gaspard2 models to synchronous models . . . 122
7.3.2 Transformation tools . . . 128
7.3.3 Template-based code generation and code generators . . . 128
7.3.4 The synchronous transformation chain . . . 128
7.4 Conclusions . . . 130

This chapter focuses on the transformation from UML-based Gaspard2 model towards synchronous executable code [135, 134]. Unlike in the previous chapter 5 where only the abstract transformation from an Array-OL model to a synchronous equational model is presented, this chapter presents a concrete transformation implementation based on the MDE approach and tools. This transformation relies on two metamodels: Gaspard2 metamodel and synchronous metamodel. These two metamodels are implemented with UML graphical modeling tools, which is called MagicDraw [1]. The Gaspard2 metamodel is detailed in the appendix A. However, the example in this chapter for the illustration of the transformation is modeled utilizing the Gaspard2 profile [10], which has only minor difference from the Gaspard2 metamodel. In order to be concise, the profile is not presented here. The synchronous metamodel is equally detailed in this chapter, which is considered as one of the contributions of this thesis. As the synchronous model is only an intermediate model, so

[1]http://www.magicdraw.com/

no profile is needed here. Model transformation is then detailed in terms of transformation rules. In summary, the modeling and model transformation are carried out in the framework of MDE.

Users specify their applications in a graphical UML modeling tool with the Gaspard2 concepts, and then the UML graphical specifications are transformed into Gaspard2 models, from which the model transformation towards synchronous model is carried out. Finally the synchronous model is used to generate different executable code in Lustre and Signal.

7.1 The Gaspard metamodel and profile

As the Gaspard2 metamodel has been briefly presented in chapter 3 and detailed in appendix A, it will not be introduced here again. The Gaspard2 UML profile is a minor modification of Gaspard2 metamodel dedicated to specify the applications with the help of UML graphical modeling tools. The concepts appeared in the profile and in the metamodel are almost similar, so here only an example is given to illustrate how to use the profiles (more details about the metamodel and profile can be found in Section 3.2.1.1 and 3.2.1.2).

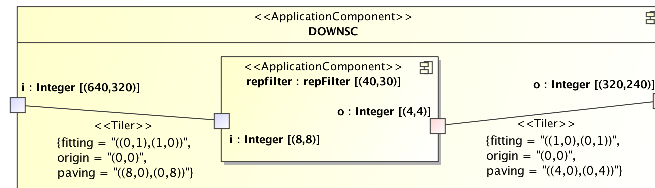


Figure 7.1: An example of downscaler.

An example of a Gaspard2 application An example, called *Downscaler*, which is always found in video or image processing, is illustrated. However, only main part of this example is shown here. The example is a downscale application that downscales from a flow of images of size [640, 480] into a a flow of images of size [320, 240] (Figure 7.1), which is the one of the standard sizes of video displayed on cell phones. A downscaling filter is supposed to work on [8, 8]-array, so it will be repeated (80×60) times for the processing of one image.

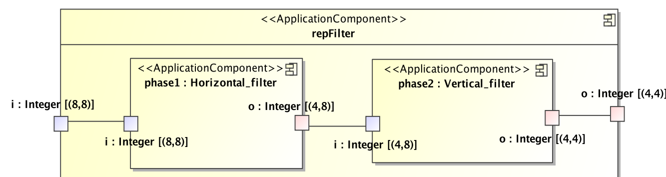


Figure 7.2: The compound task of *repFilter*.

The repeated filter can be decomposed into two filters (Figure 7.2): *Vertical_Filter* and *Horizontal_Filter*. The first one is the processing on columns of an image, and the second one concerns the lines.

Figure 7.3 illustrates how lines are processed, where eight lines are processed by *HFilter* from a tile of size [8,8]. The *HFilter* is an elementary task that should be deployed, as in

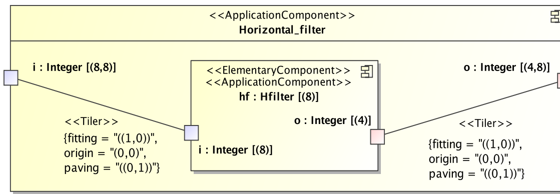


Figure 7.3: A detailed view on the horizontal filter.

Figure 7.4. This deployment gives the implementation code of the filters. More details can be found in chapter 3 and appendix A.

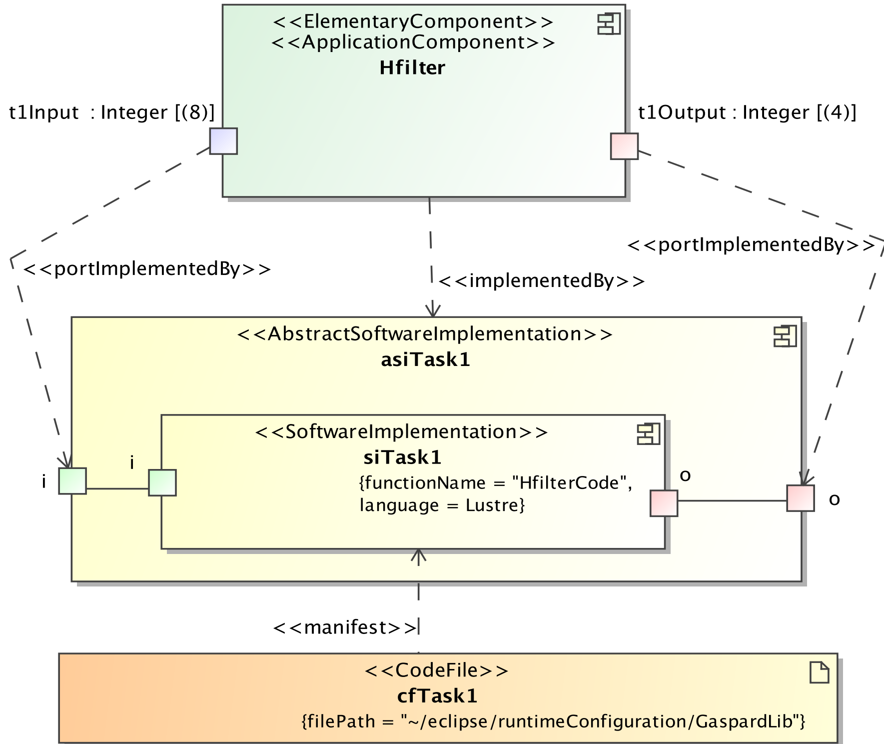


Figure 7.4: The deployment of the elementary task of *HFilter*.

7.2 Synchronous equational metamodel

7.2.1 Common aspects

The metamodel proposed here aims at three synchronous data-flow languages at the same time. These languages have considerable common aspects, which enable their code generation with the help of only one metamodel. In addition, because of the obvious differences between Gaspard2 and synchronous languages, an intermediate model is necessary to bridge the gap between them as well. A synchronous model is therefore proposed, which follows the synchronous modeling of DIP applications in chapter 5. It aims to be generic enough

to target the synchronous data-flow languages mentioned earlier and to be adequate to express data-parallel applications. So, it is not intended to have exactly the same expressivity as these languages. But this is not the case of the SIGNALMETA metamodel [22], which is specifically dedicated to the Signal language. This metamodel completely defines all programming concepts of Signal. It has been specified in the *Generic Modeling Environment* (GME), developed at Vanderbilt University.

7.2.2 From abstract syntax to synchronous metamodel

The abstract syntax presented in chapter 5 illustrates the relations between the synchronous notions in a concise way. But the metamodel illustrated here will give a more concrete definition. The metamodel is presented in several parts. Each part involves a key synchronous concept. But all these parts are not necessarily separated. First of all, `Signals` is presented, and then `Equation`, `NodeModule` are presented sequentially.

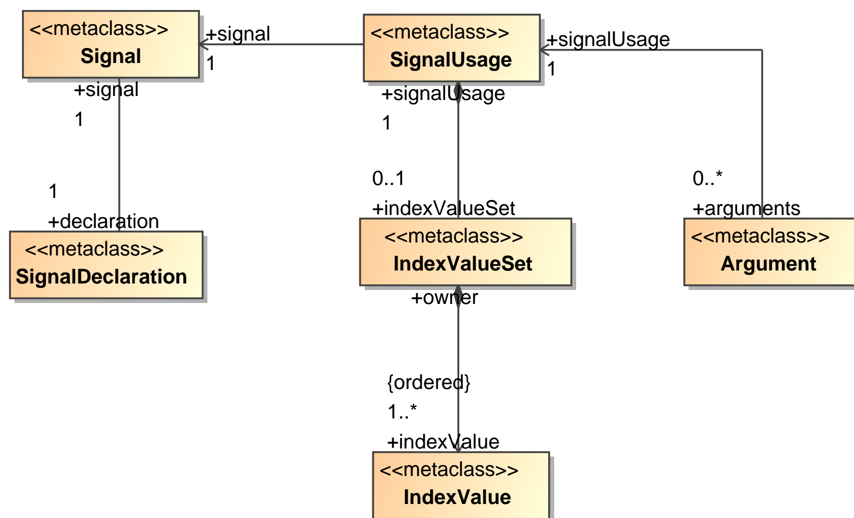


Figure 7.5: Extract of the synchronous metamodel: `Signal`.

7.2.3 Signal

According to synchronous languages, all input, output or local variables are called `Signals` (see Figure 7.5). Each `Signal` can be typed by *Boolean*, *Integer*, *Real*, etc.. These `Signals` are single-valued `Signals`. Complex `Signals` can be defined on these basic `Signals`, such as array `Signals`. An array `Signal` are defined with a shape, which indicates the size of the array. The shape of a signal is defined as its attribute.

Signal declaration. A `Signal` must be declared before using it. It is modelled by `SignalDeclaration`, which declares the name and type of the `Signal`.. But a `Signal` can be declared in different ways, such as a local declaration for a local signal and an interface declaration for an input/output signal. These aspects are captured by `SignalLocalDec` and `SignalInterDec`.

Signal usage. Once a Signal is declared, it can be used (or referenced). One SignalUsage represents one operation (read/write) on the Signal. If the Signal is an array, a SignalUsage can be an operation on a segment of this array. Hence, if this array can be divided into several segments (more precisely, it is called tiles in Gaspard2), the Signal is then associated to the same number of SignalUsage correspondingly. Each of these SignalUsages has an IndexValueSet, which is a set of IndexValue of the associated Signal. The associated IndexValues of a SignalUsages indicate the scope of one operation on a Signal.

Signals are used in both sides of equations, which are taken as equation arguments. Hence a SignalUsage is associated with at least one Argument (to be explained later) of equations.

7.2.4 Equation

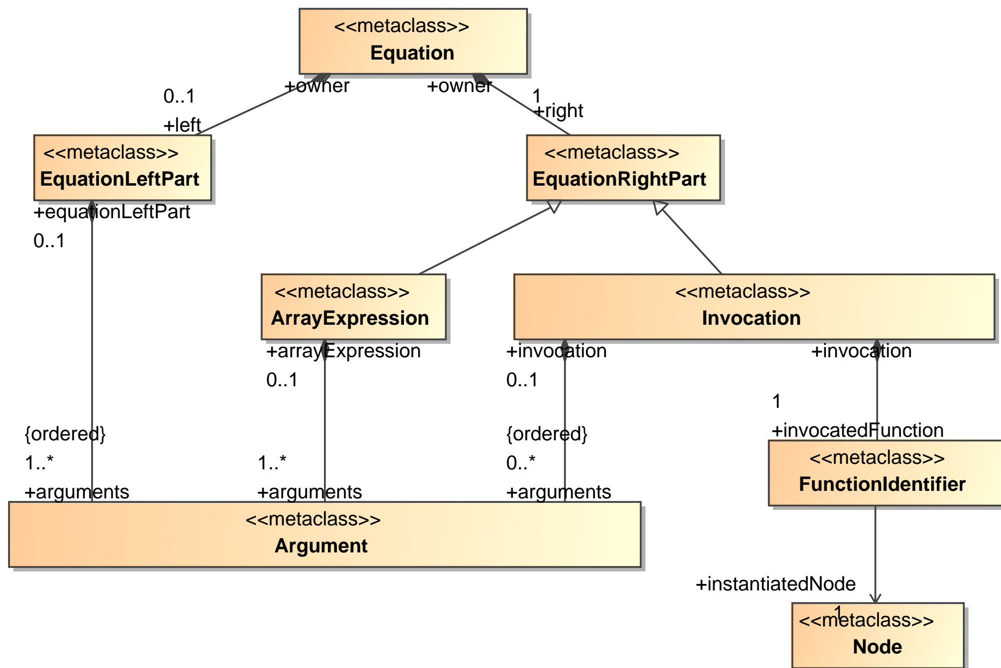


Figure 7.6: Extract of the synchronous metamodel: Equation.

Equations (Figure 7.6) indicate relations between signals. These signals used in an equation are considered as Arguments of the equation. But Signals and Arguments do not have a direct relation, SignalUsages play a intermediate role.

Structure of an equation. An Equation has an EquationRightPart and at most one EquationLeftPart. The latter has directly Arguments as Equation outputs. EquationRightPart is either an ArrayAssignment or an Invocation. ArrayAssignment has Arguments, and indicates that the Equation is an array assignment. Invocation is a call to another Node (see section 7.2.5). In an Invocation,

FunctionIdentifier indicates the called function. An Invocation may have an ordered list of Arguments, which is used as the inputs of the function call.

Figure 7.7: Extract of the synchronous metamodel: EquationSystem.

EquationSystem. Equations can be assembled together to provide a specific functionality. This set of equations are called an EquationSystem (Figure 7.7). There is no order requirement for these equations.

7.2.5 Node

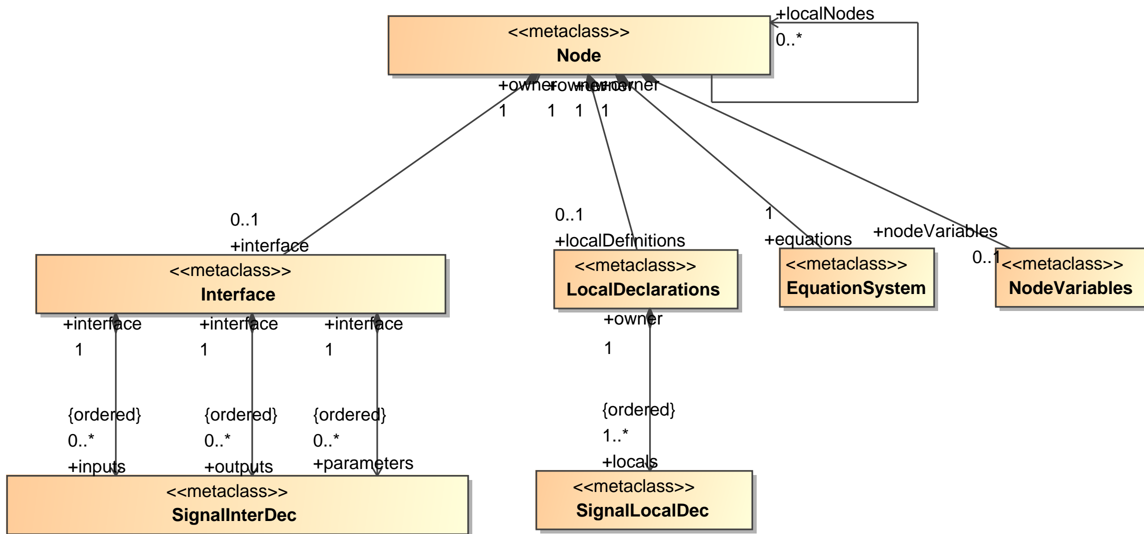


Figure 7.8: Extract of the synchronous metamodel: Node.

Synchronous functionalities are modeled as Nodes (see Figure 7.8). A Node has no more than one Interface, LocalDeclaration, NodeVariables, an EquationSystem and some Implementations and CodeFiles. NodeVariables is the container of Signals and SignalUsages.

Interface. Each input/output Signal is associated with a SignalDeclaration, which belongs to the Interface, while local Signals' SignalDeclarations belong to LocalDeclaration.

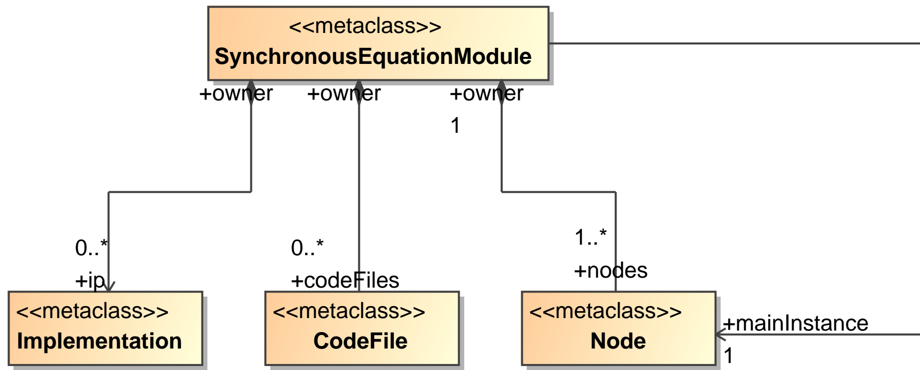


Figure 7.9: Extract of the synchronous metamodel: `Module`.

7.2.6 Module

All `Nodes` are grouped in a `Module` (Figure 7.9), which represents the package of the whole application. It contains one `Node` as the main `Node` of the application. Each `Node` is either defined in the `Module` or linked to an external function through IP deployment. These IP concepts (section 7.2.7), such as `CodeFile` and `Implementation` are also grouped in the `Module`.

7.2.7 IP deployment

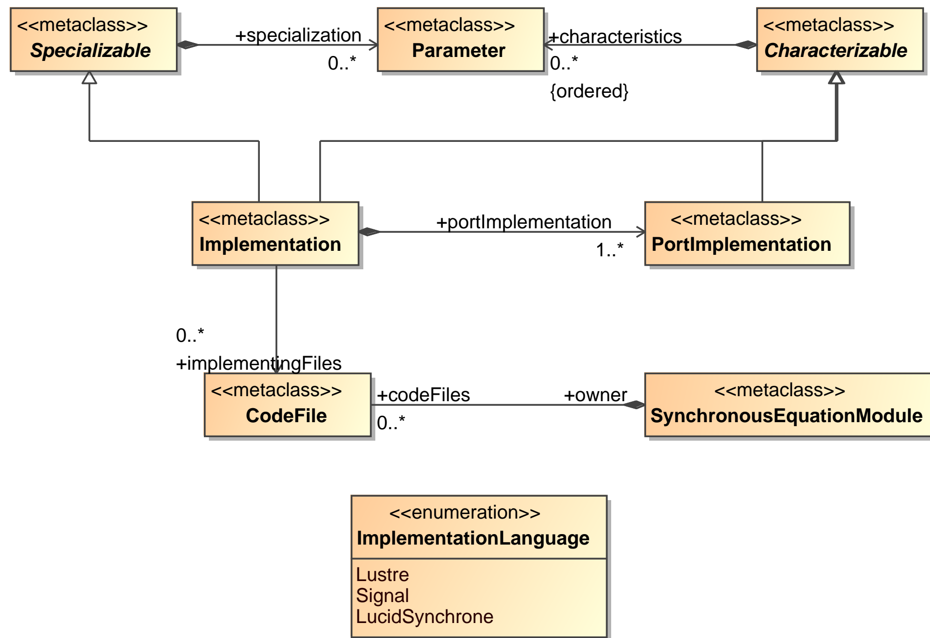


Figure 7.10: Extract of the synchronous metamodel: `Deployment`. Details of Gaspard2 deployment can be found in Appendix A.

`Nodes` that are not defined in the `Module` should be declared inside and defined outside. As a consequence, they should be deployed. The equivalent of these nodes in Gaspard2 are elementary tasks. An `Implementation` (Figure 7.10) associated with a `Node` contains the information of the external function. Parameters of external function are represented by `PortImplementations`. Their order are defined in the `Implementation` so that parameters are passed correctly to the application. An `Implementation` is associated with at least one `CodeFile`, which represents the code/source file of the external function.

7.3 The transformation rules

Transformations of Gaspard2 models into synchronous specifications consist of two steps (Figure 7.11): firstly, a transformation of Gaspard2 models into synchronous models; and then, the generation of synchronous code from synchronous models obtained from the first step.

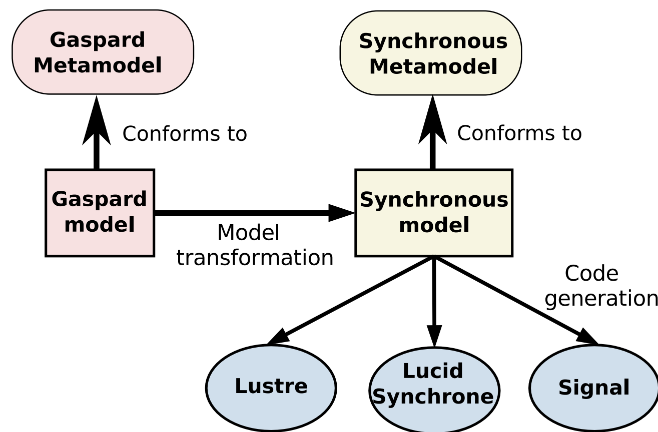


Figure 7.11: MDE-based model transformation. This figure is similar to Figure 4.12. However, an MDE-based approach is detailed in this figure.

7.3.1 From Gaspard2 models to synchronous models

Basic transformation principle is first given. `Components` and `ComponentInstances` are transformed into `Nodes` and `Equations` respectively. `Ports`, `PortInstances` and `DefaultLink` connectors in a `Component` are transformed into `Signals`, whereas `Tiler` connectors are transformed into `Equations` as well as `Nodes`. This transformation is implemented by a set of rules. The reason why we develop multiple rules is that: each notion has its distinct semantics, in this case, one rule is dedicated to one notion, which helps to separate concerns and reduce the overall complexity.

7.3.1.1 Transformation rules

The whole transformation can be represented through a tree structure (see Figure 7.12). The unique initial (root) rule is *GModel2SModel*. It transforms a whole deployed Gaspard2 application into a synchronous module. This rule then calls its sub-rules: *GApplica-*

tion2SNode, *GTiler2SNode*, *GACI2SNode*, *GAConsumer2SNode*, *GAProducer2SNode*, *GCodeFile2SCodeFile*, *GASImpl2SImpl*,etc. *GApplication2SNode* has also three sub-rules: *GElementary2SEquationSystem*, *GCompound2SEquationSystem*, *GRepetitive2SEquationSystem*. Note that not all rules in the transformation are given here. In the following, only rules presented in the Figure 7.12 are described. Among them, *GTiler2SNode* and *GApplication2SNode* are a little more detailed. The other rules are constructed in the same way.

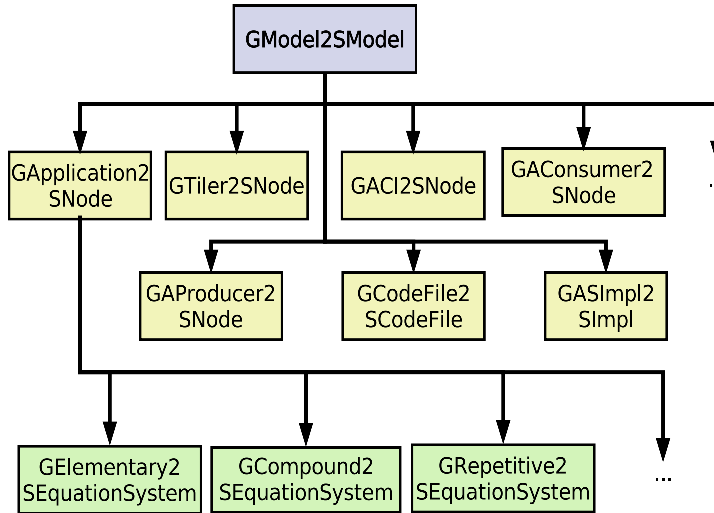


Figure 7.12: Hierarchy of the transformation rules.

- *GTiler2SNode* (Figure 7.13 in which each element is numbered). It is a rule for the transformation of `Tiler` connectors into synchronous input or output tiler `Node`. An input tiler `Node` is taken as an example for the construction of a synchronous node. First of all, the `Node` (numbered 1) is created and is associated with its `Module`. The `Port` and `PortInstance` [2] connected by this tiler are then transformed into input and output `Signals` respectively. One `Port` corresponds to one input signal, and one `PortInstance` corresponds to several output signals, whose quantity, n, is calculated from the repetition space defined in its connected `ComponentInstance`. The input signal is associated with n `SignalUsages` (4) and an output signal are associated with a `SignalUsages` (8). `Interface` (2) is then created and associated with `SignalDeclarations` (3, 9) that are associated with signals. Note that there are no `LocalDeclarations` in this node. Next, an `EquationSystem` contains n `Equations` (5). In each `Equation`, the `EquationLeftPart` has an `Argument` (6) which is associated with a `SignalUsage` of an input signal. `EquationRightPart` is directly an `ArrayAssignment`. Its `Argument` (7) is associated with a `SignalUsage` (8) of a corresponding output.
- *GRepetitive2SEquationSystem*. In this rule (Figure 7.14), an `EquationSystem` is first created. And then three types of `Equation` are created: input tiler `Equations`, repetitive task `Equation` and output tiler `Equations`. `Tiler` connectors are transformed

[2]These two concepts are first defined in UML. Port is associated with components in our case, while `PortInstance` are associated with component instances. The latter helps to distinguish dependecies specified on component and its instances.

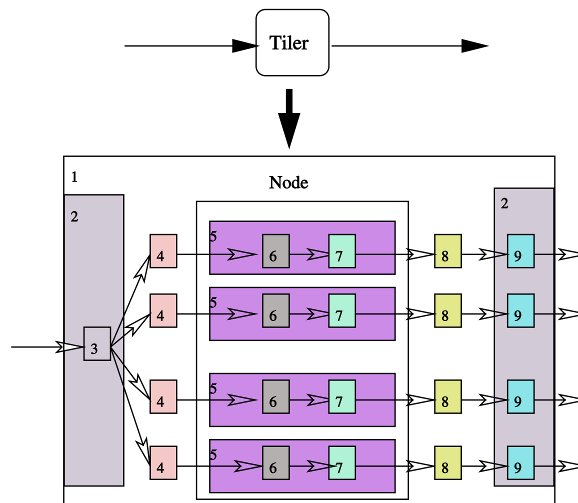


Figure 7.13: Transformation of the tiler. The numerated notions: node (1); interface (2); signal declarations (3, 9); signal usage (4, 8); equation (5); argument (6, 7).

into input/output tiler `Equations`, which are invocations to `Nodes` generated by *GTiler2SNode*, and the internal `ComponentInstance` is transformed into repetitive task `Equation`. A relevant repetitive task `Node` is then created, in which n equations invoke the task node corresponding to the component that declares the internal component instance.

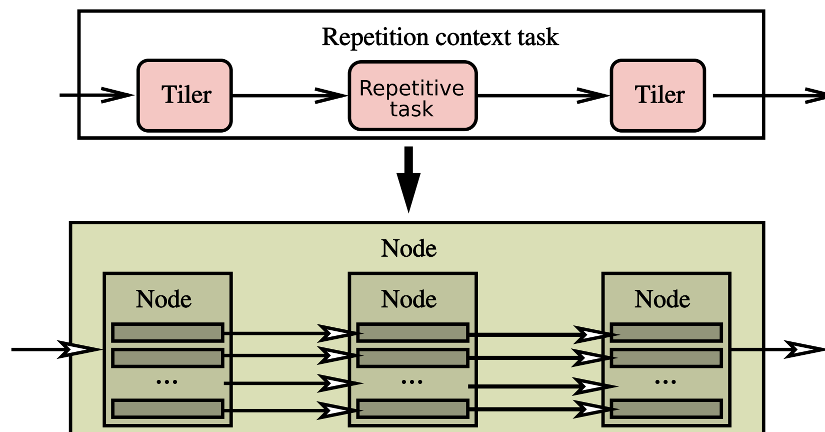


Figure 7.14: Transformation of the repetition context task.

- *GCompound2SEquationSystem*. Each internal `ComponentInstance` is transformed into an equation. Connectors between these `ComponentInstances` are transformed into local `Signals`.

- *GElementary2SEquationSystem*. No `Equation` is created because its owner `Node` is implemented externally and `Deployment` models are used to import its external declarations. However an `Interface` is created according to the component's ports.

Note that hierarchical composition in Gaspard2 models is preserved in synchronous models by node invocations.

7.3.1.2 Illustration of a rule model

These transformation rules are always difficult to describe with natural languages. The complete explanation is very tedious and takes long time for understanding, whereas a curtailed one is always ambiguous for lack of details. It is also impossible to show the implemented code for demonstration. Hence a new language is needed for the efficient description of these transformations.

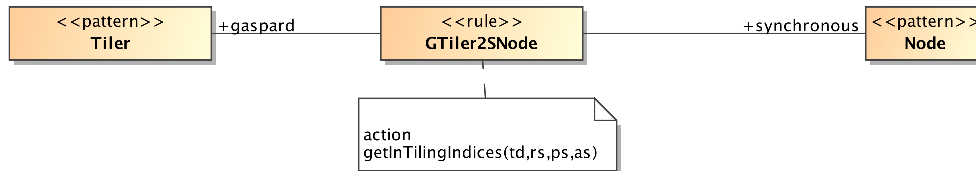


Figure 7.15: TrML patterns and rule.

TrML [37] is a TRansformation Modeling Language, which offers a graphical representation of model transformations through:

- its UML profile,
- its portable notation which is independent from any existing transformation engines,
- its mechanisms to divide transformation into rules.

Furthermore a metamodel is provided for the modeling. Note that model elements are in bold and associations between them are in italics in the following explanation. In a TrML rule (Figure 7.15), a transformation is divided into three parts: input `pattern` (`Tiler` in Figure 7.15), `rule` (`GTiler2SNode`), and output `pattern` (`Node`). The input pattern indicates the set of model elements to be transformed, which are based on the input model concepts (indicated by *gaspard*, the association between `GTiler2SNode` and `Tiler`). Similarly, the output pattern indicates the set of model elements to be generated, which are based on output model concepts (*synchronous*). `Rule` takes the input pattern and transforms it into output pattern. Some external functions used in the transformation are showed in the `note` (annotation boxes with `action`). Note that TrML allows the modeling of bidirectional transformations, but here, only one direction from Gaspard2 into synchronous is illustrated.

A typical transformation is illustrated with the help of TrML. The transformation of a Gaspard2 `Tiler` into a `Node`, called `GTiler2SNode`, is detailed in the order of input pattern, `rule`, and output pattern. Figure 7.16 shows the main part of this transformation, a finer description of the output pattern `Node` is illustrated in Figure 7.17.

The root element of the `Input pattern` is a Gaspard2 `Tiler`. In the transformation, however, more model elements execpt `Tiler` are needed. These elements can then be found through the associations connected to this Tiler. The `TilingDescription` stores the tiler's `F, O, P` information, which can be found through the *tiling*. A tiler is connected to, on one hand, a `Port` through the *Source*, which indicates the input array of the application component; on the other hand, a `PortInstance` through the *Target*, which indicates the

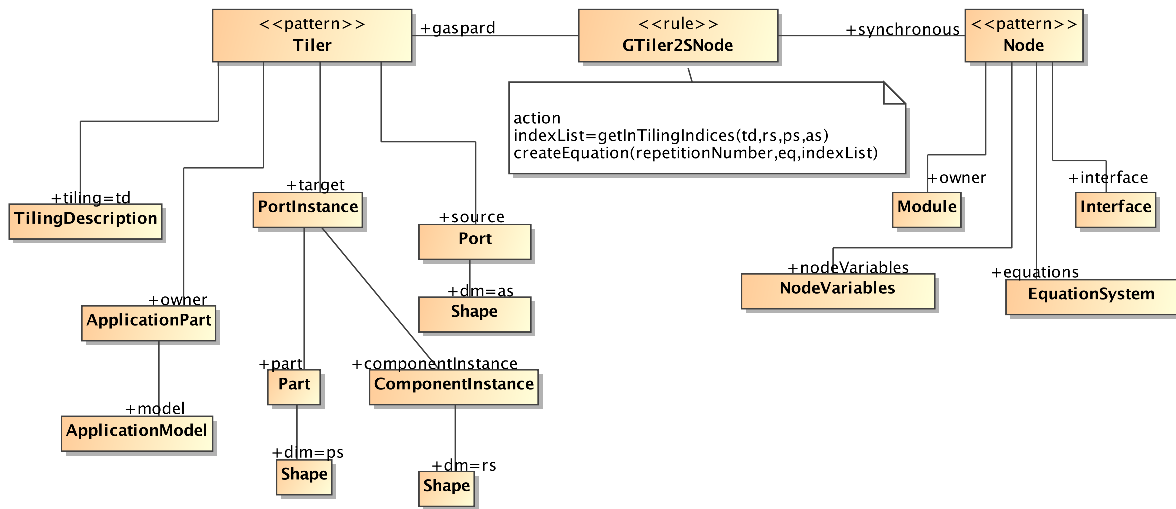


Figure 7.16: An extract of the TrML representation of a tiler transformation.

input pattern of the repetitions of the internal component. The port is connected to a `Shape` so as to indicate the array shape. The port instance is connected to `ComponentInstance` through the *componentInstance*, from which the shape of its port can be found. The port instance is also connected to `Part` by the `part`, from which the shape of repetition space can be found. Finally, from the association of `owner` of the tiler, the `ApplicationPart` and then `ApplicationModel` can be found.

`Rule` is the bridge between input and output patterns. The black box functions used by the transformation are specified in an annotation box that is linked to this rule, for instance, `getTilingIndices(td, rs, ps, as)` in the annotation box tagged with "action". This is a function implemented in Java that calculates the array indices from the tiling information. The arguments of the function come from the input pattern.

The root element of the `Output pattern` is a `Node` of the synchronous model (Figure 7.17). The node is associated with its owner, called `Module`, by the association *owner*. The `Module` can be found through the Gaspard2 `ApplicationModel` from the input pattern. All the elements required in the node are then associated to the node. For instance, `Interface`, `EquationSystem` and `NodeVariables` are associated to the node by *interface*, *equations* and *nodeVariables* respectively. Similarly, other elements are associated to `Interface`, `EquationSystem` and `NodeVariables` and so on. Some black box functions can be used during the creations of the model elements and their associations, such as `CreatIndexValue()`.

However, the transformation illustrated in the Figure 7.16 and Figure 7.17 is not a complete one because of the lack of expressivity of imperative aspects in TrML, for instance, the iterated creations of `Equation`, `Signal`, and `IndexValue` and the associations of the signals to their index values. Despite of this disadvantage, this graphical transformation language greatly helps to understand syntactic and certain semantic transformations between input/output models.

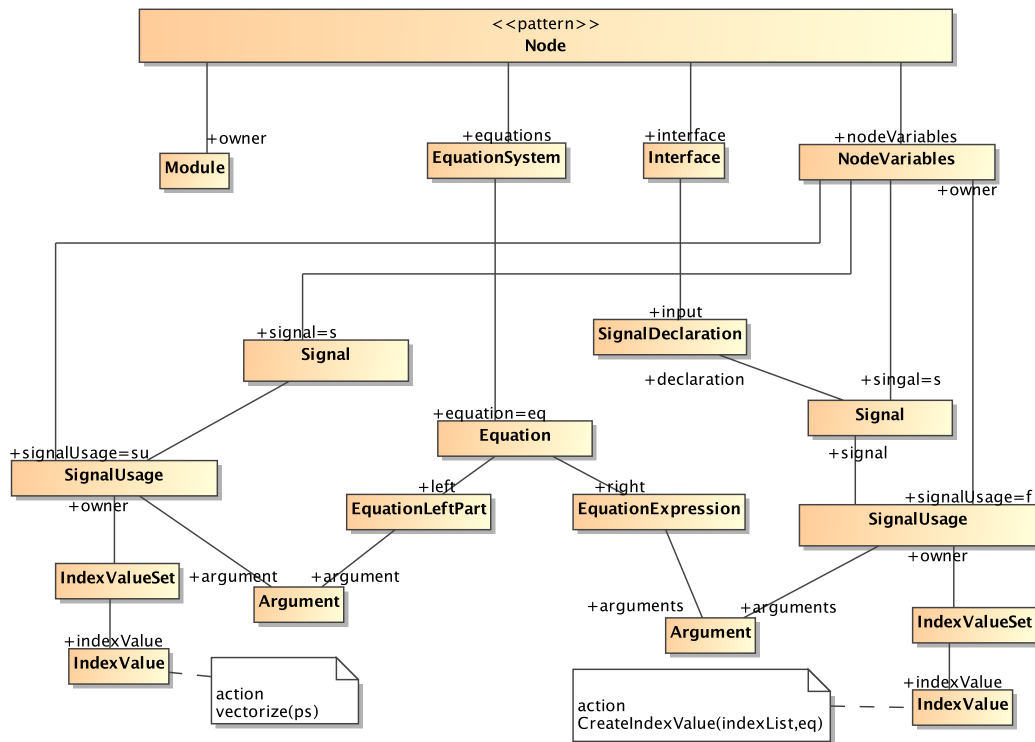


Figure 7.17: A detailed description of the output patten `Node`, which completes Figure 7.16 for the illustration of the `Tiler` transformation.

7.3.2 Transformation tools

These transformations were implemented with the help of specifications, standards and transformation languages. Some of them are briefly presented here. Although ATL and Kermeta are widely used, but they lack of extension capability specially when some external functions are needed to be integrated into the transformation during the development of this transformation. The MoMoTE (MOdel to MOdel Transformation Engine) tool is then developed in the team, which is in accord with MOF QVT. MoMoTE, which is based on the EMFT QUERY and is integrated into Gaspard2 as an internal plugin, is a Java framework that allows to perform model to model transformations. It is composed of an API and an engine. It takes input models that conform to some metamodels and produces output models that conform to other metamodels. A transformation by MoMoTE is composed of rules that may call sub-rules. These rules are integrated into an Eclipse plugin. In general, one plugin corresponds to one transformation. During model transformations, these plugins are automatically invoked one by one.

7.3.3 Template-based code generation and code generators

The implemented code generation (Figure 7.18) from synchronous models is based on EMF JET (Java Emitter Templates) [35]. MoCodE (MOdels to CODe Engine) is another Gaspard2 internal plugin, which works with JET for the code generation. MoCodE offers an API that reads the input models, and also an engine that recursively takes elements from input models and executes a corresponding JET Java implementation class on them.

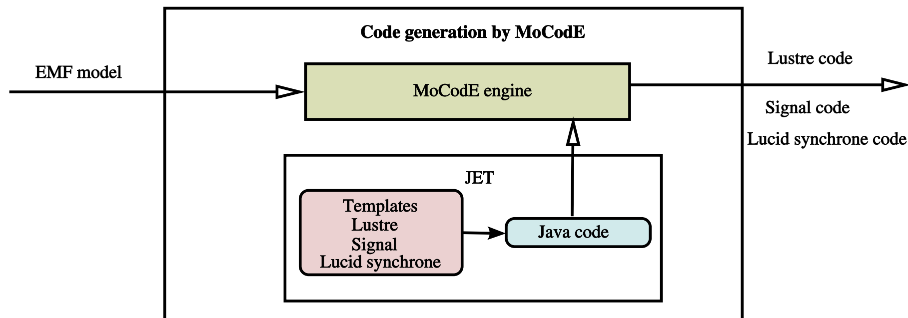


Figure 7.18: Generation of synchronous code from synchronous models.

The automatically generated code of the downscaler example takes a very big size, so an extract of the code can be found in Appendix B.

7.3.4 The synchronous transformation chain

Gaspard2 models are specified in the graphical environment MagicDraw, and are exported as Eclipse Modeling Framework (EMF) [34] models. EMF is a modeling framework and code generation facility. In the following transformation phase, these models are transformed into EMF Gaspard2 models. These two previous transformations will not be detailed here. Then the EMF Gaspard2 model is transformed into the EMF synchronous equational model, which is finally used to generate synchronous language code (e.g. Lustre and Signal code).

An automated model transformation chain (Figure 7.19) is then defined through the concatenation of these transformations from MagicDraw UML models to data-flow languages.

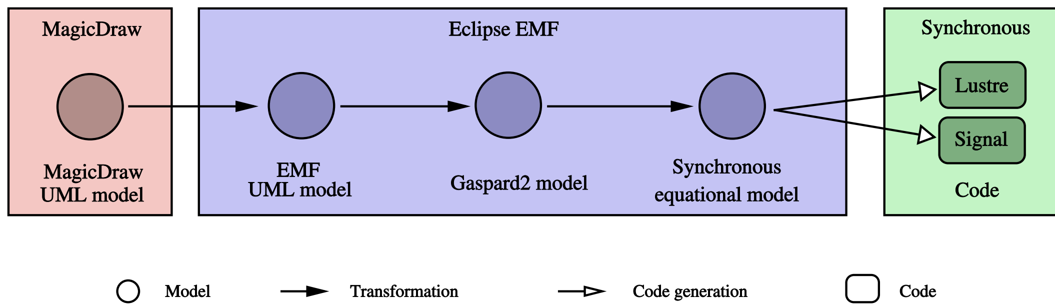


Figure 7.19: The detailed synchronous transformation chain.

A prototype model transformation tool, based on MDE has been developed in order to enable the automatic transformation of Gaspard2 models into synchronous programs. This tool is called Gaspard2 synchronous transformation chain. All the transformation presented in this chapter has been implemented by this tool. New features are still being developed and integrated into this tool, which will be described in the next chapter. Based on a generic metamodel for synchronous equational dataflow languages, the transformation targets Lustre, Lucid synchrone and Signal at the same time.

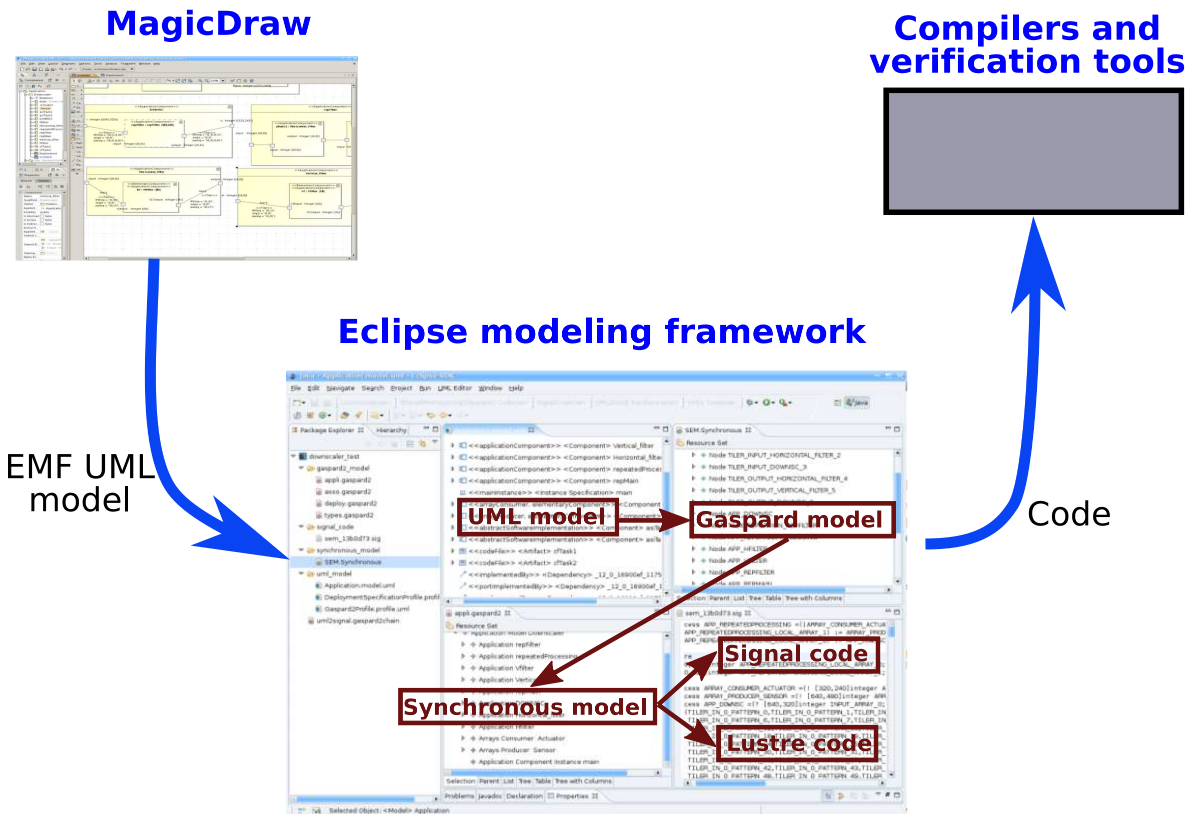


Figure 7.20: The architecture of the implementation tools.

Figure 7.20 illustrates a typical transformation chain with the Lustre and Signal as target. The DIP applications are first specified in MagicDraw with the help of the Gaspard2 UML profile. The specified model are then exported so as to be used in Eclipse environment, where automatic model transformations are carried out according to Gaspard2 and synchronous metamodels. Then the results of these transformations are Signal code.

The repetition size that could be handled in the transformation within the Eclipse capacity is about forty thousands. The implementations are carried out on a desktop computer that is equipped with two Quad-Core Intel Xeon processors for a total sum of eight cores. The computer has 2 Gigabyte memory and runs on Linux. Running on Java virtual machine, Eclipse java plugins meet a problem of memory size limit. For instance, the transformation plugin calculates array indexes and stores them in the memory temporarily, hence when the array size or the repetition number is too big, the plugin will run out of the available memory. As a result, there is a maximum repetition number. But certain optimizations on memory usage can be done to improve the repetition number that can be handled, for instance, by storing the array indexes in temporary files during the plugin execution in order to reduce the memory usage.

7.4 Conclusions

This chapter presents a concrete implementation of the synchronous modeling in the framework of MDE, in addition, a flow of design is also illustrated: the UML profiles (particularly Gaspard2 profile) are used for the high-level and heterogeneous modeling (application, architecture, association, deployment, etc.) of DIP applications; the graphical design is transformed into executable code in synchronous languages through a chain of model transformations, where the Gaspard2 model and the synchronous model act as intermediate models; execution or validation can then be carried out on the resulting code (a case study of formal validation is presented in Chapter 9).

The implementation presented in this chapter shows the advantages of the MDE approach: firstly, it simplifies the modeling of DIP applications by using simple but standard UML notations, including its extension (Gaspard2 profile). In addition, the simple organization of these notations in a UML or object manner liberates users from heavy syntax and grammar of classical languages.

Secondly, the model transformation is efficient and flexible with regard to classical compilers. As intermediate models can be introduced in this transformation, the complexity of the transformation can be reduced according to separation of concerns, i.e., one transformation addresses one certain problem. Hence each transformation can be kept simple. Another advantage is that modifications of an intermediate model will not lead to the modifications of all transformations, hence it makes it possible to follow the modern rapid software evolution. Using transformation rules is another advantage, because they are modular and hence easy to maintain. These rules are defined to be declarative, which include input pattern, output pattern and the transformation relation. However, imperative aspects of transformation can also be specified to enhance the processing capacity of rules.

Finally, the number of tools associated with or dedicated to MDE have been dramatically increased, which provide a good support for the MDE-based development.

Chapter 8

Transformation of the control extension of Gaspard2

8.1 MARTE-compatible control for Gaspard2 . . . . **132**
8.1.1 Mixed structure-behavioral modeling 132
8.1.2 System behavior modeling with the help of UML 132
8.1.3 Using UML state machines 133
8.1.4 Using UML Collaborations in mode task component 138
8.1.5 A complete Gaspard2 control structure 139
8.2 Control extension to Gaspard2 metamodel **139**
8.2.1 The metamodel of state graphs 140
8.2.2 The metamodel of events 140
8.3 Extended synchronous metamodel **140**
8.3.1 StateMachine 142
8.3.2 BooleanExpression 142
8.4 Transformations **142**
8.4.1 From a UML model to a Gaspard2 model 144
8.4.2 From a Gaspard2 model to a synchronous mixed-style model 146
8.4.3 From a mixed-style model to an equational model 147
8.4.4 From a mixed-style model to an automaton model 151
8.5 Conclusions **154**

This chapter presents the implementation of the previous reactive control modeling in Gaspard2 in Chapter 6, which results in a model transformation chain from UML/MARTE models to synchronous models. According to the different stages in the model transformation chain, the whole implementation can be presented in four parts: the graphical representation of control using UML diagrams and MARTE profile, the extension of Gaspard2 metamodel with state-based control concepts, the extension of synchronous metamodel with control concepts and transformation rules that bridge the gap between UML/MARTE and synchronous models with the help of their metamodels.

8.1 Extended Gaspard2 graphical interface in conformity with MARTE

8.1.1 From structural modeling to mixed structure-behavioral modeling

The previous version of Gaspard2 profile without control modeling is mainly based on UML diagrams for the structural specification. The internal structures and the static connections between components can be clearly specified. However the control introduced in Chapter 6 concerns behavioral aspects, which can not be expressed by structural modeling only. Hence an enrichment of Gaspard2 with behavioral modeling concepts is necessary. As a whole, we are interested in two kinds of behaviors: the first one is state-based individual behavior, which shows the state change during the life cycle of the individual in reaction to some external stimuli. In this case, we are not interested in how it communicates with other individuals. The second kind of behavior is the communication behavior between a component and its internal components in terms of connections. This connection behavior is always a consequence of state changes of another component.

8.1.2 System behavior modeling with the help of UML

In Chapter 6, the Gaspard2 control has been presented with clear syntax and semantics specifications. These specifications are declared in natural language, e.g., English. As Gaspard2 aims to provide a development environment with a graphical interface, these control specifications should also take a graphical form in alignment with other Gaspard2 concepts in terms of Gaspard2 UML profile.

In the evolution of Gaspard2, the new MARTE profile for the real-time embedded design has great influence on Gaspard2 in the sense that all Gaspard2 concepts will be integrated into or be modified in accordance with MARTE, for instance, the repetitive operator concepts has been integrated into MARTE. Following this trend,

MARTE-compatible behavioral modeling for Gaspard2 is expected. But MARTE does not provide appropriate state-based behavioral modeling concepts as we presented in Chapter 6. The basic UML behavior modeling concepts is then proposed for our control modeling, such as *State Machines* and *Collaborations* [103]. In the proposition of control extension, *Behavioral State Machines* are chosen to demonstrate the state-based behavior of an individual component, which acts as a controlling element in the system. *State Machines* are the first choice because they are compatible with the state-based modeling presented in the previous Chapter 6. As in Gaspard2, control and computation are separated, the single structure modeling of the controlled computation is not enough, *Collaborations* is used to illustrate the behavior between components, which are controlled elements in the system.

From now on, Gaspard2 actually denotes a new MARTE-compatible version, i.e., MARTE concepts are adopted in Gaspard2.

In Chapter 6, we presented the automata-based control, however UML state machines can not be used directly in Gaspard2 for the following reasons:

- Gaspard2 adopts a component-based approach, where the interfaces indicate the functionality provided by this component. UML behavioral state machines can be associated with components, however they work on *attributes* and *operations* of a component in preference to its ports, compared to UML protocol state machines.

- Gaspard2 is dedicated to the specification of DIP applications, which is different from event-driven nature of UML state machines. UML state machines are different from mode automata in synchronous languages for the same reason.

The next subsection is dedicated to the description of a specific usage of UML state machines (the same case as *collaborations*) so that they can be used in Gaspard2 in a compatible way with other existing Gaspard2 concepts. The specific usage of UML will not change the syntax of state machines or collaborations. However their semantics are changed under some conventions. The result of this change can then be considered as a variant of UML state machines.

8.1.3 Using UML state machines

8.1.3.1 The component associated to UML state machines

In Gaspard2, the component, with which UML state machines are associated, SMC (State Machine Component) for short, is always a controlling component (it is required to produce mode values for other components). An SMC (see an example in Figure 8.1) is an implementation of its associated state machines, SM for short.

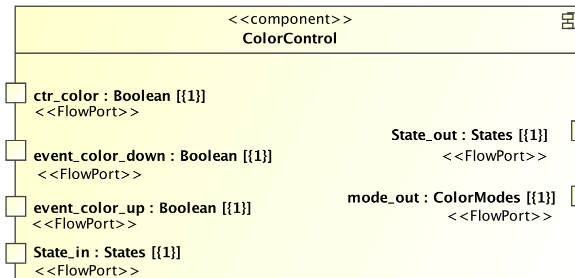


Figure 8.1: An example of state machine component. This component is associated with the state machne presented in Figure 8.2. The component has three event inputs: *ctr_color*, *event_color_down*, and *event_color_up*, which are used to trigger transitions. *State_in* is used to indicate which state of the state machine is the source state. The target state and mode are outputed through *State_out* and *mode_out* respectively.

Port specialization in an SMC

As to an SMC, only its interfaces are exhibited. And the interfaces are actually presented by *Ports*. These ports are stereotyped as *flow ports*. SM are supposed to react to the values that pass through these ports. The shape of a port indicates how many values arrive at this port at one time. As we take only dataflow into consideration, the shape is defined to correspond to one reaction of the state machines, i.e., the shape is always {1}.

The input ports of an SMC are two kinds of ports: *event ports* and *state ports*. Event ports indicate that values, which go through these ports are of Boolean type, and they are used usually in triggers of transitions. *Event ports* are UML *Behavioral ports*. Values going through *State ports* are considered as *state values*, which are identifications of states in the SM. These values provide the initial state upon entering the SM, similar to the *initial pseudostate*. As the

state machine can be a hierarchical one, so multiple initial states are needed. In this case, each state port is associated with a *region* by using the same name. As a convention, the initial state defined by the *initial pseudostate* in the region is replaced by the state obtained through the state port. Note that it is different from UML state machine, where the starting state can be only given by the *initial pseudostate*.

An SMC have also two kinds of output ports: *state ports* and *mode ports*. Similar to the input *state ports*, the output *state ports* provides the next states of the state machines (the states after the fired transitions). The output *mode ports* carry mode values that are transmitted to other components for the purpose of mode choice. These mode values are defined in the application independent from the SMC and its mode switch component for reuse purpose. As a result, any of these components (SMCs) can be replaced by a third one, if the third one is still compatible with the predefined mode values.

8.1.3.2 A subset of UML state machines

Figure 3.4 shows the metamodel of UML state machines, which seems to be too complex. The state machines used for the modeling of Gaspard2 control are only a subset of UML state machine. There are two reasons: *a*) UML state machines are intended to be applied in *all* applications, however many concepts are not needed in Gaspard2; *b*) in consideration of a concrete implementation of model transformation, Gaspard2 should remain concise but expressive enough to simplify the development. The main concepts of UML state machines used in Gaspard2 are enumerated: *StateMachine*, *Region*, *State*, *Vertex*, *Transition*, *Pseudostate*, *PseudostateKind*, etc. Relations between these concepts are illustrated in the Figure 3.4.

StateMachine

StateMachine is a *Behavior*, which is used to exhibit the behavior of part of a system, for instance, a controlling component. A state machine may have at least one regions, which contain vertices and transitions.

Region

A region is an orthogonal part of either a composite state or a state machine [103]. It contains vertices (states and pseudostates) and transitions.

States

A Gaspard2 state used in an SM can be any of the following kinds: simple state, composite state and sub-machine state. The composite state and sub-machine state make it possible to specify state machines in a hierarchical way.

A state has a specific *doActivity* , where we can specify a *behavior* carried out in this state. This *behavior* can be specified as an *OpaqueBehavior*, which can be written in any language, such as a natural language. In Gaspard2, we use this *OpaqueBehavior* in *natural language* to specify values that are sent to output ports when the state is *active*. For instance, in Figure 8.1, the SMC has two output ports: *state* port and *mode* port. The first one indicates the current

state of the state machine and second one indicates the mode to execute at this moment. The *doActivity* can be specified as:

```
region.ostate=self.name and app.mode=enumeration.M1
```

where the left part of the equations concerns the port and the right part concerns their values. For instance, `region.ostate` and `app.mode` are port names of the component. `region` is the name of a region, which owns the current state. `ostate` is a string that denotes a state port. `app` represents the name of an application that requires this mode values. `mode` is also a string that denotes a mode port. `self` indicate the state itself, it is used to distinguish its containing state, `super`, or its sub-state, `sub`, in case of state hierarchy. `self` indicates the name of the state itself in case of simple state and `M1` is a mode value, which is defined in a *enumeration* outside the state machines. The *enumeration* can be found through `enumeration`.

Vertex

A *vertex* is either a *state* or a *Pseudostate*. It is used to indicate the source and the target of *transitions*.

Pseudostate

We are interested in three types of pseudostates: *initial pseudostate*, *deepHistory pseudostate* and *shallowHistory pseudostate*. Note that these pseudostates do not exist in GSGs, but they greatly simplify the graphical design in UML, hence they are kept in the proposition, which is based on UML. They are finally translated into *DefaultLink* and *InterRepetition* in Gaspard2.

- *initial* pseudostate is connected to one of the states in a state machine (more precisely, a region), which is the initial state of the state machine (region). But if there is an input state port defined for the same region, the initial state indicated by the initial pseudostate is re-defined by the state obtained from the state port.

- *deepHistory* pseudostate represent the most recent active states in a composite state that contains this pseudostate. This pseudostate is connected to a state in case that there is no most recent active states (for instance, the composite state is entered for the first time). Otherwise, it indicates which states should be reactivated upon entering the composite state. Only one deep history pseudostate is allowed in a composite state (inconsistency in UML [43]).

- *shallowHistory* pseudostate is similar to a deep history pseudostate, except that it indicates directly the sub-states of the composite state (unlike a deepHistory, which memorizes all the states in the hierarchy). The sub-state denoted by shallow history pseudostate is actually the initial state for the composite state (inconsistency in UML [43]).

Transitions

A transition is a directed link between a source vertex and a target vertex (state or pseudostate). A transition can have several triggers, any satisfaction of these triggers can fire the transition. Figure 8.2 show an example of a transition, where the trigger is defined as `when event_color_up and ctr_color`.

Triggers

A trigger is related to an event that may cause the execution of an associated behavior (In Gaspard2, it means an associated transition), for instance, the trigger `when e1 and e2` in Figure 8.2. `when event_color_up and ctr_color` is a Boolean expression that represent an event.

Events

The events used in Gaspard2 in a trigger of a transition is generally a *ChangeEvent* which is associated to an input port. The event has an *Expression*, called *changeExpression*, which is Boolean expression that can result in a change event. Another kind of event is also used in Gaspard2 is *AnyReceiveEvent* [1], written as `all` in the example in Figure 8.2. *AnyReceiveEvent* can be considered as a default event when all the triggers of the transitions from the same vertex are not satisfied.

Expression

The expression used in Gaspard2 is an binary expression that can generate a change event when it is evaluated true. The variables in the expression are declared as MARTE primitive Boolean type. These variables are associated to the values that go through the input event ports (names of variables here are the same as those of the corresponding ports). For instance, *event_color_up*, *ctr_color* in the trigger appeared in the Figure 8.2. They are used in a binary expression: `when event_color_up and ctr_color`. And `event_color_up` and `ctr_color` are also associated with the event ports that have the same name.

From the syntactic point of view, Gaspard2 state machines are a subset of UML standard state machines. But they do not have the same semantics. The concrete semantics of Gaspard2 state graph are presented after an example of a Gaspard2 state graph in next subsection 8.1.3.4.

8.1.3.3 An example of a simple state machine

Figure 8.2 illustrate an example of a complete state machine. This state machine has three states. One of them is connected to the initial pseudostate, which indicates that the state is the initial state. Some transitions connect these states, which can be fired by triggers on the transitions. Trigger are defined on events, which are Boolean expressions of the event port variables (*event_color_up*, *ctr_color*).

[1] *AnyReceiveEvent* is defined in UML, see [103] for more details of this event type.

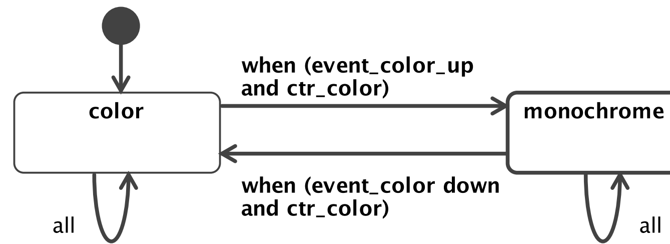


Figure 8.2: An example of a state machine.

8.1.3.4 Gaspard2 state graphs in a UML component context

From the point of view related to Gaspard2 semantics, UML state machines are not appropriate. From now on, we use the term Gaspard2 state graph (GSG for short) to distinguish UML state machines. A GSG is associated with a Gaspard2 component in order to give a more precise external view with regard to its **ports**. This is different from UML state machines, which is associated to a component to give more precise dynamical constraints on **operations** of the component.

8.1.3.5 Gaspard2 state graphs in a dataflow context

UML state machines adopt the event-driven style, i.e., the state machines react to some events issued in its context classifier. However, a Gaspard2 component associated with a state graph is a dataflow processing component, and the state graph specifies the component behavior in response to the values that pass through the flow ports of the component. Hence transitions of the state graph correspond to arrivals of new values through the ports. The GSG is supposed to accomplish its transitions before the arrivals of new values. Unchanged values lead to self transitions of the state graph through the ports. These arrivals and changes of values are supposed to be captured by some *ChangeEvents* associated with the ports, for instance *event_color_up* and *event_color_down* in Figure 8.2.

8.1.3.6 Gaspard2 state graphs in a repetition context

According to the control modeling presented in Chapter 6, a component associated with a GSG (i.e., a state graph component, SGC for short) is translated differently according to it repetition context. In case that some inter-repetition dependency is introduced in the repetition context, a GSG is translated to be repeated multiple times in a sequential way, hence the GSG associated to an SGC can be translated into an automaton. In this case, the input state ports of an SGC are set by the inter-repetition dependencies. But if inter-repetition dependency is absent in the repetitive context, an SGC is translated into multiple instances that execute in parallel. This is not handled in this thesis.

8.1.3.7 Gaspard2 state graphs in a hierarchical context

An SGC can be composed with other Gaspard2 component if their interfaces are compatible. In this case, the SGC is considered as a normal Gaspard2 component.

8.1.4 Using UML Collaborations in mode task component

The mode switch component for the mode switch task

A Gaspard2 mode switch component (MSC for short) corresponds to mode switch task introduced in Chapter 6. An MSC have several normal data ports and a mode port (a behavioral port). A MSC is the containing component of several mode task components (MTC for short). Each MTC in the context of an MSC is called a mode of the MSC. The MSC defines the collaboration behavior of these modes according to the values of its mode port. These MTCs correspond to several mutual exclusive modes, which have the same interfaces. They have also the same interfaces as their MSC, except the mode port of MSC. An MSC is connected with its MTCs with the help of *delegation connectors* through their ports.

Figure 8.3 shows an example of an MSC. The MSC, called *ColorEffectSwitch*, has three ports: *mode_color*, *i* and *o*. *mode_color* is a behavior port which is used to obtain mode values. The mode values are then used by *ColorEffectSwitch* to choose the execution of the corresponding mode. Modes in *ColorEffectSwitch*, *c* and *m*, are instances of mode tasks *ColorMode* and *MonochromeMode* respectively. The input port *i* of *ColorEffectSwitch* is connected to the input ports of *c* and *m*, in the same way, the output port *o* of *ColorEffectSwitch* is connected to the output ports of *c* and *m*. Actually, only one mode, either *c* or *m* is selected to execute according to the mode value provided by *mode_color* of *ColorEffectSwitch*. Mode values in this example are *ModeColor* and *ModeMono*, which are denoted by the collaboration names.

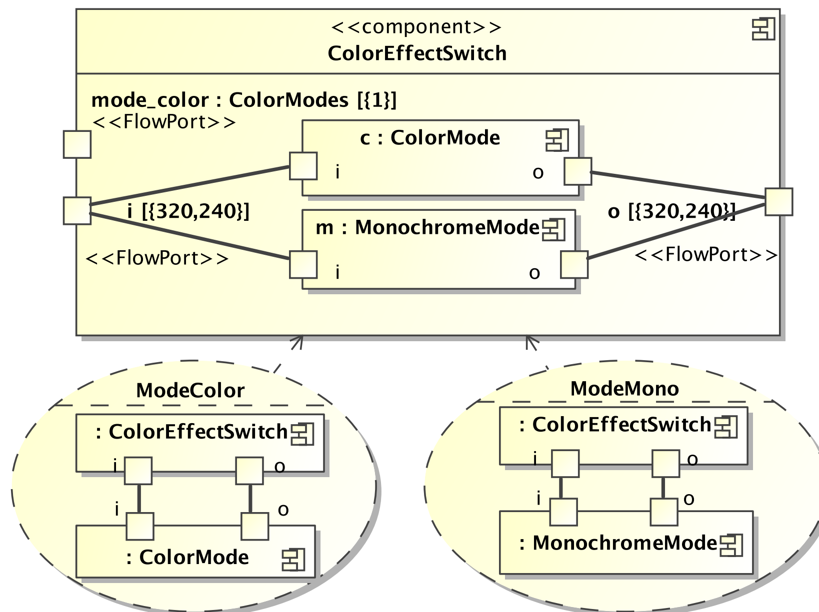


Figure 8.3: An example of *collaboration*.

The mode switch component associated with *collaborations*

The MSC is structurally presented in the previous subsection. Obviously the execution semantics or how the internal parts of the MSC collaborate is not defined. The behavior of MSC can be expressed in any natural language, e.g., choose a mode according to the mode value provided by its mode port. However this English sentence remains obscure to a machine-interpretable formalism. UML proposes to use *collaborations* to specify roles of components (instances level collaboration) by using parts and connectors in composite structures. A collaboration depicts the relation of some collaborating elements, which are called roles in the collaboration. These roles, each of which provides a specialized function, accomplish some desired functionality in a collective way. In a collaboration only the concerned aspects of a role are included, other details (identification, unused interfaces, etc.) are suppressed. Figure 8.3 illustrates the collaborations of an MSC and its MTCs. For instance, the collaboration *ModeColor* shows how the MSC *ColorEffectSwitch* works with the MTC *ColorMode*. The connections between them are showed in the collaboration. As in this mode, only *ColorMode* is supposed to execute, another mode *MonochromeMode* is suppressed. The collaboration is finally connected to *ColorEffectSwitch*.

8.1.5 A complete Gaspard2 control structure

Both SGCs and MSCs are considered as normal Gaspard2 components, hence they can be used in an application as other Gaspard2 components once they are provided compatible interfaces.

Here an example of a typical complete Gaspard2 control structure is given in Figure 8.4. In general, as a state machine cannot be modelled in the composite structure diagram, the state machine cannot be displayed with components at the same time in MagicDraw and Papyrus. Hence, in order to give a global view of a control structure, the example in Figure 8.4 is not modelled in a UML graphical environment, but it is very close to UML modeling.

8.2 Control extension to Gaspard2 metamodel

The first control extension with control proposed in [71] took a controlling component (SGC in this thesis) as an elementary component so the controlling component should be associated with an external implementation. But in the current proposition of this thesis, as mentioned in the previous subsection, the controlling component is associated with a UML state machine diagrams where state graphs can be drawn. Hence no external implementation is needed.

The state graph metamodel is based on UML state machine metamodel. More precisely, it is a simplified subset of UML state machine metamodel. First, as UML is already a standard, it is not necessary to create another similar metamodel for automata. Secondly, UML state machines are relatively stable which provide the capacity of enough expression for the modeling of automata. Thirdly, UML state machine is too complex with regard to our requirements for Gaspard2 control, so only a necessary subset is used in the Gaspard2 metamodel.

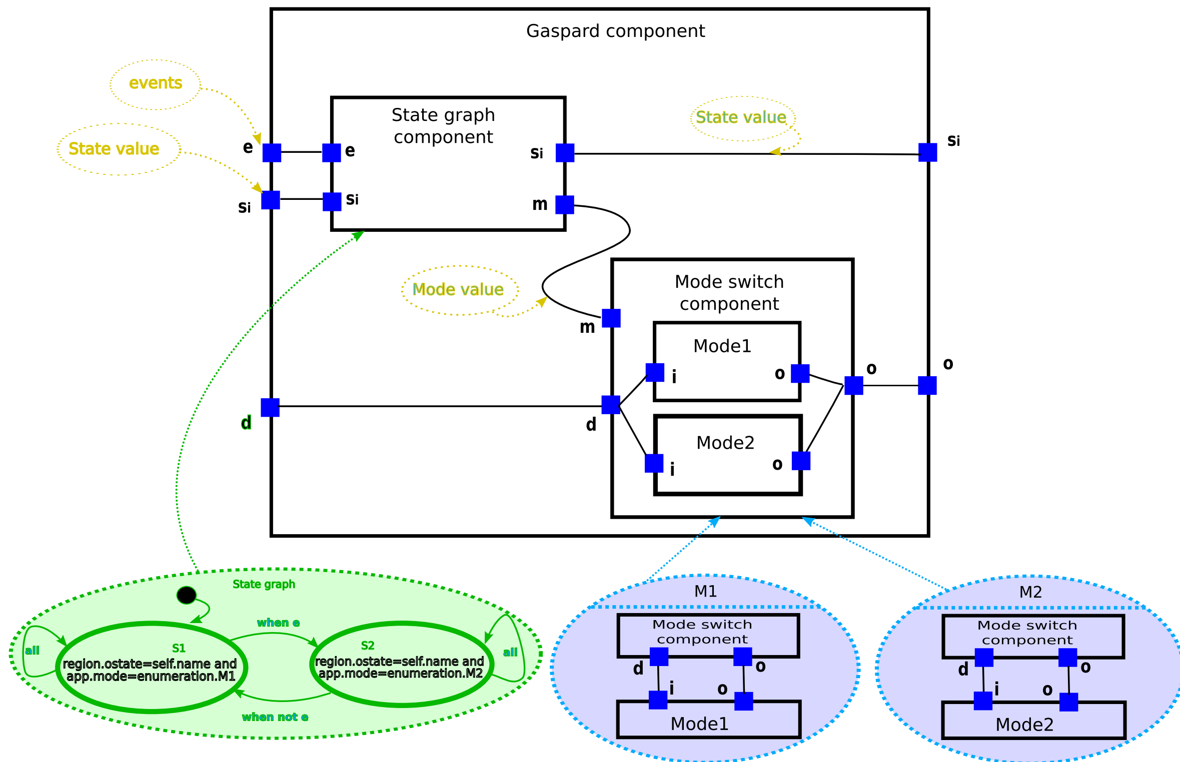


Figure 8.4: An example of a complete Gaspard2 control structure.

8.2.1 The metamodel of state graphs

This metamodel (Figure 8.5) mainly depicts the relations between *StateGraph*, *Region*, *Transition*, *State* and *Trigger*. As UML state machines have been presented in Chapter 3, the details of this metamodel are not given here.

8.2.2 The metamodel of events

This metamodel (Figure 8.6) mainly specifies the relations between *Event*, *Trigger*, *Expression* and *ValueSpecification*. For the same reason as given for the previous state graph metamodel, the details of this metamodel are not given here.

8.3 Extended synchronous metamodel

There are two ways to present the state-based control in synchronous languages. The first one is to translate the control directly into synchronous equations, i.e., all the states and transitions are coded in equations. An advantage of this method is that little modifications are needed in the synchronous model presented in previous chapter. However, this approach has also its disadvantages: *a*) the resulting equations are not easy to read; *b*) explicit-state-based verification and analysis tools, such as Matou, cannot be applied. The second approach is to integrate a metamodel of automata (state machines) in the synchronous model.

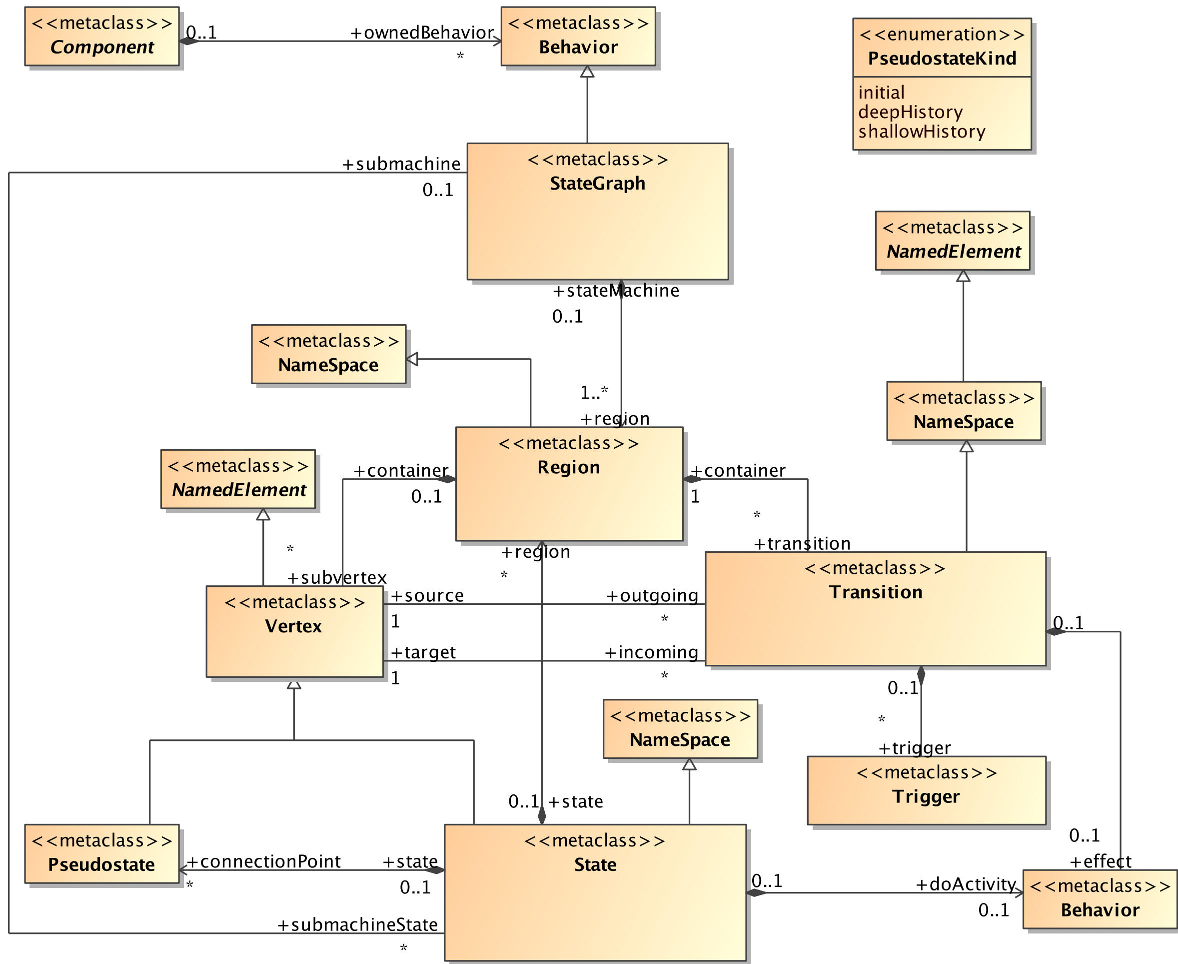


Figure 8.5: An extract of Gaspard2 *StateGraph*, which is proposed according to the metamodel of UML state machines.

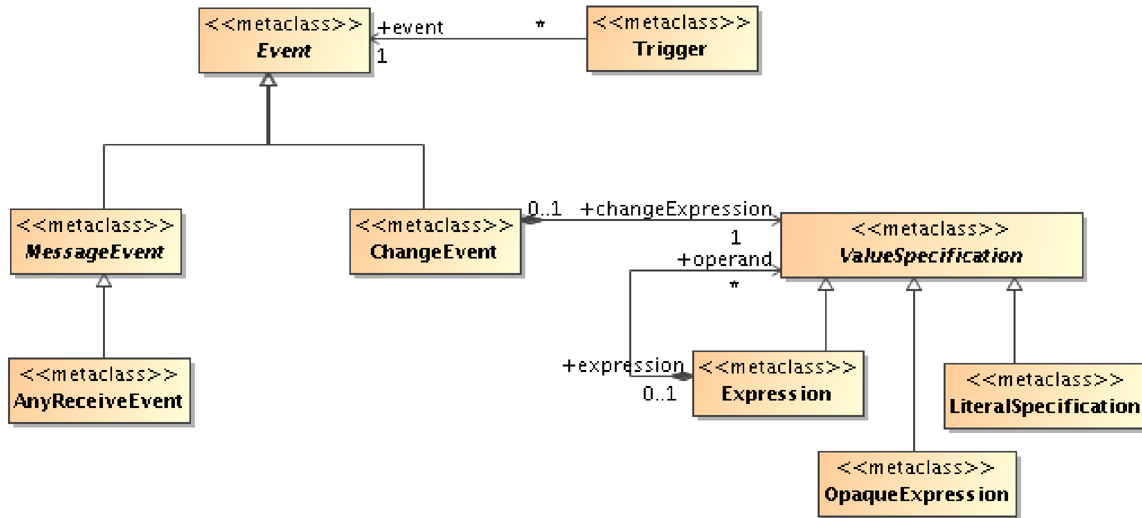


Figure 8.6: The extract of the *Event* metamodel in Gaspard2.

This explicit automata help greatly in automata-based model checking and the transformation from Gaspard2 state graphs to synchronous automata is almost straightforward.

8.3.1 StateMachine

This metamodel (Figure 8.7) mainly depicts the relations between *Node*, *Automaton*, *Transition*, *State*, *Trigger*, *BooleanExpression*, and *Equation*. This metamodel is a metamodel of flat automata, i.e., the automaton itself is not hierarchical, the hierarchy of automata is expressed by node hierarchy. A *Node* can contain an *Automaton*. The latter can have *States*, *Transitions*, *History* and *Reset*. A *State* can have *Equations* as its internal computation, where other automaton nodes can be invoked. *Transitions* are associated with source and target states. *Transitions* contain *Triggers*, which in turn have *BooleanExpressions*.

8.3.2 BooleanExpression

The metamodel of *BooleanExpression* (Figure 8.8) shows how to build Boolean expressions from operators (such as *And*, *Or*, *Not*, *IfThen* and *IfThenElse*.) and operands (such as *SignalUsage* and other *BooleanExpressions*.).

8.4 The translation rules based on metamodels

This section explains how a Gaspard2 UML model can be transformed into a synchronous model by means of transformation rules (similar to the previous chapter). First of all, some intermediate models are presented:

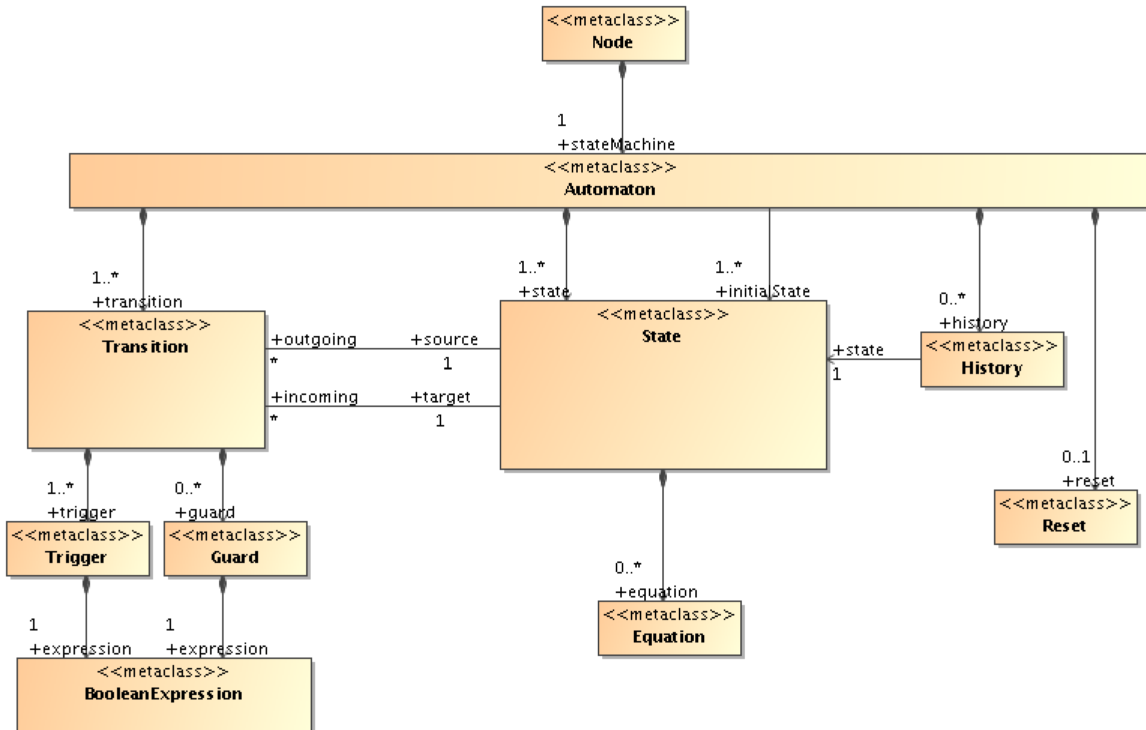


Figure 8.7: The extract of the *StateMachine* metamodel in synchronous model.

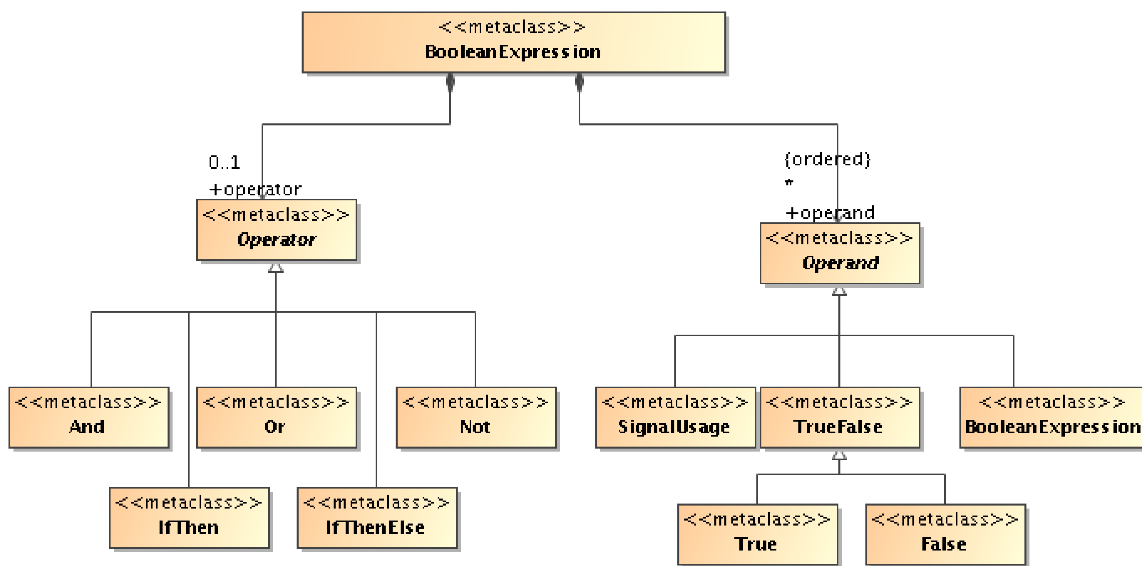


Figure 8.8: The extract of the *Expression* metamodel in synchronous model.

- Gaspard2 UML model: this model corresponds to a model obtained from a graphical UML tool, such as Papyrus[2] and MagicDraw[3]. This model contains both Gaspard2 and UML concepts at the same time.
- Gaspard2 model: this model is a pure Gaspard2 model, where only Gaspard2 domain concepts retain.
- synchronous mixed style model: the mixed style model is a mixture of an automata model and an equation model. This model enables to preserve explicit automata (compared to equations) directly, which is in favour of generating code in the form of mode automata in synchronous languages.
- synchronous equational model: this model is a pure equational model, where all automata are coded directly in equations.
- synchronous automata model: this model is a mode automata style model, which enables to generate code of mode automata for Lustre (by means of Matou), Lucid synchrone, polychronous mode automata.

Figure 8.9 illustrates a transformation flow in consideration of previously mentioned models. **T1**, the transformation from a UML model into a Gaspard2 model, is explained in Section 8.4.1. **T2**, the transformation from Gaspard2 model into synchronous mixed style model, is discussed in Section 8.4.2. **T3**, the transformation from a synchronous mixed style model into an equational model, is presented in Section 8.4.3. **T4**, the transformation from the synchronous mixed style model into a synchronous automaton model, is shown in Section 8.4.4.

8.4.1 From a UML model to a Gaspard2 model

This section concerns the transformation of the MARTE UML model into the Gaspard2 model. As transformations of MARTE UML model without concepts of state machines and collaborations have been already presented in the previous chapter, only transformations related to UML state machines and collaborations are presented here.

State machine component and mode switch component. A MARTE component associated with UML state machines is transformed into a Gaspard2 *ApplicationComponent* associated with Gaspard2 state graphs. A MARTE component associated with UML collaborations is transformed into a Gaspard2 *ApplicationComponent* associated with Gaspard2 collaborations. These transformations concern only structural aspects, hence they are very direct.

UML state machines and collaborations. As the metamodel of Gaspard2 state graphs is a subset of the metamodel of UML state machines, the translation is obviously direct and simple. In general, they are one-to-one translations on the condition that the application is only specified with the defined concepts in Gaspard2 state graph. However, a UML graphical tool provides all the features of UML state machine, so the right usage (using only concepts defined in Gaspard2 state graphs) of the graphical tool is very important.

[2]http://www.papyrusuml.org
[3]http://www.magicdraw.com

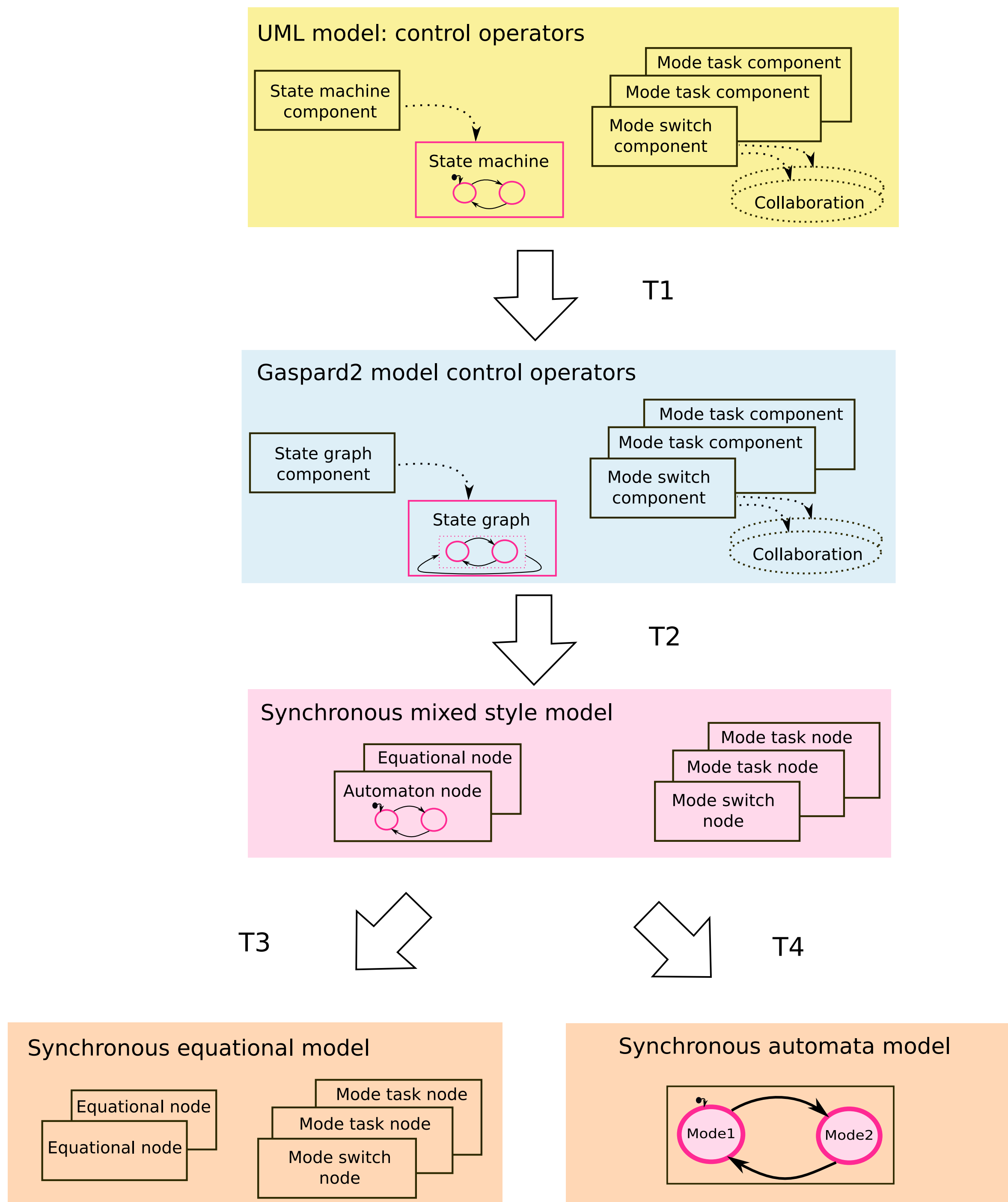


Figure 8.9: The transformation flow from UML model to synchronous equational model

8.4.2 From a Gaspard2 model to a synchronous mixed-style model

8.4.2.1 Application component and its associated state graphs

Similar to the previous Chapter 7, the transformation rules can be illustrated in a hierarchical structure (Figure 8.10). For reason of clarity, only transformations of some key concepts are showed in this figure. The root rule of the transformation involves *ApplicationComponent* that is associated to state graphs. Its sub-rule concerns the *StateGraph* rule, which transforms the state graph associated to the previous component. And It is then followed by *Regions* of the *StateGraph*. *Vertex* and *Transition* rules are invoked in the *Region* rule. *Vertex* can be divided into two sub-rules: *Pseudostate* rule and *State* rule. Finally, *Trigger* rule is a sub-rule of *Transition* rule.

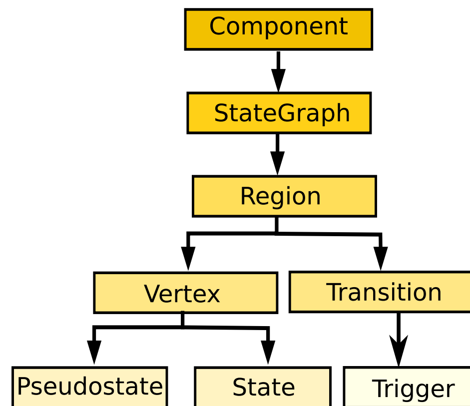


Figure 8.10: The transformation hierarchy of Gaspard2 state graph.

Component associated with state graphs. A *Component* associated with state graphs is transformed into a synchronous **node**. The interfaces of the node are defined by the ports attached to the component. This node invokes the nodes of state graphs (defined in the following rule: *StateGraph*).

StateGraph. A *StateGraph* is transformed into a synchronous node. The interfaces of the node are the same as the node that invokes it (the node from the *ApplicationComponent*). Node(s) of region(s) (defined in the following rule: Region) associated with the state graph are invoked in the *StateGraph* node.

Region. A *Region* is transformed into a synchronous automaton node. Compared to the previous transformation in Chapter 7, the automaton node does not have an **Equation-System**, it contains an **Automaton** that defines the functionality instead.

Vertex. A *Vertex* is an abstract class, so it is not transformed here. However it calls two sub-rules *State* and *Pseudostates*.

State. According to the different types of a *State*, the transformation varies too. A simple *State* is transformed into a synchronous **State**. However, the transformations of a composite state or a submachine state are more complex. Regardless of the different forms of a composite state and a submachine state, they have the same semantics. Hence they

are treated in the same way in the transformation. A composite state or a submachine state is transformed into a **State** first. Then The internal structure in a composite state or a submachine are considered as *StateGraphs*, which can be transformed by using the *StateGraph* rule. Then, the **State** transformed from a composite state or a submachine state is attached with an **Equation**, where the nodes transformed from the previous *StateGraph* rule is invoked.

Pseudostates. The transformation of Gaspard2 pseudostates, such as *Initial*, *deepHistory* and *shallowHistory*, has different rules respectively. Initial pseudostate is transformed into a dependency between the state machine and the corresponding state, namely the initial state. A ShallowHistory pseudostate is transformed into a **History**, which is connected to an automaton and a state. *deepHistory* is also transformed into a **History**. Moreover, it declares all its sub automata have their own **History**.

Reset. The *Reset* concepts in Gaspard2 is transformed into an **Reset** of the **automata**. When an automata has a **Reset**, the **History** is no longer useful in the automata.

Transition and Trigger. Gaspard2 *Transition* and *Trigger* are transformed into synchronous *Transition* and *Trigger* in a direct structural way.

Event, MessageEvent and ChangeEvent. A Gaspard2 *Event*, such as *MessageEvent* and *ChangeEvent*, is transformed into synchronous **BooleanExpressions**.

8.4.2.2 Mode switch component and its associated collaborations

A mode switch component is transformed into a synchronous **Node**, in which several mode task nodes[4] are invoked in a mutual exclusive way. This requires the *match/with*[5] or *if/then/else* control structure in a node. The pattern of the *match/with* or the condition of the *if/then/else* are obviously mode values, and the statements are invocations of these mode task nodes. The correspondence between mode values and mode task nodes is indicated by the *collaborations*, which are associated to the mode switch component. Hence *Collaborations* are transformed into dynamical invocations of nodes.

8.4.3 From a mixed-style model to an equational model

The transformation from a synchronous mixed style model into an equational model aims to obtain a pure equational model, i.e., nodes have only equations (no automata). The latter enables to generate code without explicit automata[6].

As the previous **T2** step has transformed: *a*) corresponding Gaspard2 concepts into nodes; *b*) state graph hierarchy into node hierarchy, the aim of this step **T3** is to transform automaton nodes into pure equational nodes, i.e., only automaton nodes are involved in this step.

[4]A node that implements a mode

[5]Different languages have different forms, this form belongs to Lucid synchrone à la Object Caml. Signal has the form *case/in/end*, C language has the form *switch/case*, etc.

[6]On one hand, in Lustre, there is not explicit automata concepts. On the other hand, oversampled automata could not be expressed by mode automata in synchronous languages (Chapter 6)

There are two aspects of the transformation of an automaton node into an equational node. The first one concerns the transformation of the structure of an automata, which includes states and conditioned transitions. The second one involves the execution semantics of some concepts, such as **History** and **Reset**.

8.4.3.1 Transformation of automaton structure into equations

A very simple way to transform the structure of state-transition based automaton into equations is to use *match/with* statement or *if/then/else* statement. The two transformations are described separately.

Match/with case

In the match/with case, the state/transition can be translated by two levels of nodes in a hierarchical way. The high level is a state-match node, which matches the source state. Once the source state is matched, the corresponding statement invokes a transition-match node. In the latter node, one of all the possible transitions from the source state is matched according to the Boolean expression attached on the transition, as a result the corresponding statement is carried out.

A transition can be a **strong** transition or a **weak** transition, which leads to different transformations. Matou only supports weak transitions, whereas Lucid synchrone supports both. In the following example, both two kinds of transitions are illustrated.

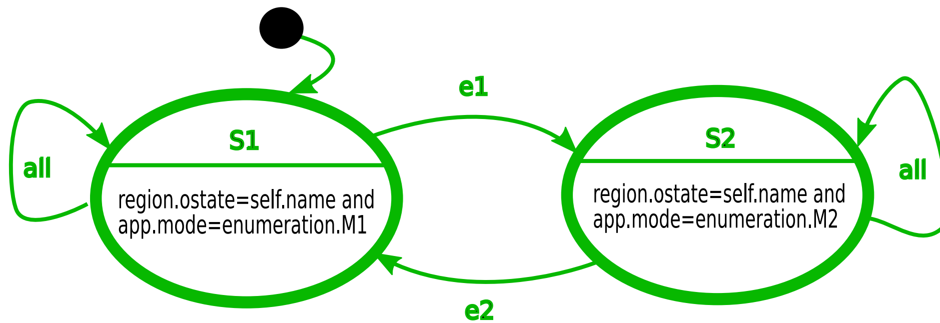


Figure 8.11: An example of automaton to be transformed.

Figure 8.11 shows an example of an automaton which is transformed into equations in the terms of match/with in Figure 8.12 and Figure 8.13. The automaton has two states: *s1* and *s2*. s1 has two outgoing transitions: one transition targets s2 in the conditions that Boolean expression *e1* is true, the other transition targets s1 itself when e1 is not true. Every state has equations attached. For example, *region.ostate = self.name and app.mode = enumeration.M1* in state s1. As the example is a simple state machine, the previous equation is directly translated in the target language: *state = s1 and mode = M1*.

Figure 8.12 (strong transition version) and Figure 8.13 (weak transition version) illustrate the results in a concise way, which try to avoid too many technical details. It adopts key words from Lucid synchrone and Lustre. A node, called *state-match-auto* is a state-match node. It matches the previous value of a signal, called *state*. According to the value of

state, node *transition-match-s1* or node *transition-match-s2* is invoked. In node *transition-match-s1*, expressions, such as *e1*, *e2* are evaluated in the match statement. According to their values, either *(true, _)* (whatever the value of *e2*), either *(_)* (whatever other values), one of the two transitions is fired (their equations are evaluated). The interfaces of these node are not involved for reason of clarity. But they are presented in the next subsection.

```
node state-match-auto
   match pre(state) with
         s1  ->
             do transition-match-s1 done
       | s2 ->
             do transition-match-s2 done
end

node transition-match-s1
   match (e1, e2) with
         (true, _)s1  ->
             do state = s2 and mode = M2 done
       | _   ->
             do state = s1 and mode = M1 done
end

node transition-match-s2
   match (e1, e2) with
         (_, true)  ->
             do state = s1 and mode = M1 done
       | _ ->
             do state = s2 and mode = M2 done
end
```

Figure 8.12: An extract of if/then/else example (without interfaces), which is obtained from the transformation of an automaton in Figure 8.11: strong transition version.

If/then/else case

As the match/with statement can be translated in an if/then/else statement naturally, we will not detail this translation. However we just give an example of transformation of the previous automaton (Figure 8.11). This is an example of the strong transition version (Figure 8.14).

8.4.3.2 Transformation of *InitialState*, *History* and *Reset*

The three concepts *InitialState*, *History* and *Reset* define which state is the starting state when the automaton is activated or reactivated. But there is a priority between them: *Reset* is prior to *History* and *History* is prior to *InitialState*. This order means that if *Reset* is true, the *InitialState* is always the starting state, otherwise, the starting state is indicated by *History* if *History* is set. When *History* is unset, the *InitialState* is always the starting state.

```
node state-match-auto
   match pre(nextstate) with
         s1  ->
             do state = s1 and mode = M1 and
                transition-match-s1 done
       | s2 ->
             do state = s2 and mode = M2 and
                transition-match-s2 done
end

node transition-match-s1
   match (e1, e2) with
         (true, _)  ->
             do nextstate = s2 done
       | _   ->
             do nextstate = s1 done
end

node transition-match-s2
   match (e1, e2) with
         (_, true)  ->
             do nextstate = s1 done
       | _ ->
             do nextstate = s2 done
end
```

Figure 8.13: An extract of if/then/else example (without interfaces), which is obtained from the transformation of an automaton in Figure 8.11: weak transition version.

```
node state-match-auto
   if pre(state)=s1 then transition-match-s1
   else if pre(state)=s2 then transition-match-s2
end

node transition-match-s1
   if e1=true then state = s2 and mode = M2
   else state = s1 and mode = M1
end

node transition-match-s2
   if e2=true then state = s1 and mode = M1
   else state = s2 and mode = M2
end
```

Figure 8.14: An extract of if/then/else example (without interfaces), which is obtained from the transformation of an automaton in Figure 8.11: strong transition version.

As *InitialState* and *Reset* are not changed once the automaton is designed, so they can be modeled by local constant signals. *History* is a little special in the sense that they can be changed according to its context. In fact it is a memory of the last recent active state(s). It can be modeled by using memory operators, such as *pre* (followed by), which supports to get the value of a signal at a previous instance (transition) and -> (default), which gives the default value.

8.4.3.3 A complete example of transformation

Figure 8.15 gives an example in consideration of *InitialState*, *History* and *Reset*. The interfaces are also included. But note that this example is only used to illustrate a skeleton of the resulting code, so it can not be submitted to be compiled. Both match/with and if/then/else are used in this example, but if/then/else has nothing to do with automata, compared to the example in Figure 8.14. This example in Figure 8.15 can also be translated into pure equational code as mentioned before.

The input arguments of the node: *e1* and *e2* are the same as in the previous example (Figure 8.11). *Active* indicates if this automaton is active in case that this automaton is a sub automaton. *Compulsivestate* indicates a compulsive starting state for the automaton no matter what the automaton has as its initial state. This compulsive state is associated with the input state port of a component (SGC). When it is set, a local Boolean signal *compulsive* is set to true. *State* and *mode* are output arguments, which indicate the current state and the corresponding mode after the fired transition.

Local signals are *initialstate, reactive, compulsive, starting state, historystate, history* and *reset*. *Initialstate, compulsive, history* and *reset* are considered as constant signals, whose values are determined at the transformation time, i.e., they can be obtained from the information of the original UML specification. *Compulsive, history* and *reset* are Boolean signals, which indicate the presence/absence of compulsive state, history and reset concepts. *Initialstate* indicates the default starting state of an automaton. *Reactive, startingstate* and *historystate* are dynamically changed according to the execution context. *Reactive* indicates if the automaton is entered or reentered. *Startingstate* indicates the resulting starting state in consideration of all the constraints, including compulsive state, reset, history and initial state.

8.4.4 From a mixed-style model to an automaton model

An automaton model is used to generate code for mode automata or Lucid synchrone. Compared to the previous transformation **T3**, which transforms the synchronous mixed-style model into equations, this transformation **T4** attempts to extract the automata structure from the mixed-style model. The metamodel is very similar to the synchronous mixed style metamodel, except some concepts, such as compulsivestate, which are absent in the automaton model.

Transformation towards Lucid synchrone

Lucid synchrone mode automata enable both weak transitions and strong transitions. Two corresponding examples are illustrated in Figure 8.16 and Figure 8.17 respectively.

In case that *History* is set but *Reset* is not set, the example of Lucid synchrone mode automaton is illustrated in Figure 8.18.

```
node state-match-auto (e1, e2, active, compulsivestate)
 returns state, mode
 var initialstate, reactive, compulsive, startingstate,
       historystate, history, reset
  initialstate = s1
  reactive = if not(true->pre active) and active then true
      else false
  historystate = if  not active and not reset and history
           then initialstate -> pre historystate
      else if active and reactive and not reset and history
           then initialstate -> pre historystate
      else if active and not reactive
           then initialstate -> pre state
      else initialstate
  startingstate = if compulsive then compulsivestate
      else  historystate
  match startingstate with
        s1 ->
           do (state, mode) =
                transition-match-s1(e1, e2, active) done
      | s2 ->
           do (state, mode) =
                transition-match-s2(e1, e2, active) done
end

node transition-match-s1 (e1, e2, active)
returns state, mode
  match (e1 and active, e2 and active) with
        (true, _)s1  ->
            do state = s2 and mode = M2 done
      | _   ->
            do state = s1 and mode = M1 done
end

node transition-match-s2 (e1, e2, active)
 returns state, mode
  match (e1 and active, e2 and active) with
        (_, true)  ->
          do state = s1 and mode = M1 done
      | _ ->
          do state = s2 and mode = M2 done
end
```

Figure 8.15: An extract of the code example, which is obtained from the transformation of an automaton in Figure 8.11. In the first node, i.e., state-match-auto, if/then/else statements, which are considered as primitive statements defined in the programming language, are only used to calculate some local signals. On the contrary, the match/with statement is used to translate the automaton.

```
let node auto e1, e2 = (state, mode) where
  automaton
    s1 -> do state = s1 and mode = M1 until e1 then s2
  | s2 -> do state = s2 and mode = M2 until e2 then s1
  end
```

Figure 8.16: An extract of the Lucid synchrone code example in terms of automaton, which is obtained from the transformation of an automaton in Figure 8.11: weak transition version.

```
let node auto e1, e2 = (state, mode) where
  automaton
    s1 -> do state = s1 and mode = M1 unless e1 then s2
  | s2 -> do state = s2 and mode = M2 unless e2 then s1
  end
```

Figure 8.17: An extract of the Lucid synchrone code example in terms of automaton, which is obtained from the transformation of an automaton in Figure 8.11: strong transition version.

```
let node auto e1, e2 = (state, mode) where
  automaton
    s1 -> do state = s1 and mode = M1 until e1 continue s2
  | s2 -> do state = s2 and mode = M2 until e2 continue s1
  end
```

Figure 8.18: An extract of the code example in terms of automaton with a history, which comes from the transformation of an automaton in Figure 8.11.

Transformation towards Lustre mode automata

When a synchronous mixed model is transformed into Lustre mode automata in terms of Matou, several concepts are not always compatible with the definitions of Lustre mode automata. For instance, the *history* semantic is not supported, hence when reentering an automaton, it acts as there is always a *reset*. Moreover, compared to Lucid synchrone mode automata, only weak transition is supported for Lustre mode automata. Lustre mode automata supports priorities on transitions, but it is not supported in UML state machines. Figure 8.19 shows an example Lustre mode automata transformed from the previous example shown in Figure 8.11.

```
AUTOMATON auto
STATES
  s1 init          [ state = s1; mode = M1; ]
  s2               [ state = s2; mode = M2; ]
TRANS
  FROM s1 TO s2 WITH rien0 [ e1 = true ]
  FROM s2 TO s1 WITH rien0 [ e2 = true ]
PROCESS auto [ in( e1, e2 ), out(state, mode) ]
```

Figure 8.19: An extract of the code example in terms of Lustre mode automaton, which comes from the transformation of an automaton in Figure 8.11.

8.5 Conclusions

This chapter presented the implementation of the state-based control in Gaspard2 (Chapter 6) in conformity to MARTE. The implementation involves: *a*) a proposition of an extension of MARTE profile with UML state machines and collaborations so as to model state-based control; *b*) an extension of Gaspard2 metamodel with state graphs; *c*) an extension of synchronous metamodel with automata; *d*) a transformation chain, which is an ongoing work, from graphical MARTE/UML descriptions to synchronous languages in consideration of language particularities. The targeted languages include Lustre, Lucid synchrone and Signal. Mode automata of these three languages are also taken into account.

This implementation of state-based control enables to generate synchronous model automata. The latter makes it possible to use: *a*) model-checking tools to verify the control correctness with regard to some specifications; *b*) discrete controller synthesis tools for safe control. The usage of these two formal techniques is illustrated in the next chapter with a case study.

Chapter 9

A case study on multimedia cell phone

9.1 Introduction **155**

9.2 Modeling of the example in Gaspard2 . . . . **156**

9.2.1 A global view of the example 156

9.2.2 A macro structure for the video effect processing 159

9.2.3 Repetitive modeling of video effect processing 159

9.2.4 A complete modeling of the phone example. 161

9.2.5 Requirements of formal application verification. 165

9.3 Application validation and analysis **168**

9.3.1 Functional validation and analysis 168

9.3.2 Validation considering non-functional aspects 172

9.4 Discrete controller synthesis **173**

9.5 Related works **175**

9.6 Conclusions **175**

9.1 Introduction

Following the advances in mobile computing in the hardware architecture and energy technology, e.g., smaller physical size, increasing computing capability and less cost, multimedia mobile devices have been spreading rapidly. As a result, these devices augment the number and improve the quality of the functionalities, which contribute to gain of a dominant foot in the commercial market. For instance, a modern cellular phone provides complex functionalities, such as camera, music/video playback, video games, GPS, mobile TV and radio.

Among these applications, those which involve multimedia processing attract a good deal of interest. They are generally DIP applications. For instance, in a camera phone, in general, the image/video processing can be divided into multi-stages [123]: *1*) image capture: images are captured by sensors, such as charged-coupled devices (CCDs) and complementary metal oxide semiconductor (CMOS); *2*) signal processing: images are rendered so as to be processable in the system; *3*) application processing: images are processed according to predefined specifications; *4*) display/storage: images are displayed on the screen or

stored in the flash memory in the cell phone. The signal and application processing stages are considered as DIP processing, which can be modeled in the Gaspard2 framework.

Mobile multimedia applications designed for cellular phones require to take user requirements and QoS into consideration in order to obtain an advantageous market evaluation, e.g., various multimedia effects to meet user's special favor, continuous and dependable service. These requirements are generally considered as, but not restricted to, application *controllability*, *adaptivity* and *dependability* .

- **Application controllability Example.** Users can watch video clips with their cellular phones, which can be obtained from the local memory, an on-line video library or the built-in camera on the phone. While watching a video clip, the phone allows to choose different video effects, such as *B&W* (Black & White), *Negative* (a tonal inversion style of a positive image), *Sepia* (a dark brown-grey color style) and *Normal* (no effect), by using buttons on the phone. Users can also change the resolution of the video, such as *High*, *Medium* and *Low*, or video color, e.g., *Color* and *Monochrome*.

- **Application adaptivity and dependability Example.** In addition to user control, previously mentioned video functionality is also supervised by the phone system to meet some QoS requirements, e.g., continuous and dependable service, minimum or optimal resource consumption. These requirements are fulfilled in consideration of the platform, hardware and environment status, which typically include the status of computing power, available memory, available communication and energy. For instance, communication quality (or available communication) can result in changes of some communication-sensitive video modes in order to obtain a better video quality, or inversely to save energy, etc.

This chapter first presents the modeling of a video effect processing module of a multimedia cellular phone within the Gaspard2 framework, which involves the embedded multimedia software design for a cell phone. The processing in the module corresponds to application processing in the multi-stage video processing. The modeling illustrates the usage of our proposed behavioral modeling constructs, which mainly concerns high-level functionality modeling of the data-parallel intensive video processing. Videos in this case study are modeled as flows of decompressed images. Access of videos through camera sensor, local memory storage, telecommunication networks, etc., are indifferent and the videos are modeled as multidimensional arrays. In addition to the modeling of the multimedia module, formal validation and discrete controller synthesis are also presented in this chapter by utilizing synchronous languages, which contribute to obtain a correct and safe application design.

9.2 Modeling of the example in Gaspard2

9.2.1 A global view of the example

The multimedia processing module in this case study (Figure 9.1) is mainly composed of seven software components, which include:

EnergyStatus : it indicates the energy level according to the events received from the energy detection component (not presented here);

CommQuality : it determines the communication level to be used in the phone, according to energy level notified by the energy component and available communication quality received from antenna component (not included here);

Controller : this component can control or validate mode change requests from the following components according to current mode configuration and available resources, such as energy and communication quality;

VideoSource : it represents the component that choose the right video source according to user change requests and the controller's authorizations, or direct commands of the controller;

Resolution : it contains a state-based control part, which can change the resolution of the video according to user requests and the controller's authorizations, or direct commands of the controller, and a DIP part, which is composed of several modes of processing.

ImageStyle : it is similar to the resolution component, except that it concerns image style processing;

ColorEffect : it is also similar to the resolution component, except that it involves the processing of video color effect;

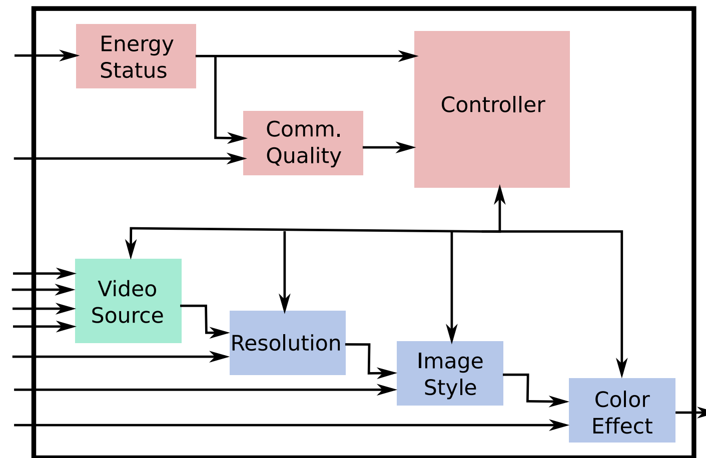


Figure 9.1: A global view of the multimedia processing module in a cellular phone example, which is composed of seven components.

In this example, modes defined in several components can be characterized by some quantitative attributes that represent certain non-functional properties, such as energy, communication quality, computing resource and memory. Table 9.1, Table 9.2, Table 9.3 and Table 9.4 illustrate the non-functional quantitative characteristics that are associated with the automata of color effect, video source, image style and resolution respectively.

The modeling of this example will be illustrated as follows: a typical components, namely the *ColorEffect* component, will be presented first. The Gaspard2 repetition and control modeling is illustrated with this component. Then, a complete modeling is illustrated. Finally, construction of model automata is discussed.

mode name	energy	communication quality	computing resource	memory
Color	30	50	40	20
Monochrome	20	40	25	20

Table 9.1: Non-functional quantitative attributes associated with the ColorEffect automata.

mode name	energy	communication quality	computing resource	memory
Online	30	40	50	20
Camera	30	0	35	25
Memory	20	0	35	30

Table 9.2: Non-functional quantitative attributes associated with the video source automata.

mode name	energy	communication quality	computing resource	memory
Normal	10	0	25	20
Negative	10	0	25	20
B&W	10	0	25	20
Sepia	10	0	25	20

Table 9.3: Non-functional quantitative attributes associated with the ImageStyle automata.

mode name	energy	communication quality	computing resource	memory
Low	30	30	30	10
Media	40	40	40	20
High	50	50	50	30

Table 9.4: Non-functional quantitative attributes associated with the Resolution automata.

9.2.2 A macro structure for the video effect processing

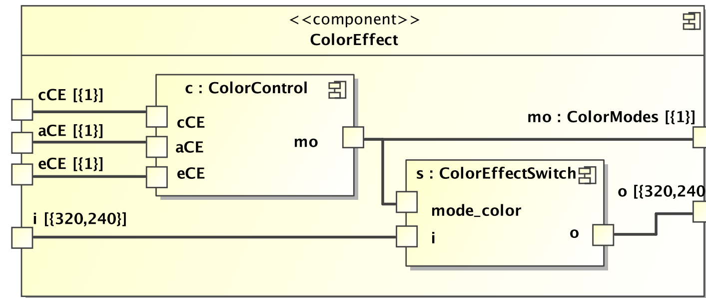


Figure 9.2: The *ColorEffect* component is composed of a state machine component and a mode switch component. The *ColorEffect* component has an interface that includes ports such as *cCE*, *aCE*, *eCE*, *mo*, *i*, and *o*. *cCE*, *aCE*, and *eCE* are ports that convey events for the *ColorControl* component in order to trigger transitions. They will be detailed in Figure 9.5. *i* and *o* are data ports, where the video passes.

The *ColorEffect* component (Figure 9.2), which represents a macro structure, is composed of two components: *ColorControl* (SMC) and *ColorEffectSwitch* (MSC). The first one plays a controller role, and the second one acts as a switch of several modes.

9.2.3 Repetitive modeling of video effect processing

The DIP parts of *Resolution*, *ImageStyle* and *ColorEffect* are similar, hence their modeling is almost the same, except the used filters are different. Here, an example of one mode in Color Effect using the MARTE RSM is shown in Figure 9.3, which shows the data parallelism specification. It illustrates how a monochrome filter (*MonoFilter*) is used for the processing of a [320, 240]-image. *MonoFilter* is modeled as a component that will be deployed with a particular IP. Because it only works on small [8, 8]-pixel patterns, it should be repeated 40×30 times to cover a whole image. The other mode of *ColorEffect*, i.e., the *ColorMode*, can be constructed in the same way, except that the filter *MonoFilter* is replaced by *ColorFilter*.

9.2.3.1 Control concept modeling in Gaspard2

Mode switch component. The mode switch component *ColorEffectSwitch*, which achieves a switch function between modes, has two modes *ColorMode* and *MonochromeMode* in the example in Figure 9.4. It has *mode_color* and *i* as inputs and *o* as outputs. *Mode_color* is a UML *behavior port*, which conveys mode values (e.g., *ModeColor* and *ModeMono* in the example). According to these values, modes are activated correspondingly. The activation behavior is specified by UML collaborations, i.e., the *ColorMode* is activated only in the mode *ModeColor*, which is indicated by the name of the collaboration.

The mode switch components of *Resolution* and *ImageStyle* can be constructed in the same way on the condition that the modes are replaced by the right ones.

State machine component. The state machine component, *ColorControl* in Figure 9.5) serves to determine mode values that are used by mode switch components to execute different computation modes. This component is associated with UML state machines. The

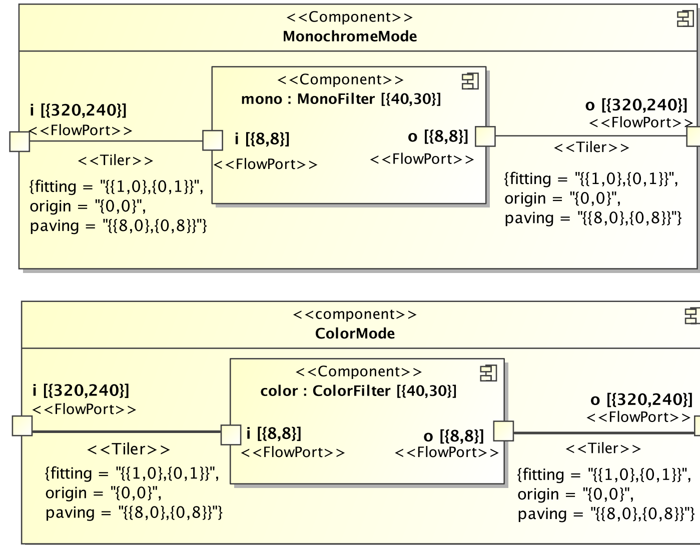


Figure 9.3: The color and monochrome effect filters in repetition context component.

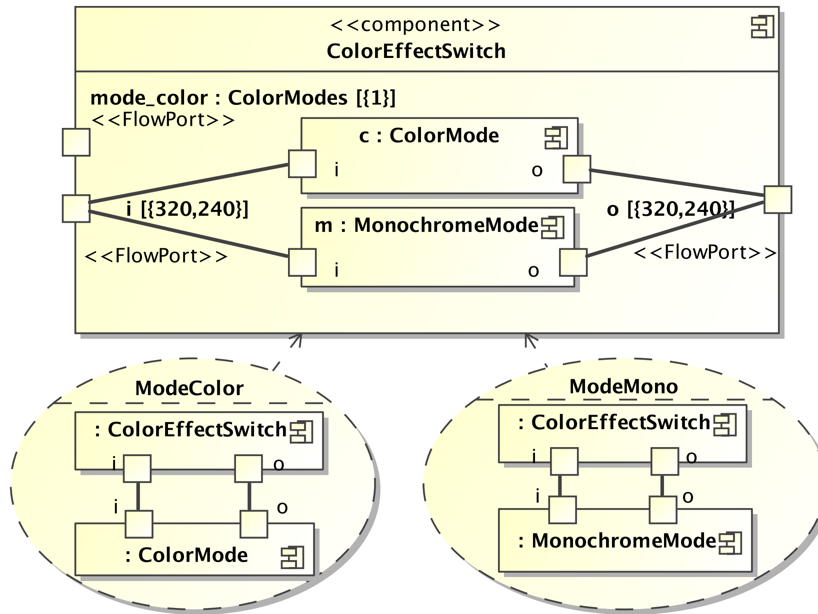


Figure 9.4: UML *collaborations* and their associated MSC. The latter acts as a switch of the color effect in the phone example. The collaboration names are defined in the enumeration *ColorModes*, which is composed of the mode values of this switch.

latter expresses transition functions carried out on the states in the machine. Each state is associated with a mode, e.g., a one-to-one mapping between states and modes. Hence, a state machine component is an ideal complement to a mode switch component. The state machines adopted in Gaspard2 are only a subset of UML state machines, where certain concepts, such as *Constraints* and *Events*, are simplified.

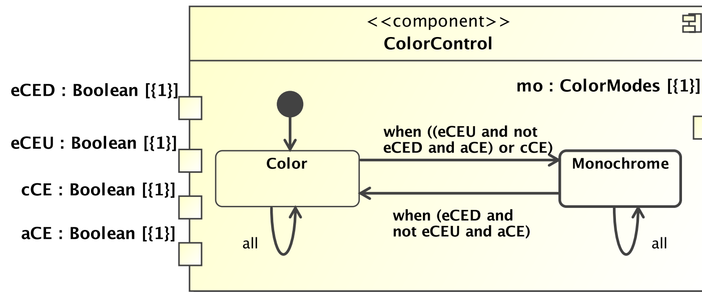


Figure 9.5: The *ColorControl* state machine component, which is associated with a *state machine*, acts as the color effect controller. *eCEU* and *eCED* indicate user's up and down selection commands respectively. They are simplified to *eCE* in Figure 9.2 and Figure 9.7. *cCE* indicates controller's active command for mode change. *aCE* signifies controller's authorization upon user's change mode requests.

As shown in Figure 9.5, a state machine component has an interface that includes the input Boolean ports *cCE*, *aCE*, *eCEU* and *eCED*, and output mode port *mo*. Values from input ports are dispatched to trigger transitions. *cCE* conveys the commands of controller that are used to change mode even without user request. In contrast, *aCE* indicates controller's authorizations in relation to the requests of users (through *eCEU* and *eCED*). *eCEU* and *eCED* indicate up and down (of modes) respectively. Transition conditions are prefixed by *when*. A transition with an *all* tag represents a self-transition. A mode value, which is specified in the *doActivity* in a state, is conveyed through the *mo* port. An important required property of the state machines is determinism, i.e., for each state, input events lead to the firing of at most one transition .

Figure 9.5 shows the state machine defined in the *ColorEffect* component. Figure 9.6 illustrates all the state machines defined in the *Resolution*, *Resolution*, *ImageStyle* and *ColorEffect*. Their controller component are specified in the same way as *ColorControl* in *ColorEffect*.

9.2.4 A complete modeling of the phone example.

With the help of the mode automata, the overall cellular phone multimedia functionality is modeled and shown in Figure 9.7. The component *CellPhoneExample* shows the processing of only one frame, which can be repeated to process a video clip. The ports on the left-hand side of the component represent its inputs, including events that indicate: the image resolution, style or color, the energy level, the computing resource, the communication quality and the image inputs (e.g. local storage, online library, camera). The output of the component is the processed image.

Apart from the previous mentioned component, the *Controller* component is implemented as an IP, which can be manually programmed by the developer. It is used to decide if a request of mode change from users is valid or not according to energy, communication

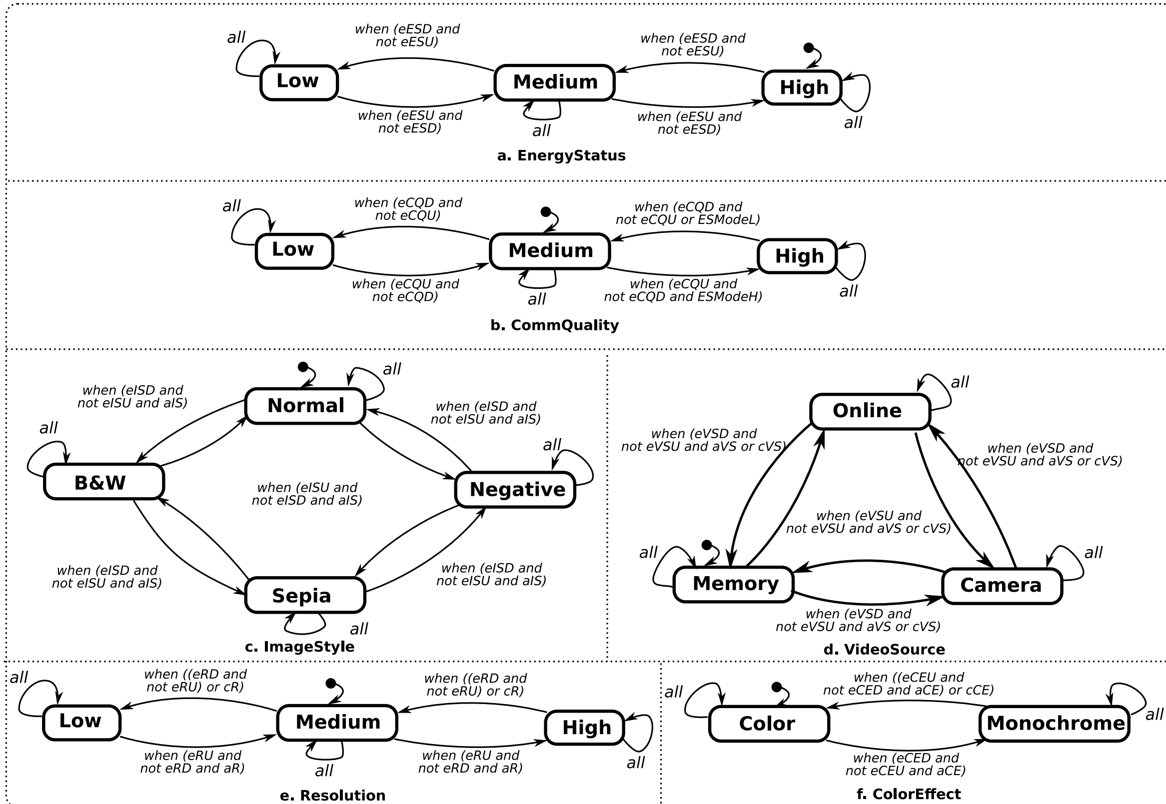


Figure 9.6: State machines associated with *EnergyStatus*, *CommQuality*, *VideoSource*, *ColorEffect*, *ImageStyle* and *Resolution* component.

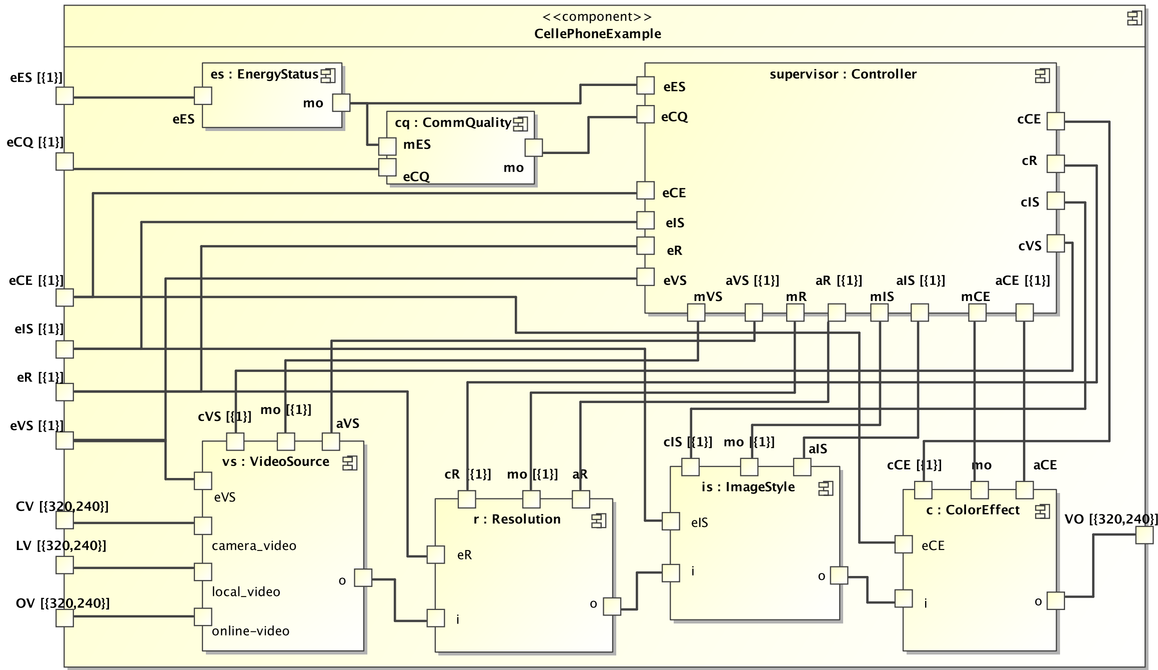


Figure 9.7: The main component of the cellular phone example, which is designed with the UML tools: MagicDraw.

levels and status of other components. The *EnergyStatus* component is composed of a three-states automata, which is used to indicate the energy status according to the energy change events. The *EnergyStatus* component is composed of a three-states automata, which is used to indicate the communication quality level according to available communication quality and energy.

9.2.4.1 Construction of mode automata

Mode automata can be constructed from the previously defined concepts. First, the structure of mode automata is presented by the macro structure as illustrated by the *ColorEffect* component in Figure 9.2. In this example, the state machine component produces mode values consumed by the mode switch component in order to achieve a switch function between the modes. In order to simplify the illustration, *eCEU* and *eCED* are only shown as *eCE*.

Secondly, this typical composition should be placed in a repetition context with at least one IRD specification. The reasons are twofold: *ColorEffect* suggests the processing of one frame of a video clip, so it should be repeated; an IRD specifies the sequential processing of these frames. Hence the repetition context of the *ColorEffect* is a serialized one. In this manner, the mode automata can be built and executable. On the contrary, if this composition is placed in a parallel repetition context, state machines will be translated differently. Figure 9.8 shows the *CellPhoneExample* is placed in the repetition context defined by the *MainRepetition* component. IRDs are not shown in the UML modeling for the sake of simplicity. As state machine components are defined in the *CellPhoneExample*, IRDs will be added by default during the transformation.

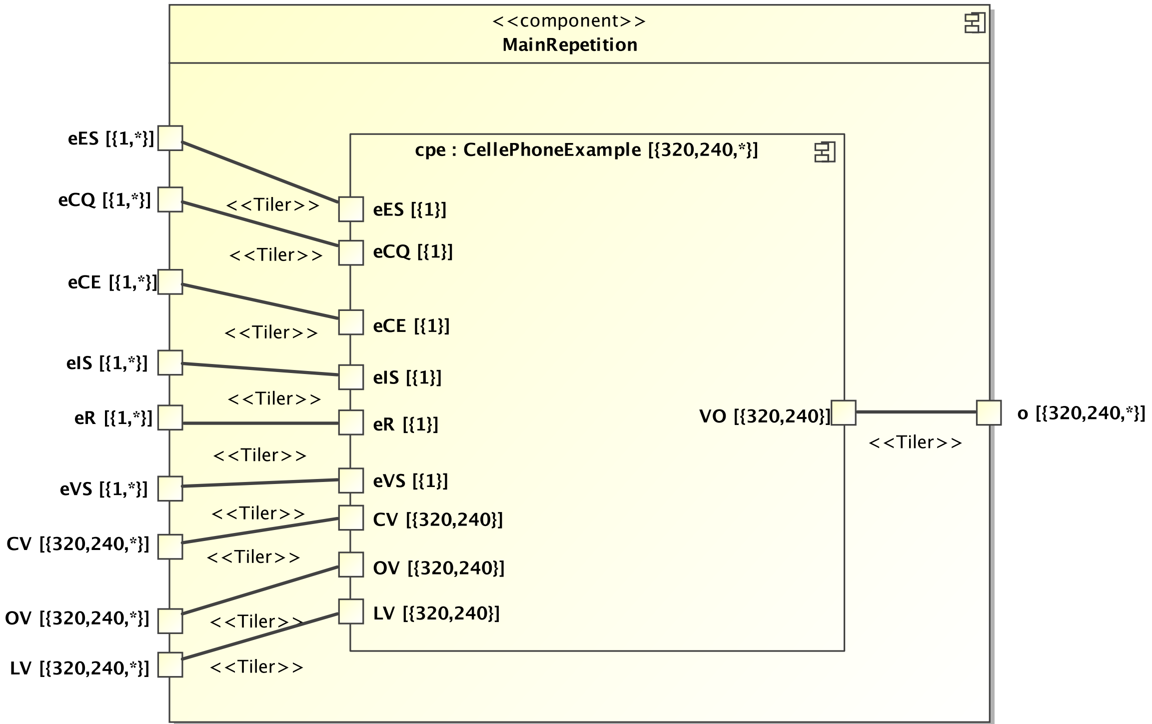


Figure 9.8: The *CellPhoneExample* component is placed in a repetition context component, i,e., *MainRepetition*, which makes it possible to construct mode automata with SMCs and MSCs defined in *CellPhoneExample*. Note that inter-repetition dependencies are considered to be specified implicitly at this repetition level as state machines are defined in *CellPhoneExample*.

9.2.5 Requirements of formal application verification.

Design correctness is one of the main concerns of Gaspard2. Here, we are only interested in high-level validation issues. With the help of the automatically or manually generated code in synchronous languages, it is possible to validate the high-level Gaspard2 models, which is one of the motivations that we connect these two technologies. Figure 9.9 shows the methodology used here for the purpose of Gaspard2 application validation. The overall design process is divided into four stages: *application specification*, *specification implementation*, *executable implementation* and *application verification*.

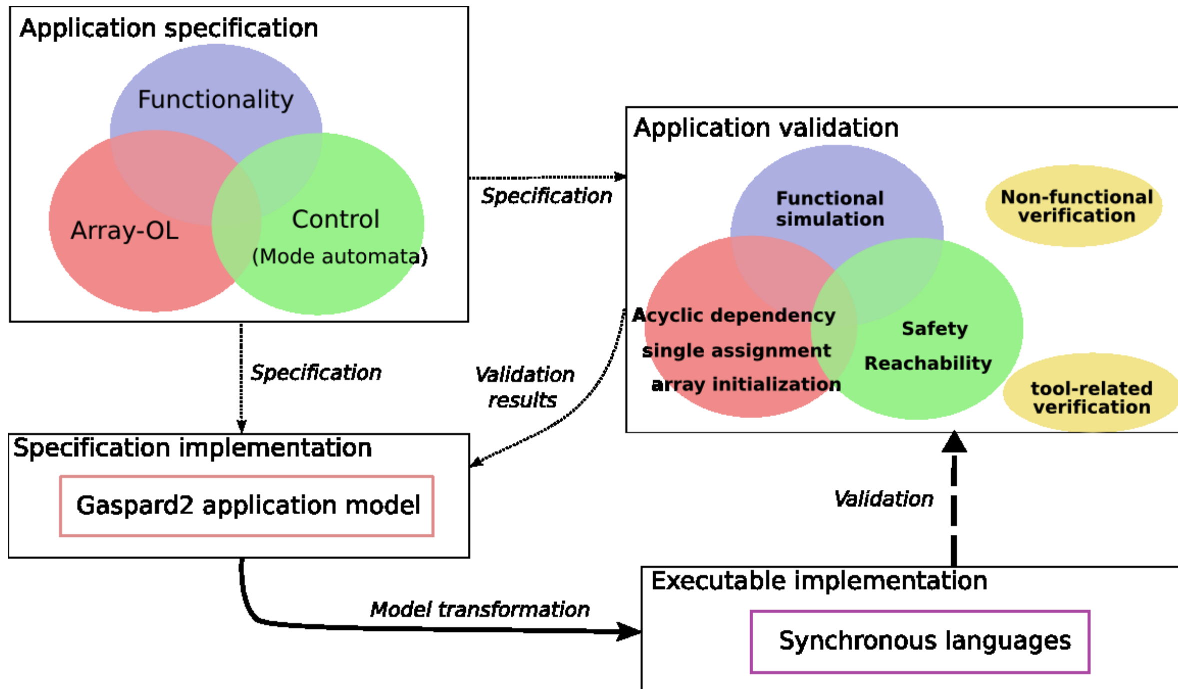


Figure 9.9: The general schema of (software) application validation for Gaspard2.

- *Application specification.* The first stage concerns (software) *application specification*, which is an abstract specification of an application, i.e., although it involves certain specific technologies that will be used to implement the application, concrete implementation details, in particular tool-related details are avoided at this stage. In the context of our work, the specification is considered as a combination of three different aspects: A*rray-OL* (repetitive structure modeling and language properties defined in the Array-OL language and passed to Gaspard2), *Functionality*, *Control* (in the form of mode automata), which are illustrated by three circles in the application specification box in Figure 9.9.
  - A*rray-OL* aspect concerns data parallelism that can be specified by existing DIP specifications (e.g., Gaspard2). For instance, how to use the Gaspard2 ODTs to design the data-parallel computing part of the application.
  - *Control* aspect enables dynamic behavior of the applications. It involves the design of automata and mode selection according to the automata.

  - *Functionality* aspect denotes what is the functionality or usage of an application. For instance, the application is implemented as a downscaler or a black and white filter, etc.

- *Specification implementation.* The second stage is *specification implementation*, which signifies the implementation of the first-stage specification in the Gaspard2 development environment. All the aspects considered in the first stage are implemented by concrete Gaspard2 concepts, which also include MARTE and UML concepts.

- *Executable implementation.* Model transformations are carried out on the *specification implementations* in order to obtain *executable implementations* at the third stage. The execution implementation denotes the generated code in synchronous languages, such as Lustre, Signal and mode automata.

- *Application verification.* The last stage in the design process consists in *application validation*, which is carried out by using compilers and formal verification tools built around synchronous languages. According to the different aspects of applications introduced in *application specification*, a similar partition of the validation is presented here.

  However, as implementation languages and tools (e.g., UML) are involved in the process, additional verifications are required to verify the properties related to these tools. In addition, some non-functional properties can also be checked if some non-functional aspects are considered. These last two verifications are represented by two ellipses in the application verification box in Figure 9.9. A detailed discussion about these verifications are given hereinafter:

  - A*rray-OL*-related verification:
    * *data dependency analysis* contributes to find cyclic dependency in the application, which is not allowed in Gaspard2. Data dependency analysis corresponds to causality analysis in synchronous languages. The causality issue is caused by the self dependency under the instantaneous semantics of synchronous languages. Two approaches of causality analysis are distinguished according to the different mechanisms defined in Lustre and in Signal. In Lustre, specifications are analyzed by the compiler syntactically, and those which have potential causality issues are rejected by the compiler. In Signal, the compiler considers as well a clock analysis to determine the causality, which provides a finer analysis than that of Lustre. This analysis helps to check the existence of cyclic dependencies specified in Gaspard2. They are not always obvious to be detected because the specification and component hierarchy can probably conceals the potential real data dependency (or dependency in execution).
    * *single assignment* indicates that no data element is ever written twice but it can be read several times. This property can be easily checked by compilers of Lustre and Signal.
    * *array initialization* may cause problems when there is only partial initialization of Gaspard2 arrays, e.g., the non-initialized array elements are used in the following processing. Lustre and Signal are distinguished while addressing such an issue. Lustre imposes complete initialization, hence the compiler

rejects the programs that have partially initialized arrays. Whereas Signal enable to fill the non-initialized array elements with default values.

- *Control* introduction in the system benefits application flexibility and adaptivity, however, safe control of the application is not always ensured. Correctness verification is therefore necessary. Control-related (pure functional) verification involves safety and reachability. The former can be checked if an invariance of system states are defined for the model checking. Reachability of certain system status can also be checked.

- *Functionality* can be checked by functional simulation through the simulators provided by synchronous language. Simulation enables both functional and non-functional verification, performance analysis, etc. Functional simulation allows the verification of the application correctness during its execution. Both Lustre and Signal provide simulators. Typical examples of Gaspard2 are image processing, such as rotation and filtering, whose processing can simulated and the result can be checked through some image display tools.

- U*ML-related verification* involves the problems of good usage of UML concepts, such as class, component and data types. This issue is not addressed in this thesis, but a related work [25] proposes to use OCL for the UML model validation.

- *Non-functional aspects* involve the problems of the incompatibility with the environment of the system, hardware architecture, etc. For instance, the synchronizability between components in consideration of environment constraints, execution time, etc.

  - synchronizability: Gaspard2 application components can be deployed onto different hardware nodes, which may have different rates, the synchronizability can be checked by the Signal compiler if the rates of these hardware nodes can be expressed by related clocks [47, 46].
  - resource load: resources, such as computing power, energy, communication and memory, can be modeled in a quantity manner, which can be integrated in the automata-based systems. Then resource load related safety and reachability can also be checked by model checkers [136].
  - execution time: certain non-functional simulation is also studied in the Signal language, such as performance evaluation for temporal validation [70]. Temporal information is associated to the Signal program, from which an approximation of the program execution time can be calculated.

Some of the previous verifications are guided by the *application specification* of the first stage, which can be considered as verification objectives. Consequently, the validation results are used to expose the problems between *application specification* and *specification implementation*, for instance implementations do not comply with their specifications. In this case, the specification implementations should be modified according to the original specifications and the validation results. Note that this process can be iterated until the conformance relation is satisfied.

9.3 Application validation and analysis

9.3.1 Functional validation and analysis

This section involves data-dependency analysis, functional simulation and verifications of single assignment and array initialization. As the last two ones can be checked directly by compilers of Lustre and Signal, they are not detailed here.

9.3.1.1 Safe array assignment

Single assignment is a basic property of Gaspard2. It indicates that no data element is ever written twice but it can be read several times. This constraint can be easily checked by compilers of Lustre and Signal. Another concern is the partial initialization of Gaspard2 arrays that may cause problems when the non-initialized array elements are used later. Lustre and Signal provide different way to address such an issue as mentioned previously.

9.3.1.2 Data dependency analysis

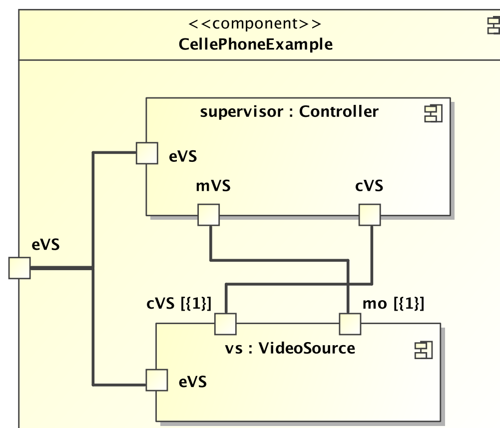


Figure 9.10: A cyclic data dependency example in the cell phone.

Currently, dependency analysis of Gaspard2 specifications is still not available in any Gaspard2 tools. Hence, we propose to do it with the help of synchronous language compilers. Causality analysis have been experimented with different Lustre programs that are automatically generated from Gaspard2 models in order to verify the absence of dependency cycles. For instance, in this case study, if we take a closer look at the dependencies illustrated in Figure 9.7, some cyclic data dependency specification can be found. One of them is illustrated in Figure 9.10. The dependencies between the four ports, e.g., *mVS* and *cVS* of *supervisor*, and *cVS* and *mo* of *vs* form a dependency cycle. However, this cycle maybe a false cycle, e.g., Figure 9.11 shows a possible implementation of *supervisor* and *vs*, where these two components are illustrated with their internal structure. Hence, the analysis can be carried out in a finer grain manner. In this example, the dependency cycle does not exit any more.

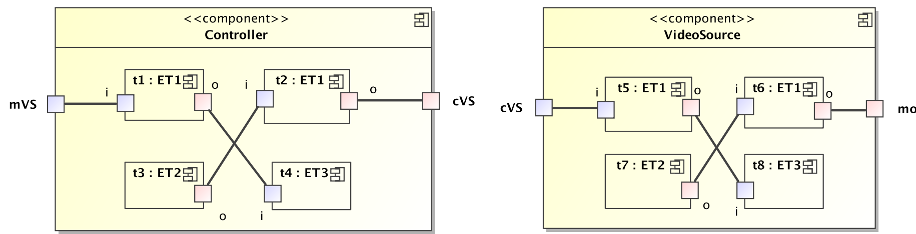


Figure 9.11: Causality analysis

Dependency in specification or dependency in execution? From the previous analysis, the dependency specified in Gaspard2 can be divided into two families: dependency in specification and dependency in execution (or dependency at run time). Obviously, the cyclic dependency that is to be avoided in Gaspard2 is the cyclic dependency in execution, as only this kind of dependency causes problems, such as the deadlock problem. Gaspard2 cyclic dependencies in specification does not necessarily lead to cyclic dependencies in execution. Synchronous languages help to find cyclic dependencies in execution when interface specifications of the components are given. These interface specifications describe the routing of the dependencies at run time.

The dependency cycle illustrated in 9.10 shows a cyclic dependency in specification, which can be easily broken by add a *pre* operator to the input port of *Controller* or *VideoSource* in their implementations. This results in a different dependency routing that resolves the cycle problem.

9.3.1.3 Functional simulation

In order to verify the functional correctness of applications, the functional simulators are then used [134]. The simulation is carried out through the graphical simulator SIMEC [79], which is distributed with the Lustre environment.

9.3.1.4 Model checking

The cellular phone example has been modeled within the Gaspard2 environment. The correctness of the controller can be checked by model checking first. As this verification only involves control-part of the application, data parallel computation is not necessary, so they can be removed thanks to the control-data separation. Hence, only control part of the example is translated into model automata. The *ColorEffect* component is taken as an example to illustrate the translation. Other components are translated in the same way.

Translation into mode automata. The *ColorEffect* component can be translated into mode automata [137]. However, as the objective of this translation is to obtain the automata defined in the application, which are required to be verified, the data computation part, i.e., the mode switch component *ColorEffect* component, which is not necessary in the verification, is removed form the generated code. The translation result is illustrated in Figure 9.12.

Other components, which are similar to *ColorEffect* can be translated in the same way, hence, we can obtain a system composed of the mode automata of *Energy*, *CommQuality*,

```
AUTOMATON ColorEffect
STATES
Color        init [CEModeC = true; CEModeM = false;]
Monochrome        [CEModeC = false; CEModeM = true;]
TRANS
FROM Color TO Monochrome  WITH rien0
            [ (eCEU and not eCED and aCE) or cCE ]
FROM Monochrome TO Color  WITH rien0
            [ eCED and not eCEU and aCE ]
```

Figure 9.12: The automaton of *ColorEffect* in the format of Targos. This automaton is extracted from the *ColorControl* component. For the sake of simplicity, it is called *ColorEffect*.

Controller, *VideoSource*, *Resolution*, *ImageStyle*, *ColorEffect*. The *Controller*, which avoids the concurrent occurrences of the *B&W* and *Color* states, is coded in equations (Figure 9.13).

```
aCE = not(CEModeM and  (ISModeB or
          (ISModeN and eISD and not eISU and aIS) or
          (ISModeS and eISU and not eISD and aIS)));
aIS = true;
aR = not(RModeM and
  (CEModeC or (CEModeM and eCED and aCE)) and
  (ESModeL or (ESModeM and not (eEnergyU and not eEnergyD))
  or (ESModeH and (eEnergyD and not eEnergyU))));
aVS = true;
cR = RModeH and
  (CEModeC or (CEModeM and eCED)) and
  (ESModeL or (ESModeM and not (eEnergyU and not eEnergyD))
  or (ESModeH and (eEnergyD and not eEnergyU)));
cCE = (ISModeN and eISD and not eISU and aIS) or
  (ISModeS and eISU and not eISD and aIS);
cVS = false;
```

Figure 9.13: The controller manually coded in equations.

Once all the mode automata enumerated previously are translated and regrouped in an automaton system, the interface and composition structure can be defined for the system (Figure 9.14). All the outputs of the automata are declared in the `out` statement, and all the inputs, which exclude the outputs, are declared in the `in`. All the automata used in the example are composed together in a parallel way, which is specified by the `PAR` statement. The complete specification of these automata and their composition can be found in Appendix B.

Model checking through Sigali. Figure 9.15 shows the tools used in the validation work. Matou automata are first specified in *Targos* format (with explicit state and transition declarations), which can be compiled into polynomial dynamical systems over $\mathbb{Z}/3\mathbb{Z}$ ({-1, 0, 1}). The latter is used by Sigali to carry out model-checking.

```
PROCESS CellularPhoneExample [ in(CEU,eCED,...: bool),
  out(CEModeC=true,CEModeM=false,...,
        aCE=true,cCE=false,...:bool)]

PAR
        RAFF ColorEffect
        RAFF EnergyStatus
        RAFF CommQuality
        RAFF Controller
        RAFF VideoSource
        RAFF Resolution
        RAFF ImageStyle
        RAFF ColorEffect
ENDPAR
```

Figure 9.14: Declaration of the interface and composition of automata.

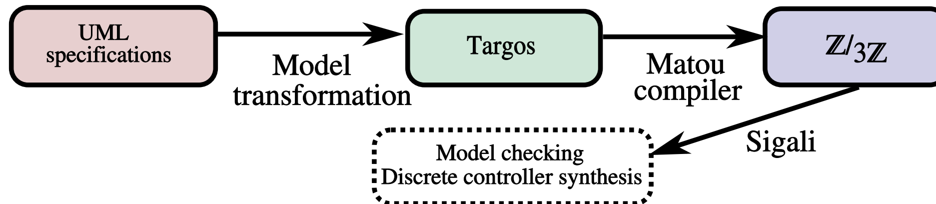


Figure 9.15: The tools involved in the model checking.

The previous automaton system is compiled into the polynomial system, which is represented by $\mathbb{Z}/3\mathbb{Z}$. This system is then used in the Sigali, together with some specified properties to be verified, the model checking can be carried out. Figure 9.16 shows this process in Sigali. A proposition, called `Prop1`, is specified in the verification. This proposition is true when the two states `B&W` and `color` are active at the same time. Then the reachability can be checked through `Prop1` on the system `S`. The model checker answers `false` as these two states are not reachable at the same time.

```
read("task.z3z");
S : processus(conditions, etats, evolutions,
   initialisations, [gen(contraintes)],controlables);

read("Property.lib");
read("Verif_Determ.lib");
p1 : BNW_de_ImageStyle_3 and Color_de_ColorEffect_2;
Prop1 : B_True(S,p1);
Reachable(S,Prop1);
```

Figure 9.16: The model checking process carried out in Sigali.

9.3.2 Validation considering non-functional aspects

9.3.2.1 Model checking considering non-functional aspects

The previous section shows the classical model checking of the translated automaton system, which include verification of some safety and reachability properties. This section presents the model checking considering non-functional aspects, which is represented by quantitative attributes of used resources, for instance, resources provided by the environment, the platform or the hardware. These kinds of verification helps to handle the complex system evaluation at a low cost in a fast way with regard to other methods that take all the elements (software, hardware, etc.) in the system into account.

Cost functions. In Sigali , we encode non-functional properties, e.g., cost requirements on energy, communication quality, computing resource, and memory, by utilizing a mechanism of *cost functions*, which can be associated with Boolean states or *event variables*. A cost function associated with a state defines a set of quantitative attributes with regard to state status, for instance, *active* and *inactive*. A global cost can also be defined with the composition of states, e.g., by adding local costs of parallel states. A bound can be associated with a cost function and one can check invariance of the property that *the global cost is less than the bound in all reachable states*.

In the cellular phone example, we associate cost functions with each state of the automata specified in Figure 9.6. For instance, the color effect automaton has two states, each state is associated with cost functions that characterize its required energy, communication quality, computing resource and memory. These cost functions describe the bound, in the form of percentage, of used resources by each state when it is active. For *Color* and *Monochrome* states, the cost functions are respectively (30, 50, 40, 20) and (20, 40, 25, 20) with regard to

the previously mentioned four resources (Table. 9.1). (0, 0, 0, 0) and (0, 0, 0, 0) are specified for these states when they are inactive. The sum of the resource consumption of the color effect filter is computed directly from these cost functions. Other cost functions defined for the automaton of video source (Table 9.2), image style(Table 9.3) and resolution (Table 9.4). (0, 0, 0, 0) are defined as the cost function of all inactive states in these automata.

By associating each state of the automata in Figure 9.6 with similar cost functions, a global cost function has been computed when composing these automata in parallel. This global function defines for each resource, the sum of its cost functions computed from all possible combinations of active states in the global automaton. A bound is then specified, for example (90, 90, 90, 90), in the system. Finally, a reachability of certain states is checked under this configuration of resource costs.

Verification of correctness of the controller. As cost functions are associated with the states of the automata, the *controller* can then be verified in consideration of these cost functions [136]. Figure 9.17 shows an extract of this checking, which only takes energy resource into account:

CE_Color	$: a_var(Color, 0, 30, 0);$	L1
$CE_Monochrome$	$: a_var(Monochrome, 0, 20, 0);$	L2
$CE_ColorEffect$	$: CE_Color + CE_Monochrome;$	L3
CE_GL	$: CE_ColorEffect + CE_ImageStyle + ...;$	L4
MAX_En	$: 110;$	L5
$CE_Limitation$	$: a_sup(CE_GL, CE_MAX);$	L6
$Reachable(S, CE_Limitation);$		L7

Figure 9.17: An extract of Sigali command for the modeling checking.

a_var command in Sigali takes four parameters: the name of the state, the cost value when the state is absent, the cost value when it is present and true and the cost value when it is present and false. The energy cost functions of `Color` and `Monochrome` (states of the ColorEffect automaton) are specified in L1 and L2. `CE_ColorEffect` (L3) represents the cost function associated with the automaton `CE_ColorEffect`, which is a sum of the cost functions of the states of `CE_ColorEffect`. `CE_GL` (L4) is the cost function of all the automata in relation to energy resource. L5 specifies a bound of the energy resource, and L6 applies this bound to the `CE_GL`. The reachability can then be checked, i.e., certain states, which uses more energy resource than the bound, can be reached even if a controller is used to control the system. The model checker answers *false* to the reachability analysis in this example, i.e., the controller is a correct one.

9.4 Discrete controller synthesis

From the previous example, a controller that takes all the bound of resource usage is not easily programmed manually. Thus, a controller that can be synthesized into the system and guarantee all the expected properties are highly demanded. Discrete controller synthesis (DCS) [85] is proposed to answer this demand. In this case study, a controller can be synthesized in the system, which contributes to guarantee that the pre-defined bounds are not surpassed. This controller is not the *Controller* presented in the cellular phone example. In

order to distinguish the two controllers, the former, which is synthesized into the system, will be called *supervisor* form now on. Therefore, the controlled system is a correct system as the supervisor is used to guarantee the expected properties.

The previous example are still used for DCS. Particularly, the automata and the cost functions associated to their states. Figure 9.18 shows the change of some output variables in the model checking, which are changed to the controllable variables in the system. These controllable variables, which is prefixed by `*`, can be controlled by the supervisor. In addition, the controller is removed from the system, because a supervisor will be integrated into the system.

```
PROCESS CellularPhoneExample [
 in(eESD,eESU,eCQU,eCQD,eCEU,eCED,eRU,eRD,
     eVSU,eVSD,eISU,eISD,*aCE,*aIS,*aR,*aVS,
     *cCE,*cIS,*cR,*cVS: bool),
 out(ESModeH=true,ESModeM=false,ESModeL=false,
    CQModeH=false,CQModeM=true,CQModeL=false,
    CEModeC=true,CEModeM=false,ISModeN=true,
    ISModeB=false,ISModeE=false,ISModeS=false,
    VSModeC=false,VSModeO=false,VSModeM=true,
    RModeH=false,RModeM = true,RModeL=false:bool)]
PAR
        RAFF EnergyStatus
        RAFF CommQuality
        RAFF ColorEffect
        RAFF ImageStyle
        RAFF Resolution
        RAFF VideoSource
ENDPAR
ENDTARGOS
```

Figure 9.18: Declaration of the interface and composition of automata.

Figure 9.19 shows the commands used in Sigali for DCS. L1 to L2 specify a proposition, which the two states, *BNW* and *Color* are exclusive. L3 synthesizes a supervisor that satisfies this property of exclusivity. L4 to L6 specify the property that the energy used used by the system will not go above current energy status indicated by the EnergyStatus automaton. L7 synthesizes a supervisor that satisfies this property. In order to verify the synthesized controller, a simulation tool, called Sigalsimu, can be used. In the simulation, the supervisor does not allow the simultaneous occurrences of certain states by forbidding some controllable events, e.g., when $ColorEffect$ is $Color$, $ImageStyle$ can not be changed to $B\&W$ through controlling the event aIS.

Reachability checking. The resulting system of DCS are guaranteed to be correct in relation to some properties (we can call them *supervisor properties* for short). However, it is still interesting to check other properties, such as reachability, of the new system. As these properties are not necessarily included in the supervisor properties, it is feasible to carry out the

$p1$:	$BNW_de_ImageStyle_3$ and $Color_de_ColorEffect_2$;	L1
$Prop1$:	$B_False(S, p1)$;	L2
S:	$S_Security(S, Prop1)$;	L3
CE_Prod:	$CE_EnergyStatus$;	L4
CE_Global:	$CE_Cons + CE_Prod$;	L5
$CE_Limitation$:	$a_inf(CE_Global, 0)$;	L6
S:	$S_Security(S, CE_Limitation)$;	L7

Figure 9.19: An extract of Sigali commands for discrete controller synthesis.

check. A similar work can be found in [48]. In the phone example, a tenth command (L8) can be appended in Figure 9.19 so that the reachability of certain states (*Color* state here) can be checked on $S1$.

$$Reachable(S1,\ B_True(S, Color));\ \text{L8}$$

9.5 Related works

The OMEGA project [51] provides a model-based framework for real-time and embedded systems. It proposes the OMEGA UML profile, which allows to specify functionality, timing, architecture, etc. A set of analysis and verification tools have been associated with the profile, which involve syntax checker, model checkers, proof-based tools, etc. Compared to OMEGA, we concentrate on DIPs, and adopt model transformations to bridge UML and synchronous validation tools.

DIPLODOCUS [9] adopts a similar approach. Applications can be specified using DIPLODOCUS UML profile, from which LOTOS or UPPAAL code can be generated. The last two languages enable formal verifications to check certain given properties, such as presence of deadlock, reachability, liveness. However, DIPLODOCUS does not adopt an MDE approach or a standard UML profile, compared to our work.

Compared to some previous work on model checking of UML state machines and collaborations [117, 73], our work makes several distinctions: non functional aspects are considered in the model checking. Hence we can reach a wider scope, extending from strictly Boolean properties, i.e., some non-functional properties are handled using cost functions associated with states. The ongoing automatic MDE model transformation bridges the gap between UML models and synchronous languages; other static verification can also be carried out besides model checking; UML state machines and collaborations are applied in a dataflow context, hence mode automata semantics of state machines is adopted.

9.6 Conclusions

In this chapter, a multimedia processing functionality on cellular phone is modeled in the Gaspard2 environment. The modeling of DIP and control is illustrated with concrete examples. Based on this cellular phone example, Gaspard2 validation has been carried out and illustrated, which include formal analyses and verifications of pure functional and non-functional properties. These analyses and verifications are carried out at a high level, as a result it allows a fast verification feed-back rhythm for the application design.

Although many tools associated with synchronous languages are used, users are not required to know much about synchronous languages to achieve the validation. But at least, they need to be acquainted with the specification of the properties to be verified and to know how to use the verification and synthesis tools.

Conclusions

The work presented in this dissertation is carried out in the context of SoC design dedicated to DIP applications, particularly in the Gaspard2 environment. It mainly involves the synchronous modeling of DIP applications, which bridges Gaspard2 specifications with synchronous languages. The latter enables high-level formal validation of Gaspard2 specifications. Reactive control modeling for Gaspard2 is also described, which is based on a previous control proposition. An MDE-based approach contributes to support the implementation of the previous mentioned work, which includes modeling and model transformations.

Contributions

Synchronous modeling. We first propose the synchronous modeling of DIP applications, which leads to an intermediate parallel model, i.e., the synchronous equational model, between data-parallel applications and synchronous data-flow languages. On one hand, this model preserves the properties of original Gaspard2 specifications, such as multidimensional array data structure, data dependency, single assignment, task parallelism, data parallelism, etc. The preserved properties contributes to guarantee that verifications carried out on this model or the executable code generated from this model is also valid for the Gaspard2 model. On the other hand, it only keeps common aspects of synchronous languages (Lustre, Signal and Lucid synchrone) so that it is simple and generic, hence the complexity and particularity of target languages is not involved in this model. It enables to generate these three languages with only one model. While the parallel model is a direct modeling of original Gaspard2 models, two other models are also proposed, e.g., serialized and partitioned models. These two models can be obtained by different space-time mappings in consideration of the non-functional constraints. These two models offer certain refined view of the basic parallel model at a high level.

A major objective of the synchronous modeling is that the resulting models allow the formal validation of high-level Gaspard2 models, which is indispensable for the safe application design. As low-level validation, e.g., SystemC simulation or circuit verification, is considered to be costly, a high-level validation helps to reduce the validation cost by finding faults at high-level.

Model transformations. According to the synchronous modeling, a synchronous metamodel has been proposed. Corresponding model transformations from DIPs specified in Gaspard2 into synchronous languages have also been developed in the Eclipse environment, through an MDE approach. A flow of design is also illustrated: the UML profiles (particu-

larly Gaspard2 profile) are used for the high-level and heterogeneous modeling (application, architecture, association, deployment, etc.) of DIP applications; the graphical design is transformed into executable code in synchronous languages through a chain of model transformations. In this transformation, the Gaspard2 model, the synchronous model act as intermediate models; execution or validation can then be carried out on the resulting code.

Reactive control modeling. Another main contribution is the reactive control modeling of Gaspard2. Gaspard2 control mechanism is first proposed in [71] based on mode automata. However this proposition has some constraints, e.g., lack of formal semantics and parallel and hierarchical composition operators, etc., hence an extension and improvement are proposed here. The Gaspard2 control is first discussed at a high-level, which is independent from any execution models. The change of tiler, task, array and repetition space is presented separately. Based on these discussions, an extension proposal for the Gaspard2 control is presented, where formal semantics is given, parallel and hierarchical composition is formally defined. The control introduced in Gaspard2 remains generic and high-level, because no execution model involved. The control can be projected onto different execution models, for instance, the synchronous execution model, which is also discussed. The problem of synchronization of control flow and dataflow is illustrated and discussed with examples. However, not all state graphs of Gaspard2 can be translated into synchronous automata, particularly the state graphs that enable parallel execution. Once the Gaspard2 control is transformed into synchronous automata, formal techniques, such as model checking and discrete controller synthesis can be carried out for the purpose of safe design.

The graphical implementation of the Gaspard2 control in accord with MARTE is presented. This implementation is based on the UML state machines and collaborations. Extensions of Gaspard2 metamodel and synchronous metamodel and a transformation chain (not implemented yet) from graphical MARTE/UML descriptions into synchronous languages are also described. The targeted languages include Lustre, Lucid synchrone and Signal, and mode automata.

Formal validation and analysis. While the synchronous model is suitable to express data-parallelism and task parallelism in DIP applications, the code generated from this model allows to reason about critical design properties of these applications, i.e., formal validation and analysis based on the generated code enable the safe design of Gaspard2. The validation and analysis involve single assignment, acyclic dependency, array initialization, model checking, etc. The first three is carried out according to the corresponding properties defined in Gaspard2, and the fourth one is intended to check the safe control in Gaspard2 applications. Functional simulation is also illustrated with the generated code through the SIMEC simulator available in the Lustre environment. It contributes to check the functional correctness of the application. Non-functional aspects related verification, particularly model checking, is equally shown. It is used for the check of some properties, such as invariance and reachability under the non-functional constraints of the system. Discrete controller synthesis is also involved in non-functional aspects. By synthesizing a proper task controller, this technique can be used to enforce safety properties in the system. Finally, all these verification and analysis results can be exploited by Gaspard2 users to achieve a correct application design.

In order to illustrate the previous work, a case study of a multimedia processing func-

tionality on cellular phone is presented. The case study is modeled with the repetition and control operators in the Gaspard2 environment. Some verifications and analyses are also carried out on this case study.

Advantages of MDE The presented implementation in the framework of MDE shows the advantages of the MDE approach: firstly, it simplifies the modeling of DIP applications by using simple but standard UML notations, including its extension (Gaspard2 profile). In addition, the simple organization of these notations in a UML or object manner liberates users from heavy syntax and grammar of classical languages. Secondly, the model transformation remains efficient and flexible with regard to classical compilers. As intermediate models can be introduced in this transformation, the complexity of the transformation can be divided according to separation of concerns, i.e., one transformation addresses one certain problem. Hence each transformation can be kept simple. Another advantage is that modifications of an intermediate model will not lead to the modifications of all transformations, hence it makes it possible to follow the modern rapid software evolution. Using transformation rules is another advantage, because they are modular and hence easy to maintain. These rules are defined to be declarative, which include input pattern, output pattern and the transformation relation. However, imperative aspects can also be specified to enhance the processing capacity of rules. Finally, the tools associated with or dedicated to MDE have been dramatically increased, which provide a good support for the MDE-based development.

Perspectives

Code optimization. The presented transformation chain shows promising results. However, a main limitation involves the size problem of resulting synchronous models, which can be very huge due to the explicit instantiation of Gaspard2 data-parallel constructs. As these instances are supposed to run in parallel, hence no loop statements are introduced. Consequently, it leads to big number of repetitive equations. In implementation, the Eclipse plugin, which is used to generate these equations, also suffers from this problem, as too much memory is used for the computing of patterns. A serialized model, where repetitions can be specified using a loop-similar operator, can help to reduce the repetition number. Moreover, external files can be used in the Eclipse plugin to store information of computed patterns in order to reduce the memory load of this plugin.

Automatic control transformation. One of our ongoing works concerns the extension of the transformation chain [135] with control concepts. This mainly includes an extension of metamodels and transformation rules with UML state machines and collaborations. The metamodels of Gaspard2 model and synchronous model with control concepts are very similar to the metamodel of UML, hence the transformation is almost a direct one. The UML collaborations are not transformed in a structural way, because they only express the dynamic behavior of the cooperating elements. The code generations from these models are different, for instance, when pure equations are needed, UML state machines will be transformed into if/then/else statements, etc. But when automata are needed, UML state machines will be transformed into mode automata. As the code examples are already illustrated, the extension is not a different work.

Control application in FPGA and SystemC. The presented control is mainly applied in the context of synchronous reactive systems for the purpose of application control validation, however, it is possible to apply it in other context, such as FPGAs and SystemC for application control. A simple version of the control has been developed [75] for the control of applications implemented on FPGA accelerator. However, it is not involved in the reconfigurability of FPGAs, i.e., the control uniquely acts as a switch between different regions of FPGA, where each region implements a mode task. It is interesting to introduce the control for the reconfigurability of FPGA [113]. The introduction of control in SystemC is another perspective, as the current SystemC model in Gaspard2 does not include control feature. GSGs that cannot be mapped onto the synchronous execution model, such as parallel execution of SGTs, can be possibly achieved in a SystemC execution model. However, the SystemC model does not ensure a synchronous execution of control, which needs more studies for safe design reasons.

Space-time mapping and clock based analysis. The synchronous modeling adopts a simple version of space-time mapping. However, it is possible to use a more complex one in consideration of non-functional aspects. There are several possibilities of this extension: arrays can have different clocks according to the different architecture when they are mapped onto flows; arrays and computation carried on these arrays may have different clocks due to the same reason; memory is needed when multi-processor and serialized execution are considered. The first two result in a synchronizability analysis of different clocks. [47] presents an analysis based on affine-clock system. The last one leads to a memory usage analysis, which is similar to the problem presented in [27]. The synchronous model is also expected to be extended with clock system, so that the parallel, serialized and partitioned models can be used for sophisticated clock-related analysis.

Bibliography

[1] M. Maroti A. Ledeczi and P. Volgyesi. The Generic Modeling Environment. In *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP'01)*, 2001. 61

[2] M. D. Adams. The JPEG-2000 still image compression standard. Technical Report Report N2412, ISO/IEC JTC 1/SC 29/WG 1, JPEG, septembre 2001. 16

[3] N. Aizenbud-Resher, R. F. Paige, J. Rubin, Y. Shalam-Gafni, and D. S. Kolovos. Operational semantics for traceability. In *ECMDA Traceability Work-shop (ECMDA-TW) 2005*, 2005. 32

[4] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA*, December 2005. 24

[5] VHDL Analysis and Standardization Group. VHDL. http://www.eda.org/vhdl-200x/, 2008. 12

[6] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, July 1996. IEEE-SMC. 53

[7] Charles André. SyncCharts: a Visual Representation of Reactive Behaviors. Research Report 96.56, I3S, Sophia Antipolis, April 1996. 53

[8] Charles André. Semantics of SSM (Safe State Machine). http://www.esterel-technologies.com, April 2003. 53

[9] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A UML-based Environment for System Design Space Exploration. *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 1272–1275, Dec. 2006. 175

[10] R. B. Atitallah, P. Boulet, A. Cuccuru, J.-L. Dekeyser, O. Labbani A. Honoré, Sébastien Le Beux, É. Piel P. Marquet, J. Taillard, and H. Yu. Gaspard2 UML profile documentation. Technical Report RR-0342, INRIA, DaRT team, September 2007. 39, 42, 115

[11] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(Issue: 1):70–78, Jan 2002. 11

[12] A. Benveniste, P. Le Guernic, and P. Aubry. *Compositionality in Dataflow Synchronous Languages: Specification and Code Generation*, volume LNCS 1536, page 61. Springer, Malente, Germany, 1997. Proceedings of the 1997 Workshop on Compositionality Albert Benveniste, Paul Le Guernic, and Pascal Aubry. 78

[13] G. Berry. The constructive semantics of pure Esterel. 63

[14] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. 53, 62

[15] G. Berry and E. Sentovich. Multiclock esterel. In *Proc. CHARME'2001*, Correct Hardare Design and Verification Methods, Edinburgh, 2001. Springer-Verlag. LNCS 2144. 62

[16] L. Besnard, T. Gautier, and P. Le Guernic. *Signal Reference Manual.*, 2006. www.irisa.fr/espresso/Polychrony. 17

[17] L. Besnard, H. Marchand, and E. Rutten. The Sigali Tool Box Environment. In *Workshop on Discrete Event Systems, WODES'06*, Ann-Arbor (MI, USA), July 2006. 59

[18] J. Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005. 26

[19] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, http://hal.inria.fr/inria-00128840/en/, February 2007. 17, 18, 19, 22, 23, 40, 41, 74, 105

[20] P. Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Technical Report 6467, INRIA, France, March 2008. http://hal.inria.fr/inria-00261178/en. 17, 18

[21] F. Boussinot and R. de Simone. The Esterel language. another look at real time programming. *Proceedings of the IEEE*, Special Issue 79(9):1293–1304, 1991. 53, 62

[22] C. Brunette, J.-P. Talpin, L. Besnard, and T. Gautier. Modeling multi-clocked dataflow programs using the Generic Modeling Environment. In *Synchronous Languages, Applications, and Programming*. Elsevier, March 2006. 61, 118

[23] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 4:155–182, April 1994. Special issue on Simulation Software Development. 63

[24] 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. ACM Press, 1987. 54

[25] A. Charfi, A. Gamatié, A. Honoré, J.-L. Dekeyser, and M. Abid. Validation de modèles dans un cadre d'IDM dédié à la conception de systémes sur puce. In *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles - IDM'08*, Mulhouse - France, June 2008. 167

[26] T. A.C.M. Claasen. System on a Chip: Changing IC design today and in the future. *IEEE Micro*, 23(3):20–26, May/Jun 2003. 11, 13

[27] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-sychronous Kahn networks. In *ACM Symp. on Principles of Programming Languages (PoPL'06)*, Charleston, South Carolina, USA, January 2006. 180

[28] A. Cuccuru. *Modélisation Unifiée des Aspects Répétitifs dans la Conception Conjointe Logicielle/Matérielle des Systèmes sur Puce à Hautes Performances*. PhD thesis, Université des Sciences et Technologies de Lille, Lille, France, December 2005. 40

[29] K. Czarnecki and S. Helsen. Classification of model tranformation approaches. In *Proceeding of OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2003. 31

[30] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, October 1998. 17

[31] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Colloque GRETSI sur le Traitement du Signal et de l'Image, Juan-Les-Pins, France*, September 1995. 17, 40

[32] P. Dumont. *Spécification multidimensionnelle pour le traitement du signal systématique*. PhD thesis, Université des Sciences et Technologies de Lille, France, December 2005. Dumont Philippe. 22, 73

[33] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating Array-Ol in Ptolemy II. Technical Report 5516, INRIA, France, March 2005. available at www.inria.fr/rrrt/rr-5516.html. 24

[34] Eclipse. Eclipse Modeling Framework. http://www.eclipse.org/emf. 29, 128

[35] Eclipse. EMFT JET. http://www.eclipse.org/emft/projects/jet. 32, 128

[36] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, andsynthesis. *Proceedings of the IEEE*, 85:366–390, 1997. 13

[37] A. Etien, C. Dumoulin, , and E. Renaux. Towards a Unified Notation to Represent Model Transformation. Research Report 6187, INRIA, 05 2007. 31, 125

[38] M. Eva. *SSADM Version 4: A User's Guide*. McGraw-Hill Publishing Co, april 1994. 27

[39] A.D. Falkoff and K.E. Iverson. The design of APL. *IBM Journal of Research and Development*, 17(5):324–334, 1973. 76

[40] J-M. Favre, J. Estublier, and M. Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles, au-delà du MDA*. Hermès Science, Lavoisier, Jan. 2006. 26, 27

[41] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I One Dimensional Time. *International journal of parallel programming*, 21(5):313–348, October 1992. 74

[42] P. Feautrier. Some efficient Solutions to the Affine Scheduling Problem. Part II Multidimensional Time. *International journal of parallel programming*, 21(6):389–420, December 1992. 74

[43] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005. 33, 37, 135

[44] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12):11–14, Dec. 1983. 11

[45] A. Gamatié, E. Rutten, and H. Yu. A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems. Research Report RR-6589, INRIA, http://hal.inria.fr/inria-00293909/fr, July 2008. 45, 97, 107, 110

[46] A. Gamatié, É. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser. Modeling and formal validation of high-performance embedded systems. In *7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, Krakow Poland, July 2008. 167

[47] A. Gamatié, É. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser. Synchronous modeling and analysis of data intensive applications. *EURASIP Journal on Embedded Systems*, 2008. To appear. Also available as INRIA Research Report: http://hal.inria.fr/inria-00001216/en/. 86, 87, 167, 180

[48] A. Girault and H. Yu. A flexible method to tolerate value sensor failures. In *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'06)*, Prague, Czech Republic, September 2006. 175

[49] C. Glitia and P. Boulet. High level loop transformations for multidimensional signal processing embedded applications. In *International Symposium on Systems, Architectures, MOdeling, and Simulation (SAMOS VIII)*, Samos, Greece, July 2008. 73

[50] R. Goering. SoC value linked to software. EE Times, Dec. 2005. 13

[51] S. Graf. Omega – Correct Development of Real Time Embedded Systems. *SoSyM, int. Journal on Software & Systems Modelling*, 7(2):127–130, 2008. 33, 175

[52] M. Griebl, P. Faber, and C. Lengauer. Spacetime mapping and tiling: a helpful combination. *Concurr. Comput. : Pract. Exper.*, 16(2-3):221–246, 2004. 73

[53] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993. 50, 51

[54] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), September 1991. 53, 54, 55, 56

[55] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag. 56, 57

[56] N. Halbwachs and D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986. 17, 57

[57] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI 2008*, June 2008. 57

[58] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 34, 52, 91, 96, 97, 105

[59] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990. 52

[60] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. 105

[61] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. 50

[62] High Performance Fortran Forum. High performance fortran language specification, January 1997. http://hpff.rice.edu/versions/hpf2/index.htm. 17, 76

[63] Open SystemC Initiative. SystemC. http://www.systemc.org/home. 13

[64] INRIA Atlas Project. ATL. http://modelware.inria.fr/rubrique 12.html. 32

[65] INRIA DaRT Project. Gaspard2. https://gforge.inria.fr/projects/gaspard2. 3, 39, 40

[66] INRIA Triskell Project. Kermeta. http://www.kermeta.org/. 32

[67] A. Jerraya and W. Wolf, editors. *Multiprocessor Systems-on-Chip*. Elsevier Morgan Kaufmann, San Francisco, California, 2005. 10

[68] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress, vol 74 of Information Processing*, pages 471–475, 1974. 53, 54

[69] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Ak¸sit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–. Springer-Verlag, 1997. 27

[70] A. Kountouris and P. Le Guernic. Profiling of Signal programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, Bristol, UK, February 1996. HP Labs. 167

[71] O. Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. PhD thesis, USTL, 2006. 89, 91, 94, 96, 97, 139, 178

[72] O. Labbani, J.-L. Dekeyser, P. Boulet, and E. Rutten. Introducing control in the Gaspard2 data-parallel metamodel: Synchronous approach. In *Int'l Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES'05), Montego Bay, Jamaica*, October 2005. 45, 97

[73] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Aspects Computing*, 11:637–664, 1999. 175

[74] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991. 64

[75] S. Le Beux, P. Marquet, O. Labbani, and J.-L. Dekeyser. FPGA implementation of embedded cruise control and anti-collision radar. In *9th Euromicro Conference on Digital System Design (DSD'2006)*, Dubrovnik, Croatia, August 2006. 180

[76] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sep 1991. 53

[77] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003. 87

[78] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. 17

[79] Lustre. Vérimag. http://www-verimag.imag.fr/SYNCHRONE/index.php?page=lang-design. 169

[80] R. Manduchi, G. M.Cortelazzo, and G. A. Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 3:325–340, 1993. 16

[81] F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In *International Symposium: Compositionality - The Significant Difference*, Malente/Holstein, Germany, September 1997. 78

[82] F. Maraninchi and Y. Rémond. Mode-Automata: About Modes and States for Reactive Systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag. 63, 97

[83] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27:61–92, 2001. 53

[84] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003. 45, 49, 91, 99, 105

[85] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000. 60, 173

[86] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan. Formal Verification of programs specified with Signal : Application to a Power Transformer Station Controller. *Science of Computer Programming*, 41(1):85–104, August 2001. 60

[87] G. Martin and W. Mueller, editors. *UML for SoC Design*. Springer, June 2005. 13

[88] C. Mauras. A*lpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, France, December 1989. 17

[89] Planet MDE. Model-Driven Engineering. http://planetmde.org. 25, 28

[90] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142, March 2006. 30

[91] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. 27

[92] J. Miller and J. Mukerji. Model Driven Architecture (MDA). Technical report, OMG, 2001. 33

[93] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983. 51

[94] L. Morel. Array iterators in Lustre: From a language extension to its exploitation in validation. *EURASIP Journal on Embedded Systems*, 2007, 2007. 17, 57, 77

[95] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50:3306–3309, 2002. 17

[96] Object Management Group Inc. http://www.omg.org. 33

[97] Object Management Group Inc. Model-driven architecture (mda). http://www.omg.org/mda. 33

[98] Object Management Group Inc. Modeling and analysis of real-time and embedded systems (MARTE). http://www.omgmarte.org/. 13, 38, 44

[99] Object Management Group Inc. MOF 2.0 core final adopted specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, Oct. 2003. 29

[100] Object Management Group Inc. Uml profile for schedulability, performance and time. Technical report, OMG, Sept. 2003. 38, 40

[101] Object Management Group Inc. MOF Query / Views / Transformations. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, Nov. 2005. 32, 33

[102] Object Management Group Inc. Final adopted omg sysml specification. http://www.omg.org/cgi-bin/doc?ptc/06-0504, mai 2006. 38, 40

[103] Object Management Group Inc. Omg unified modeling language (OMG UML), superstructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/, Nov. 2007. 34, 35, 38, 40, 132, 134, 136

[104] Object Management Group Inc. Uml 2 infrastructure (final adopted specifcation). http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/, Nov. 2007. 27, 33

[105] OMG. UML Profile for System on a Chip (SoC), June 2006. 13

[106] OpenMP Architecture Review Board. OpenMP application programme interface. http://www.openmp.org/drupal/mp-documents/spec25.pdf, May 2005. 43

[107] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall PTR, second edition edition, February 2003. 12

[108] N. Pernet and Y. Sorel. A design method for implementing specifications including control in distributed embedded systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA'05*, Catania, Italy, September 2005. 64

[109] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*. Springer, 1996. Guy-René Perrin and Alain Darte. 19

[110] E. Piel. *Ordonnancement de systèmes parallèles temps-réel*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), 2007. 196

[111] M. Pouzet. Lucid synchrone: Reference manual. www.lri.fr/~pouzet/lucid-synchrone. Marc Pouzet. 61

[112] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: www.lri.fr/~pouzet/lucid-synchrone. 53, 64

[113] I. Quadri, S. Meftali, and J.-L. Dekeyser. High Level Modeling of Partially Dynamically Reconfigurable FPGAs with MDE and MARTE. In *The Reconfigurable Communication-centric Systems-on-Chip workshop 2008 (ReCoSoC'08)*, 2008. 180

[114] F. Rocheteau. *Extension du langage Lustre et application la conception de circuits: le langage Lustre-V4 et le système Pollux,*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1992. 17, 57, 77

[115] E. Rutten and F. Martinez. SIGNAL GTi: implementing task preemption and time intervals in the synchronous data flow language SIGNAL. In *7th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'95)*, pages 176–183. IEEE, 1995. 61

[116] D.-C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. 25, 28

[117] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In *CAV 2001 Workshop on Software Model Checking*, ENTCS 55(3), Paris, France, 2001. 175

[118] B. V. Selic. On the semantic foundations of standard UML 2.0. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2004*, volume 3185 of *Lecture Notes in Computer Science*, Bertinoro, Italy, septembre 2004. Springer-Verlag. 33

[119] I. Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps réel en utilisant les langages* Signal *et* Alpha*.* PhD thesis, Université de Rennes 1, Rennes, France, October 1998. 59

[120] I.M. Smarandache, T. Gautier, and P. Le Guernic. Validation of mixed Signal-Alpha real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999. 59

[121] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. PhD thesis, Université de Lille 1, December 2001. In French. 152 pages Julien Soula. 22, 73

[122] P. Stevens. A landscape of bidirectional model transformations. In *Summer school on Generative and Transformational Techniques in Software Engineering 2007 (GTTSE'07)*, 2007. 31

[123] S. Sundaram. More is more: Improvements in camera phone performance drive increased sales. Toshiba America Electronic Components, Inc. White papar., 2008. 155

[124] SystemVerilog. http://www.systemverilog.org/. 13

[125] J. Taillard, F. Guyomarc'h, and J.-L. Dekeyser. OpenMP code generation based on an Model-Driven Engineering approach. In *The 2008 High Performance Computing & Simulation Conference (HPCS 2008)*, Nicosia, Cyprus, June 2008. 24

[126] J.-P. Talpin, C. Brunette, T. Gautier, and A. Gamatié. Polychronous mode automata. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 83–92, New York, NY, USA, 2006. ACM. 64

[127] H. Tardieu, A. Rochfeld, and R. Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991. 27

[128] West Team. Gaspard classic: Graphical array specification for parallel and distributed computing. http://www2.lifl.fr/west/gaspard/classic.html. 23

[129] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag. 17

[130] M. von der Beeck. A comparison of statecharts variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag. 53

[131] D. Wilde. The ALPHA Language. Technical Report 827, IRISA - INRIA, Rennes, 1994. available at www.irisa.fr/bibli/publi/pi/1994/827/827.html. 59

[132] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(Issue: 4):38– 43, April 2003. 13

[133] H. Yu, A. Gamatié, É. Rutten, P. Boulet, and J.-L. Dekeyser. Vers des transformations d'applications à parallélisme de données en équations synchrones. In *9ème édition de SYMPosium en Architectures nouvelles de machines(SympA'2006)*, Perpignan, France, Octobre 2006. 24

[134] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser. Model transformations from a data parallel formalism towards synchronous languages. Research Report RR-6291, INRIA, http://hal.inria.fr/inria-00172302/en/, Sept. 2007. 115, 169

[135] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser. *Embedded Systems Specification and Design Languages, Selected Contributions from FDL'07 Series*, volume 10 of *Lecture Notes Electrical Engineering*, chapter 13: Model Transformations from a Data Parallel Formalism towards Synchronous Languages. Springer Verlag, 2008. ISBN: 978-1-4020-8296-2. 115, 179

[136] H. Yu, A. Gamatié, É. Rutten, and J.-L. Dekeyser. Safe design of high-performance embedded systems in a mde framework. *Innovations in Systems and Software Engineering (ISSE)*, 4(3), 2008. NASA/Springer journal ISSE. 167, 173

[137] Huafeng Yu, Abdoulaye Gamatié, Éric Rutten, and Jean-Luc Dekeyser. Safe design of high-performance embedded systems in a mde framework. In *1st IEEE International UML & Formal Methods workshop (UML&FM'08, hosted by ICFEM 2008)*, Kitakyushu, Japan, october, october 2008. Accepted in a special issue of NASA/Springer journal ISSE. 169

Appendices

Appendix A

The Gaspard2 metamodel

Some general ideas of Gaspard2 are presented in Chapter 3. In addition, a Gaspard2 abstract syntax is introduced in Chapter 5, which helps to understand some general concepts of Gaspard2. Here the Gaspard2 metamodel is presented, which enables the model transformations in an MDE framework. The metamodel is presented according to their different owning packages: *Component*, *Factorization*, *Application*, *Deployment*, etc..

Component package

Gaspard2 adopts component-based design method, so the *Component* (Figure A.1) package is first presented. A *Component* is a named element, which may have ports as its interface. As a structured element, the *Component* may have internal structures, such as *Elementary*, *Repetitive*, *Compound*, etc.. *Elementary* indicates that the component is an elementary function component, which is a black box.

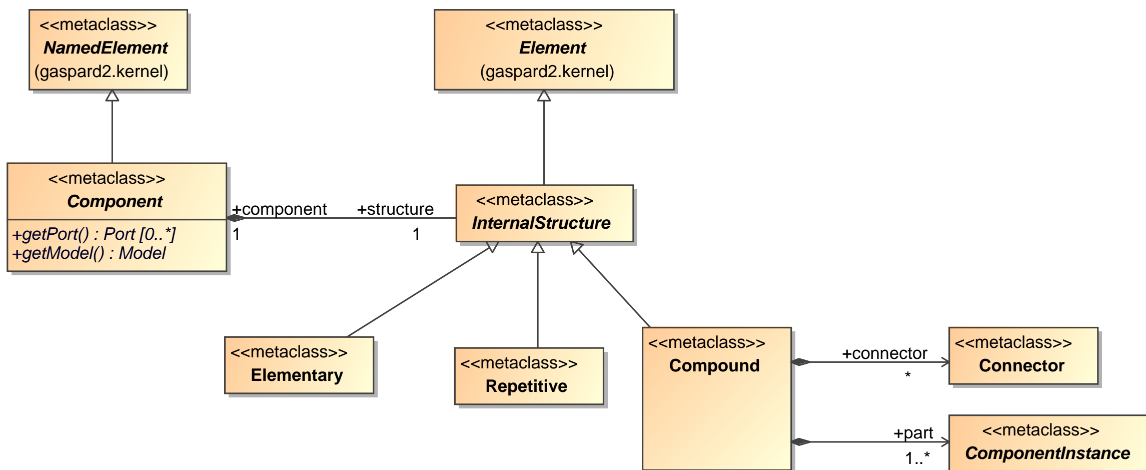


Figure A.1: Extract of the Gaspard2 metamodel: `Component`.

Port (Figure A.2) may have name and be connected to other elements. It is associated with a *DataType*, which indicates the type of the data that pass through the port. According to the

direction of data that pass through the port, the port is classified into *InputPort*, *OutputPort*, *InputOutputPort*.

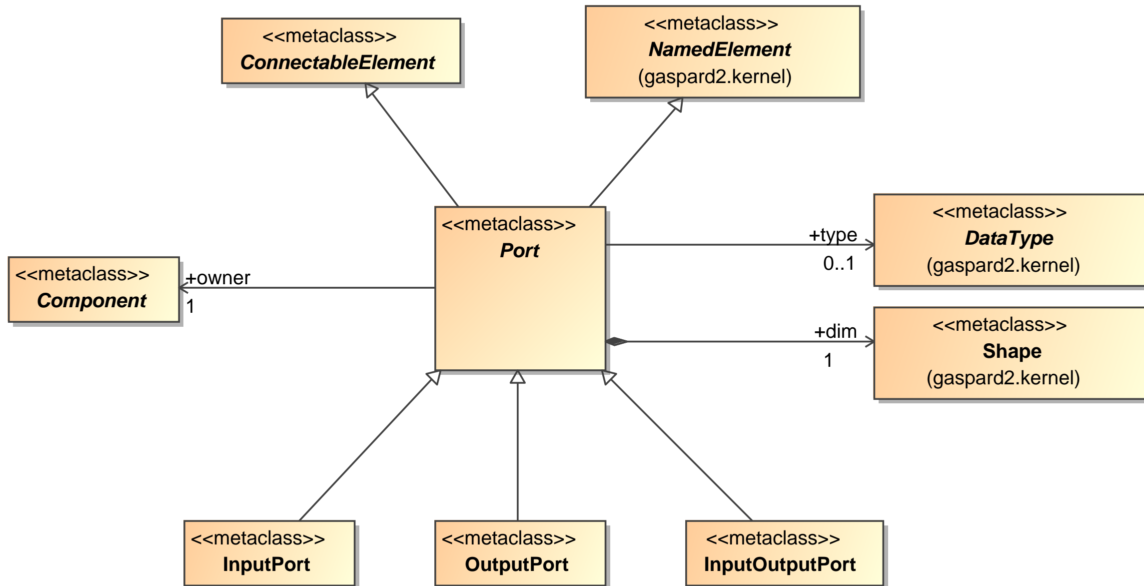


Figure A.2: Extract of the Gaspard2 metamodel: `Port`.

Factorization package

The key notion in the factorization package is *Repetitive* (Figure A.3) and *Tiler*. As an internal structure, *Repetitive* has port that can connect to the port of the owning component. It has also the *ComponentInstance*.

In a repetitive component, which has a *Repetitive* as internal structure, the connectors that link the component with the internal structure are generally *Tilers* (Figure A.4). A *Tiler* is associated to a *TilingDescription* in order to indicate how to tile the arrays. It has a *intVector* to store the original point of the array, and two *Matrix*, which are filing and paving matrix respectively.

In the case of inter-repetition dependency, the additional connectors are introduced: *InterRepetition*, *DefaultLink*. An *InterRepetition* connector links an output port of a repetitive instance to one of its input port. But note that the input/output ports do not belong to the same repetition defined in the repetition space, this is denoted by the *Shape* associated with the *InterRepetition*. *DefaultLink* is used to give the default value on the occasion of the absence of input value for some input ports.

Application package

The application package contains all the software application elements. *ApplicationModel* (Figure A.5) is the root of the *Application* package. One of the basic elements in the Application package is *ApplicationComponent* (Figure A.6), which models functionalities (called task

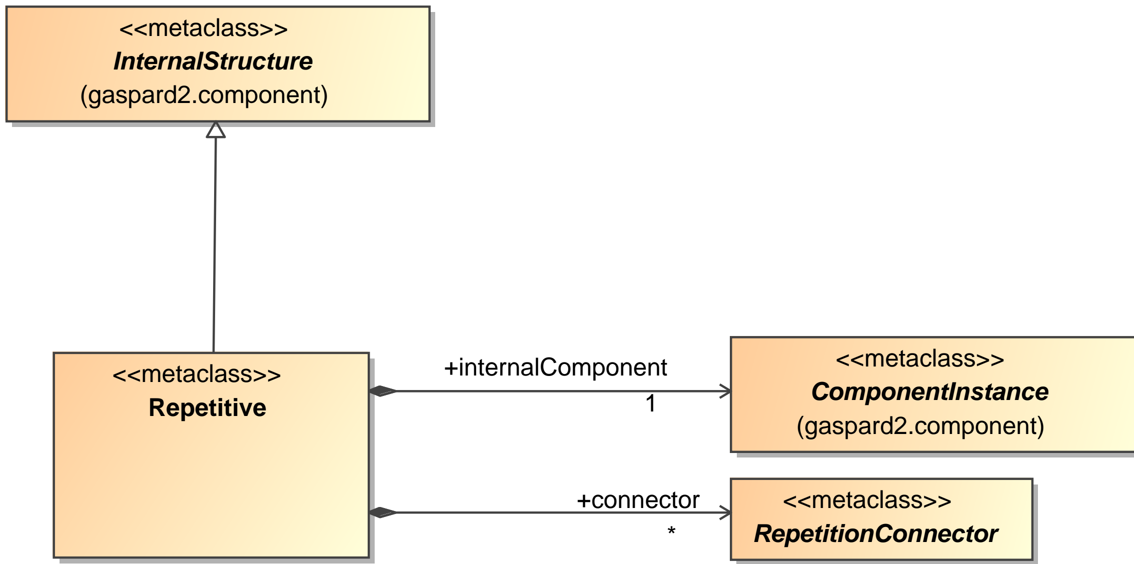


Figure A.3: Extract of the Gaspard2 metamodel: `Repetitive`.

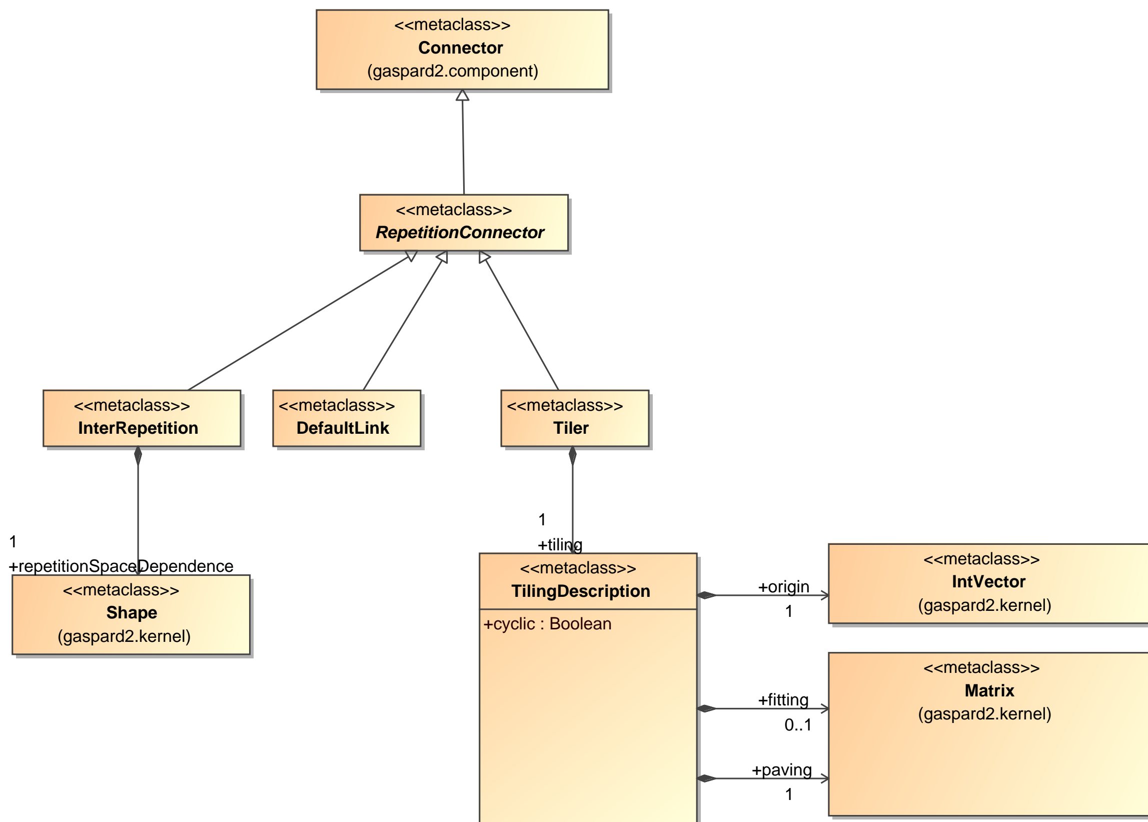


Figure A.4: Extract of the Gaspard2 metamodel: `Tiler`.

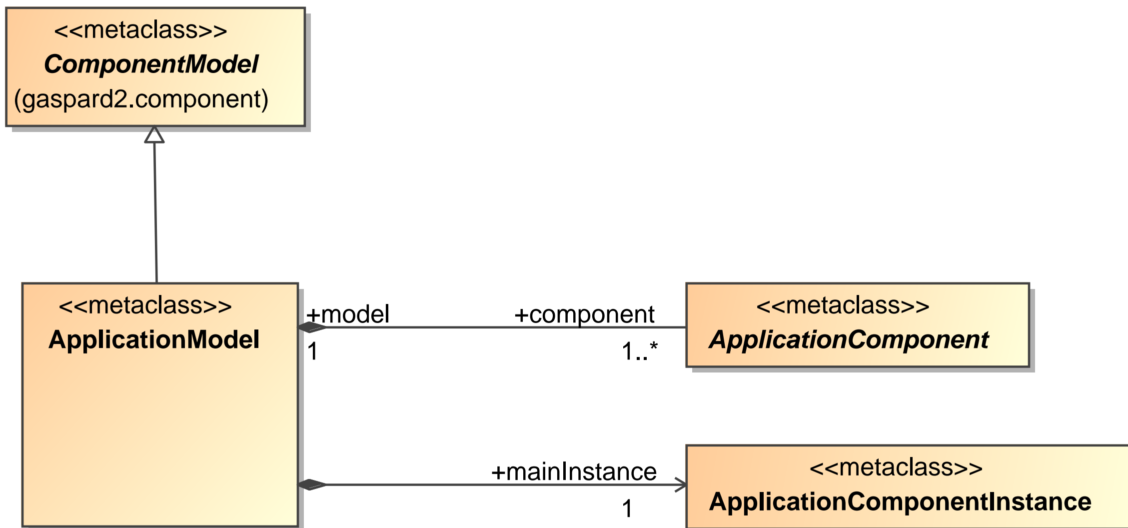


Figure A.5: Extract of the Gaspard2 metamodel: `ApplicationModel`.

in Array-OL). *ApplicationComponent* has *Ports*, from which the component receives/sends data. An *ApplicationComponent* also has an *InternalStructure*, such as *Elementary*, *Compound* and *Repetitive*. These internal structures are defined according to the component instances inside the *ApplicationComponent*. *Elementary* means no component instance defined in an *ApplicationComponent*, so it should be deployed with an IP. If the *ApplicationComponentInstances* inside an *ApplicationComponent* have no multipilicity and are connected in a concatenation or parallel way, the internal structure is called *Compound*. The component instances are connected through their *PortInstances* and *Connector*. But if an *ApplicationComponentInstance* has multipilicity defined on itself, its internal structure is called *Repetitive*. The internal component is connected to its owning component through *Tilers*. Some information about tiling is given, such as *FittingMatrix*, *PavingMatrix* and *OriginalPoint*.

Deployment package

The deployment package (Figure A.7) in the Gaspard2 metamodel is dedicated to provide enough information about the deployment of IPs in Gaspard2. According to the usage of these pieces of information, two classes are distinguished: *specilizable* and *characterizable*. The first one denotes the information that is passed to the compiler of an IP so as to have a specialized implementation from this IP. Whereas *characterizable* indicates the information to be delivered to the transformation so that the implementation of an IP is integrated in the application. *AbstractImplementation* is used to indicate a functionality that could be implemented by several *Implementations*. Each *Implementation* is linked to a *CodeFile*, which indicates the physical source or implementation of an IP.

The deployment in this section only concerns the software application deployment. The complete modeling and explanation of Gaspard2 deployment can be found in the theses of Eric Piel [110].

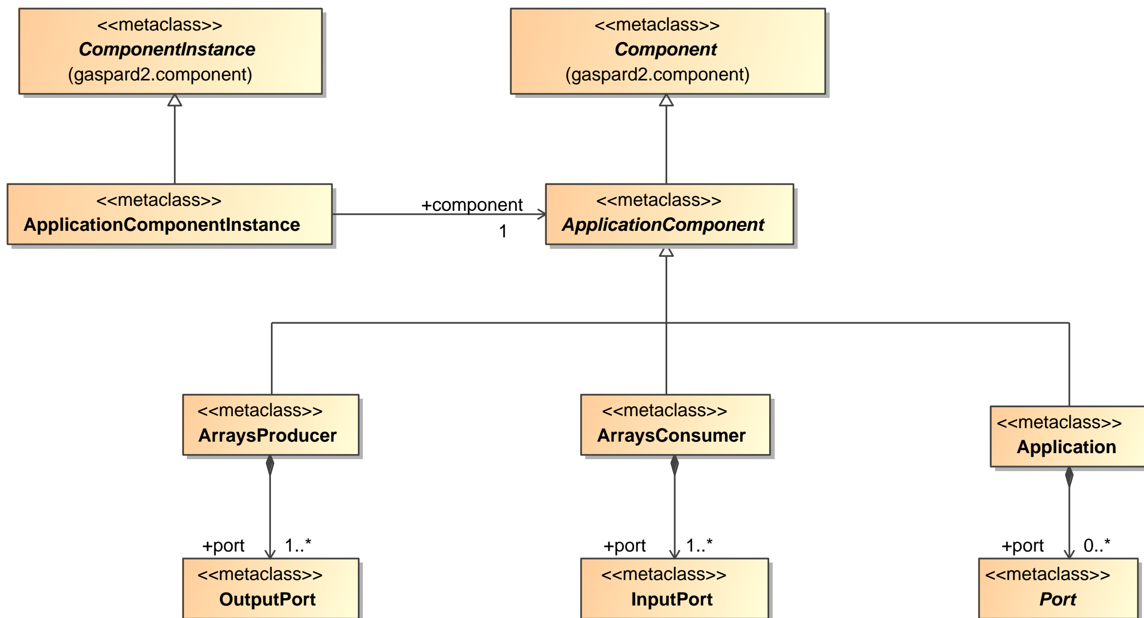


Figure A.6: Extract of the Gaspard2 metamodel: `ApplicationComponent`.

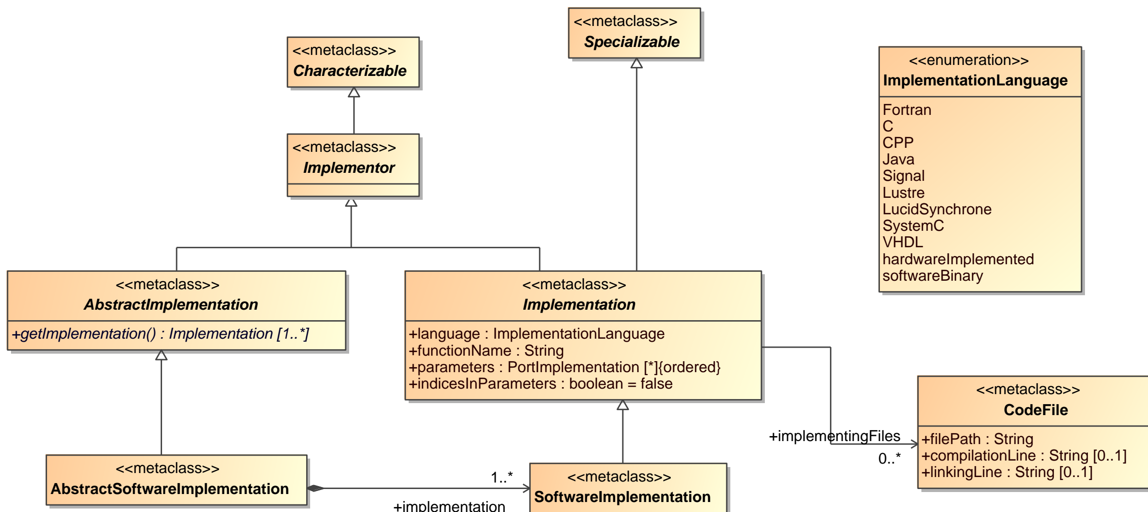


Figure A.7: Extract of the Gaspard2 metamodel: software deployment.

Appendix B

Code examples

The generated code of the downscaler example

Section 7.1 shows an example of downscaler modeled with the Gaspard2 profile, the extract of the generated code from the downscaler can be found in the following:

```
node TILER_INPUT_VERTICALFILTER_1 (
   Tiler_Input_Array:int^4^8)
 returns
   (Tiler_Pattern_0:int^8;Tiler_Pattern_1:int^8;
   Tiler_Pattern_2:int^8;Tiler_Pattern_3:int^8);
 let
   (Tiler_Pattern_0[0],...,Tiler_Pattern_0[7])
   =(Tiler_Input_Array[0,0],...,Tiler_Input_Array[0,7]);

   (Tiler_Pattern_1[0],...,Tiler_Pattern_1[7])
   =(Tiler_Input_Array[1,0],...,Tiler_Input_Array[1,7]);

   ...

   (Tiler_Pattern_3[0],...,Tiler_Pattern_3[7])
   =(Tiler_Input_Array[3,0],...,Tiler_Input_Array[3,7]);
tel

node TILER_OUTPUT_VERTICALFILTER_2 (
   Tiler_Pattern_0:int^4;
   Tiler_Pattern_1:int^4;Tiler_Pattern_2:int^4;
   Tiler_Pattern_3:int^4;)
 returns
   (Output_Array:int^4^4);
 let
   (Output_Array[0,0], Output_Array[0,3]) =
     (Tiler_Pattern_0[0],...,Tiler_Pattern_0[3]);
   ...
   (Output_Array[3,0], Output_Array[3,3]) =
```

```
     (Tiler_Pattern_3[0],...,Tiler_Pattern_3[3]);
tel

node TILER_INPUT_DOWNSCALER_3 (
   Tiler_Input_Array:int^640^480)
returns
   (Tiler_Pattern_0:int^8^8,
   ...
   Tiler_Pattern_479:int^8^8);
let
   (Tiler_Pattern_0[0,0],Tiler_Pattern_0[1,0],
   Tiler_Pattern_0[2,0],Tiler_Pattern_0[3,0],
   Tiler_Pattern_0[4,0],Tiler_Pattern_0[5,0],
   Tiler_Pattern_0[6,0],Tiler_Pattern_0[7,0],
   Tiler_Pattern_0[0,1],Tiler_Pattern_0[1,1],
   Tiler_Pattern_0[2,1],Tiler_Pattern_0[3,1],
   ...
   Tiler_Pattern_0[4,7],Tiler_Pattern_0[5,7],
   Tiler_Pattern_0[6,7],Tiler_Pattern_0[7,7]) =
   (Tiler_Input_Array[0,0],Tiler_Input_Array[0,1],
   Tiler_Input_Array[0,2],Tiler_Input_Array[0,3],
   Tiler_Input_Array[0,4],Tiler_Input_Array[0,5],
   Tiler_Input_Array[0,6],Tiler_Input_Array[0,7],
   Tiler_Input_Array[1,0],Tiler_Input_Array[1,1],
   ...
   Tiler_Input_Array[7,6],Tiler_Input_Array[7,7]);
   ...
tel

node TILER_OUTPUT_DOWNSCALER_4 (
   Tiler_Pattern_0:int^4^4,...,
   Tiler_Pattern_479:int^4^4, )
 returns
   (Output_Array:int^320^240);
let
   (Output_Array[0,0],Output_Array[1,0],
   Output_Array[2,0],Output_Array[3,0],
   ...
   Output_Array[0,3],Output_Array[1,3],
   Output_Array[2,3],Output_Array[3,3]) =
     (Tiler_Pattern_0[0,0],Tiler_Pattern_0[1,0],
     Tiler_Pattern_0[2,0],Tiler_Pattern_0[3,0],
     Tiler_Pattern_0[0,1],Tiler_Pattern_0[1,1],
     ...
     Tiler_Pattern_0[0,3],Tiler_Pattern_0[1,3],
     Tiler_Pattern_0[2,3],Tiler_Pattern_0[3,3]);
   ...
```

```
tel

node TILER_OUTPUT_HORIZONTALFILTER_5 (
   Tiler_Pattern_0:int^4;
   ...
   Tiler_Pattern_5:int^4;Tiler_Pattern_6:int^4;
   Tiler_Pattern_7:int^4)
returns
   (Output_Array:int^4^8);
let
   (Output_Array[0,0],...Output_Array[3,0]) =
     (Tiler_Pattern_0[0],Tiler_Pattern_0[1],
     Tiler_Pattern_0[2],Tiler_Pattern_0[3]);

   ...
   (Output_Array[0,7],...Output_Array[3,7]) =
     (Tiler_Pattern_7[0],Tiler_Pattern_7[1],
     Tiler_Pattern_7[2],Tiler_Pattern_7[3]);
 tel

node TILER_INPUT_HORIZONTALFILTER_6 (
   Tiler_Input_Array:int^8^8)
returns
   (Tiler_Pattern_0:int^8;Tiler_Pattern_1:int^8;
   Tiler_Pattern_2:int^8;Tiler_Pattern_3:int^8;
   Tiler_Pattern_4:int^8;Tiler_Pattern_5:int^8;
   Tiler_Pattern_6:int^8;Tiler_Pattern_7:int^8);
let
   (Tiler_Pattern_0[0],...,Tiler_Pattern_0[7]) =
   (Tiler_Input_Array[0,0],...,Tiler_Input_Array[7,0]);
   ...
   (Tiler_Pattern_7[0],...,Tiler_Pattern_7[7]) =
   (Tiler_Input_Array[0,7],...,Tiler_Input_Array[7,7]);

tel

node APP_VFILTER (VFILTERINPUT_ARRAY_0:int^8)
 returns
   (VFILTEROUTPUT_ARRAY_1:int^4);
 let
   (VFILTEROUTPUT_ARRAY_1) =
     INVOCATION_VFILTER(VFILTERINPUT_ARRAY_0);
 tel

node APP_HFILTER (HFILTERINPUT_ARRAY_0:int^8)
 returns   (HFILTEROUTPUT_ARRAY_1:int^4);
 let
```

```
   (HFILTEROUTPUT_ARRAY_1) =
     INVOCATION_HFILTER(HFILTERINPUT_ARRAY_0);
 tel

node APP_VERTICALFILTER (INPUT_ARRAY_0:int^4^8)
 returns   (OUTPUT_ARRAY_1:int^4^4);
 var
   TILER_IN_0_PATTERN_0:int^8;
   ...
   TILER_IN_0_PATTERN_3:int^8;
   TILER_OUT_1_PATTERN_0:int^4;
   ...
   TILER_OUT_1_PATTERN_3:int^4;
let
   (TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
   TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3) =
    TILER_INPUT_VERTICALFILTER_1(INPUT_ARRAY_0);

   (OUTPUT_ARRAY_1) =
    TILER_OUTPUT_VERTICALFILTER_2(
    TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
    TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3);

   (TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
   TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3)=
     TASK_REPETITION_VFILTER(
     TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
     TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3);
tel

node TASK_REPETITION_VFILTER(
   PATTERN_IN_0_0:int^8; PATTERN_IN_1_0:int^8;
   PATTERN_IN_2_0:int^8; PATTERN_IN_3_0:int^8;)
returns
   (PATTERN_OUT_0_1:int^4;PATTERN_OUT_1_1:int^4;
   PATTERN_OUT_2_1:int^4; PATTERN_OUT_3_1:int^4);
 let
   (PATTERN_OUT_0_1)=APP_VFILTER(PATTERN_IN_0_0);
   (PATTERN_OUT_1_1)=APP_VFILTER(PATTERN_IN_1_0);
   (PATTERN_OUT_2_1)=APP_VFILTER(PATTERN_IN_2_0);
   (PATTERN_OUT_3_1)=APP_VFILTER(PATTERN_IN_3_0);
 tel

node APP_HORIZONTALFILTER (INPUT_ARRAY_0:int^8^8)
 returns (OUTPUT_ARRAY_1:int^4^8);
 var
   TILER_IN_0_PATTERN_0:int^8; TILER_IN_0_PATTERN_1:int^8;
```

```
   TILER_IN_0_PATTERN_2:int^8; TILER_IN_0_PATTERN_3:int^8;
   TILER_IN_0_PATTERN_4:int^8; TILER_IN_0_PATTERN_5:int^8;
   TILER_IN_0_PATTERN_6:int^8; TILER_IN_0_PATTERN_7:int^8;
   TILER_OUT_1_PATTERN_0:int^4;TILER_OUT_1_PATTERN_1:int^4;
   TILER_OUT_1_PATTERN_2:int^4;TILER_OUT_1_PATTERN_3:int^4;
   TILER_OUT_1_PATTERN_4:int^4;TILER_OUT_1_PATTERN_5:int^4;
   TILER_OUT_1_PATTERN_6:int^4;TILER_OUT_1_PATTERN_7:int^4;
 let
   (TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
   TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3,
   TILER_IN_0_PATTERN_4,TILER_IN_0_PATTERN_5,
   TILER_IN_0_PATTERN_6,TILER_IN_0_PATTERN_7) =
     TILER_INPUT_HORIZONTALFILTER_6(INPUT_ARRAY_0);

   (OUTPUT_ARRAY_1) =
     TILER_OUTPUT_HORIZONTALFILTER_5
     (TILER_OUT_1_PATTERN_0, TILER_OUT_1_PATTERN_1,
     TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
     TILER_OUT_1_PATTERN_4, TILER_OUT_1_PATTERN_5,
     TILER_OUT_1_PATTERN_6, TILER_OUT_1_PATTERN_7);

   (TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
   TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
   TILER_OUT_1_PATTERN_4,TILER_OUT_1_PATTERN_5,
   TILER_OUT_1_PATTERN_6,TILER_OUT_1_PATTERN_7) =
     TASK_REPETITION_HFILTER(
     TILER_IN_0_PATTERN_0, TILER_IN_0_PATTERN_1,
     TILER_IN_0_PATTERN_2, TILER_IN_0_PATTERN_3,
     TILER_IN_0_PATTERN_4, TILER_IN_0_PATTERN_5,
     TILER_IN_0_PATTERN_6, TILER_IN_0_PATTERN_7);
 tel

node TASK_REPETITION_HFILTER (PATTERN_IN_0_0:int^8;
   PATTERN_IN_1_0:int^8;PATTERN_IN_2_0:int^8;
   PATTERN_IN_3_0:int^8;PATTERN_IN_4_0:int^8;
   PATTERN_IN_5_0:int^8;PATTERN_IN_6_0:int^8;
   PATTERN_IN_7_0:int^8)
 returns
   (PATTERN_OUT_0_1:int^4;PATTERN_OUT_1_1:int^4;
   PATTERN_OUT_2_1:int^4;PATTERN_OUT_3_1:int^4;
   PATTERN_OUT_4_1:int^4;PATTERN_OUT_5_1:int^4;
   PATTERN_OUT_6_1:int^4;PATTERN_OUT_7_1:int^4);
 let
   (PATTERN_OUT_0_1) = APP_HFILTER(PATTERN_IN_0_0);
   (PATTERN_OUT_1_1) = APP_HFILTER(PATTERN_IN_1_0);
   ...
   (PATTERN_OUT_6_1) = APP_HFILTER(PATTERN_IN_6_0);
```

```
   (PATTERN_OUT_7_1) = APP_HFILTER(PATTERN_IN_7_0);
 tel

node APP_DOWNSCALER (INPUT_ARRAY_1:int^640^480)
 returns
   (OUTPUT_ARRAY_0:int^320^240);
 var
   TILER_OUT_0_PATTERN_0:int^4^4;
   ...
   TILER_OUT_0_PATTERN_479:int^4^4;
   TILER_IN_1_PATTERN_0:int^8^8;
   ...
   TILER_IN_1_PATTERN_479:int^8^8;
 let
   (OUTPUT_ARRAY_0) =
    TILER_OUTPUT_DOWNSCALER_4(TILER_OUT_0_PATTERN_0);

   (TILER_IN_1_PATTERN_0,...,TILER_IN_1_PATTERN_479)=
     TILER_INPUT_DOWNSCALER_3(INPUT_ARRAY_1);

   (TILER_OUT_0_PATTERN_0,...,TILER_OUT_0_PATTERN_479)=
     TASK_REPETITION_HVFILTER(
     TILER_IN_1_PATTERN_0,...,TILER_IN_1_PATTERN_479);
 tel

node TASK_REPETITION_HVFILTER (PATTERN_IN_0_1:int^8^8,
   ..., PATTERN_IN_0_479:int^8^8)
 returns
 (PATTERN_OUT_0_0:int^4^4,...,PATTERN_OUT_0_479:int^4^4);
 let
   (PATTERN_OUT_0_0) = APP_HVFILTER(PATTERN_IN_0_1);
   ...
   (PATTERN_OUT_0_479) = APP_HVFILTER(PATTERN_IN_0_479);
 tel


node APP_REPDSMAIN (
   APP_REPDSMAIN_INPUT_ARRAY_0:int^640^480)
 returns
   (APP_REPDSMAIN_OUTPUT_ARRAY_1:int^320^240);
 let
   (APP_REPDSMAIN_OUTPUT_ARRAY_1) =
     APP_DOWNSCALER(APP_REPDSMAIN_INPUT_ARRAY_0);
 tel

node APP_HVFILTER (APP_HVFILTER_INPUT_ARRAY_0:int^8^8)
 returns
```

```
   (APP_HVFILTER_OUTPUT_ARRAY_1:int^4^4);
 var
   APP_HVFILTER_LOCAL_ARRAY_0:int^4^8;
 let
   (APP_HVFILTER_OUTPUT_ARRAY_1) =
     APP_VERTICALFILTER(APP_HVFILTER_LOCAL_ARRAY_0);

   (APP_HVFILTER_LOCAL_ARRAY_0) =
     APP_HORIZONTALFILTER(APP_HVFILTER_INPUT_ARRAY_0);
 tel

node MAIN_APPLICATION(
   APP_REPDSMAIN_INPUT_ARRAY_0:int^640^480)
 returns (APP_REPDSMAIN_OUTPUT_ARRAY_1:int^320^240);
 let
   (APP_REPDSMAIN_OUTPUT_ARRAY_1)=
     APP_REPDSMAIN(APP_REPDSMAIN_INPUT_ARRAY_0);
 tel
```

Automata in Targos and Sigali commands of the case study

Automata of the case study

The automata in the form of *Targos* that are used in the case study are illustrated in the next:

```
TARGOS

AUTOMATON EnergyStatus
STATES
   High init [ESModeH=true; ESModeM=false;ESModeL=false;]
   Medium  [ESModeH=false;ESModeM=true;ESModeL=false;]
   Low     [ESModeH=false;ESModeM=false;ESModeL=true;]
TRANS
   FROM High TO Medium  WITH rien0  [eESD and not eESU]
   FROM Medium TO Low   WITH rien0  [eESD and not eESU]
   FROM Low TO Medium  WITH rien0   [eESU and not eESD]
   FROM Medium TO High WITH rien0   [eESU and not eESD]

AUTOMATON      CommQuality
STATES
   Hign   [CQModeH=true; CQModeM=false; CQModeL=false;]
   Medium init [CQModeH=false; CQModeM=true;
                        CQModeL=false;]
   Low   [CQModeH=false; CQModeM=false; CQModeL=true;]
TRANS
```

```
   FROM Hign TO Medium  WITH rien0 [eCQD and
             not eCQU or ESModeL]
   FROM Medium TO Low   WITH rien0 [eCQD and not eCQU]
   FROM Low TO Medium   WITH rien0 [eCQU and not eCQD]
   FROM Medium TO Hign  WITH rien0 [eCQU and
            not eCQD and ESModeH]

AUTOMATON      ColorEffect
STATES
   Color   init [CEModeC=true; CEModeM=false; ]
   Monochrome   [CEModeC=false; CEModeM=true; ]
TRANS
   FROM Color TO Monochrome WITH rien0 [
            (eCEU and not eCED and aCE) or cCE]
   FROM Monochrome TO Color WITH rien0 [
             eCED and not eCEU and aCE ]

AUTOMATON      ImageStyle
STATES
   Normal init [ISModeN=true;ISModeB=false;
           ISModeE=false;ISModeS=false;]
   BNW      [ISModeN=false;ISModeB=true;
           ISModeE=false;ISModeS=false;]
   Negative    [ISModeN=false;ISModeB=false;
           ISModeE=true;ISModeS=false;]
   Sepia    [ISModeN=false;ISModeB=false;
           ISModeE=false;ISModeS=true;]
TRANS
   FROM Normal TO BNW WITH rien0
            [eISD and not eISU and aIS]
   FROM BNW TO Sepia WITH rien0
            [eISD and not eISU and aIS]
   FROM Sepia TO Negative WITH rien0
            [eISD and not eISU and aIS]
   FROM Negative TO Normal WITH rien0
            [eISD and not eISU and aIS]
   FROM BNW TO Normal WITH rien0
            [eISU and not eISD and aIS ]
   FROM Normal TO Negative WITH rien0
            [eISU and not eISD and aIS ]
   FROM Negative TO Sepia WITH rien0
            [eISU and not eISD and aIS ]
   FROM Sepia TO BNW WITH rien0
            [ eISU and not eISD and aIS ]

AUTOMATON      VideoSource
STATES
```

```
   Camera    [VSModeC=true; VSModeO=false; VSModeM=false;]
   Online    [VSModeC=false; VSModeO=true; VSModeM=false; ]
   Memory init  [VSModeC=false; VSModeO=false; VSModeM=true; ]
TRANS
   FROM Online TO Memory WITH rien0
            [eVSD and not eVSU and aVS or cVS]
   FROM Memory TO Online WITH rien0
            [eVSD and not eVSU and aVS or cVS]
   FROM Camera TO Online WITH rien0
            [eVSD and not eVSU and aVS or cVS]
   FROM Online TO Camera WITH rien0
            [eVSU and not eVSU and aVS or cVS]
   FROM Camera TO Memory WITH rien0
            [eVSU and not eVSU and aVS or cVS]
   FROM Memory TO Online WITH rien0
            [eVSU and not eVSU and aVS or cVS]

AUTOMATON      Resolution
STATES
  High  [RModeH=true; RModeM=false; RModeL=false;]
  Medium init [RModeH=false; RModeM=true; RModeL=false;]
  Low  [RModeH=false; RModeM=false; RModeL=true;]
TRANS
  FROM High TO Medium WITH rien0 [(eRD and not eRU) or cR]
  FROM Medium TO Low WITH rien0 [(eRD and not eRU) or cR]
  FROM Low TO Medium WITH rien0 [eRU and not eRD and aR]
  FROM Medium TO High WITH rien0 [eRU and not eRD and aR]
```

Sigali commands for model checking

The Sigali commands for model checking in consideration of cost functions:

```
read("task.z3z");
S : processus(conditions, etats, evolutions,
  initialisations, [gen(contraintes)],controlables);
read("Property.lib");
read("Synthesis.lib");
read("Verif_Determ.lib");
read("Simul.lib");
read("Synthesis_Partial_order.lib");

CE_Color_de_ColorEffect_2: a_var(
   Color_de_ColorEffect_2,0,30,0);
CE_Monochrome_de_ColorEffect_2:
   a_var(Monochrome_de_ColorEffect_2,0,20,0);

CE_Normal_de_ImageStyle_3: a_var(
```

```
   Normal_de_ImageStyle_3,0,10,0);
CE_BNW_de_ImageStyle_3: a_var(
   BNW_de_ImageStyle_3,0,10,0);
CE_Negative_de_ImageStyle_3: a_var(
   Negative_de_ImageStyle_3,0,10,0);
CE_Sepia_de_ImageStyle_3: a_var(
   Sepia_de_ImageStyle_3,0,10,0);

CE_High_de_Resolution_4: a_var(
   High_de_Resolution_4,0,50,0);
CE_Medium_de_Resolution_4: a_var(
   Medium_de_Resolution_4,0,40,0);
CE_Low_de_Resolution_4: a_var(
   Low_de_Resolution_4,0,30,0);

CE_Camera_de_VideoSource_5: a_var(
   Camera_de_VideoSource_5,0,30,0);
CE_Online_de_VideoSource_5: a_var(
   Online_de_VideoSource_5,0,30,0);
CE_Memory_de_VideoSource_5: a_var(
   Memory_de_VideoSource_5,0,20,0);

CE_ColorEffect: CE_Color_de_ColorEffect_2 +
   CE_Monochrome_de_ColorEffect_2;
CE_ImageStyle: CE_Normal_de_ImageStyle_3 +
   CE_BNW_de_ImageStyle_3 +
   CE_Negative_de_ImageStyle_3 +
   CE_Sepia_de_ImageStyle_3;
CE_Resolution: CE_High_de_Resolution_4 +
   CE_Medium_de_Resolution_4 +
   CE_Low_de_Resolution_4;
CE_VideoSource: CE_Camera_de_VideoSource_5+
CE_Online_de_VideoSource_5 +
   CE_Memory_de_VideoSource_5;
CE_Cons: CE_ColorEffect +
   CE_ImageStyle+CE_Resolution +
   CE_VideoSource;

CE_High_de_EnergyStatus_0: a_var(
   High_de_EnergyStatus_0,0,-120,0);
CE_Medium_de_EnergyStatus_0: a_var(
   Medium_de_EnergyStatus_0,0,-110,0);
CE_Low_de_EnergyStatus_0: a_var(
   Low_de_EnergyStatus_0,0,-110,0);

CE_EnergyStatus: CE_High_de_EnergyStatus_0 +
   CE_Medium_de_EnergyStatus_0 +
```

```
   CE_Low_de_EnergyStatus_0;
CE_Prod: CE_EnergyStatus;
CE_Global: CE_Cons + CE_Prod;
CE_Limitation : a_sup(CE_Global, 1);

Reachable(S, CE_Limitation);
```

Nomenclature

Array-OL Array Oriented Language

DIP Data-Intensive Processing

EMFT Eclipse Modeling Framework Technology

EMF Eclipse Modeling Framework

MARTE Modeling and Analysis of Real-Time and embedded System

MOF Meta-Object Facility

MoMoTE MOdel to MOdel Transformation Engine

OMG Object Management Group

QVT Query/View/Transformation

AOP Aspect-Oriented Programming

ATL ATLAS Transformation Language

CAD Computer-Aided Design

DCS Discrete Controller Synthesis

DIP Data-Intensive Processing

DSL Domain-Specific Language

DSP Digital Signal Processor

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GMA Gaspard2 Mode Automata

GPS Global Positioning System

GSG Gaspard2 state graph

HDL Hardware Description Language

HPF High-Performance Fortran

IC Integrated Circuits

IP Intellectual Property

IRD Inter-Repetition Dependency

JET Java Emitter Templates

MDA Model-Driven Architecture

MDE Model-Driven Engineering

MoC Model of Computation

MPSoC Multi-processor System on Chip

MST Mode Switch Task

MT Mode Task

MTC Mode Task Component

MTD Multidimentional-Time Dimension

NoC Networks on Chip

OCL Object Constraint Language

ODT Opérateurs de Distribution de Tableaux or array distribution operators in English

OOAD Object-Oriented Analysis and Design

PDA Personal Digital Assistant

PIM Platform-Independent Model

PSM Platform-Specific Model

RCT Repetition Context Task

RSM Repetitive Structure Modeling

RT Repetitive Task

RTOS Real-Time Operating System

SGC State Graph Component

SGT State Graph Task

SMC State Machine Component

SoC System on Chip or System-on-a-Chip

SPT Schedulability, Performance and Time

SQL Structured Query Language

SSADM Structured Systems Analysis and Design Methodology

SysML System Modeling Language

UML Unified Modeling Language

XML eXtensible Markup Language

Un Modèle Réactif Basé sur MARTE Dédié au Calcul Intensif à Parallélisme de Données : Transformation vers le Modèle Synchrone

Résumé: Les travaux de cette thèse s'inscrivent dans le cadre de la validation formelle et le contrôle réactif de calculs à haute performance sur systèmes-sur-puce (SoC).

Dans ce contexte, la première contribution est la modélisation synchrone accompagnée d'une transformation d'applications en équations synchrones. Les modéles synchrones permettent de résoudre plusieurs questions liées à la validation formelle via l'usage des outils et techniques formels offerts par la technologie synchrone. Les transformations sont développées selon l'approche d'Ingénierie Dirigé par les Modèles (IDM).

La deuxième contribution est une extension et amélioration des mécanismes de contrôle pour les calculs à haute performance, sous forme de constructeurs de langage de haut-niveau et de leur sémantique. Ils ont été défini afin de permettre la vérification, synthèse et génération de code. Il s'agit de déterminer un niveau d'abstraction de représentation des systèmes où soit extraite la partie contrôle, et de la modéliser sous forme d'automates à états finis. Ceci permet de spécifier et implémenter des changements de modes de calculs, qui se distinguent par exemple par les ressources utilisées, la qualité de service fournie, ou le choix d'algorithme remplissant une fonctionnalité.

Ces contributions permettent l'utilisation d'outils d'analyse et vérification, tels que la vérification de propriétés d'assignement unique et dépendance acyclique, model checking. L'utilisation de techniques de synthèse de contrôleurs discrets est également traitée. Elles peuvent assurer la correction de faĉon constructive: à partir d'une spécification partielle du contrôle, la partie manquante pour que les propriétés soient satisfaites est calculée. Grâce à ces techniques, lors du développement de la partie contrôle, la spécification est simplifiée, et le résultat est assuré d'être correct par construction.

Les modélisations synchrone et de contrôle reposes sur MARTE et UML. Les travaux de cette thèse sont été partiellement implémentés dans le cadre de Gaspard, dédié aux applications de traitement de données intensives. Une étude de cas est présentée, dans laquelle nous nous intéressont à une application de système embarqué pour téléphone portable multimédia.

Mots clefs: MARTE, parallélisme, contrôle réactive, systèmes-sur-puce, calculs à haute performance, Gaspard, ingénierie dirigée par les modèles, UML, model checking, synthèse de contrôleurs, langages synchrones.

A MARTE-Based Reactive Model for Data-Parallel Intensive Processing: Transformation Toward the Synchronous Model

Abstract: The work presented in this dissertation is carried out in the context of System-on-Chip (SoC) and embedded system design, particularly dedicated to data-parallel intensive processing applications (DIPs). Examples of such applications are found in multimedia processing and signal processing. On the one hand, safe design of DIPs is considered to be important due to the need of Quality of Service, safety criticality, etc., in these applications. However, the complexity of current embedded systems makes it difficult to meet this requirement. On the other hand, high-level safe and verifiable control, is highly demanded in order to ensure the correctness and strengthen the flexibility and adaptivity of DIPs.

As an answer to this issue, we propose to take advantage of synchronous languages to assist safe DIPs design. First, a synchronous modeling bridges the gap between the Gaspard framework, which is dedicated to SoC design for DIPs, and synchronous languages that act as a model of computation enabling formal validation. The latter, together with their tools, enable high-level validation of Gaspard specifications.

Secondly, a reactive extension and improvement to a previous control proposition in Gaspard, is also addressed. This extension is based on mode automata and contributes to conferring safe and verifiable features onto the previous proposition. As a result, model checking and discrete controller synthesis can be applied for the purpose of correctness verification.

Finally, a Model-Driven Engineering (MDE) approach is adopted in order to implement and validate our proposition, as well as benefit from the advantages of MDE to address system complexity and productivity issues. Synchronous modeling, MARTE-based (the UML profile for Modeling and Analysis of Real-Time and Embedded system) control modeling, and model transformations, including code generation, are dealt with in the implementation.

Keywords: MARTE, parallelism, reactive control, systems-on-chip, high-performance computing, Gaspard, model-driven engineering, UML, model checking, controller synthesis, synchronous languages.