Building and analyzing processing graphs on FPGAs with strong time and hardware constraints

Ke Du

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques
UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

Building and analyzing processing graphs on FPGAs under strong real-time environment and hardware constraints

KE DU

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

N° 0 0 5

Création et analyse de graphes de traitements sur FPGA, sous contraintes matérielles et contexte temps réel dur

présentée par

Ke DU

pour obtenir le

Grade de Docteur de

Université Bourgogne Franche-Comté

Spécialité : **Informatique**

Dissertation Committee:

ERIC MONMASSON	Rapporteur	Professeur à l'Université de Cergy Pontoise
FRANÇOIS AUGER	Rapporteur	Professeur à l'Université de Nantes
SERGE WEBER	Examinateur	Professeur à l'Université de Lorraine
MICHEL LENCZNER	Directeur de thèse	Professeur à l'Université Bourgogne Franche-Comté
STÉPHANE DOMAS	Co-directeur de thèse	Maître de Conférences à l'Université Bourgogne Franche-Comté

ABSTRACT

Building and analyzing processing graphs on FPGAs under strong real-time environment and hardware constraints

Ke Du
University of Burgundy Franche-Comté, 2018

Supervisors: Michel Lenczner, Stéphane Domas

In recent years, embedded systems has been widely used in both scientific environment and daily lives of common people. For some applications with strong real-time constraints, FPGA chips constitute a good choice. Their sizes and capacities are increasing continuously, allowing to build more and more complex applications. Thus, it is harder and harder to manage the application designs manually. This problem has been addressed through several ways. One is to use a model that is a more or less realistic abstraction of the behavior of the design. Nevertheless, it introduces another problem, which is the efficient implementation of the model on real architectures, like FPGAs. For example, some model characteristics may lead to a waste of resources, which can even make a design infeasible for a particular target architecture.

In this dissertation, we focus on overcoming some limitations yield by unfaithful descriptions of hardware behaviors for some existing models and the drawbacks of available tools. The Static/Synchronous Data Flow (SDF) based models, especially the version with Access Patterns (SDF-AP), are investigated. From the analysis of the problems of the existing models and EDA tools, our researches yield a new model: Actors with Stretchable Access Patterns (ASAP), and a new EDA tool called BlAsT (Block Assembly Tool). The model shares some basic principles of SDF-AP model but with other semantics and goals, which allows to model a wider range of behaviors and to obtain greater analysis capacities. Indeed, we propose a complete framework to check whether a design processes the input data streams correctly and if it is not the case, to modify the graph automatically to obtain this correctness. It is verified by experiments carried out on the realistic cases that clearly point out the advantages of ASAP over SDF-AP model, notably in terms of resources consumption. The BlAsT proposes a graphical interface to create designs by putting functional blocks on a panel and connecting them. It integrates the analysis principles defined by ASAP. It is also able to produce the VHDL code for the whole design. Thus, BlAsT offers the possibility for users without any knowledge in VHDL to create designs for FPGAs and with the insurance that it will produce correct results.

KEY WORDS: Field Programmable Gate Arrays (FPGAs), Embedded Systems, System on Chips (SoC), Static Analysis and Scheduling, Synchronous Data Flow (SDF), Model Based Design, Electronic Design Automation (EDA).

RÉSUMÉ

Création et analyse de graphes de traitements sur FPGA, sous contraintes matérielles et contexte temps réel dur

Ke Du
Université Bourgogne Franche-Comté, 2018

Encadrants: Michel Lenczner, Stéphane Domas

Ces dernières années, les systèmes embarqués ont envahit tant les environnements scientifiques que la vie quotidienne. Pour les applications avec des fortes contraintes temps réel, les FPGA sont un choix pertinent. Leur taille et leurs capacités évoluent constamment, ce qui permet de créer des applications de plus en plus complexe. Cependant, cette augmentation va de pair avec une difficulté croissante à créer le design de ces application à la main. Ce problème a été abordé de diverses façons. L'un d'entre elles consiste à élaborer un modèle qui abstrait le comportement d'un design de façon plus ou moins réaliste. Cependant, cela conduit à un autre problème, qui est la transposition du modèle sur une architecture réelle telle qu'un FPGA. Par exemple, certaines caractéristiques du modèle peuvent entraîner un gâchis de ressources logiques, au point de rendre le design inapplicable sur certaines architectures.

Dans cette thèse, nous nous intéressons à comment dépasser les limitations de certains modèles en terme d'expressivité de comportement. Nous abordons également celles des outils d'aide au développement de designs. Les modèles basés sur les flux de données synchrones (SDF) et plus spécialement la version avec patrons d'accès (SDF-AP) ont été pris comme référence. A partir de l'étude des limitations de ces modèles, nous avons produit un nouveau modèle nommé Acteurs avec patrons d'accès extensibles (ASAP), ainsi qu'un nouvel environnement d'aide au développement nommé BIAsT. Ce modèle a des caractéristiques communes avec SDF-AP mais en leur donnant des nouvelles définitions afin d'élargir le nombre de comportements modélisés et les possibilité d'analyse du design. En effet, nous proposons un cadre d'analyse complet qui vérifie si le design traite correctement les flux de données entrants et si ce n'est pas le cas, qui fait automatiquement les modifications minimales pour assurer des résultats corrects. Ce cadre a été testé sur une application réelle qui montre clairement les avantages que procure notre modèle comparé à SDF-AP, notamment en terme de consommation de ressources logiques. Quant au logiciel BIAsT, il propose une interface graphique pour créer un design, simplement en posant des blocs fonctionnels sur un panneau et en les connectant.

Il intègre les principes d'analyse tels que définis par ASAP. Enfin, il permet de générer automatique le code VHDL d'un design. En conclusion, il offre la possibilité de créer des designs FPGA sans aucun connaissance sur VHDL, tout en ayant l'assurance d'obtenir un code fonctionnel.

MOTS-CLÉS: Réseaux de portes logiques programmables (FPGAs), Systèmes embarqués, Système sur copeaux (SoC), Analyse et ordonnancement statique, Flux de données synchrones (SDF), Modèle basé sur la conception, Automatisation de la conception électronique (EDA).

Contents

List of Figures

List of Tables

List of Algorithms

NOTIONS AND ABBREVIATIONS

In order to help the reader, the main notions and acronyms used in this manuscript are reported here, together with a short description in English and French. They are given in their approximate order of appearance.

- **SDF graph:** Static Data Flow graph (graphe à flux de données statiques).

  SDF is a model that abstracts a design with a graph composed of actors linked by channels. An Actor represents a process that consumes and produces a fixed (i.e. static) number of data during each of its executions. Data are received/sent via the channels.

  SDF est un modèle qui abstrait un design grâce à un graphe composé d'acteurs reliés par des canaux. Un acteur représente un processus qui consomme et produit un nombre fixe (d'où le mot statique) de données à chacune de ses exécutions. Les données sont reçues/envoyées via les canaux.

- **SDF-AP:** Static Data Flow with Access Patterns (flux de données statiques avec schéma d'accès).

  SDF-AP is a model that uses access patterns to describe at which clock cycles an actor consumes/produces data during a single execution. A pattern is a sequence of 1 and 0, with a length equal to the duration of the execution in clock cycles. For a given clock cycle, a 1 means that the actor consumes/produces a data.

  SDF-AP est un modèle qui utilise des schémas d'accès pour décrire à quel cycle horloge un acteur consomme/produit une donnée lors de son exécution. Un schéma d'accès est une suite de 1 et 0, dont la longueur est celle d'une exécution en terme de cycles horloge. Pour un cycle donnée, un 1 signifie que l'acteur consomme/produit une donnée.

- **ASAP:** Actors with Stretchable Access Patterns (Acteurs avec schéma d'accès extensible).

  ASAP is a model also using access patterns. Nevertheless, patterns describe the maximum pace of consumption/production of an actor. When connected to other actors, it may receive/send data at a lower pace, which means that the pattern contains more 0 than expected: it is stretched. This model is the central contribution of this PhD.

  ASAP est un modèle qui utilise également les schémas d'accès. Néanmoins, ces schémas décrivent le rythme maximum de consommation/production. Quand un acteur est connecté à d'autres, il peut recevoir/envoyer des données à un rythme plus lent. Cela implique que le schéma contient plus de 0 que prévu : il est donc étiré.

- P_I, P_O: Number of input/output ports (nombre de ports d'entrée/sortie).

For an actor within a graph, P_I is the number of its input channels, and P_O the number of output channels. The word “port” is a reference to the name used for inputs/outputs signals of a VHDL component.

Pour un acteur au seind d'un graphe, P_I est le nombre de canaux entrants et P_O celui des canaux sortant. Le mot “port” est une référence à celui utilisé pour les signaux d'entrée/sortie d'un composant VHDL.

- **CP, PP:** Consumption/Production pattern (Schéma de consommation/production).

  The consumption/production pattern represents the maximum pace of consumption/production of an actor, during one execution. For example, $CP = [101]$ means that the actor can consume at most a data during the first and third clock cycle of its execution.

  Le schéma de consommation/production représente le rythme maximal de consommation/production d'un acteur durant son exécution. Par exemple, $CP = [101]$ signifie que l'acteur peut au mieux consommer une donnée lors des premier et troisième cycles horloge de son exécution.

- δ: Production delay (délai de production).

  It represents the latency of an actor, i.e. the number of clock cycles needed to produce the first result. It corresponds to the number of 0 at the beginning of PP.

  Il représente la latence d'un acteur, c'est-à-dire le nombre de cycles horloge pour produire le premier résultat. Il correspond au nombre de 0 au début de PP.

- **PC:** Production counter (Compteur de production).

  The production counter represents the number of data that must be consumed to produce a particular data. For example, $PC = [2, 3]$ means that the actor must consume 2 data to produce the first result, and one more (thus 3) to produce the second.

  Le compteur de production représente le nombre de données qui doivent être consommées pour produire une donnée en particulier. Par exemple, $PC = [2, 3]$ signifie que l'acteur doit consommer 2 données pour produire le premier résultat, et une de plus (donc 3) pour produire le second.

- **IP, OP:** Input/Output pattern (Schéma d'entrée/de sortie).

  The input/output pattern represents what an actor really receives/produces all along its executions when it is connected to other actors. IP must be compatible with the admittance pattern (see below) so that the actor can consume and process the data correctly. In that case, OP is computed from IP, PP and PC.

  Le schéma d'entrée/sortie représente ce qu'un acteur va réellement recevoir ou produire tout au long de ses exécutions, lorsqu'il est connecté à d'autres acteurs. IP doit être compatible avec le schéma d'admissibilité (voir ci-dessous) pour que l'acteur consomme et traite correctement les données. Dans ce cas, OP est calculé à partir de IP, PP et PC.

- **IS, OS, PS:** Input, Output, Production Schedules (Timings d'entrée, sortie, production).

  The input, output or production schedules are an alternative representation of the patterns. The schedule is the list of the clock cycles at which there is a 1 in the associated pattern.

Les timings d'entrée, sortie ou production sont une représentation alternative des schémas d'accès. Un timing est la liste de cycles horloge où apparaît un 1 dans le schéma associé.

- **AP:** Admittance pattern (Schéma d'admissibilité).

  The admittance pattern is built from the consumption pattern. If removing some 0 in the input pattern leads to the admittance pattern, they are declared to be compatible. It means that the actor consumes and processes correctly the input data.

  Le schéma d'admissibilité est créé à partir du schéma de consommation. Si en supprimant des 0 du schéma d'entrée, on obtient le schéma d'admissibilité, ils sont déclarés comme étant compatibles. Cela signifie que l'acteur consomme et traite correctement les données.

- Δ: Delay between executions (délai entre les exécutions).

  It describes the number of data that must be consumed by an actor before it can start another execution. Depending on its value and the number of 1 in the consumption pattern, it may lead to concurrent executions of the actor.

  Il décrit le nombre de données qui doivent être consommées par un acteur avant qu'il ne puisse être exécuté une nouvelle fois. Selon sa valeur et le nombre de 1 dans le schéma de consommation, cela peut conduire à des exécutions concurrentes de l'acteur.

- Γ: Topology matrix (matrice de topologie).

  It represents the relations between actors and their consumption/production on the channels of a graph. Assuming actors and channels are labeled with unique numbers, $\Gamma_{i,j}$ is the number of data consumed/produced (depending on the sing value) by actor j on channel i.

  Elle indique les relations entre les acteurs et leur consommation/production sur les canaux. En supposant que chaque acteur/canal reçoit un numéro unique, $\Gamma_{i,j}$ donne le nombre de données consommées/produites (selon de signe de la valeur) par l'acteur j sur le canal i.

- q: Repetition vector (vecteur de répétition).

  This vector only exists if all data produced by actors during a certain number of executions are finally consumed in a finite time. It expresses the existence of a cycle. Taking into account the same labels used for Γ, q_i gives the number of executions of actor i to obtain that cycle.

  Ce vecteur n'existe que si toutes les données produites par les acteurs durant un certain nombre d'exécutions sont consommés dans un temps fini. Cela exprime une notion de cycle. En se basant sur la même numération de Γ, q_i indique le nombre d'exécution de l'acteur i pour obtenir ce cycle.

- O: Traversal order (ordre d'analyse).

  This vector gives an order to analyze the actors, for example when input pattern compatibility is checked. It ensures that all the precursors of a given actor are analyzed before itself.

  Ce vecteur indique un ordre pour analyser le graphe, par exemple afin de tester la compatibilité des schémas d'entrée. Cet ordre assure que tous les précurseurs d'un acteur sont évalués avant lui-même.

- CM, PM: Consumption/Production matrix (matrice de consommatio/production).

  They give the number of data sent/receive between two given actors. They are used to check if sample rates of consumption/production are consistent between actors.

  Elles indiquent le nombre de données envoyées/reçues entre deux acteurs donnés. Elles sont utilisées pour vérifier que les taux de consommation/production entre deux acteurs sont cohérents.

- D: Downsampling matrix (matrice de réechantillonage).

  This matrix indicates the downsampling (i.e. the decimation rate) that must be applied between two actors so that their rates of consumtion/production are consistent.

  Cette matrice indique le rééchantillonnage (c.a.d. le taux de décimation) à appliquer entre deux acteurs afin que leur taux de consommation/production soient cohérents.

- DS: Decimation schedule (Timings de décimation).

  DS contains a vector for each input port of an actor. Each vector gives the clock cycles at which a decimation occurs (i.e. a valid data that is ignored).

  DS contient un vecteur pour chaque port d'entrée d'un acteur. Chaque vector indique à quel cycles horloge une décimation doit avoir lieu (c.a.d. une donnée valide ingorée).

- DM: Delay matrix (Matrice des délais).

  DM contains the delay that must be applied to each valid value received by an actor. It may be 0. These delays allows to enforce a corect processing.

  DM contient le délai à appliquer pour chaque donnée valide reçue par un acteur. Cela peut être 0. Ces délais permettent d'assurer un traitement correct.

- **BlAsT:** Block Assembly Tool (Outil d'assemblage de blocs).

  BlAsT is the software developped in the framework of this PhD. It allows to create FPGA designs graphically, to analyze and to modify them with the principles of ASAP. It also generates the VHDL code.

  BlAsT est l'outil logiciel développé dans le cadre de cette thèse. Il permet de créer de façon graphique des designs pour FPGA, de les analyser et les modifier grâce aux principes d'ASAP. Il génère également le code VHDL.

ACKNOWLEDGEMENTS

The long journey of my Ph.D. study has finished. It is with great pleasure to express my most sincere gratitude to Prof. Michel Lenczner and Assoc. Prof. Stéphane Domas, my supervisors, for giving me the opportunity to work on my dissertation, and for their dedicated guidance, thoughtful advices and endless patience during the course throughout the entire process of my study. During the past three years, They have been constantly available to discuss our results and provide insightful suggestions. Without their brilliant and illuminating instructions on my research and even about the writing, this dissertation could not reach its present form. This dissertation would never have been possible without their elaborative direction and meticulous corrections.

I wish also to acknowledge the members of AND(*Algorithmique Numérique Distribuée*) for the warm and friendly atmosphere in which they allowed me to work. Especially for the helps from Prof. Raphaël Couturier and my colleague Yousra Ahmed Fadil, Amor Lamar, Nesrine Khernane and Ali Kadhum Idrees both in life and study. I will never forget all the colleagues: Jean-François Couchot, Mourad Hakem, Gilles Perrot, Michel Salomon, Jean-Claude Charr, Karine Deschinkel, Arnaud Giersch, Abdallah Makhoul, Fabrice Ambert, Christophe Guyeux, Mohammed Bakiri, Joseph Azar, Gaby Boutayeh, Zeinab Fawaz, Christian Salim, Carol Habib, Anthony Nassar and Ahmed Badri Muslim Fanfakh.

I would further like to give my gratitude to the financial support from the program of China Scholarships Council (CSC), and to UBFC where I did my dissertation.

My sincere thanks also goes to my friends that I passed an amazing journey with them in France. I will never forget the beautiful moments I shared with you: Ruifeng Zhu, Yan Wang, Jie Qiu, Chunjie Huang, Chaoyue Chen, Hui Shang, Renfei Han, Zhao Zhang, Dongxue Lu, Chen Song, Jianding Guo, Yingchun Xie, Rongrong Liu, Jian Zhang, Tao Jiang, Hailong Wu, Mengli Yin, Lei Zhang, Daming Zhou, Bei Li, Yu Wu, Jinjian Li, Jin Wei.

I would also like to express my thanks to the high speed development of China and the fast growing of real estate prices, which have been encouraging me to non-stop hard working.

Finally, I wish to take this opportunity to express my appreciation and thanks to all my family for the emotional support. I would especially like to thank my parents who provide me the mental support and encouragement to explore the unknowns. Thanks especially to my wife Shuyi for her love. The future will be much better.

1

INTRODUCTION

1.1/ GENERAL INTRODUCTION

With the development of electronic industry, a growing number of projects require real-time streaming applications on embedded platforms. These comprise increasingly high hardware and timing constraints, which leads to the use of FPGAs (Field Programmable Gate Arrays). Usually, the designer should have a good knowledge of programming with VHDL or Verilog HDL. Unfortunately, only specialists can do it, because this needs a lot of training and practices to master the skill. Furthermore, even for specialists, the process of development is quite time consuming. Therefore, how to develop a tool to help non-expert users working on FPGA is a promising but challenging work.

In order to manage the ever-increasing size and complexity of designs, the abstraction is gradually more and more essential. As a result of a trend called “raising the level of abstraction”, the developer can focus on the design at higher-level properties that matter most, which helps to avoid being bothered by the lower-level details. This is true in both software programming and hardware design, which have historically evolved toward higher-level languages and models. For software, programming languages have evolved from process-oriented assembly to object-oriented programming, such as C++ and Java. In the meanwhile, coming with the advances of chips and EDA tools functionalities, hardware design has evolved from basic logic elements transistor and gate layout to logic synthesis and high-level synthesis.

In another aspect, coping with large and complex systems, current hardware design practice often relies on integration of components. Although it makes hardware development much easier by allowing modularization and component reuse, because of a lack of support with rigorous methodologies, theories and tools are still managed in a mostly ad-hoc process. Some disturbing troubles or difficulties may be caused by the informal description documents of components. For example, designers can only get descriptions of structural but non-behavioral specifications in IP-XACT, which are usually incomplete. Moreover, the ability to read the files in English is a prerequisite.

Not only some models but also some tools have emerged in recent years, based on the concept of HLS (High Level Synthesis). They can make transformation in high-level languages, such as C and VHDL. These tools are increasingly effective but limited to low stress applications. Other tools exist, such as Simulink / HDL coder based chaining function blocks [65, 64]. This is the easiest way to create, debug and test a processing via simulation. Nevertheless, the results are seldom applicable when actually implemented

in FPGAs.

1.2/ Motivation and Objectives

In general, the existing tools suffer from two flaws. One is that they do not take the physical characteristics of the target architecture of the application into account, including that of the selected FPGA. The other one is that they do not check whether a data stream is processed correctly by the chain, besides creating many test-benches, which is tedious and time consuming for the developer. In fact, these tools can neither analyze the process of a data stream when it passes through the processing graph nor test whether the blocks are actually able to produce the expected outcomes based on the entries and the selected target FPGA. Therefore, they are not suitable to produce applications in real-time environment and high hardware constraints.

These problems have been partially addressed in a previous thesis [32] that proposed a software environment named CoGen. It allows to create chains of blocks to process video streams (or similar ones), and check the capacity of each block to process the stream it receives. Finally, it was able to produce VHDL code for some FPGAs.

The primary objective of this dissertation is to generalize these results to an acyclic graph of blocks, by providing a model of them that allows to determine mathematically the correctness of the result, and thus without launching complex simulations. Every component used in FPGAs is regarded as a block. If the inputs and outputs of connected blocks can be approved compatible, the designed system can be implemented on an FPGA by hardware design languages (VHDL or Verilog HDL). But if the analysis result is negative, algorithms for modifying the design are needed. This is another objective of this thesis. After some modifications, some designs can be implemented on hardware and the correct designs can be derived from our approaches by algorithms. Otherwise, the designs are regarded as unable to be implemented correctly on FPGAs.

The final goal of this thesis is to develop a software tool which can produce VHDL code from a graph of functional blocks. It can do far more than existing ones based on the same concepts. One major new possibility, based on our models and algorithms, is to check whether the result will be correctly carried out by the proposed model and algorithms for a given input stream. The production of final VHDL code is made automatically by assembling previously developed components. This process is called block assembly. Then, the blocks are able to self-schedule their executions because they know the status of executions. Thus, users without VHDL programming skills are able to generate correct code by assembling blocks.

1.3/ Contributions of this Thesis

In this dissertation, we concentrate on the study of the static analysis of block and graph models and the software tool that can help non-expert users for automatic design of FPGA implementations correctly. The main contributions are summarized as follows:

i) The limitations of existing SDF based models, in particular those of the SDF-AP model, are described and illustrated by the analysis of characteristic examples. The

two most common problems encountered in block assembly implementations are the production of incorrect results and the infinite growth of buffer size.

ii) We propose a new model called Actors Stretchable Access Patterns (ASAP) that describes the hardware behaviors as efficiently and precisely as possible. This is a novel way to address the scheduling problem of actors dedicated for FPGA architectures. It opens the possibility to determine the execution correctness mathematically without launching complex simulations. It can not only model actors' behaviors properly, but also avoid the above mentioned drawbacks. Algorithms of transmutations of the patterns and corresponding schedules and output pattern generation are also provided.

iii) We investigate strategies and related algorithms to analyze and schedule graphs of systems. The correctness of the designed systems can be analyzed by a series of algorithms, such as sample rate checking and pattern compatibility checking. Using the proposed ASAP model, the rate decimation and actor's input pattern modifications are applied when a correctness failure is detected. This increases the number of possible real FPGA implementations covered by the block assembly method.

iv) A software tool based on the concept of functional block graph is also developed. It is called BlAsT (Block Assembly Tool) and aims to compensate the drawbacks of other tools based on the same concepts, as for example Simulink + HDL coder. In BlAsT, the proposed ASAP model and related algorithms are used to check that for a given input stream, whether the system can produce a correct result and finally generate VHDL code directly usable on a real FPGA-based board. Otherwise, the tool determines the required decimations and modifications on the graph automatically. It makes a user without any programming skills able to make designs on FPGAs thanks to the friendly graphic interface.

1.4/ THESIS OUTLINE

The dissertation is organized as follows: the next chapter is a review of the related literature dedicated to models for static analysis and related tools oriented to FPGA implementation. A brief analysis of each model and tool is also discussed. Chapter 3 discusses the main limitations of the SDF-AP model and presents the principles of the proposed analysis model: Actors with Stretchable Access Patterns (ASAP). Both basic algorithms for pattern generation and transmutation are given together. Chapter 4 introduces the strategies to analyze a design modeled with the ASAP principles. Algorithms achieving this analysis are given in detail with examples. Furthermore, the proposed principles are verified to be feasible and efficient by some metrics and tests on realistic cases. Chapter 5 describes our EDA tool BlAsT developed based on the concept of block assembly, the proposed ASAP model and related approaches. Functionalities and the methods for using the tool are illustrated by a real application. Finally, the conclusions of the work and some perspectives are given in the last chapter.

I

SCIENTIFIC BACKGROUND

2

SUMMARY OF BIBLIOGRAPHY

2.1/ INTRODUCTION

Hardware devices are widely used nowadays both in scientific environment and daily lives of common people. Among the different types of architectures, FPGAs have become indispensable choices for designers. Regardless of the relative high price, they really improve the reliability and the integration of systems, and they are especially suitable for small batch systems. Modern embedded systems often execute parallel applications. For instance smartphones have four or more processors and several applications can be run for different purposes simultaneously. Thus, there are many expectations for the systems, such as a robust behavior, stable performances and less resource consumption especially minimal energy cost for battery limitation, etc [19, 79]. Thanks to the high degree in integration, it is possible to design complex and large size systems. This leads to the conception of Multi-Processor Systems-on-Chips (MPSoCs). Some discussions about related theories and techniques can be found in [44, 78].

The abstraction of systems allows the designers to deal with large and complex systems. Cooperating with the abstraction, in the meanwhile, the reusable components work as functional blocks in different levels play an important role in hardware design. Both of them help a lot for FPGA design and yield more powerful FPGA applications. But programming MPSoCs is a challenging work, especially timing problems may be caused by the interaction between the components [63]. The design of the requisite communication and the control logic to connect the blocks are still made manually which is time consuming and error-prone. Designing hardware systems is usually done by taking system abstraction and usable components into account to meet the performance requirements. In practice, in order to chose the components to build a system correctly, the real-time environment and hardware constraints should be taken into consideration at the same time. Then, designers must deal with the interfaces of blocks in a manual way. Thus, they are usually faced with low-level control and timing artifacts. Even when all parts of systems are already built, designers still need to spend considerable time to debug [20]. Therefore, it is really hard to design MPSoCs on FPGAs. Some key issues regarding MPSoC design and programming is discussed in [62]. This includes the number of processors, interprocessor communications, concurrency, memory hierarchy, platform scalability, models of programming and control, etc. Although there are some commercial ESL and EDA tools which can provide help to designers, such as Xilinx ISE, they are still like the IP industry requiring to offer large components of the solution.

The question of whether it is possible to find a way to automatically reduce the

complexity of creating designs still remains. Many researchers have worked on it for a long time. With the efforts of researchers, series of models have been proposed. Nowadays, designing hardware often involves model based analysis and techniques [67, 82, 72, 61, 36, 10], and the related tools have also been developed, which can help to check the correctness of systems and give performance evaluations.

The basic problem is discussed in [88] with a vivid name as the glue design problem. An abstract model of hardware systems is provided to analyze the system correctness and performance properties. There are also other models dealing with the similar problems since 30 years ago, but surprisingly none of them described the hardware behavior precisely. Therefore, important information about system are omitted yielding rough system descriptions and resulting in either defensive or aggressive analysis results.

As for tools, there has been an increasing interest in the development and use of standard formats for system structure and IP configuration parameters. Among them, XML-based formats and data flow interchange format are widely used in the software tools development. But the core part to analyze the performance of the system is the model. Therefore, the analysis results of tools based on existing models can be seldom used directly and correctly.

It is promising to find a model that can describe the behaviors of hardware exactly and capture the properties of systems faithfully. If such a model is achieved, the related analysis results can be used in hardware design directly, especially for FPGAs. A software tool based on such a model will be more promising.

2.2/ Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are reprogrammable silicon chips. In 1985, the first FPGA was invented by Ross Freeman, the cofounder of Xilinx. In fact, FPGAs combine the best parts of application-specific integrated circuits (ASICs) and processor-based systems. This why FPGA chips are adopted across all industries. FPGAs provide hardware-timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design. Figure 2.1 shows two FPGA chips produced by Xilinx (on the left) and Altera (on the right), which are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percents of the worldwide market.

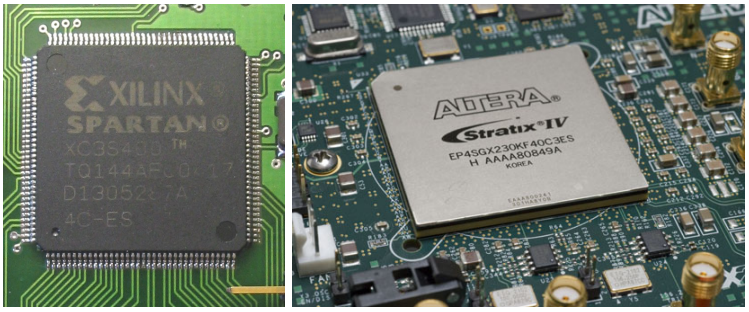

Figure 2.1: FPGA chips produced by Xilinx and Altera.

Reprogrammable silicon has also the same flexibility of software running on a

processor-based system, but it is not limited by the number of available processing cores. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. This makes it possible for designers to get around the basic limitations of sequential processors. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when adding more processing.

2.2.1/ COMPONENTS OF FPGAS

Until now, most of the FPGAs are based on look up tables (LUTs) technique. Every FPGA chip is made up of a finite number of predefined resources. Configurable logic blocks (CLB) are the basic unit of FPGAs. As shown in Figure 2.2, it works with programmable interconnects to implement a reconfigurable digital circuit, I/O blocks (IOB) to allow the circuit to access the outside world, embedded block RAM to make it more flexible in applications and digital clock management modules (DCM).

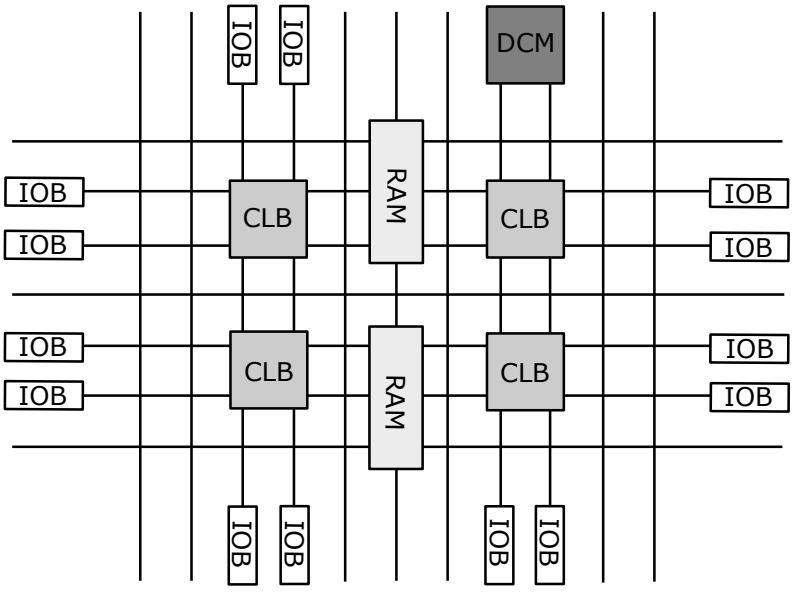

Figure 2.2: The different parts of an FPGA.

Generally, the metrics to evaluate FPGA resource specifications include the number of configurable logic blocks, the number of fixed function logic blocks such as multipliers and the size of memory resources like embedded block RAM. The parts shown in the above figure are typically the most important in FPGA chips. When selecting FPGAs for a particular application, these metrics must be taken into consideration.

The CLBs, sometimes referred to as slices or logic cells, are made up of two basic components: flip-flops and lookup tables (LUTs). Various FPGA families differ in the way flip-flops and LUTs are packaged together, so it is important to understand the principle of flip-flops and LUTs operation.

2.2.1.1/ FLIP-FLOPS

Flip-flops are binary shift registers used to synchronize logic operations and save logical states between clock cycles within an FPGA circuit. On every clock edge, a flip-flop latches the 1 (TRUE) or 0 (FALSE) value on its input and holds that value constant until the next clock edge. As shown in Figure 2.3, a flip-flop has an input, an output and a clock signal.

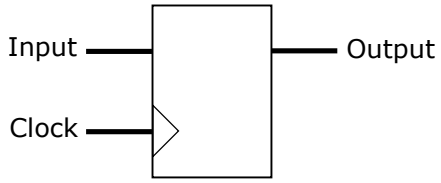

Figure 2.3: The symbol of flip-flop.

2.2.1.2/ LOOKUP TABLES (LUTS)

Much of the logic in a CLB is implemented using very small amounts of RAM in the form of LUTs. Figure 2.4 shows a 4-input LUT. It is easy to assume that the number of system gates in an FPGA refers to the number of NAND gates and NOR gates in a particular chip. But, in reality, all combinatorial logic (ANDs, ORs, NANDs, XORs, and so on) is implemented as truth tables within LUT memory. A truth table is a predefined list of outputs for every combination of inputs.

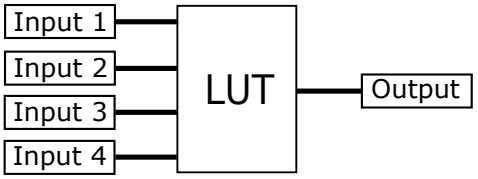

Figure 2.4: An 4-input LUT.

The Boolean AND operation, for example, is shown in Figure 2.5 and the corresponding truth table for the two inputs of an AND operation is shown in Table 2.1.

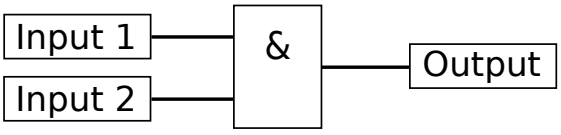

Figure 2.5: Boolean AND operation.

2.2.1.3/ MULTIPLIERS AND DSP SLICES

Figure 2.6 shows a multiply function for the simple task of multiplying two numbers, which can get extremely resource intensive and complex to implement in digital circuitry. This

Table 2.1: Truth table for boolean AND operation

Input1	Input2	Output
0	0	0
0	1	0
1	0	0
1	1	1

is illustrated in Figure 2.7 that shows the schematic drawing of one way to implement a 4-bit by 4-bit multiplier using combinatorial logic. When it comes to multiplying two 32-bit numbers, it ends up with more than 2000 operations. For this reason, FPGAs have prebuilt multiplier circuitry to save on LUT and flip-flop usage in math and signal processing applications.

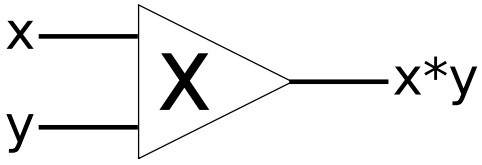

Figure 2.6: Multiply function.

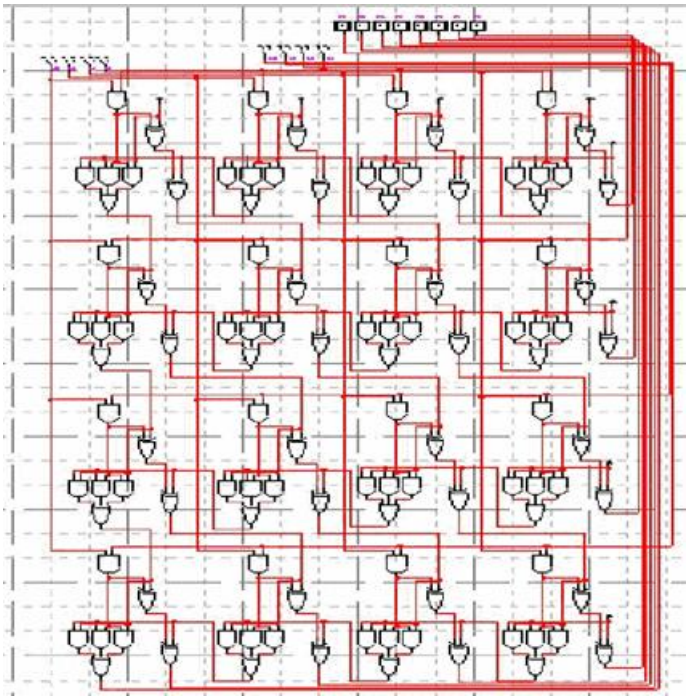

Figure 2.7: Schematic drawing of a 4-bit by 4-bit multiplier.

Many signal processing algorithms involve keeping the running total of numbers being multiplied, and, as a result, higher-performance FPGAs like Xilinx Virtex-5 FPGAs have

prebuilt multiplier-accumulate circuitry. These prebuilt processing blocks, also known as DSP48 slices, integrate a 18×18 bits multiplier with adder circuitry.

2.2.1.4/ BLOCK RAM

Memory resources are another key specification to consider when selecting FPGAs. User-defined RAM, embedded throughout the FPGA chip, is useful for storing data sets or passing values between parallel tasks. Their size and number depends on the FPGA family and model. For example, a Spartan 6 LX100 integrates 536 RAM blocks of 9Kbits. There is still the option to implement data sets as arrays using flip-flops; however, large arrays quickly become expensive for FPGA logic resources. A 100-element array of 32-bit numbers can consume more than 30 percent of the flip-flops in a Virtex-II 1000 FPGA or take up less than 1 percent of the embedded block RAM. Digital signal processing algorithms often need to keep track of an entire block of data, or the coefficients of a complex equation, and without on-board memory, many processing functions do not fit within the configurable logic of an FPGA chip.

The inherent parallel execution of FPGAs allows for independent pieces of hardware logic to be driven by different clocks. Passing data between logic running at different rates can be tricky, and on-board memory is often used to smooth out the transfer using first-in-first-out (FIFO) memory buffers.

2.2.2/ FPGA DESIGN AND TOOLS

With the understanding of the fundamental FPGA components, the advantage of implementing a design in hardware circuitry can be seen clearly: it allows improvements in execution speed, reliability, and flexibility. However, in the process of FPGA design some trade-offs should be made based on an FPGA for the processing and I/O connectivity in a system.

For FPGA design, the designer defines digital computing tasks in software using development tools and then compile them down to a configuration file or bitstream that contains information on how the components should be wired together. Although there are many development tools, the challenge in the past with FPGA technology was that the low-level FPGA design tools could be used only by engineers with a deep understanding of digital hardware design. However, the rise of high-level synthesis (HLS) design tools, such as Simulink developed by MathWork and the NI LabVIEW system design software, changes the rules of FPGA programming in some degree.

2.2.2.1/ THE DESIGN FLOW

As the design is the main part of FPGA implementations, it is necessary to talk about the entire process. Figure 2.8 shows the design flow of hardware devices, not only for FPGA but also for others, such as ASIC and CPLD. Following the steps in the design flow can guarantee the best chance to get a correct prototype of the designed system.

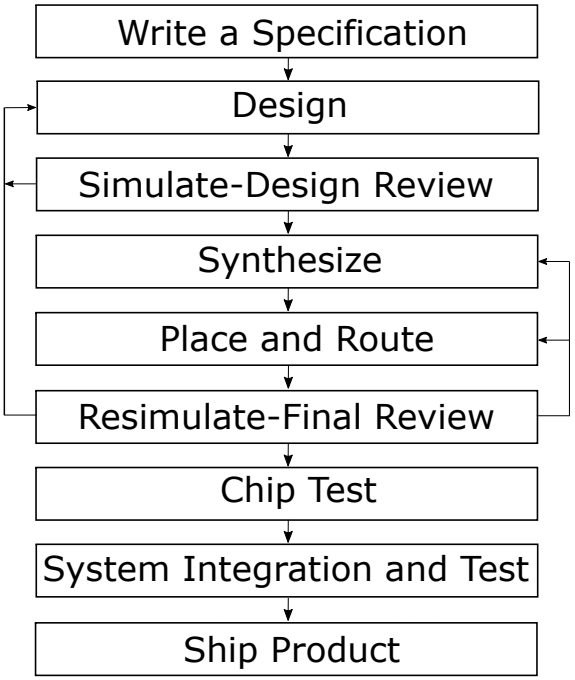

Figure 2.8: The flow chart of FPGA design.

Writing a specification is the first step. This plays a quite important role as a guide for designers to understand the entire design and choose the proper devices. It allows to design the correct interface to the rest of the pieces of the chip. It also helps to save time and avoid misunderstanding. The following information should be included in a specification:

- An external block diagram showing how the chip fits into the system.
- An internal block diagram showing each major functional section.
- A description of the I/O pins, including output drive capability and input threshold level.
- Timing estimates, including setup and hold times for input pins, propagation times for output pins and clock cycle time.
- Estimated gate count.
- Package type.
- Target power consumption.
- Target price.
- Test procedures.

With the specification, the designer can find the best vendor with a technology and structure that best meets the requirements of the project. At this point, a design entry method should be chosen. Generally speaking, for small chips, schematic entry is used,

especially if the designer is familiar with the tools. But for large designs, a hardware description language (HDL) such as VHDL or Verilog HDL is used for its portability, flexibility, and readability. When using a high level language, synthesis software is required to "synthesize" the design. This means that the software creates low level gates from the high level description. Thus, at the same time, the designer should choose a synthesis tool, which is important since each synthesis tool has recommended or mandatory methods of designing so that it can work properly.

After knowing the aim of the system to be designed and get all the preparation ready, it comes to the core step designing the chip. This is the most essential and difficult work for hardware design, which mainly involves programming using the chosen hardware description language.

In order to get the correct functionality, simulation is a process while the design is being done. Small sections of the design should be simulated separately before hooked up to larger sections. There should be many iterations of design, simulation and review for a final system.

When the design is finished, the designer should use the synthesis software to synthesize the chip. It involves translation of the register transfer level (RTL) design into a gate level design which can be mapped to logic blocks in the FPGA. And then, the design should be programmed into chip, which is called place and route. The design results in a real layout for a real chip.

After layout, another simulation is necessary to check whether the real chip goes on well and the results agrees with the predicted ones. If every part of the system performs correctly, finally, system integration and system testing is carried out to insure all parts of the entire system work correctly together. When there is no problem in the system, the product of the design is obtained at last.

2.2.2.2/ Traditional Design Tools

In the whole process of hardware design, the tools used are critical to the design. Through the first 20 years of FPGA development, hardware description languages (HDLs) such as VHDL and Verilog evolved into the primary languages for designing the algorithms running on FPGA chips. These low-level languages integrate some of the benefits offered by other textual languages taking into account that on an FPGA, a circuit is architectured. The resulting hybrid syntax requires signals to be mapped or connected from external I/O ports to internal signals, which ultimately are wired to the functions that house the algorithms. These functions execute sequentially and can reference other functions within the FPGA. However, the true parallel nature of the task execution on an FPGA is hard to visualize in a sequential line-by-line flow. HDLs reflect some of the attributes of other textual languages, but they differ substantially because they are based on a dataflow model where I/O is connected to a series of function blocks through signals.

To verify the logic created by an FPGA programmer, it is a common practice to write test benches in HDL to wrap around and exercise the FPGA design by asserting inputs and verifying outputs. The test bench and FPGA code are run in a simulation environment that models the hardware timing behavior of the FPGA chip and displays all of the input and output signals to the designer for test validation. The process of creating the HDL test bench and executing the simulation often requires more time than creating the original

FPGA HDL design itself.

Once an FPGA design using HDL is created and verified, it needs to be fed into a compilation tool that takes the text-based logic and, through several complex steps, synthesizes the HDL down into a configuration file or bitstream that contains information on how the components should be wired together. As part of this multistep manual process, it often requires a mapping of signal names to the pins on the FPGA chip that is used.

Ultimately, the challenge in the design flow is that the expertise required to program in traditional HDLs is not widespread, and as a result, FPGA technology has not been accessible to the vast majority of engineers and scientists.

2.2.2.3/ HIGH-LEVEL SYNTHESIS DESIGN TOOLS

Thanks to the emergence of graphical HLS design tools, such as LabVIEW, some of the major obstacles of the traditional HDL design process are removed. The LabVIEW programming environment is distinctly suited for FPGA programming because it clearly represents parallelism and data flow, so users who are both experienced and inexperienced in traditional FPGA design processes can leverage FPGA technology. In addition, existing VHDL codes can be used to be integrated within designs. The Intellectual Property (IP) can be used, among which IP blocks from native and third-party sources are widely used. The most familiar IP sources are Xilinx CoreGen [40], National Instruments LabVIEW FPGA [42], and the OpenCores library [1].

Then to simulate and verify the behavior of a FPGA logic, LabVIEW offers features directly in the development environment. Without knowledge of the low-level HDL language, one can create test benches to exercise the logic of the design. In addition, the flexibility of the LabVIEW environment helps more advanced users model the timing and logic of their designs by exporting to cycle-accurate simulators such as Xilinx ISim.

2.2.3/ ANALYSIS

The adoption of FPGA technology continues to increase as higher-level tools such as LabVIEW and Simulink, the standard microprocessor, and the FPGA RIO architecture are making FPGAs more accessible. It is still important, however, to look inside the FPGA and appreciate how much is actually happening when block diagrams are compiled down to execute in silicon. Comparing and selecting hardware targets based on flip-flops, LUTs, multipliers, and block RAM is the best way to choose the right FPGA chip for your application. Understanding resource usage is extremely helpful during development, especially when optimizing for size and speed. The existing HLS design tools deliver new technologies that convert graphical block diagrams into digital hardware circuitry, but they may lead to unusable results or resources waste when actually implement in FPGAs for some limitations of the analysis models. They are either defensive or aggressive, even ignoring some important characteristics. Thus, there are much room to improve the high-level synthesis tools to make hardware design easier.

2.3/ Models for Static Analysis

In order to improve the tools for hardware design, we should try to remove the obstacles for designers. For the implementation of complex systems, what is the most difficult thing for designers? It is obvious that when dealing with a single processor system, every component executes sequentially, which can be built easily. Therefore, it is concurrency that makes it complex [13, 34]. In order to solve this problem, some models are proposed and have shown to be effective. Among them, Synchronous (or Static) Data Flow (SDF) are widely developed and used, notably for digital signal processing applications implemented on parallel hardware. This model was introduced by Lee and Messerschmitt in the founding article [55]: "Data Flow is a natural paradigm for describing DSP applications for concurrent implementation on parallel hardware" in 1987. This remark was done when the first FPGAs emerged but thirty years later, even if FPGAs are far more powerful and the field of applications much larger than DSP, the same problem occurs: how to build a hardware design by connecting blocks so that it produces correct results? Indeed, since hand-coding a whole design in VHDL is a very fastidious and time consuming task, describing it with a Data Flow Graph, composed of functional blocks that consume and produce data requires less efforts. Nevertheless, this way leads to two main problems to be solved:

- Finding the best conditions under which the graph produces correct results.
- Transfer the model to a real and functional implementation (e.g. in VHDL) for a chosen architecture.

The first problem has been investigated for different models, all based on the original one presented in [55] and [59] named Synchronous Data Flow (SDF). They all rely on the fact that the number of data consumed and/or produced during the execution of a block is fixed and known a priori. The execution time is also in the same case. The discussion of the possibility to make static analysis is argued in [4, 5, 50, 51]. Compared with Finite State Machine (FSM) [30], it leads to a simpler method in most cases. Although, FSMs abstracts hardware properties at a high level, it causes the problem known as state explosion problem [91].

A comprehensive survey on concurrent models of computation can be found in [56]. Prior research has shown that data flow and its variants are proposed to capture the task and data parallelism in streaming applications.

2.3.1/ Synchronous Data Flow

2.3.1.1/ Principles

The Synchronous Data Flow is a model to compute the operation of infinite streams of data. This model is described mathematically as a directed graph, in which nodes represent *actors* and edges denote inter-actor communication of data. These edges are usually called *channels*, which connect actors with *ports*. This means that, in an SDF model, data are passed in the form of tokens among actors linked by channels. The model can be used to analyze processing chains of data in a system, which is the same as Petri nets [69]. The execution (or firing) of actors consumes tokens on input ports,

and in the meanwhile, produces new tokens on output ports. Thus, the number of tokens consumed and produced in a firing (named as consumption and production rates, respectively) should be fixed and pre-specified at design time. If the rates in an SDF graph are not the same, it is named multi-rate graph, otherwise it is homogeneous [57, 81].

Let us take an SDF model of a Source-Downsampler for example, which is shown in Figure 2.9. A 3:1 decimation downsampler actor *D* with one input port and one output port would consume three tokens from the source actor *S* via the channel *C*. In fact, there is a *buffer* to temporarily store produced tokens. When *S* fires, the execution lasts for two cycles (marked as ET in the figure) and at the end it produces one token. For *D*, the execution lasts for four cycles and during the first three cycles it consumes three tokens from the input port and at the end produces one token to the output port. It is mandatory that *S* must execute three times, so that *D* can consume three tokens (polled from the buffer) for its own execution. With the rates specified statically, it can be analyzed to verify properties such as consistency of production and consumption rates and to ensure that enough communication buffer space is allocated for correct execution. From this process, it guarantees decidability of main model properties: existence of deadlock-free and memory-bounded infinite computation, throughput, latency, and execution schedule [7, 73]. The expressiveness of SDF model capturing streaming applications naturally, coupled with its strong compile-time predictability properties, has been widely used for specifying embedded real-time applications in the domains of digital signal processing, such as DSP and FPGA hardware designs.

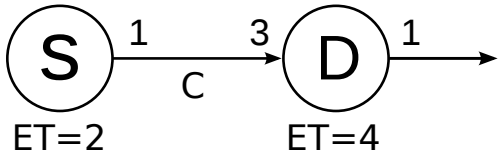

Figure 2.9: A Source-Downsampler presented by SDF.

Once the number of needed tokens is available on the input channels and enough vacant space appears on its output channels to store the tokens it will produce, the SDF model can be executed in a self-timed manner [84]. An actor reads its inputs from the respective buffers in the order in which the input tokens were produced and put into the buffer, and outputs its tokens to its production buffers. Typically, the buffers used between actors are FIFOs (First In, First Out). Handshaking is often used to implement the self-timing: an actor stalls until necessary resources (both input tokens and output space) are available, and its neighbors inform it when necessary inputs have been produced or output space has been released.

Static scheduling is applicable with SDF models. Without execution, SDF actors can be statically scheduled according to the information of consumption and production rates. The number of data tokens produced or consumed on each input and output port is specified a priori. Lee and Messerschmitt also present necessary conditions for static scheduling programs described in SDF graphs onto processors (single or multiple) in [7]. Self-timed implementations are also widely used, for a long time, which was considered as the best approach for data flow based models [53]. However, sometimes it is needed to support multiple applications running on a single system without prior knowledge of the properties of the applications at design-time. Under this circumstance, run-time scheduling approaches are needed as explained in [47].

2.3.1.2/ ANALYSIS

The SDF model is usually used to describe the abstract of hardware components [37, 14]. With the efforts of researchers, some efficient algorithms appeared, which can be used to compute performance metrics of SDF model, such as throughput, buffer sizes, as well as execution schedule. It has been investigated that for analyzing the timing behavior of applications, using the standard SDF model is a common practice to associate worst-case execution time models [39, 68, 75, 96, 95]. These timing information makes it possible for static analysis of SDF model. Moreover, it can provide mapping solutions to specific platforms under resource and performance constraints. These models have been applied to capture behavior of SDF actor executions for software and hardware implementations. However, losing information about the precise timing of consumption and production of tokens by an actor during a firing cycle is the main drawback of timing models. When using SDF models to analyze hardware implementation behavior, this problem is more obvious. For hardware IP blocks, data tokens should be delivered to them at precise clock cycles. But for SDF model, it can only describe the number of needed tokens and produced during an execution. This loss of exact timing information in SDF model leads to more latency and resource usage than necessary and finally results in sub-optimal analysis and implementations. Thus, the results are usually conservative.

We can take a simple design for example. Assume a producer *A* is connected to a consumer *B*. *A* fires and produces one token per clock cycle, and each execution of *B* lasts for six clock cycles and consumes six tokens per firing. But this behavior of implementing can not be captured effectively by SDF timing model. The SDF model assumes that an actor should wait to execute until there are sufficient tokens available at the inputs. For this example, the IP block of *B* needs six tokens in six consecutive clock cycles. Thus, *B* cannot fires until *A* completes six firings and produces six tokens. It results in the usage of a FIFO between *A* and *B*, the size of which should be at least six. And, *B* can execute only after the buffer has collected six tokens from *A*. It is obvious that this is a valid implementation, but the result is sub-optimal in terms of throughput and allocation of buffer resources. In practice, this example can be achieved by a better implementation using only a FIFO of size one.

The above problem leads to a main question that when actors can start their firing. An obvious answer to reach maximal throughput is as soon as possible, as the timed actor interface theory "*the earlier the better*" as Geilen proposed in [23]. But SDF model based schedules, not only self-timed but also statically scheduled [84], share one key characteristics-the actors do not execute until all necessary inputs are available and all required output space are free. Because it takes time to create tokens and fill output FIFOs, actors are guaranteed to be stalled some portion of the time while one of those two processes is going on. Although there are some run-time scheduling algorithms, due to the fact that they assume independent periodic or sporadic tasks, such simple task model is not usable for modern embedded systems. Basically, it may lead to infinite buffers. Thus, we have to come back to solve the problem mentioned above. In order to compute a valid and optimal schedule with finite buffers, some assumptions should be set:

- Tokens are all produced at the end of the executions, in a single "shot".
- These tokens are stored in a buffer, the size of which should be big enough.

- Tokens used as inputs stay in the buffer until the end of the execution.

These assumptions yield a model quite distant from the behavior of real design in VHDL. Indeed, output data are produced sequentially, sometimes in the middle of the execution. Furthermore, such a buffer is more complicated than a classical FIFO since a block can ask for tokens without flushing them from the buffer immediately, which is like a mix of FIFO and memory.

2.3.2/ Cyclo-Static Data Flow

2.3.2.1/ Principles

In the middle of 1990's, Bilsen and Engels and al. introduce a model named Cyclo-Static Data Flow (CSDF) for system analysis, which makes improvement based on SDF by breaking a firing of actor into finer-grained *phases*. Generally speaking, CSDF makes up the shortcomings in some degree. It allows the consumption or production rates of an actor to vary periodically according to the given cyclic pattern. Thus, the consumption and production of tokens are specified for each phase. For CSDF model, every firing of an actor is refined to correspond to a phase, which is different from the firing of an actor for SDF model. The authors give the necessary and sufficient conditions for the possibility of a static schedule of a CSDF graph and details of methods for a static analysis for a system in [16, 9, 8].

In order to illustrate the CSDF model more clearly, the same example shown in Figure 2.9 can be expressed by CSDF model in Figure 2.10. The execution time of *S* is two, so it is presented as two phases, each taking one clock cycle to execute: in phase 1, *S* produces nothing; in phase 2, it produces one token. Similar as *S*, *D* contains four phases in one firing, also taking one cycle for each: in first three phases, *D* consumes one token in each phase and produces nothing; in phase 4, *D* consumes nothing and produces one token. Compared with SDF model, it makes the schedules of firings more efficient by shortening the waiting times for the needed tokens. Moreover, Thomas and al. made their efforts to express a transformation from CSDF graphs to SDF graphs. Thus, some of existing SDF scheduling techniques can be used in CSDF model. But it is not always feasible for every case. This transforming sometimes introduces deadlock. The detail comparison is discussed in [71].

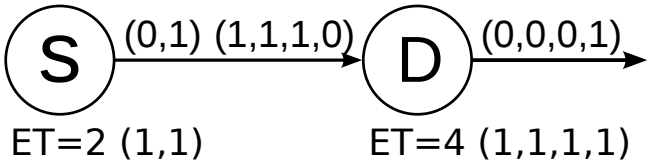

Figure 2.10: A Source-Downsampler presented by CSDF.

2.3.2.2/ Analysis

CSDF model refines the unit of firing to phase, which is much smaller than original execution of actor and makes the latency of scheduling shorter. It does not need to wait for

all the tokens needed for an actor available, only for the basic unit (phase) is enough. In this aspect, CSDF model has a better performance than SDF model.

CSDF model still relies on the same basic hypothesis as SDF model that the actor of execution must wait until sufficient tokens have been available at the input channels before firing a phase. However, this hypothesis can not meet the requirements related to the precise timing of tokens. In the above example of producer *A* and consumer *B*, *B* requires to receive six tokens in six consecutive clock cycles once it starts execution. This constraint can not be taken into consideration either and as a result can lead to incorrect implementations [88], which is more worse than for SDF model. Actually, if the execution time of producer *A* is two, the CSDF model would make the conclusion that only a buffer of size one is needed between *A* and *C*, but this is a conflict to the timing requirement. Therefore, it leads to incorrect analysis result.

A possible method to solve the incorrect scheduling is to make a switch to turn on or turn off an actor. There is no doubt that it can be easily implemented in software. When it comes to hardware implementation, an enable signal can be used to regulate the execution. The signal 0 (false) can be used to disable the actor when input data are unavailable on input ports, vice versa. However, the increasing numbers of enable signals have further impacts to achieve high frequency. Thus, in practice, the enable logic is undesirable for the increased latency. As for the drawbacks, therefore, researchers have never stopped exploiting for better solutions.

When it comes to eliminate the deadlock problem mentioned above, some techniques have been proposed, such as the polyphases filtering algorithm [90] and the mixer (a multi-rate actor) introduced in [80]. But the improvements got in this aspect are in application of CSDF model, which are not substantive modifications in the sense of model principles.

2.3.3/ Static Data Flow with Access Patterns

2.3.3.1/ Principles

The Static Data Flow with Access Patterns (SDF-AP) model of computation was introduced informally in [88] and established in [29]. SDF-AP model attempts to overcome the limitation and overlap execution of the actors by initiating execution at the earliest possible time, rather than waiting for all inputs to become available, which specifies in addition to token rates the specific cycles (relative to the start of the firing of an actor instance) in which individual tokens are produced or consumed. SDF-AP strikes a balance between the analysis capacities of SDF and CSDF while accurately capturing the interface timing behavior. The latter is achieved by specifying **access patterns** that capture the precise timing behavior of token productions and consumptions. Access patterns describe when tokens are produced/consumed, in terms of clock cycles, from the beginning of the execution of an actor. It models the actor's behavior in a similar fashion to that of a real block. Buffers can be simple FIFO driven by a controller and finally, the buffer size and latency can be reduced and throughput rate increased (sometimes drastically) when comparing real applications implemented with SDF or SDF-AP approaches [93].

An SDF-AP model is similar to other SDF based models, except that each input and output terminal of an actor is annotated by a vector called **consumption pattern** (CP) for an input terminal, and **production pattern** (PP) for an output terminal. Each such

vector has length equal to the execution time of the actor. Each element of the vector describes the number of tokens to be processed in the corresponding cycle. It is most common for hardware implementations to process at most 1 token per cycle, so CPs and PPs are assumed to consist of only 0 and 1. For each clock cycle from the moment the actor is triggered, a sequence of 1 and 0 describes the fact that it consumes/produces a token or not. Nevertheless, the concept can be easily adapted to capture actors that process more than 1 token per cycle by standardizing the hardware protocol of terminals. SDF-AP model should not be confused with CSDF model, despite the fact that the two models share great similarity in syntactic notation. The main difference is that SDF-AP model defines strict timing for all the tokens in one firing, while CSDF model defines the timing for each phase but is not at all strict between the firing of phases (in particular, CSDF model allows stalling between phases).

Figure 2.11 shows an SDF-AP model for the Source-Downsampler example, which also contains two actors, *S* and *D*. *S* produces one token every time it fires, but its execution time is two clock cycles. The access pattern additionally specifies that, each time *S* is triggered, it produces nothing during its first clock cycle and one token during the second. When *D* is triggered, it consumes three tokens and produces one token every time it fires, and its execution lasts four clock cycles. To be more precise, the access pattern at the input port to *D* specifies that it consumes one token every clock cycle for the first three of one firing. The access pattern at the output port of *D* specifies that it does not produce any token in the first three clock cycles, and produces one token at the last clock cycles.

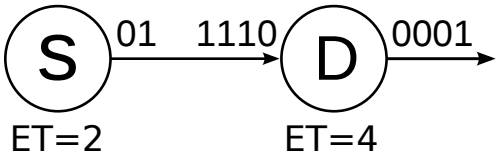

Figure 2.11: A Source-Downsampler presented by SDF-AP.

2.3.3.2/ ANALYSIS

It is important to notice that, similar to the tokens in SDF and CSDF models, these patterns must be strictly matched. Thus, in the example in Figure 2.11, if there is a valid data presented on the input of *D* at the fourth clock cycle of its execution, or if there are no valid data at clock cycles 1, 2 or 3, the actor will produce incorrect results. This has great consequences. For example, assuming that *S* is triggered every two clock cycles, the *production pattern*, i.e. the clock cycles at which it produces valid data, is $\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \dots \end{bmatrix}$. Nevertheless, without buffering, it is impossible to find a schedule to trigger *D* so that it produces correct results, since it will always consume tokens at clock cycles that do not correspond to a valid output of *S*. Moreover, this remark is true for any triggering schedules of *S*.

To solve this problem, SDF-AP model assumes that, as other SDF based models, the channel between two actors is a buffer. In [89] and [88], the same group of authors proposed a realistic representation (from the hardware point of view) of these buffers: a FIFO with a controller that manages the store and poll requests. This general structure is shown in Figure 2.12. Generally, a store request is issued as soon as a token is produced by *S* on `data_out`, that is when `data_o_enb` is asserted. Poll requests occur at clock cycles

computed thanks to the access patterns and the scheduling. In order to lower the global latency and to minimize the size of the buffer, tokens must be polled as soon as possible but in a sequence that matches the input pattern of *D*.

To illustrate this behavior, we take the design of Figure 2.11 and the production pattern given above. The couple FIFO/controller is supposed to behave as a VHDL implementation synchronized on a global clock. It means that for a poll request at clock cycle t, the result is available on the FIFO's output at clock cycle $t + 1$ (same with store requests). Under theses conditions, the SDF-AP model implies the following sketch:

- At clock cycle 2: *S* produces its first token and the controller issues a store request.
- At clock cycle 3: the first token is available in the FIFO.
- At clock cycle 4: *S* produces its second token and the controller issues a store request. It also issues a poll request.
- At clock cycle 5: the second token is available in the FIFO and the first token is available on `data_in` of *D*. This latter starts its execution and consumes it. The controller issues a poll request.
- At clock cycle 6: the second token is available on `data_in` of *D*, that consumes it. *S* produces its third token and the controller issues a store request.
- At clock cycle 7: the third token is available in the FIFO and and the second token is available on `data_in` of *D*. This latter starts its execution and consumes it. The controller issues a poll request.
- At clock cycle 8: the third token is available on `data_in` of *D*, that consumes it. *S* produces its fourth token and the controller issues a store request.
- At clock cycle 9: the fourth token is available in the FIFO and *D* produces its result.
- ...

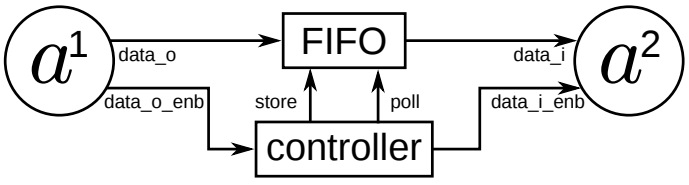

Figure 2.12: General structure (FIFO+controller) to interconnect two actors in SDF-AP.

This leads to an execution of *D* every six clock cycles, the first being at clock cycle 5 and a minimum size of two for the FIFO. With a SDF model, the throughput would be the same but with a first execution at clock cycle 8 (or even more depending on the access policy of the buffer) and a minimum buffer size of four. In the presented case, the gain is small but for actors that consumes hundreds of tokens, it can be huge [88, 93].

2.3.4/ Other Data Flow Based Models

2.3.4.1/ Some Efforts in Data Flow Based Models

Many efforts have been done to improve the performance of the methods based on data flow. In 1994, Feautrier presents fine-grain scheduling of loops under resource constraints for micro-processors [17]. More generally, Fimmel and Muller present an approach for optimal software pipelining of a loop under resource constraints in 2001, where the schedule period and periodicity are treated as variable and the objective is to minimize the schedule initiation interval [18]. One year later, Govindarajan and al. use ILP to minimize buffer requirements under rate-optimal schedule for multi-rate SDF graphs [35]. In their approach, a rate-optimal schedule is one that achieves the best throughput when no actor auto-concurrency is considered. To achieve an ILP formulation, a good amount of approximation is introduced, which leads to sub-optimality.

In 2010, Benazouz and al. presented an ILP-based approach to minimize buffer capacities for embedded systems specified using marked timed weighted event graph, which is a sub-class of Petri nets [3]. It turns out that this representation is equivalent to the SDF graphs with back edges used to encode the buffer capacities. Their work was extended to CSDF graphs with regular periodic schedules in [2]. Their computation of the sufficient condition on the minimum delay between the first firing start times may introduce artificial constraints and thus it may lead to sub-optimality.

In 2000, Goddard and Jeffay exposed transformation techniques for designing real-time systems described as processing graphs, which are similar to SDF graphs [33]. One key difference between a processing graph and an SDF graph is the threshold specification for each channel in the processing graph. A node in a process graph can start its execution only when all its input channels have at least a certain number of the required input tokens defined by the channel thresholds and its output channels have at least a certain number of vacancies defined by respective channel production rates.

In 2007, Wiggers and al. showed that analyzing designs based on CSDF models may lead to a significant buffer capacity reduction compared to a multi-rate dataflow model specification [96]. They formulated minimum buffer capacity computation for CSDF graphs under regular periodic schedules as a network flow problem. In their formulation, what is actually minimized is the sum of start times of the first firings of all actors. They argue how minimization on this sum could lead to buffer capacity reduction. However, in general, minimization on total buffer capacity does not correspond to minimization on the sum of actor start times.

In 2008, Stuijk and al. propose a model checking-based approach to trade off throughput and buffer storage for SDF and CSDF models [84]. Based on the charted Pareto space, the minimal buffer space is determined to meet a user-specified throughput constraint. To avoid the potential long run-times, an approximation technique is proposed to reduce the number of distributions in the design space.

In 2010, Kee and al. present computation of an upper bound on buffer distributions for throughput-optimal scheduling of SDF graphs implemented onto FPGA devices. Their sizing algorithm works only for tree-structured graphs [48]. Recent variants like Heterochronous Data Flow (HDF) [31], Scenario-Aware Data Flow (SADF) [86, 87, 85], and Core Functional Data Flow (CFDF) [39] extend SDF or CSDF model with specifications for control. In a syntactical perspective, the former two are the same. There difference

is that FSM-based SADF focus on timing analysis of parallel executions [86, 28]. In contrast, the analysis techniques based on HDF model focus on sequential executions. It does not have a timed version which can be used for timing analysis.

There are also some models based on data flow that takes dynamics into account allowing design-time analysis, so that can be implemented reasonably efficiently. Differing from other models, in Parameterized Synchronous Data Flow (PSDF) model [6], the rates of the ports are parameterized rather than constant. Parameterized schedules and buffer sizes can be computed according to the given parameters. However, options to express dynamics are limited by its principles. Similar as PSDF model, Variable Rate Data Flow (VRDF) model [97] also does not require constant rates. Port rates are allowed to vary arbitrarily within a specified range. Variable Phased Data Flow (VPDF) model [98] is an extension of VRDF model based on CSDF model where the number of repetitions of CSDF phases can be parameters from some finite intervals. Existing analyses of VRDF and VPDF models are limited to computing buffer sizes under a throughput constraint.

Data flow based models such as Boolean Data Flow (BDF) model and Dynamic Data Flow (DDF) model [12] allow data-dependent firing rules. This makes them Turing-complete in the sense that they can operationally simulate a Turing machine. Consequently, it is impossible to realize an exact analysis of their timing behavior and buffer sizes at design-time. These models require run-time scheduling and deadlock detection. This makes their implementation far less efficient compared to all models discussed so far.

The Kahn Process Network (KPN) model [46, 70, 21] is another model that can be used to express application dynamism. The Reactive Process Network (RPN) model [22] extends KPN with state transitions that allow it to change the function of the process network based on events. Both KPN model and RPN model do not allow for design-time analysis and require a complex run-time mechanism that incurs a large implementation overhead. We consider DDF model more expressive than KPN model because the definition of DDF model given in [12] allows non-functional behavior, which cannot be expressed in KPN model.

2.3.4.2/ ANALYSIS

Data flow based models make it possible for strong compile-time analyzability of hardware implements (DSPs and FPGAs). The number of data produced and consumed during each firing can be explicitly specified. This results in the execution properties of designed systems can be analyzed with efficient algorithms, such as deadlock absence, channel boundness, and throughput at the design-time statically [25, 94, 26, 27, 24]. This is the most successful aspect of these models. Generally, systems specified using these models can be implemented as both software and hardware [84, 48, 97, 99].

On another side, all of these approaches based on SDF/CSDF model have the limitations about available information. For the execution, actors in all these models should wait until all necessary tokens are available on input ports. Schedule and throughput vary with the amount of memory allocated for inter-actor communication. Analysis techniques for these models usually overestimate communication storage requirements, which has very detrimental effects on embedded real-time systems. This is the main reason for sub-optimal results (more resources than necessary) or even incorrect results [14, 49, 45]. Thus, to achieve resource optimal implementations, capturing the precise timing of token

accesses is the most important. For this point, no one is equipped.

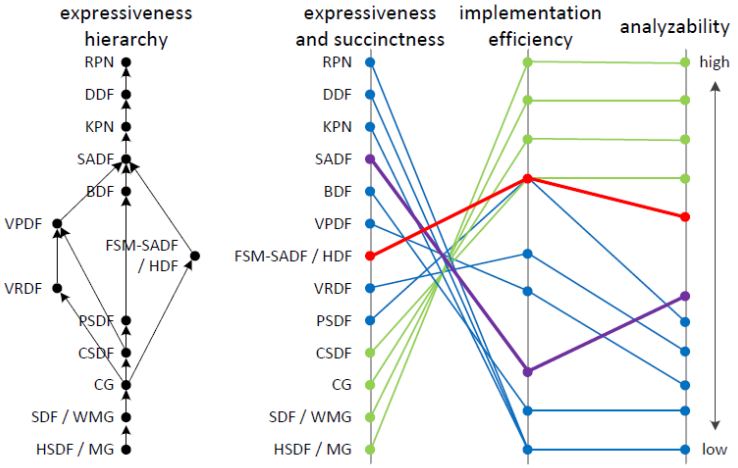

Figure 2.13: Hierarchy graph of dataflow based models.

As it was discussed in [85], the comparison of some models talked above is shown in Figure 2.13. It shows the relations studied in the literature and straightforward ones. On the left side, an expressiveness hierarchy for the models is visualized. Some edges are drawn from one model from another for the reason that there is either a same instance for them, or a transformation that can transform one model to another semantically equivalent model. The right side shows a deeper comparison in three aspects: expressiveness, analyzability and implementation efficiency. We can see clearly that from HSDF/MG model to RPN model the expressiveness and succinctness go up. In the meanwhile, the analyzability shows the same trend as the implementation efficiency going down generally. It is easy to understand that the more the succinctness of the model, the more details of the hardware are omitted, so that the less analyzability and efficiency the model has.

2.3.5/ Scheduling of Hardware Systems

Combined with the analysis models, scheduling plays an important role to analyze the behavior of the systems, even more important than the models themselves. Scheduling of the models includes two aspects, one is determining the order of the actors to fire, the other is the time of when they are fired. For multi-processor systems, there is another aspect to arrange the execution to idle processors. But for FPGAs, it can be regarded as having enough processors for every execution. Therefore, it is necessary to take it into consideration.

Generally, there are three classes of scheduling strategies based on different models, dynamic scheduling, self-timed scheduling, and static scheduling. In dynamic scheduling, actors are scheduled at run-time only. When all inputs for a given actor are available, the actor is fired. In self-timed scheduling, the compiler determines the order in which actors fire. At run-time, the processor waits for data to be available for the next actor in its ordered list, and then fires that actor. Because of its similarity to self-timed circuits, it is called self-timed scheduling. The last type of scheduling is fully static scheduling, where

the compiler determines the exact firing time of actors, as well as their ordering. This is analogous to synchronous circuits. As discussed in [53], if taking the assignment to different processors, there will be more classes. The boundary between these categories is not quite rigid. But the key point to classify the scheduling strategies is based on the time of scheduling at compile time or run-time.

Different scheduling strategies for existing models with various details are provided by researchers. Usually, the automatic scheduling strategies cause more or less additional cost and each strategy is limited to a specific domain. The main principle for the research of scheduling is to reduce the implementation cost, which often means the more is done at compile time the better. In fact, for the nature of the models, static and self-timed scheduling strategies are suitable for SDF based models. From the basic SDF model to the latest SDF-AP model, the researchers have been trying to make a balance among different aspects, such as complexity, function, cost, etc.

2.3.6/ Design Frameworks of Data Flow Based Models

Combined with the analysis models, there are also many software tools and frameworks based on data flow with different characteristics. Design frameworks like Ptolemy [11, 54], Ptolemy-II [15, 60], SDF^3 [83], SPIRIT [41], DIF [38, 39] and OpenDF [66, 43] deliver hardware and software implementations. In the industry, LabVIEW FPGA from National Instruments supports FPGA deployment for homogeneous static data flow models [42]. System Generator from Xilinx supports FPGA implementations from synchronous reactive and discrete time models of computation. These frameworks provide extensive libraries for common math and signal processing functions and enable easy integration of Intellectual Property (IP) blocks from native and third-party libraries.

Ptolemy is a framework that supports data flow programming for simulation and prototyping of heterogeneous systems. Several data flow (DF) models are supported, such as synchronous data flow (SDF), dynamic data flow (DDF) and discrete-event (DE). Object-oriented software technology (C++) is used to model each subsystem in a natural and efficient manner, and to integrate these subsystems into a whole system. Ptolemy encompasses practically the designing signal processing and communications systems, ranging from algorithms and communication strategies, simulation, hardware and software design, parallel computing, and generating real-time prototypes [74]. In order to realize a computational model appropriate for a particular type of subsystem, Domain is employed as a basic abstraction in Ptolemy. Domains can be mixed as appropriate to realize an overall system simulation. The applications of Ptolemy include networking and transport, call-processing and signaling software, embedded micro-controllers, signal processing, scheduling of parallel digital signal processors, board-level hardware timing simulation, and combinations of these.

Ptolemy II is an improved version of Ptolemy with more functions supporting experimentation with actor-oriented design. It inherits the characteristics from the previous version, such as actors, models and Domains. In Ptolemy II, the semantics of a model is not determined by the framework, but rather by a software component in the model called a director, which implements a model of computation. The Ptolemy project has developed directors supporting process networks (PN), discrete-events (DE), data flow (DF), synchronous/reactive(SR), rendez-vous-based models, 3-D visualization, and continuous-time models. Each level of the hierarchy in a model has its own director, and distinct

directors can be composed hierarchically. Distinct directors can be composed hierarchically with state machines to make modal models [52]. A hierarchical combination of continuous-time models with state machines yields hybrid systems [100]; a combination of synchronous/reactive with state machines yields StateCharts [58]. The core of Ptolemy II is a collection of Java classes and packages, layered to provide increasingly specific capabilities. The kernel supports an abstract syntax, a hierarchical structure of entities with ports and interconnections. A graphical editor called Vergil supports visual editing of this abstract syntax. An XML concrete syntax called MoML provides a persistent file format for the models. Various specialized tools have been created from this framework, including HyVisual (for hybrid systems modeling), Kepler (for scientific workflows), VisualSense (for modeling and simulation of wireless networks), Viptos (for sensor network design), and some commercial products. Various experiments with synthesis of implementation code and abstractions for verification are included in the project.

The open-source SDF^3 tool set offers analysis, transformation, generation, and implementation techniques for the SDF, CSDF and SADF models. The graph generation algorithms can construct random graphs which are connected, consistent, and deadlock-free. The user can restrict relevant properties of the generated graph, such as port rates, or construct only acyclic or strongly connected graphs. All algorithms and techniques implemented in SDF^3 can be accessed through a set of command line tools as well as a C/C++ API. Algorithms are provided to transform data flow graphs from one model to another. The rich set of algorithms offered by SDF^3, makes it a versatile tool set for the development of novel data flow based design approaches.

SPIRIT [41] is an XML format based tool, derived originally from the XML format used by Mentor Platform Express, have been developed and promoted, although actual industrial usage remains rather low. Although XML tends to be verbose and inelegant, XML-based formats and schemes can be quickly extended, parsed and generated and are an interesting way both to store system structure and parameters and to pass this information between tools.

The data flow interchange format (DIF) is a textual language that is geared towards capturing the semantics of graphical design tools for DSP system design. It can accommodate a variety of data flow related modeling constructs, and to facilitate experimentation with and technology transfer involving such constructs by providing a common, extensible semantics for representing coarse-grain data flow graphs, and recognizing useful sub-classes of data flow models. DIF captures essential modeling information that is required in data flow based analysis and optimization techniques, such as algorithms for consistency analysis, scheduling, memory management, and block processing, while optionally hiding proprietary details such as the actual code that implements the dataflow blocks. DIF is not centered around any particular form of data flow, and is designed instead to express different kinds of data flow semantics, which supports for SDF, CSDF, and BDF semantics. Accompanying DIF is a software package of inter-mediate representations and algorithms that operate on application models that are captured through DIF. It can also be read and written by designers who wish to understand the data flow structure of applications or the data flow semantics of a particular design tool, or who wish to specify an application model for one or more design tools using the features of DIF. It should be noticed that the later developed DIF-to-C software synthesis framework can generate monolithic C-code implementations from DSP system specifications that are programmed by DIF.

The open source simulation and compilation framework OpenDF can be used together with the CAL language and the DIF/TDP analysis tools. There exists a backend for generation of HDL (VHDL/Verilog), and another backend for that generates C for integration with the System C tool chain [77]. A third backend targeting ARM11 and embedded C is under development [92]. It is also possible to simulate CAL models in the Ptolemy II environment. Where, CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values), which has been superseded by the Open Dataflow environment (OpenDF[2] for short).

2.3.7/ REMARKS

Since the first model of SDF was launched in 1987, it has been 30 years. During this period, many researchers have pushed it forwards. SDF model enables compile time analysis of key execution properties, such as, absence of deadlocks and consistency of execution rates, via efficient algorithms. But for it is limited in its ability to capture how data is accessed in time. Analysis techniques for these models generally overestimate communication storage requirements. Therefore, using these models often leads to sub-optimal results. Even other similar data flow based models and scheduling algorithms also have been proposed, they still have not overcome the drawbacks. They can not ensure that the abstraction of the hardware system is faithful. Thus, the analysis results are not trustworthy. For producing usable VHDL code automatically, there is still a great gap to be bridged.

In recent years, many new SDF based models was proposed, especially the latest SDF-AP model. The semantics of SDF-AP model can be derived by starting from the same class of possible schedules as in the corresponding CSDF model and then restricting this class by removing those schedules that do not satisfy the strictness requirements, such as schedules where an actor stalls during a firing. The original motivation for SDF-AP model comes from modeling hardware IP blocks, where access patterns are precisely characterized and presented as timing diagrams. Nevertheless, the timing extensions that access patterns provide are general and applicable to actors implemented in software as well.

Although the SDF-AP model constitutes a major improvements and is really more efficient compared with former models based on data flow to capture the timing behavior of actor interfaces, some classical behaviors of actors cannot be expressed correctly with this model and some very simple designs lead to an infinite buffer growth while it is perfectly possible to build them without any buffer, provided some simple assumptions are set on how an actor is implemented. Furthermore, there are situations where delays are sufficient to synchronize the different inputs of a single actor. Even if a delay is functionally similar to a FIFO, there is a huge gap of complexity between their VHDL implementation and logic resources consumed. Finally, source actors (actors without any input) are taken into account in the schedule computation but if the graph matches a real design to be put on a FPGA, these actors would surely represent peripherals connected to FPGA I/O pins. In this case, there are a lot of peripherals with a fixed (or nearly) execution schedule.

As for the design tools, there is not an ideal tool to help the designer to realize automatic design. All the tools and frameworks introduced above suffer from different kinds

of limitations. Generally, they share the same limitations from the models they are based on, which lead to some analysis far from the real behavior of hardware. Due to the research interest of developers, the tools are also limited to different domains. No one is developed for non-expert users to work on FPGA development. Therefore, research is needed in this area.

2.4/ CONCLUSION

In this chapter, Field Programmable Gate Arrays (FPGA) is introduced in two aspects: the basic components of FPGA and the process of FPGA design. In addition, the difficulties of designs and the limitations of existing design tools are pointed out. Models for static analysis playing an important role in the EDA tools to help the designers analyze the hardware system being designed are discussed in details. The principles, abilities and drawbacks of Data Flow (DF) based models, such as SDF, CSDF and SDF-AP are analyzed ccarefully. There are also a general comparison among other models in different aspects.

Different expressivenesses of different models and related scheduling strategies are typically traded off between analyzability and implementation efficiency. Based on this common truth, some interesting improvements can be done to the SDF-AP model in order to limit resource consumption and to obtain a valid schedule on more designs. They mainly rely on a new definition of the auto-concurrency property, an optional buffering, using simple delays and constraints on actor implementation. It leads to a new model, Actors with Stretchable Access Patterns (ASAP), that is presented in details in the next Chapter. And some metrics analysis of scheduling are also talked in Chapter 4.

This chapter also makes a brief summary of some software tools or frameworks developed with different concepts and techniques who can help designers to make some analysis and simulations. Some of them are concentrate on SDF based models and hardware design, such as SDF^3, DIF and OpenDF, some tend to help the designer to make some transformation between tools, such as SPIRIT, and others are huge systems with many modules using different models or analysis techniques who can deal with different cases not only in hardware systems but also in software programming, such as Ptolemy. Although they have some limitations in many aspects, we are inspired to develop a new tool. Referring to the existing frameworks and tools, our software tool BlAsT (Block Assembly Tool) based on the novel proposed model and strategies are introduced in Chapter 5.

II

CONTRIBUTIONS

3

Actors with Stretchable Access Patterns

3.1/ Introduction

The problems of existing SDF based models discussed in Chapter 2, especially those of the SDF-AP model, cause unfaithful descriptions of hardware behaviors. In practice, the SDF-AP model suffers some limitations on concurrency, strict pattern conformance and buffering, which lead to some incorrect expressions of actors. They result in limited actor expressions far from the hardware behaviors and yield either invalid analysis or resources waste.

In order to improve the performance of SDF-AP model, in this chapter, we propose a new formalism that relies mostly on SDF-AP concepts but that addresses the scheduling problem in an novel way, closer to the hardware – **Actors with Stretchable Access Patterns (ASAP)**. Firstly, we suppose that actors have a maximum rate of data consumption (expressed by a theoretical pattern) but that they are able to consume data more slowly. This assumption yields strong constraints on how to implement an actor, but no real increase of coding complexity. Thus, actors have **stretchable access patterns** in the sense that the pattern of a real execution may be longer (i.e. with more 0) than the theoretical one. Secondly, considering that channels are all FIFOs with controllers requires a lot of FPGA resources. Thus, instead of computing an optimal schedule and buffer size, we start from a graph with no buffers and try to find the minimum set of buffers and delays so that the graph processes data produced by sources correctly. In case of linear graphs, it may lead to no buffers at all. Basically, "stretchable" means that an actor has a set of "theoretical patterns" that have the same meaning as in SDF-AP, but these patterns may be expanded with a bounded number of 0, placed anywhere, without causing the actor to work incorrectly. For example, if the theoretical input pattern of an actor is $\left[\begin{array}{cccc} 1 & 1 & 1 & 0 \end{array}\right]$ but if it receives $\left[\begin{array}{cccccc} 1 & 0 & 1 & 0 & 1 & 0 \quad \ldots \end{array}\right]$ from the source, it will work correctly as well. It corresponds to the fact that an actor is able to wait a new valid data during an unfixed number of clock cycles. Nevertheless, as soon as an actor has several inputs and outputs, this property is not so simple to define.

In this chapter, we start by going further into the analysis of the limitations of the SDF-AP model firstly, and then provide the detailed definitions and principles of the ASAP model. Explanations on how to model hardware behaviors using the newly proposed model are also illustrated. In the whole description, we use as much as possible terms from related works but redefine them or define new ones when needed. The correctness

of the principles and computations in the model are tested by comparing the computed output patterns and the results of an implementation of a realistic FPGA IP provided by Xilinx CoreGen.

3.2/ LIMITATIONS OF SDF-AP MODEL

As said in the introduction, if the goal is to obtain a data flow graph that models as close as possible the hardware behaviors during an execution on an FPGA, some properties of SDF-AP model prevent to express certain types of actors correctly or find a valid schedule with finite buffers. This section describes the four main limitations and problems yield by the auto-concurrency property, the fact that patterns must be strictly matched, and the mandatory buffering.

3.2.1/ AUTO-CONCURRENCY

In the literature [29], A. Ghosal, et al. use the notion of *auto-concurrency* in SDF-AP graphs to describe the fact that "multiple instances of an actor can execute simultaneously". They define a parameter ii that represents the **minimum** number of clock cycles between two executions of the same actor. They also add that: "this may be not feasible in practice due to restrictions like finite resources, IP properties, etc.". Despite the fact that they do not give precisions about what they consider to be a real auto-concurrent actor (at the VHDL level for example), we notice that actors using a sliding window on input data are more or less in this case.

To point out the SDF-AP limitations with such actors, we take the following example: a 1D average filter with a mask of size 3. Assuming that it consumes a sequence of data: d_1, d_2, d_3, ..., it produces $\frac{d_1+d_2+d_3}{3}$, $\frac{d_2+d_3+d_4}{3}$, ... (to simplify, we omit averages at bounds). Generally, such a filter uses an accumulator (noted $accum$ in the following) and an internal FIFO of size three to store input data. For example, when the filter has consumed d_1, d_2, d_3, $accum = d_1 + d_2 + d_3$ and the FIFO contains these values. When d_4 is consumed, the filter polls the FIFO to retrieve d_1, computes $accum \leftarrow accum + d_4 - d_1$ and finally stores d_4 in the FIFO. There are two main ways to implement that filter, depending on the fact that the data flow is finite and has a size known a priori, or if the data flow is infinite with an unknown size.

The first case consists in coding the filter for a particular size or provide an input that allows to store the size in a register. Thus, the filter has an "internal" knowledge on when it must end the processing, the definition of ii in [93] is relevant. We say that it does an **auto-driven computation**. For example, if the filter operates on data flows of size 5 then its input pattern is $\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$ and its output pattern is $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$. We assume that the filter is able to consume one data, to update the accumulator and to do the division by 3 in the same clock cycle so that the result is available on the output at the next clock cycle. This is why the first 1 in output pattern corresponds to the fourth clock cycle of the execution (N.B.: a functional FPGA implementation would surely need more cycles).

In this case, the actor cannot execute once again before five inputs have been consumed (i.e. $ii \geq 5$), otherwise it would produce incorrect results. We call this situation a

weak auto-concurrency since the overlap occurs only when all needed input data have been consumed. Thus, the same input data is not used by several concurrent executions.

The second case consists in adding a boolean input to the filter to indicate the end of the data flow. Thus, the filter has no knowledge on the end of its processing, which may never arise if the signal is never asserted. We say that the filter does an **externally-driven computation**. In this case, its input pattern is $\begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix}$ and its output pattern is $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$.

In order to compute a sequence of averages correctly, the filter **must** execute once again as soon as there is a valid input data. We call this situation a strong auto-concurrency since the same input data will be used for three executions (except at the bounds). This constitutes the first problem with SDF-AP model that defines ii as a minimum value. With such an actor, $ii = 1$ is the single possible value.

3.2.2/ Strict Pattern Conformance

The second limitation can be illustrated with the example shown in Figure 3.1. It corresponds to a source that emits frames of 16 data at each execution, followed by a decimator that keeps one data out of two to feed an average filter with a mask of size 3. Patterns use the standard regular expression syntax to specify groups (with parenthesis) and repetitions (with embraces). For example $(01)\{8\}$ means 01 repeated 8 times. Assuming that the source execution starts at clock cycle 1, then the decimator produces data at clock cycles $2, 4, \ldots$ Nevertheless, in SDF-AP model, the filter must consumes 8 data during 8 consecutives clock cycles. A certain amount of data must therefore be stored in the FIFO after the decimator, before triggering the filter execution. In this example, the minimum reachable time to start the filter is clock cycle 11 and a FIFO size of 4 (or 5 depending on the priority between the store and poll requests). In the general case, if N is the number of data produced by the decimator, the filter can start its execution at clock cycle $N + 3$ and the buffer size is $\lceil \frac{N}{2} \rceil$.

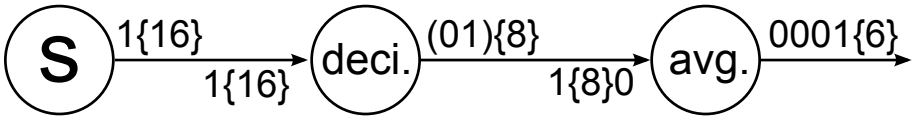

Figure 3.1: A decimator connected to an average filter with fixed size data flows modeled by SDF-AP.

This behavior is not in itself a problem but a waste of resources. Indeed, buffering can be totally avoided if the filter is able to consume data without strictly respecting its input pattern, that is, only when the decimator produces. In terms of VHDL code, it represents a very little change: a simple test on `data_i_enb` must be added in the accumulating process so that values available on `data_i` are taken into account not at each clock cycle but only when the decimator tells that they are valid.

3.2.3/ Infinite Buffering

The third problem comes from the combination of the two previous and is illustrated by the example given in Figure 3.2. The design is functionally equivalent to the previous example, but it operates on an infinite data flow. It implies that the filter implementation must match the second option exposed above and thus, a new instance of the filter must start at each clock cycle. Since patterns must be strictly respected in SDF-AP model, it also implies that the filter must be fed **at each clock cycle** with a valid data. Unfortunately, whatever may be the rhythm of production of the source, the decimator will never produce data at contiguous clock cycles. The only solution is to fill the FIFO after the decimator with an infinite number of data before polling them, which is impossible.

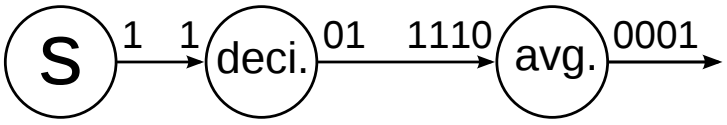

Figure 3.2: A decimator connected to an average filter with an infinite data flow modeled by SDF-AP.

The same as the problem talked above, the infinite buffer problem can be simply solved by using a filter that consumes data only when they are available.

3.2.4/ Mandatory Buffering

Mandatory buffering plays a major role in the problems exposed in the previous sections. The last two limitations mentioned above can be solved without buffer but with constraints on the implementation of the actors. There are also some cases where basic actors can be connected without buffers or with simple delays, provided a good schedule is chosen. Figure 3.3 illustrates such a situation.

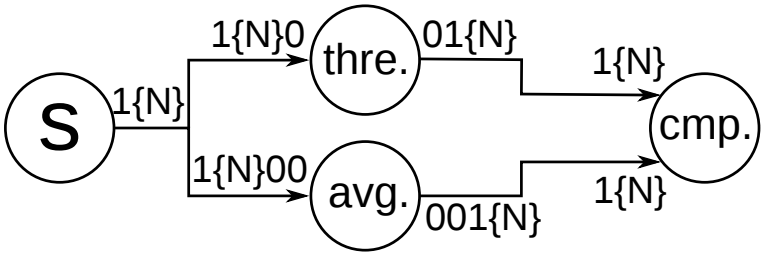

Figure 3.3: An average filter and a threshold filter in parallel, feeding a comparator modeled by SDF-AP.

A source S emits N pixels that are processed by an average filter (N.B.: a correct version that takes into account the bounds, and not simplified as above) and a threshold filter in parallel. Their results are then passed into a comparator that does an undetermined computation. It is reasonable to assume that the threshold produces its first result one clock cycle after the first pixel is consumed, meanwhile the average produces it after the consumption of the first two pixels.

Assuming that the source S produces a pixel at each clock cycle, it is clear that the comparator execution can start as soon as the average produces its first result. Even a FIFO of size one is useless since the comparator execution will be triggered directly by the `data_o_enb` signal from the average. Nevertheless, since the threshold latency is smaller, a simple delay must be put after the output of the threshold so that the two input streams are synchronized. Someone can argue that a delay is a FIFO but it needs no controller and consumes very few resources.

3.3/ PRINCIPLES

3.3.1/ ACTOR'S CONTEXT AND STRUCTURE

Firstly, we consider that all actors in a graph are synchronized on the same global clock signal that is always enabled. Clock cycles are numbered ranging from 1 to ∞. In the following, each event, like the beginning of an actor's execution, can be referenced using that clock. The notation $\tau[x]$ represents the clock cycle at which occurs an event x.

Secondly, we set the following notations and assumptions on the actor's ports (i.e. its interfaces to receive and emit data):

- An actor has $P_I \in \mathbb{N}^*$ input ports and $P_O \in \mathbb{N}^*$ output ports. An actor without inputs is called a **source** and without outputs a **sink**.

- An actor is considered to be enabled (but not necessarily executing) since clock cycle 1. Thus, as in FPGAs, values are available on inputs/outputs at each clock cycle. In order to detect what values are pertinent for an actor's computation, each input receives and output produces a couple of signals `(data,validity)`. The type of `data` is let undefined. `Validity` is a boolean signal. If it is true (=1), then the value of `data` is considered as a valid entry for the actor's computation or a valid result. If it is false (=0), `data` is not valid and must be considered as if it does not exist.

- The variations of the `validity` signal received by an input or produced by an output can be represented by a vector of 1 and 0. For all inputs/outputs of an actor, these vectors form a matrix called respectively an **input pattern** (IP) or an **output pattern** (OP). Assuming we only consider clock cycle from 1 to T, they are noted respectively

  $IP = [IP_{i,t}], i \in \{1,\dots,P_I\}, t \in \{1,\dots,T\}$, where $IP_{i,t}$ is the value of the `validity` signal received by an input i at clock cycle t.

  $OP = [OP_{o,t}], o \in \{1,\dots,P_O\}, t \in \{1,\dots,T\}$, where $OP_{o,t}$ is the value of the `validity` signal produced by an output o at clock cycle t.

  Example 1: An IP with 2 inputs ($P_I = 2$) and $T = 8$: $IP = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$.

  Example 2: An OP with 3 outputs ($P_O = 3$) and $T = 6$: $OP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$.

- The **input schedule** (IS) defines the clock cycles for which the `validity` signal is true for at least one input. Thus, it is a simple vector.

  For Example 1, it gives $IS = \begin{bmatrix} 1 & 3 & 4 & 6 & 8 \end{bmatrix}$.

- The **output schedule** (OS) defines the clock cycles for which the `validity` signal is true for each output. Since for a given interval of time, the number of 1 may be different for each output, it is expressed as a list of vectors $OS_o, o = 1 \dots P_O$.

  For Example 2, it gives $OS = \begin{bmatrix} 3 & 6 \end{bmatrix} \\ \begin{bmatrix} 3 & 4 & 6 \end{bmatrix} \\ \begin{bmatrix} 4 \end{bmatrix}$.

It should be noticed that it is easy to obtain the output schedule from the pattern and vice-versa, but it is not the case for the input schedule since the schedule does not indicates where is the 1 for each input. The algorithms to make the transmutations between patterns and schedules are given in Algorithm 1 (from input pattern to input schedule), Algorithm 2 (from output pattern to output schedule) and Algorithm 3 (from output schedule to output pattern).

In the following, some additional notations for patterns and schedules will be used, notably in algorithms. For XP a given type of pattern (resp. schedule), $XP_{p,*}$ represents the input/output pattern of input/output port p, that is the $p\text{-}th$ horizontal vector in XP. $XP_{*,t}$ represents the state of `validity` signal for all inputs/outputs at clock cycle t, that is the $t\text{-}th$ vertical vector in XP. The combination of the two gives $XP_{p,t}$.

Algorithm 1: Transmutation from input pattern to input schedule.

```
1 j ← 1
2 for k = 1 to length(IP) do
3 |   if IP_{k,*} contains 1 then IS_j ← k
4 |   j ← j + 1                          /* counter for columns in IS */;
5 end
```

Algorithm 2: Transmutation from output pattern to output schedule.

```
1 for i = 1 to P_O do
2 |   j ← 1
3 |   for k = 1 to length(OP) do
4 |   |   if OP_{i,k} = 1 then OS_{i,j} ← k
5 |   |   i ← i + 1                    /* counter for columns in OS */ ;
6 |   end
7 end
```

Algorithm 3: Transmutation from output schedule to output pattern.

```
1 OP ← [0]_{P_O,max(OS)}
2 for i = 1 to P_O do                  /* counter for lines in OS */
3 |   for j = 1 to length(OS_i) do     /* counter for columns in OS */
4 |   |   OP_{i,[OS_{i,j}]} ← 1
5 |   end
6 end
```

Besides input/output pattern/schedule, there are some other items related to input/output need to be defined and explained.

- An **input data group** is a set of values of the `data` signal received on the inputs, at a given clock cycle.
- An **output data group** is a set of values of the `data` signal produced on the outputs, at a given clock cycle, which shares the similar meaning with the **input data group**.
- A **valid data group** is a data group for which **at least one** associated `validity` signal is equal to 1. It implies that there may be invalid data within a data group and obviously, for inputs, they should not be used by the actor during its execution, as explained in Section 3.3.2.1.

To go further, these definitions imply that:

- At a given clock cycle t, if the $t\text{-}th$ column of IP or OP contains only 0, the associated data group is invalid.
- IS contains the clock cycles of all valid input data groups, in an increasing order.

In the following, for concision, the term data group will always refer to a valid data group. Otherwise, we will explicitly use the term invalid data group.

3.3.2/ Actor's Behavior

3.3.2.1/ Computation

The received and produced data groups of an actor are two sequences of vectors of `data` signals $X_k = (X_{i,k})_i$ and $Y_k = (Y_{j,k})_j$. The index $k \in \mathbb{N}^*$ refers to the clock cycle while i and j refer to the indexes of the input and output ports. They are ordered in time i.e. $\tau[X_k] < \tau[X_{k+1}]$ and $\tau[Y_k] < \tau[Y_{k+1}]$.

What an actor computes is represented by a function F. Assuming that it is triggered for the $n\text{-}th$ execution, and the concurrent execution uses $C \in \mathbb{N}^*$ input data groups to produce $R \in \mathbb{N}^*$ output data groups, then it computes $Y = F(X)$ with:

$$X = \begin{bmatrix} X_{1,1+(n-1)\times C} & \cdots & X_{1,n\times C} \\ \vdots & \ddots & \vdots \\ X_{P_I,1+(n-1)\times C} & \cdots & X_{P_I,n\times C} \end{bmatrix} \text{ and } Y = \begin{bmatrix} Y_{1,1+(n-1)\times R} & \cdots & Y_{1,n\times R} \\ \vdots & \ddots & \vdots \\ Y_{P_O,1+(n-1)\times R} & \cdots & Y_{P_O,n\times R} \end{bmatrix}.$$

Matrix notation is a convenience since F may use only a limited set of X elements to compute a limited set of Y elements. Other elements of X are simply ignored and elements of Y assigned with an undefined value.

3.3.2.2/ Execution and Concurrency

An **execution** of an actor is noted as E. It covers the time between the reception of a first input data group that triggers the computation of F, and the production of the last result. In the following, the $n\text{-}th$ execution ($n \in \mathbb{N}^*$) of the actor is noted as E_n.

An actor is **self-triggering**. It means that when it is idle (i.e. active but not executing), the first data group received on inputs triggers the actor's execution. Moreover, depending on how the actor is implemented and the structure of the input pattern, it can trigger itself again when it is executing. This lead to **concurrent executions**. The most basic case is when an actor outputs results of an execution while it consumes new data groups for another execution. Nevertheless, concurrent executions can overlap more largely, for example when a new execution starts while the previous one has not totally consumed all needed data groups. This case is called a **concurrent consumption**. It occurs as soon as different computations of F share at least one input data group. Note that concurrent consumptions imply concurrent executions but not the inverse.

We can take the following example to illustrate concurrent consumptions. Assuming an actor with $C = 3$, where *C* represents the number of tokens for one execution, if it is implemented, the computations can be in different cases as following:

- *Case 1*: $F(X_1, X_2, X_3)$, $F(X_2, X_3, X_4)$, $F(X_3, X_4, X_5)$,..., etc. then there are concurrent consumptions.
- *Case 2*: $F(X_1, X_2, X_3)$, $F(X_3, X_4, X_5)$, $F(X_5, X_6, X_7)$,..., etc. then there are concurrent consumptions.
- *Case 3*: $F(X_1, X_2, X_3)$, $F(X_4, X_5, X_6)$, $F(X_7, X_8, X_9)$,..., etc. then there is **no** concurrent consumption.

3.3.2.3/ Delay between Executions

Δ represents the delay between two executions of an actor. It covers the same notion than ii in SDF-AP but with a different definition [93]. Indeed, if the constraint of strict conformance to patterns is released, the number of clock cycles needed to consume a certain amount of data groups is variable. This is why Δ corresponds to the number of input data groups (and not of clock cycles) that must be consumed by an actor before it starts another execution, thus $1 \leq \Delta \leq C$.

In terms of the mapping F, it implies that, for an undefined j, if E_n computes $F(X_j, \ldots, X_{j-1+C})$, the next execution E_{n+1} computes $F(X_{j+\Delta}, \ldots, X_{j-1+C+\Delta})$.

From the point of view of the consumption, Δ represents a strict value and not a minimum as ii. It implies that as soon as Δ data groups have been consumed, the next input data group will automatically trigger a new execution of the actor. Note that Δ has a great influence on the correctness of execution. It is discussed in details in Section 3.3.3 and Section 4.3.4.

3.3.3/ Actor's Patterns and Schedules

3.3.3.1/ Execution

The **execution pattern** (EP) defines which values of X are used by F. It is represented by a matrix $EP = [EP_{i,k}], i \in \{1, \ldots, P_O\}, k \in \{1, \ldots, C\}$, composed of 1 and 0.

If $X_{i,k}$ is used by F then $EP_{i,k} = 1$ while $EP_{i,k} = 0$ otherwise.

From matrix EP, we can express a necessary conditions to be sure that the actor produces a correct result.

Condition C1 - The correctness of a result requires that if $\forall X_{i,k}$ for which $EP_{i,k} = 1$, the value of `validity` signal of input i for the k-th input data group is true. If it is not the case, the result is incorrect.

3.3.3.2/ CONSUMPTION

It is worth noting that the condition *C1* is not a sufficient condition. Indeed, EP just tells what data must be used in each input data group but not **when** this data group may be consumed by the actor from the moment it begins its execution. In SDF-AP, IP integrates the "when": each 1 or 0 is associated to a different clock cycle, meaning that the actor consumes or not the data on the associated input at that clock cycle. But the correctness is not guaranteed. If IP contains a 0 at a given clock cycle but there is a valid data on the input, it will be lost and the result of the computation will be incorrect. Conversely, if there is a 1 in IP and there is not a valid data on the input the result is also erroneous. The mandatory buffering and FIFO controller are there to avoid such cases (if possible), enforcing a strict conformance to the pattern.

Nevertheless, there is another problem if we take concurrent executions into account. In Section 3.2, we pointed out the fact that the auto-concurrency parameter ii is defined as a minimum in SDF-AP but it should be a fixed value for some actors. Assuming such an actor has a consumption pattern (in the sens of SDF-AP) equal to $\begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$ and $ii = 2$, let t be the clock cycle at which it is triggered for the first time. Then, it will be triggered again at $t + 2 \times k$ for any $k \in \mathbb{N}^*$. The problem is that at $t + 2$, the first execution does not consume any data (0 in the pattern), so applying the strict pattern conformance, there must not be a valid data available on the input. However, there is also the second execution that starts at $t + 2$ and applying the same conformance, it must consume data. The consumptions of different executions at the same clock cycle are in contradiction.

This problem can be easily solved by changing the definition of the 0. It corresponds to the fact that the actor do not need to consume for the current execution. But if there is an available valid data, it can be consumed by another execution.

Moreover, there are some cases where an actor must effectively not consume data for any current execution. Such a situation occurs for example when an actor consumes data that are stored in registers before using them to compute something. In some cases, the value of these registers **must not** change during this computation, otherwise, the result would be incorrect. It implies that during this computation, concurrent consumptions on the associated inputs are forbidden. We need a third notation to identify such special clock cycles in a pattern. Thus, symbol $\times$ is introduced to the patterns instead of 0 to express these clock cycles.

We define the **consumption pattern** (CP) as the minimal pattern in length that represents the consumption policy at each clock cycle of an actor's execution to produce correct results. It is similar to the input pattern defined in SDF-AP but it takes into account more precisely the cases of concurrent consumptions. For convenience, it is expressed as a matrix but should be interpreted as a sequence of column vectors: the t-th column describes the consumption policy of the actor at clock cyle t, relatively to the beginning of its execution. Thus, if L_{CP} is this length, then

$CP = [CP_{i,t}], i \in \{1, \ldots, P_I\}, t \in \{1, \ldots, L_{CP}\}$, $CP_{i,t} \in \{1, 0, \times\}$, and

- $CP_{i,t} = 1$, if the actor must consume that data on input i to compute a correct result,
- $CP_{i,t} = 0$, if the actor does not need that data for the current execution, but another execution may consume it,
- $CP_{i,t} = \times$, if the actor must not consume that data for any current execution.

Example 3: A pattern CP with $P_I = 2$, $L_{CP} = 7$, and $C = 4$:

$$CP = \begin{bmatrix} 1 & \times & 1 & 0 & 0 & 0 & 1 \\ 1 & \times & 0 & 0 & 1 & 0 & \times \end{bmatrix}.$$

This theoretical example is not realistic but possible. It exhibits the different combinations of 1, 0, and $\times$ within a single column. The pattern CP is in fact EP with some 0 transformed in $\times$ and expanded with columns of 0 or $\times$. In this example, we would have:

$$EP = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}.$$

It is important to notice that CP is simply a "theoretical" pattern that represents a maximum but not an absolute pace of consumption. Thus, during a "real" execution, the associated portion of input pattern may not correspond but can be longer, with columns of 0 inserted between some columns of CP. It represents the fact that input data groups are available at a slower pace than the theoretical maximum given by CP. The process that checks whether IP and CP match is called **compatibility checking** discussed in details in Section 4.3.4. Nevertheless, in order to clearly understand the CP definition, we give a simple example that illustrates the principles of the checking.

Example 4: For $P_I = 2$, $L_{CP} = 3$, $C = 2$, and $\Delta = 1$,

$IP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \ldots \end{bmatrix}$ and $CP = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ are compatible.

- At $t = 3$: the first valid data group is available on inputs. The actor starts its first execution and consumes the group, that is a data on input 1.
- At $t = 5$: normally, the first execution should consume a data on input 2 but since it is not a valid data group, it simply waits.
- At $t = 6$: the second valid data group is available, thus the first execution consumes a data on input 2. Since $\Delta = 1$ and the first execution has already consumed one data group, a second execution automatically starts, consuming a data on input 1.
- At $t = 8$: the third valid data group is available, thus the second execution consumes a data on input 2 and the third execution automatically starts, consuming a data on input 1.
- ...

But if $IP = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 1 & 0 & \dots \end{bmatrix}$, it is incompatible with CP.

- At $t = 3, 4$: the first and second executions start, consuming the data on input 1.
- At $t = 5$: the first execution consumes the data on input 2 but, since a valid data group is available, the third execution starts, consuming an invalid data on input 1.

It must be noted that there is a close relation between CP and Δ. In the case of real blocks, their values are consistent because Δ and CP are given by the implementation and cannot be chosen. But in theoretical examples, there may be impossible combinations of Δ and CP as illustrated in Example 5.

Example 5: $CP = \begin{bmatrix} 1 & \times & 1 \\ 1 & 1 & 0 \end{bmatrix}$ and $\Delta = 1$ are inconsistent.

Assuming the first data group triggers the first execution at clock cycle t, since $\Delta = 1$, the second data group automatically triggers the second execution at $t + 1$. For that execution, the actor must consume a valid data on both inputs to produce a correct result. Nevertheless, for the first execution, the consumption policy is given by the second column of CP, that forbids consumption on input 1 for any execution. These two constraints are in contradiction thus Δ cannot be equal to 1 (but 2 and 3 are possible).

3.3.3.3/ PRODUCTION

The **production pattern** (PP) is the counterpart of CP but for results on outputs. The t-th column of PP describes if the actor produces valid results or not on its outputs at clock cycle t, relatively to the beginning of its execution. Thus, if L_{PP} is its length, then

$PP = [PP_{o,t}], o \in \{1, \dots, P_O\}, t \in \{1, \dots, L_{PP}\}$, $PP_{o,t} \in \{1, 0\}$, and

- $PP_{o,t} = 1$ if the actor produces a valid result,
- $PP_{o,t} = 0$ if the actor do not produce a valid result.

Example 6: A PP with $P_O = 2$, $L_{PP} = 6$, and $R = 3$: $PP = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$.

The **production schedule** (PS) defines the clock cycles for which PP is equal to 1. Like OS, it is expressed as a list of vectors $PS_o, o = 1 \dots P_O$.

For Example 6, it gives $PS = \begin{bmatrix} 3 & 5 \\ 3 & 5 & 6 \end{bmatrix}$.

As for sharing the same format with output pattern and schedule, Algorithm 2 and Algorithm 3 can also be used to make the transmutation between production pattern and schedule.

Generally, an execution takes a certain amount of clock cycles to produce the first output data group. This amount corresponds to the number of null columns (i.e. composed of 0) at the beginning of PP and is called **production delay**, noted as δ. It is closely related (but not mandatory equal) to the number of input data groups that are

needed to compute the first output data group. For example, the two versions of the average filter described in Section 3.2.1 (Auto-concurrency), have a $\delta = 3$. Nevertheless, the "correct" version in Section 3.2.4 (Mandatory buffering), that works with finite data flow has a $\delta = 2$ but that could be more if the accumulation and division take more than one clock cycle to complete.

It is important to notice that PP does not indicate what inputs must be consumed to produce a particular result. This is a problem if IP does not strictly corresponds to CP because actors with the same PP may have different output patterns.

For example, an average filter on three values would have $CP = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, and $PP = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \end{bmatrix}$. A threshold filter on three successive values could have exactly the same patterns. If these filters are able to wait for valid data groups, then they can process correctly even if $IP = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$. Since the threshold needs a single input to produce an output, its output pattern is PP with extra zeros inserted at the same places as in IP: $OP^{thre.} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$. The average needs two or three values, leading to $OP^{avg.} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$.

The **production counter** (PC) is used to solve the previous problem. It defines the number of input data groups that must have been consumed before producing a particular output. It is expressed as a list of vectors $PC_o, o \in \{1, \ldots, P_O\}$. For example, the filters mentioned above have $PC^{thre.} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ and $PC^{avg.} = \begin{bmatrix} 2 & 3 & 3 \end{bmatrix}$.

As for CP, there is a close relation between PP and Δ, with some combinations of values that are inconsistent. For example, $PP = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$ is inconsistent with $\Delta = 1$ because two successive executions would yield a result at the same clock cycle, which is not physically feasible. Thus, for real blocks, Δ and PP are necessarily consistent and this fact is taken into account in the following.

3.3.3.4/ OUTPUT

Under the assumption that an input pattern IP is compatible with a consumption pattern CP, an output pattern OP can be computed according to the principles of execution, consumption and production. The following example is used to explain the principles for output pattern generation.

Example 7: $IP = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$,

$CP = \begin{bmatrix} 0 & \times & 1 & \times & 1 \\ 1 & \times & 0 & \times & 1 \end{bmatrix}$, $PP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$, $PC = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$, $\Delta = 1$,

$P_I = 2$, $L_{CP} = 5$, $C = 3$, and $n_{exe} = 4$.

For simplicity, we give C, P_I and L_{CP} directly which, in fact, can be derived from the other items listed.

A detailed illustration is given by Figure 3.4. For clarity, IP is recalled on the first row. The clock cycles where an execution is triggered are surrounded by a circle, and the extra null columns in IP (compared with CP) are represented in rounded boxes with gray background. For each execution, PP is copied on a different row. For the first one, the first output group is produced two clock cycles after the first input group, at $t = 5$. According to

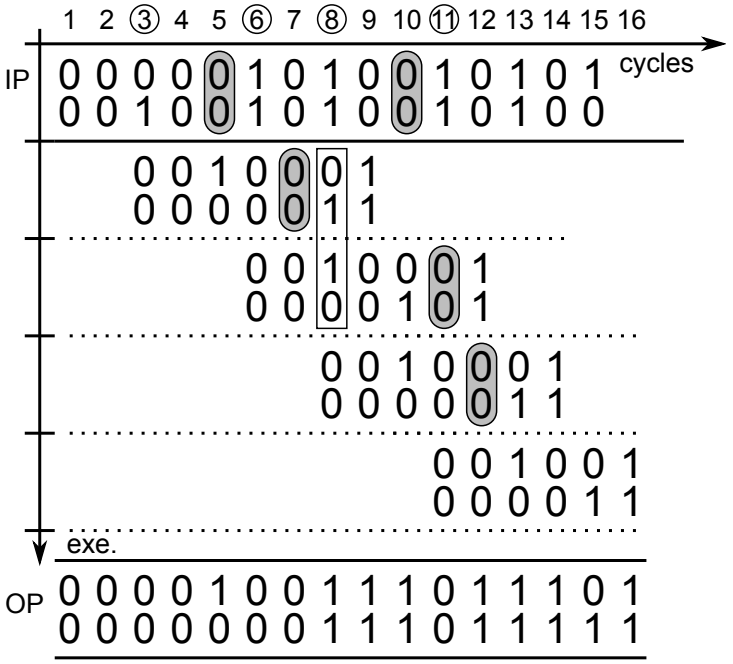

Figure 3.4: The process of computing output pattern.

PP, the second output should be at $t = 7$. Nevertheless, an extra null column exists in IP between the first and second input data group. Thus, this column must be reported in the copied PP, delaying the output by one clock cycle until $t = 8$. There is no extra null column for the third output, can appear just after the second at $t = 9$. For the second execution, the same principles apply, except that the extra null column is between the second and third inputs, leading to delay the third output by one clock cycle until $t = 12$. After all executions have been considered, OP is obtained with a logical OR on each column, as for AP. It is worth noting that the combination between the different executions leads to a compatible OP. Indeed, there are never two 1 on the same output. For example, at $t = 8$, the box points out the fact that the production of the first execution uses the second port, and the second execution uses the first port, preventing any conflict.

These principles are implemented in Algorithm 4. For each execution n, it starts searching the triggering clock cycle t (line 6 for the first and line 26 to 29 for the following). Then, the output is computed using several indexes (line 8): p for PP, $t + o$ for OP, c_{CP} for CP, and $t + c_{IP}$ for IP. Two counters, n_{IP} and n_{CP}, are used to know how many input groups have been consumed during the current execution. The goal is to combine the valid output groups of PP with OP at the right index. For that, the algorithm loops over valid output groups (line 9), searching their location p in PP (line 10). Then, it computes the gap between the consumption rhythm in IP and CP (line 12 to 15, and 16 to 19). It corresponds to the number of null columns that have been added in IP compared with CP. Since each output group is produced at a fixed amount of clock cycles after a number of consumed inputs, this gap is added to o (line 20) yielding the right index for the

combination of $PP_{*,p}$ and $OP_{*,t+o}$ (line 21) with a logical OR.

Algorithm 4: Output pattern generation.

```
1  for i = 1 to length(IP) + length(PP) do
2      OP_{*,i} ← null column
3  end
4  t ← 1                                              /* counter for clock cycles */
5  n ← 1                                              /* counter for n execution.  */
6  while IP_{*,t} is a null column do t ← t + 1 ;
7  while n ≤ n_exe do
       /* initialize counters for the current execution                          */
8      p ← 1; o ← 0; c_IP ← 0; c_CP ← 0; n_IP ← 0; n_CP ← 0
9      for m = 1 to length(PC) do
10         while PP_{*,p} is null column do p ← p + 1; o ← o + 1 ;
11         gap ← 0
12         while n_IP < PC_m do
13             if IP_{*,t+c_IP} is valid input then n_IP ← n_IP + 1;
14             c_IP ← c_IP + 1; gap ← gap + 1
15         end
16         while n_CP < PC_m do
17             if CP_{*,c_CP} is valid data group then n_CP ← n_CP + 1;
18             c_CP ← c_CP + 1; gap ← gap − 1
19         end
20         o ← o + gap
21         OP_{*,t+o} ← OP_{*,t+o} OR PP_{*,p}
22         p ← p + 1; o ← o + 1
23     end
24     t ← t + 1
25     n_IP ← 0
26     while n_IP < Δ do                          /* search start of next execution */
27         if IP_{*,t} is valid input then n_IP ← n_IP + 1 ;
28         if n_IP < Δ then t ← t + 1;
29     end
30     n ← n + 1
31 end
```

3.3.3.5/ REMARKS

With the proposed principles and algorithms we can compute the output of every actor. But the correctness of the output depends on the compatibility of IPs and CPs. If they are incompatible, the input can not execute directly and other approaches should be applied. Therefore, the compatibility checking approach is needed and is an important part of the model. The detailed strategies applied to deal with various cases will also be investigated in this dissertation.

3.4/ EVALUATION THROUGH EXISTING IPS

In order to illustrate and evaluate the principles of actors' behaviors in our model, we find some examples in existing FPGA IPs for testing. The goal is to check whether our model can be applied to existing blocks without precise descriptions of their behaviors and applicable VHDL code. A perfect one is given by the Xilinx CoreGen tool and its FIR compiler that allows to generate different types of FIRs with a variable number of coefficients. In fact, CoreGen can only generate a wrapper and a netlist for a filter. Thus, the whole VHDL code is not available. Nevertheless, the generator proposes different versions for the available interfaces. In the oldest version that is used for the following tests, it is possible to add a `nd` boolean signal that is true when the value on the input port must be consumed. If the signal is false, the block does not consume the value. It is clearly similar to a validity signal of our model. Moreover, in order to save DSP48 multipliers used by the filter, it is possible to specify the minimum number of clock cycles between two inputs. The recent version allows to generate a filter compliant with the AIX4-stream protocol, which is discussed at the end of this section.

3.4.1/ TESTS ON THE ORIGINAL VERSION OF FIR FILTER

Without exploitable code, the technical documentation could be a valuable source of information. Unfortunately, it is not sufficient to deduce CP, PP and PC from the parameters that are used to generate the blocks. Thus it is mandatory to guess them from simulations. The question is: can they really represent the behaviors of the blocks in all situations?

To answer to this fundamental question, we chose to generate interpolators that can accept at least a new data every four clock cycles, with three possible ratio between input and output: $3 \rightarrow 5$, $5 \rightarrow 7$ and $5 \rightarrow 8$. The information allows to deduce:

$CP_{3\rightarrow5}$ = [(1 0 0 0) {2} 1] and $CP_{5\rightarrow7} = CP_{5\rightarrow8}$ = [(1 0 0 0) {4} 1].

Then, we performed simulations with different input patterns, more or less stretched compared with the minimum requirement, and a clock period of 10ns. Results reported below are based on simple cases: IP_{4cc} = [(1 0 0 0) $*$], IP_{5cc} = [(1 0 0 0 0) $*$] and IP_{6cc} = [(1 0 0 0 0 0) $*$], which corresponds to a new input every 4, 5 and 6 clock cycles respectively.

In the following figures, `nd` is asserted to 1 each time an input is received and consumed by the block. `rdy` is asserted to 1 each time the block produces an output.

Figures 3.5, 3.6 and 3.7 concern the $3 \rightarrow 5$ interpolator. In the first one, IP matches the nominal rate of consumption of the block. Thus, we can deduce that PP = [0 {14} (1 0) {4} 1)]. 14 represents the latency to produce the first result, which can be computed from the time indexes in yellow.

In the second figure, it can be noticed that stretching IP has no impact on the clock cycle at which the second and the fourth outputs are produced. The logical assumption is that outputs 1 and 2 only depend on the first input, outputs 3 and 4 depend on the second one, and output 5 depends on the third one. It gives a likely PC = [1 1 2 2 3] that can be checked with other IPs.

Applying Algorithm 4 to compute the output pattern with IP_{6cc}, we obtain the OP = [1 0 1 0 0 0 1 0 1 0 0 0 1] for a single execution. There are 5 idle clock

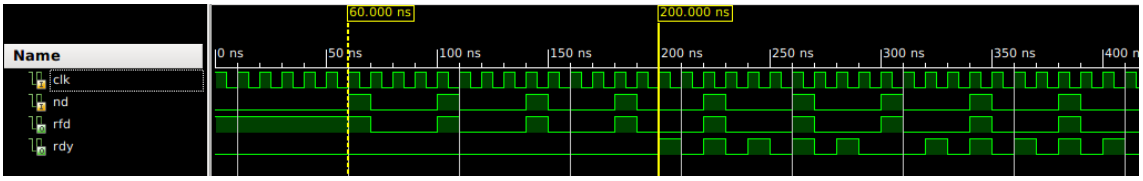

Figure 3.5: Simulation 1 - 3→ 5 interpolator for IP_{4cc}.

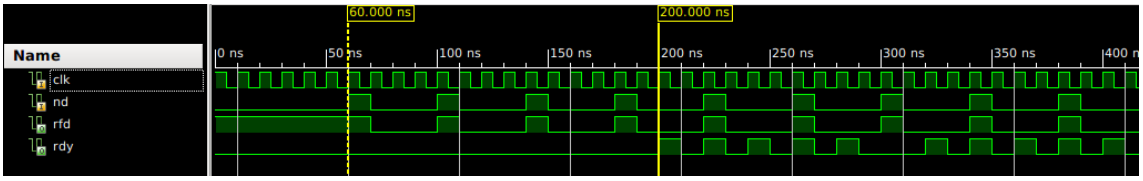

Figure 3.6: Simulation 2 - 3→ 5 interpolator for IP_{5cc}.

cycles between each execution, which can be verified that the block really has the same behavior as shown in Figure 3.7.

We also tried more complex IPs with combinations of different cycles so that the output is not so regular. For example, with $IP = [\ (1\quad 0\quad \{4\}\quad 1\quad 0\quad \{3\})\ *]$, the $OP = [\ 0\quad \{14\}\quad (10100101010000101010100 1000)\ *]$ is computed from Algorithm 4, which is also confirmed by simulations. Until now, we have not found any IP that could lead to results different between simulations and our algorithm.

The same type of study has been conducted with the $5 \rightarrow 7$ and $5 \rightarrow 8$ interpolators. Their characteristics are summarized in Table 3.1. The latency is smaller because of less coefficients are used in the filter. Otherwise, as shown in the following figures, each tested IP gives the same results from the Algorithm 4 and simulations.

Table 3.1: Characteristics of 5 → 7 and 5 → 8 interpolators

	CP	PP	PC
$5 \rightarrow 7$	$(1000)\,\{4\}\,1$	$0\,\{10\}\,(10101000)\,\{2\}\,1$	1123345
$5 \rightarrow 8$	$(1000)\,\{4\}\,1$	$0\,\{10\}\,(10)\,\{5\}\,0010101$	11223445

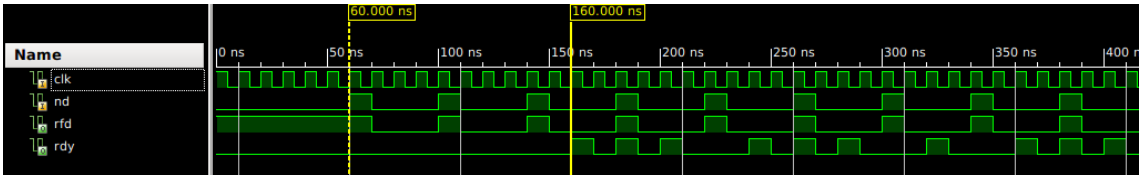

Figure 3.8: Simulation 4 - 5→ 7 interpolator for IP_{4cc}.

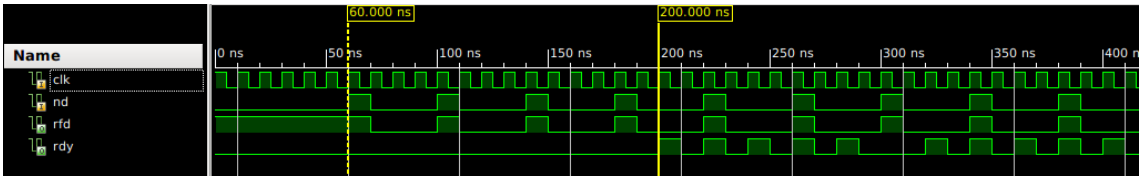

Figure 3.7: Simulation 3 - 3→ 5 interpolator for IP_{6cc}.

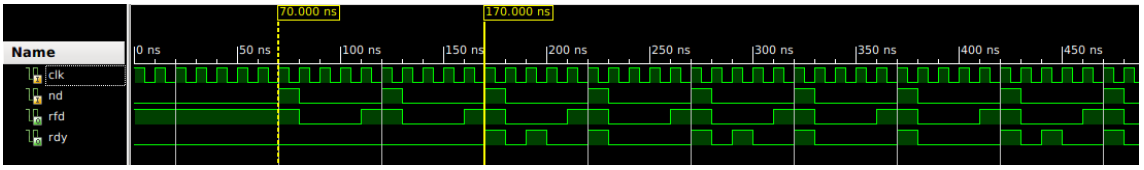

Figure 3.9: Simulation 5 - 5→ 7 interpolator for IP_{5cc}.

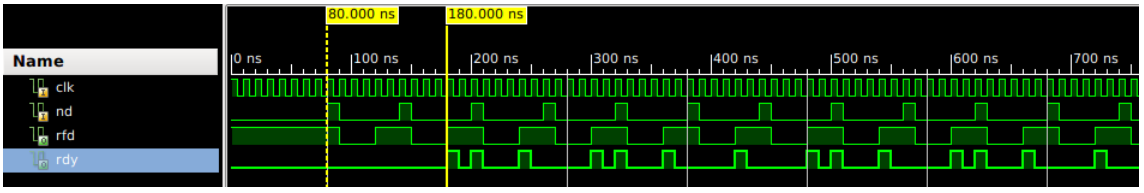

Figure 3.10: Simulation 6 - 5→ 7 interpolator for IP_{6cc}.

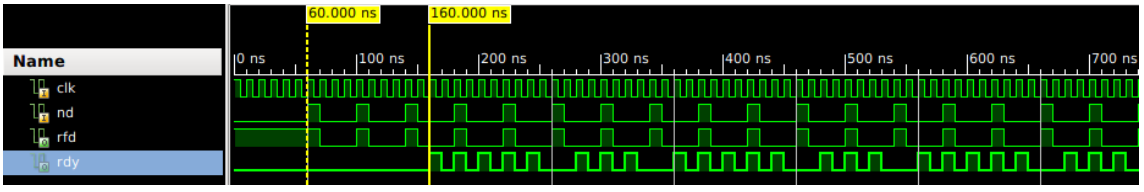

Figure 3.11: Simulation 7 - 5→ 8 interpolator for IP_{4cc}.

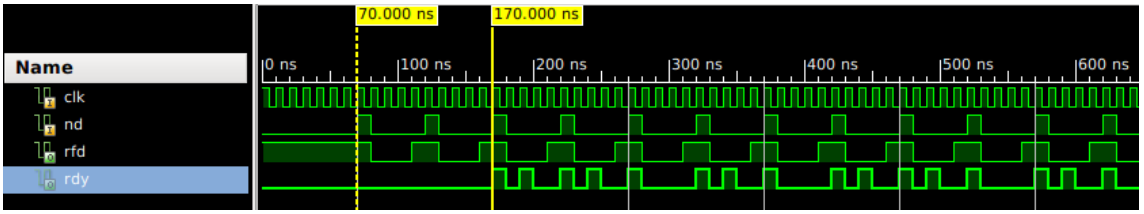

Figure 3.12: Simulation 8 - 5→ 8 interpolator for IP_{5cc}.

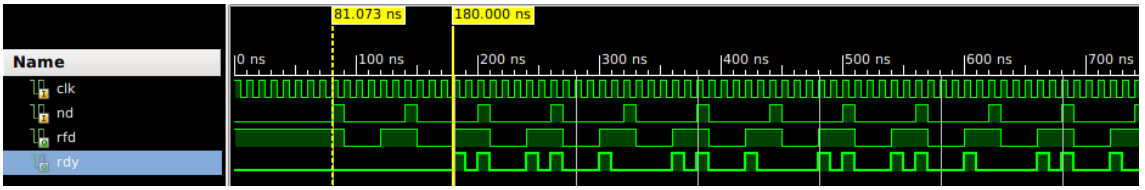

Figure 3.13: Simulation 9 - 5→ 8 interpolator for IP_{6cc}.

All these tests seem to confirm that an existing block can be modeled correctly with our approach. Nevertheless, it is not always the case and some “strange” behavior cannot be taken into account. Such a case occurs for example with a $5 \rightarrow 7$ interpolator that is able to consume at least a new data **every 3 cycles**, instead of 4 for the previous. This block has $CP = [\ (1\quad 0\quad 0)\quad \{4\}\quad 1\ ]$.

Figure 3.14 shows the simulation result using IP_{5cc}. An equal gap of 10 cycles can be observed between input 1 and output 1, input 2 and output 3, input 3 and output 4, ... Thus, it seems that this block has also $PC = [\ 1\quad 1\quad 2\quad 3\quad 3\quad 4\quad 5\ ]$. Nevertheless, since CP is different, it corresponds to a different $PP = [\ 0\quad \{10\}\quad (1\quad 0\quad 1\quad 1\quad 0\quad 0)\quad \{2\}\quad 1\ ]$.

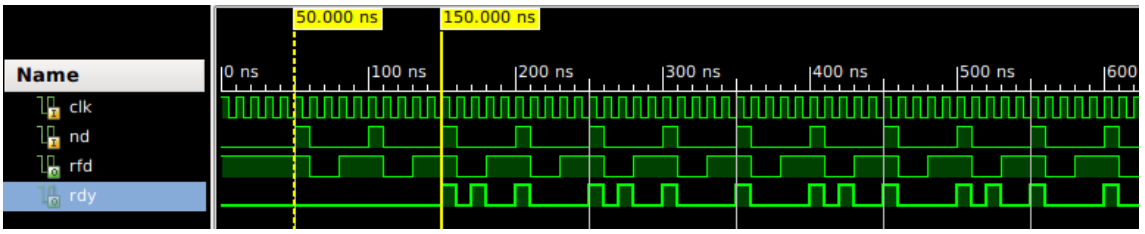

Figure 3.14: Simulation 10 - 5→ 7 interpolator v2, for IP_{5cc}.

If we use $IP_{3cc} = [\ (1\quad 0\quad 0)\ *]$ as an input, Algorithm 4 computes an output $OP = [\ 0\quad \{10\}\quad (101100101100100)\ *]$, which is normal because IP is just a concatenation of CP. Nevertheless, the simulation gives another result, shown in Figure 3.15.

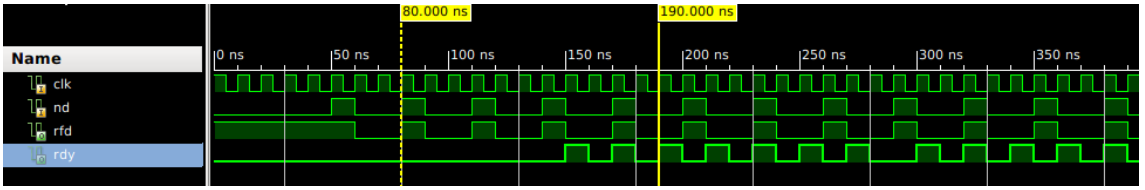

Figure 3.15: Simulation 11 - 5→ 7 interpolator for IP_{3cc}.

For an unknown reason, the latency between input 2 and output 3 is 11 instead of 10. The same behavior appears with input 4 and output 6. Without the VHDL code, it is impossible to analyze the reasons of this irregular behavior. It is even more strange that a $5 \rightarrow 6$ interpolator produces the same irregularities but not a $5 \rightarrow 8$ one. Neither of them appear with interpolators that are able to consume an input every two clock cycles.

But even if an explanation could be found, our model is not able to match such irregularities, as for blocks with a latency depending on the value of the inputs.

In conclusion, IPs generated with obfuscated code represent a real problem to be integrated in our model.

3.4.2/ THE AIX4-STREAM PROTOCOL

As mentioned above, this filter can be generated with a set of interfaces compliant with the AIX4-stream protocol. From the point of view of available interfaces, it is mainly a "cosmetic" change compare to the version used above: the "new data" (`nd`), "new output" (`rdy`) and "ready for data" (`rfd`) interfaces are just named differently in the AIX4-stream version. They are generally referred as "valid" interfaces for inputs and outputs, and "ready" interfaces to manage the fact the IP is ready to consume input data. Nevertheless, there are major differences in the behavior because the filter includes internals buffers for inputs and outputs.

More generally, IPs that are compliant with the AIX4-stream protocol may be generated to work in blocking (default) or non-blocking mode.

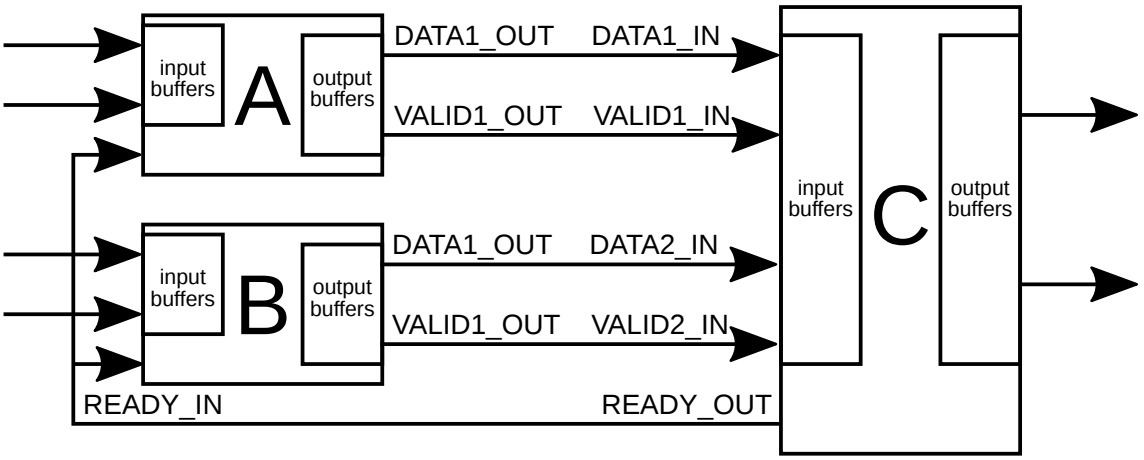

Figure 3.16: AIX4-stream - IPs using the blocking mode.

The first one is illustrated by Figure 3.16 with two IPs named `A` and `B` that are connected to `C`. The blocking mode implies the following constraints:

- if `C` is not ready to consume (i.e. `READY_OUT` asserted to '0'), `A` and `B` put there output data in output buffers.
- if `C` is ready to consume (i.e. `READY_OUT` asserted to '1'), `A` and `B` can release data out of their buffers or deliver them directly if buffers are empty.
- Each time a data is presented on a `DATA_OUT` interface, the associated `VALID_OUT` interface is asserted to `1` (as in ASAP).
- if `C` does not receive a `1` on all its `VALID_IN` interfaces at the same clock cycle, inputs are buffered.
- As soon as `C` has a data for all its input interfaces (whether in the input buffers or directly on the input), it consumes and processes them.

Apart from the READY signals, these constraints lead to a behavior quite similar to the SDF model except the fact that buffers are embedded in actors instead of channels. Indeed, assuming that C is always ready, there is no output buffering in A and B. Nevertheless, C buffers inputs until it has a sufficient number to process them. The problem is that there is no fine control on the size of these buffers. So they may be too large compared to the real needs, or worst, they may overflow, yielding an incorrect processing of data streams.

Figure 3.17 illustrates the same configuration in non-blocking mode. In this case, there are no internal buffers. The receiving IP is considered to always be ready (i.e. READY signal is ignored) so the sender always delivers its outputs directly. Since there are no input buffers, the receiving IP consumes and processes only if all its input interfaces are fed with a valid data (i.e. all VALID signals asserted to 1 at the same clock cycle). Thus, there may be some data that are lost because the condition is not satisfied. Such a behavior is a particular case in ASAP. It corresponds to actors with $CP = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$.

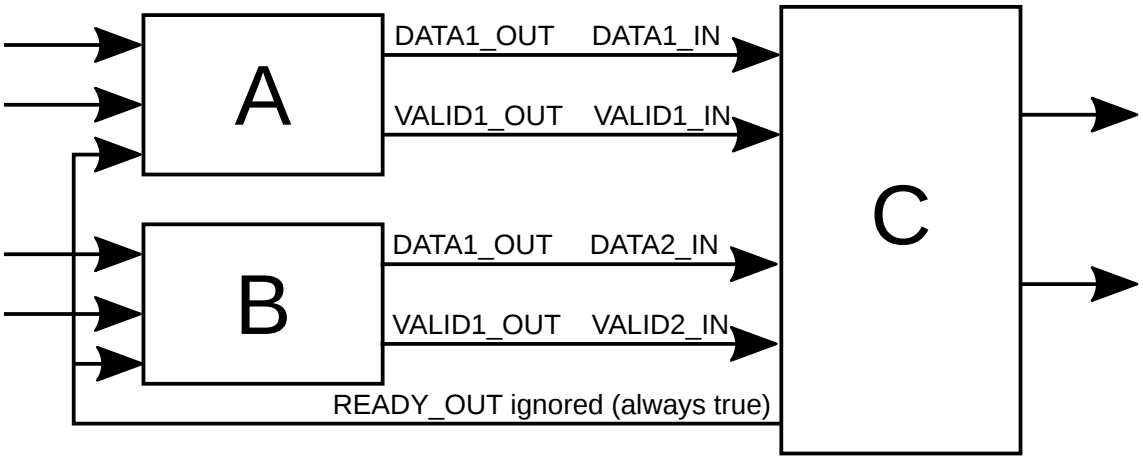

Figure 3.17: AIX4-stream - IPs using the non-blocking mode.

To summarize, even if AIX4-stream has similarities with SDF and ASAP models, it is only in terms of actor's behavior. It is just a protocol that defines a specific behavior during the execution: it cannot be used as a base for static analysis. Nevertheless, IPs that are using the non-blocking mode of AIX4-stream can be modeled by stretchable patterns. Thus, they can be integrated in our approach, as the FIR filter presented above.

3.5/ CONCLUSION

In this chapter, a very detailed analysis of SDF-AP model is carried out to bring light to its main limitations: auto-concurrency property, the fact that patterns must be strictly matched, and the mandatory buffering. Then, we have introduced Actors with Stretchable Access Patterns (ASAP), a novel way to address the scheduling problem of actors. It allows to model actor's behavior close to the hardware and to avoid some of the mentioned

drawbacks of previous SDF and SDF-AP models. The ASAP model is presented from the aspect of actor, where actor's context and structure is firstly introduced with the definitions of patterns and schedules. The algorithms to make the transmutations between patterns and schedules are given, too. The examples make it easier to understand the behaviors of actors including computation, execution, concurrency and delay between executions. After illustrating the related concepts of actor in the model, the algorithm of output pattern generation is also provided.

A FIR filter generated from the Xilinx CoreGen tool has been tested to evaluate whether the principles of the ASAP model are feasible with some real FPGA components. We compared the boolean signals of the FIR filter generated by CoreGen and the output patterns computed from our algorithm. Except for a strange case, the output results are the same between the simulations and the algorithm. Thus, the effectiveness of our model is confirmed. Finally, a quick study of the AIX4-stream protocol pointed out that IPs that are compliant with its non-blocking mode can be modeled by ASAP.

The detailed strategies to analyze the designs will be elaborated in the next chapter. The core algorithms for compatibility checking and pattern modification will also be provided together.

4

STRATEGIES FOR DESIGN ANALYSIS BASED ON ASAP MODEL

4.1/ INTRODUCTION

The novel model Actors with Stretchable Access Patterns (ASAP) has been introduced in Chapter 3. It describes the behavior of blocks using executions of actors with patterns and schedules for the inputs and outputs. It allows to model a wide range of real IP behaviors. The main aim of this chapter is to present how to analyze the designed system. First of all, we must provide approaches to check the pattern compatibility of all the actors. Indeed, if all the actors in the design have compatible IPs and CPs, the outputs should be correct. Otherwise, some modifications are needed. In practice, incompatibilities come from the fact that some inputs come too early, or there are too many compared with those are required to be consumed. Basically, it implies to delay some inputs, put them in a buffer, or at worst to decimate them. Thus, in order to ensure the compatibility for all blocks, some solutions to determine a minimal set of modifications that must be applied on incompatible inputs are needed.

This chapter is organized as follows. Some additional assumptions on the graph are set and some properties about compatibility resulting from the ASAP model are explained. Then, the details of the analysis process with corresponding algorithms are explored in different aspects involving sample rate checking, graph traversal, ratio checking and re-sampling, pattern compatibility checking and pattern modifications. Representative examples are also provided. At last, a realistic case is tested to verify the effectiveness of the proposed approach by making comparisons between the analysis results of both SDF-AP model and ASAP model.

4.2/ PRELIMINARY REMARKS ABOUT GRAPH ANALYSIS

4.2.1/ ADDITIONAL ASSUMPTIONS ON THE GRAPH OF ACTORS

Presently, the ASAP modeling approach assumes that designs do not contain feedbacks, i.e. there are no cycles. Thus, a whole design is modeled by a **Direct Acyclic Graph** (DAG), as illustrated in Figure 4.1. There is one source actor S, and four other actors a^1, a^2, a^3 and a^4, connected by channels. Patterns are marked on the channels. For a given

actor, the consumption pattern is indicated on input arrows, and the production pattern on output arrows. For example, CP and OP of actor a^1 are $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ respectively, which means that it has two input ports and one output port.

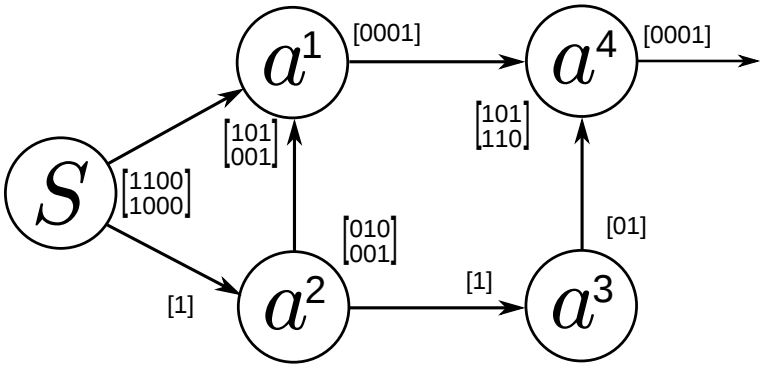

Figure 4.1: An graph presented by ASAP model.

Nevertheless, in some realistic designs, we regard some "parallel" channels between two actors as a single channel in the graph. Figure 4.2 shows a deserializer (actor D) for a video processing system, connected to a filter (actor F). Among the output ports, R, G and B represent the component signals for each pixel, and they share the same pattern. HS means a horizontal synchronization signal that is asserted at the end of every row of the frame. Similarly, VS is a vertical synchronization signal that is asserted at the end of the frame. In this example, the five output ports of D (as shown in the top of Figure 4.2) have direct relationships with the input ports of F. Thus, all channels can be regarded as a single channel, as shown in the bottom of Figure 4.2.

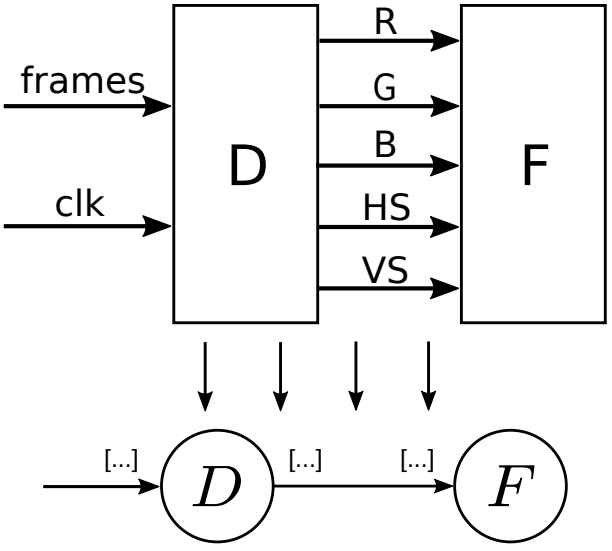

Figure 4.2: An example of channels aggregation.

4.2.2/ CORRECT PROCESSING CONDITIONS RESULTING FROM ASAP MODELING

There are mainly two necessary conditions so that the graph processes the source streams correctly. The first one is already presented in SDF based models because it is only related to the production and consumption rates of the actors. It consists in checking whether sample rates between actors are consistent. It is illustrated by Figure 4.3. An actor d has 3 precursors a, b and c. The numbers of data produced or consumed during an execution on each port are indicated on arrows that figure the channels.

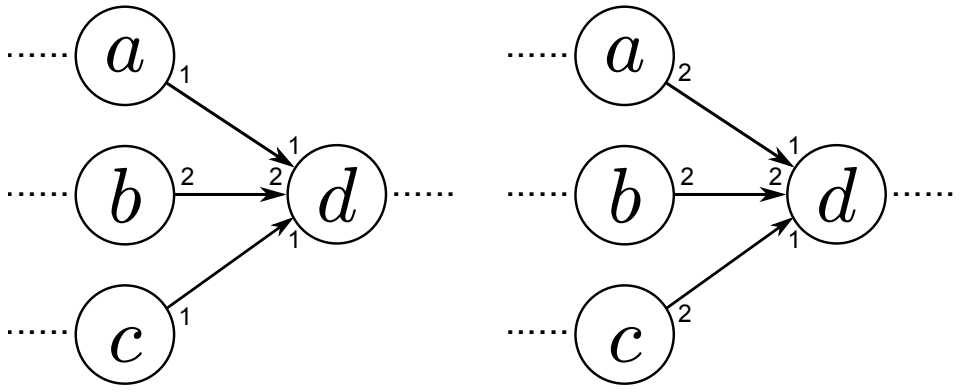

Figure 4.3: Consumption rates: the most favorable case and the unfavorable case.

For the left graph of Figure 4.3, for a single execution, what is produced by a, b and c matches exactly what is consumed by d. This is the most favorable case and no further computation need to be done to ensure that the sample rates are consistent.

For the right graph in Figure 4.3, for a single execution, the sample rates are inconsistent. It is quite straight forward to compute the number of executions of each actors to ensure the consistency. b and d must execute twice the times of a and c. Nevertheless, a, b and c are themselves connected to some precursors and the test of consistency may lead to a number of executions that turn to be contradiction with the results for d. In conclusion, enforcing the sample rates of all actors to be consistent implies to use a global approach at the graph level.

A solution has been proposed for SDF model [55, 59]. Its principles are recalled in Section 4.3.1 and explained with characteristic examples. Nevertheless, there are some adaptations to do because of the properties of ASAP model. Indeed, concurrent consumptions must be taken into account to obtain the real consumption rates. Figure 4.4 illustrates this remark. The single difference between the two graphs is the value of Δ for c. For graph (1), $\Delta_1^c = 2$ which corresponds to the number of data groups in CP^c and no concurrent consumptions. PP^a, PP^b and CP^c indicate that for a single execution a and b produces two data, c consumes one on its first input and two on the second. Thus, the sample rates are inconsistent, unless b and c execute twice the times of a. For graph (2), $\Delta_2^c = 1$ yielding concurrent consumptions. Every input data group triggers a new execution of c that will consume a data from a and b. So sample rates are inconsistent but it can be the case if c executes twice the times of a and b. The solutions are different for both graphs, showing that Δ has an impact while checking the sample rates. In fact, for a given input, the consumption rate is the minimum value between Δ and the number of 1

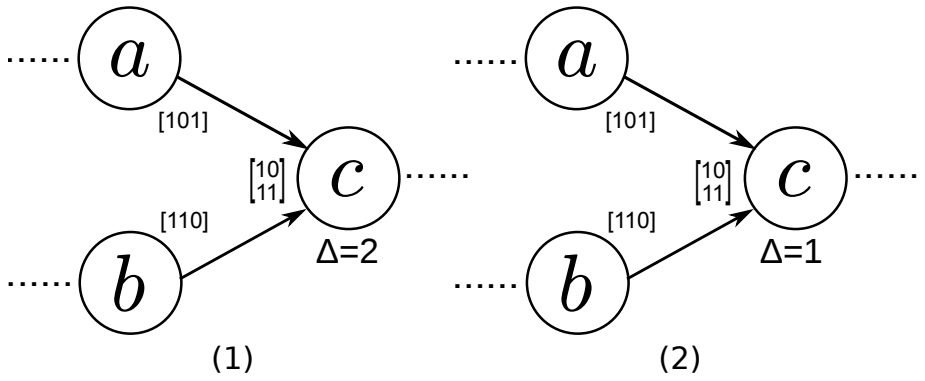

Figure 4.4: Consumption rates of an actor in different cases.

in CP.

This first condition can be expressed differently, taking into account the self-scheduling property of the actors. Assuming that sources are executing regularly, they have an output pattern that is a concatenation of their PP. These output patterns constitute the input patterns of actors that are the successors of the sources. Thus, these actors are themselves executing repeatedly and regularly, producing a cyclic output pattern. This process repeats for the whole graph. Nevertheless, in the ASAP model, the execution pace of an actor cannot be set freely because it is self-scheduled. Thus, the duration of a given number of executions depends on Δ and the output pattern of the precursors. Since patterns are cyclic, for a given actor, it is possible to search for a bounded number of executions during which it consumes all data produced by its precursors. If this number does not exists, it means that some data will be lost. Even FIFOs on channels cannot solve this problem because they will grow infinitely. In this case, we choose to apply a decimation on the streams in order to obtain consistent sample rates. It is discussed in Section 4.3.3.

The second necessary condition is that IP and CP are compatible for each block. Indeed, consistent sample rates do not imply that input data received on different ports will be perfectly synchronized, as expected in CP. And even for a single port, some data may arrive at an incorrect clock cycle during the actor's execution. The principles to check the compatibility are explained in Section 4.3.4.

4.3/ Strategies for Design Analysis

Based on the above remarks, we can deduce practical rules and obtain a procedure to use the model for system analysis. Figure 4.5 shows the flow chart of the procedure to check and possibly achieve a correct processing.

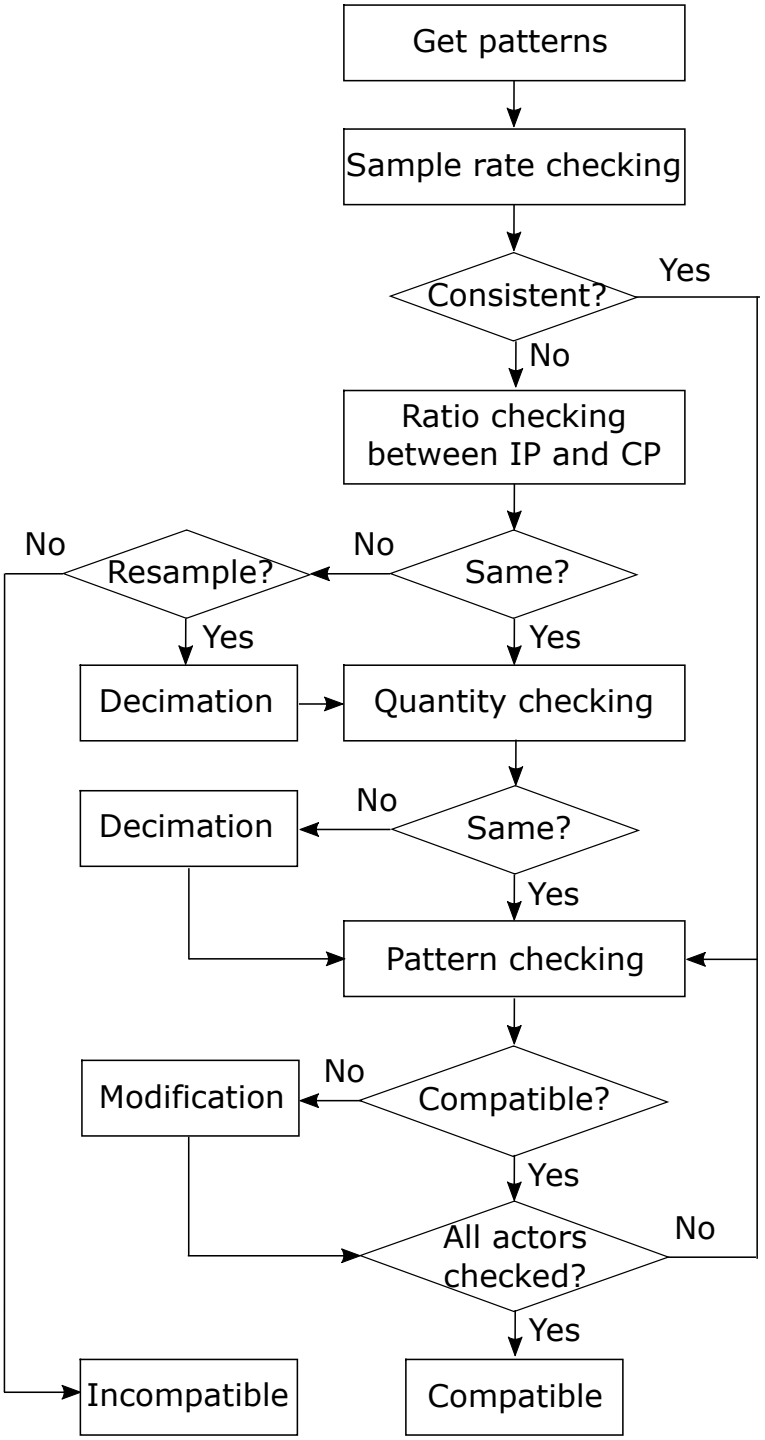

Figure 4.5: The flow chart of conformance checking and modification.

Firstly, the sample rates for the whole graph are checked. If they are consistent, then it is possible that the design is feasible. But if IP and CPs are incompatible, modifications of the graph are necessary, which mainly consist in inserting delays, stream restructurers or even FIFOs before some input ports. Otherwise, if the sample rates are inconsistent, data can not be processed correctly. In some cases, data streams can be resampled via

decimation yielding acceptable results. But if the nature of the design does not allow for sufficient decimation, it is considered to be incompatible with our requirements because data streams cannot be processed correctly. It is worth noting that these different checks need to traverse the whole graph in a pertinent order to achieve a correct analysis.

All these phases are discussed in details in the following sections.

4.3.1/ Sample Rate Checking

In the whole system, all produced data groups must be consumed during a finite number of executions. This means that the data groups may be stored temporarily and be consumed at last. Sometimes buffers are needed, but the sizes of the buffers should be finite. There is a notion defined in SDF based models as **repetition vector** to test whether the sample rates are consistent. If there is an existing repetition vector q for the graph of the designed system, it can come back to its original state after a fixed number of executions. Then, it is said to be bounded [55, 59, 76]. The repetition vector belongs to the kernel of the **topology matrix** Γ, i.e.

$$\Gamma q = \mathbf{O} \tag{4.1}$$

It can be noticed that $\mathbf{O}$ corresponds to no changing state after a finite number of executions (indicated in q), thus there are no more data to be consumed/produced. If this equation has a non-zero solution, the system can return to the initial state cyclically. The topology matrix describes the relationship between the channels and the actors. Each column corresponds to an actor, and each row to a channel. $\Gamma_{i,j}$ gives the number of data produced or consumed by the actor j on channel i. If this value is positive, it reflects a production, and a negative value represents a consumption.

There is a condition for the existence of q: for a graph with s actors, the repetition vector q exists only if $rank(\Gamma) = s - 1$.

$rank(\Gamma) = s - 1$ is a sufficient and necessary condition for the existence of q and also a necessary condition for the existence of an executable schedule for the system represented by the graph [55].

To illustrate this result, we can redraw the graph in Figure 4.1 as in SDF based models, with the token rates (parameter C of the patterns in ASAP model) marked on the channels of the graph. Figure 4.6 is the result of this operation and Figure 4.7 shows the labels that are assigned to actors and channels. The topology matrix has the following form,

$$\Gamma^1 = \begin{bmatrix} 2 & -2 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -2 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{4.2}$$

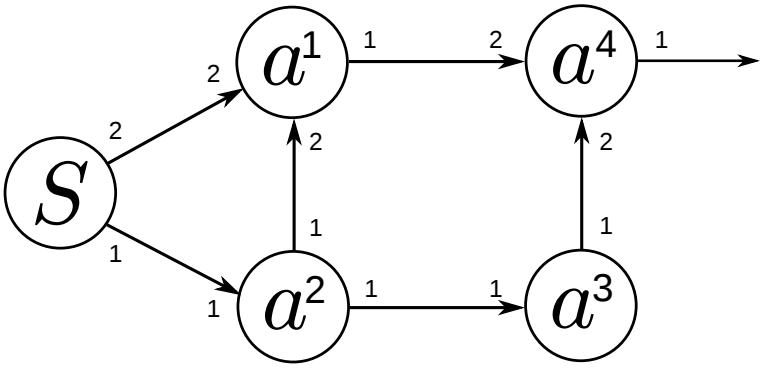

Figure 4.6: A graph (consistent) presented in SDF model.

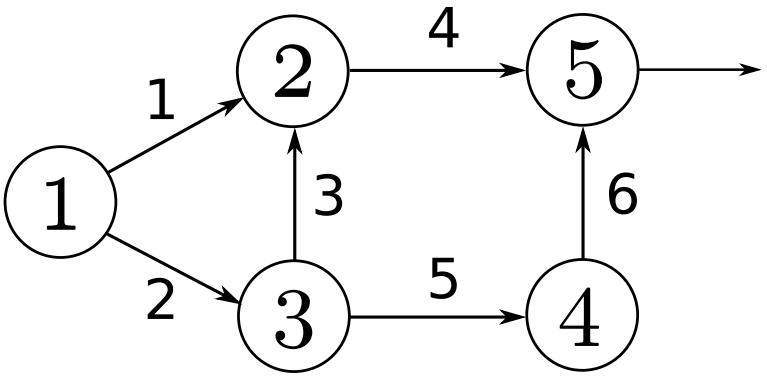

Figure 4.7: A labeled graph.

The five columns in the matrix Γ^1 represent the five actors, and the six rows the six channels. Their order is given by their labels. For example, column 1 is associated to the source labeled by 1, which effectively produces two tokens on channel 1 ($\Gamma^1_{1,1} = 2$) and one token on channel 2 ($\Gamma^1_{2,1} = 1$). Using the matrix Γ^1, we deduce that $rank(\Gamma^1) = 4$ thus $rank(\Gamma^1) = s - 1$ because $s = 5$. Therefore, the repetition vector exists and can be calculated according to equation 4.1,

$$q_1 = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 1 \end{bmatrix} \tag{4.3}$$

From q_1, we can get that when the actors S, a^1, a^2 and a^3 execute two times and a^4 one time, all the data produced can be consumed and the system can go back to the original state. In other words, for the graph shown in Figure 4.6, the sample rates are consistent, and the next step is to check the pattern compatibility of inputs and consumptions.

However, the sample rates are sometimes inconsistent, as in the graph shown in Figure 4.8.

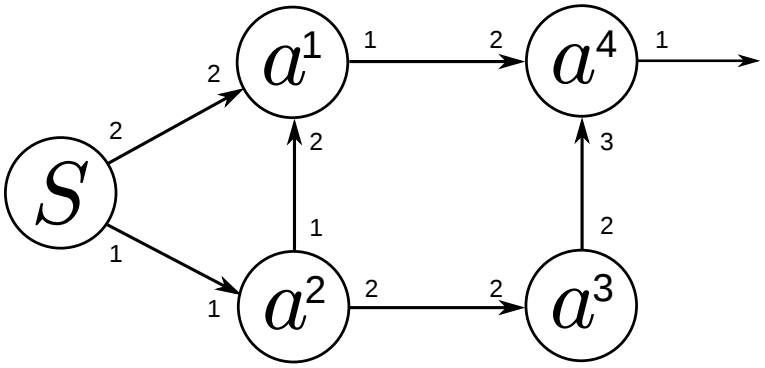

Figure 4.8: A graph (inconsistent) presented in SDF model.

The topology matrix is

$$\Gamma^2 = \begin{bmatrix} 2 & -2 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & -2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -2 \\ 0 & 0 & 2 & -2 & 0 \\ 0 & 0 & 0 & 2 & -3 \end{bmatrix} \tag{4.4}$$

The rank of the matrix Γ^2 is 5, which is equal to the number of actors. Therefore, there is no non-zero solution to the equation 4.1 and the corresponding vector q does not exist. The solution is to decimate some data to enforce the sample rate consistency, if possible.

4.3.2/ GRAPH TRAVERSAL

When it comes to the detailed analysis, ratio checking and pattern compatibility checking are not the only issues. The traversal strategy of the whole graph is also needed. The general rule is that the graph should be traversed one actor after another from the source to the sinks. Moreover, the successors must be checked after their precursors. Indeed, for a given actor, ratio or compatibility check relies on the rate or the pattern of its precursors.

Since the graph is acyclic we can get the traversal order of actors by referring to topological sort algorithms meeting the above requirements. Algorithm 5 is a feasible approach to get the order. It fills a vector O, called the **order vector** in the following, that represents the traversal order based on actors' labels. For example, assuming that X and Y are the labels of two actors, then if $O_i = X$ and $O_{i+1} = Y$, it means that Y must be evaluated just after X.

This algorithm requires to build a matrix P of size $s \times s$, that indicates which actor is the precursor of another one. It can be easily deduced from a labeled graph. $P_{i,j}$ is equal to 1 if actor i is a precursor of j, and otherwise 0. For the graph in Figure 4.7, it gives

$$P = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.5}$$

This matrix allows to build a vector d^p that contains the number of precursors for each actor, i.e. $d^p_j = \sum_{i=1}^{s} P_{i,j}$. For the P matrix above, it gives $d^P = \begin{bmatrix} 0 & 2 & 1 & 1 & 2 \end{bmatrix}$

Algorithm 5: Traversal order determination.

```
1  Compute P and d^P.
2  for i = 1 to s do // for each actor
3      for j = 1 to s do
4          if d^P_j = 0 then
5              O_i ← k                    /* append the actor label in order vector O */
6              d^P_j ← −1
7              for k = 1 to s do
8                  if P_{j,k} = 1 then
9                      d^P_k ← d^P_k − 1
10                 end
11             end
12             break
13         end
14     end
15 end
```

For each element i of O (line 2), the algorithm searches for an actor without precursor (line 3). As soon as one is found (line 4), its label is appended in O_i (line 5) and it is marked as already treated (line 6). Then, for each of its successors (line 7), it is removed from their count of precursors (line 9).

Applying this algorithm on the graph of Figure 4.7, we obtain the following processing. For $i = 1$, the first actor without precursors is found at $j = 1$. Thus, $O = \begin{bmatrix} 1 \end{bmatrix}$ and D^P is updated to $\begin{bmatrix} -1 & 1 & 0 & 1 & 2 \end{bmatrix}$.

For $i = 2$, the first actor without precursors is found at $j = 3$. Thus, $O = \begin{bmatrix} 1 & 3 \end{bmatrix}$ and d^P is updated to $\begin{bmatrix} -1 & 0 & -1 & 0 & 2 \end{bmatrix}$.

It repeats until all the actors have been sorted. At last, we can get the order vector $O = \begin{bmatrix} 1 & 3 & 2 & 4 & 5 \end{bmatrix}$, which means the feasible traversal order is $S \rightarrow a^2 \rightarrow a^1 \rightarrow a^3 \rightarrow a^4$. In this order, all the ancestors can be investigated before their successors.

4.3.3/ Ratio Checking and Resampling

Designs with inconsistent sample rates cannot be analyzed as other SDF based models. Since it is always possible to decimate the inputs/outputs, the problem is to find a strategy to obtain consistent sample rates and if it is possible, with a minimal decimation. This section details the principles of this decimation.

The main idea is to cross the graph in the traversal order and for each actor to determine whether some of its input streams must be resampled or not. In this case, it is always downsampling since the goal is to decimate the stream. So the resampling factor corresponds to the decimation rate. Taking the decision is based on a simple computation called ratio checking. Its principle is as follows. From a given number of executions of each actor, we can deduce the tokens produced and consumed on each channel. Then, for every actor with more than one input channels, we can check the ratios between the production and consumption rates. If the ratios are not equal, there are two cases. When the consumption rate is less than the production rate, decimation is needed. Otherwise, the number of executions of the producing actor needs to be multiplied by a certain value. Nevertheless, these changes may have an impact on other actors, leading to update their decimation rate and/or number of executions to keep the sample rate consistent. The whole process is achieved with Algorithm 6 that computes a **downsampling matrix** D of size $s \times s$ and the repetition vector q that ensure the sample rate consistency. $D_{i,j}$ indicates the resampling rate that must be applied between actors i and j. For example, if $D_{i,j} = \frac{2}{3}$, it implies to decimate one data out of three.

This algorithm uses two matrices called the **production matrix** PM and the **consumption matrix** CM. $PM_{i,j}$ is the number of data produced by actor i that are sent to actor j. $CM_{i,j}$ is the number of data consumed by actor i coming from actor j. Thus, if there is no connection between i and j, the value is 0. PM and CM corresponding to the graph in Figure 4.8 are as follows,

$$PM = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.6}$$

$$CM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 2 & 0 & 3 & 0 \end{bmatrix} \tag{4.7}$$

It is worth noting that these two matrices are related to the **adjacency matrix** A, which is equivalent to the topology matrix for describing a labeled graph. Adjacency matrix reflects the relationships between the actors. $A_{i,j}$ is the number of data sent (positive value) or received (negative value) between actors i and j. For the graph shown in Figure 4.8, it is

$$A = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 \\ -2 & 0 & -2 & 0 & 1 \\ -1 & 1 & 0 & 2 & 0 \\ 0 & 0 & -2 & 0 & 2 \\ 0 & -2 & 0 & -3 & 0 \end{bmatrix} \tag{4.8}$$

The relationship between A, PM and CM can be expressed as follows,

$$PM - CM = A \tag{4.9}$$

Algorithm 6: Ratio checking and resampling.

```
1  q ← [0]_S ; q_{O_1} ← 1                                  /* initialize the repetition vector q */
2  D ← [1]_{S×S}                                            /* initialize the downsampling matrix D */
3  for k = 1 to s − 1 do // main loop, using the order given by O
4      i ← O_k // get the actor's id to be evaluated according to O
       /* step 1: check if decimation needed between i and its precursors */
5      for j = 1 to s do
6          if CM_{i,j} != 0 then // j is a precursor of i
7              if q_j != 0 AND PM_{i,j} × q_i × D_{i,j} > CM_{j,i,j} × q_j then D_{i,j} = (CM_{j,i} / PM_{i,j}) × (q_j / q_i) ;
8          end
9      end
       /* step 2: check if too less prod. between i and its successors */
10     max ← 1
11     for j = 1 to s do
12         if PM_{i,j} != 0 then // j is a successor of i
13             if q_j = 0 AND q_i × PM_{i,j} < CM_{j,i}) then
14                 if max < CM_{j,i} then max ← CM_{j,i} ;
15             else if q_i × PM_{i,j} < q_j × CM_{j,i} then
16                 if max < q_j × CM_{j,i} then max ← q_j × CM_{j,i} ;
17             end
18         end
19     end
20     for j = 1 to i do q_{O_j} ← q_{O_j} × max;
       /* step 3: update q and D for successors of i */
21     for j = 1 to s do
22         if PM_{i,j} != 0 then // j is a successor of i
23             if q_i × PM_{i,j}/CM_{j,i} is an integer then
24                 if q_j = 0 OR q_i × PM_{i,j}/CM_{j,i} < q_j then
25                     q_j ← q_i × PM_{i,j}/CM_{j,i}
26             else
27                 if q_j = 0 OR ⌊q_i × PM_{i,j}/CM_{j,i}⌋ < q_j then
28                     q_j ← ⌊q_i × PM_{i,j}/CM_{j,i}⌋
29                     D_{i,j} ← CM_{j,i}/PM_{i,j} × qj/q_i
30             end
31         end
32     end
33 end
```

For each actor i according to O (line 3-4), Algorithm 6 starts to compare its consumption with the production of all its precursors j, taking the current resampling rate and number of executions into account (lines 5 to 9). If j produces more than i consumes, then $D_{i,j}$ must be updated to figure a decimation (line 7). The second step checks the opposite case but for the successors (lines 10 to 19). If i produces less than j consumes, then q_i must be multiplied by an integer factor that enforces a production greater or equal to the consumption (lines 14 and 16). Nevertheless, if i has several successors, several factors may be found. The maximum factor is applied but only on the previous and cur-

rently evaluated actors (line 20). The last step is to compute the number of executions of the successors j of i (lines 21 to 32). There are two cases. If an integer number can be found to ensure the sample rate consistency, it is assigned to q_j (line 25). Otherwise, the floor value is taken (line 28) and it implies to set an additional decimation to obtain the consistency (line 29). Applying this algorithm on the graph in Figure 4.8, we obtain the following process.

For $i = 1$, sample rates between S and its successors a^1 and a^2 are already consistent. Thus, we obtain $q = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \end{bmatrix}$.

For $i = 2$, $O_2 = 3$ thus a^2 is evaluated. During step 2, condition in line 15 applies for a^2 and a^1. Indeed, $q_3 \times PM_{3,1} = 1 \times 1$ while $q_1 \times CM_{1,3} = 1 \times 2$. Thus, $max = 2$ and q is updated to $\begin{bmatrix} 2 & 1 & 2 & 0 & 0 \end{bmatrix}$. During step 3, condition in line 23 applies for a^3 yielding $q = \begin{bmatrix} 2 & 1 & 2 & 2 & 0 \end{bmatrix}$.

For $i = 3$, a^1 is evaluated. Condition in line 7 applies between S and a^1 because $q_2 \times CM_{2,1} = 1 \times 2$ while $q_1 \times PM_{1,2} = 2 \times 2$. Thus, a decimation is needed, leading to update $D_{1,2} = \frac{1}{2}$. During step 2, $max = 2$ and during step 3 the number of execution of a^4 is set to 1. It yields $q = \begin{bmatrix} 4 & 2 & 4 & 2 & 1 \end{bmatrix}$.

For $i = 4$, a^3 is evaluated. During step 1, a decimation is needed between actors a^2 and a^3, indicated by $D_{3,4} = \frac{1}{2}$. Step 2 modifies nothing. During step 3, condition in line 27 applies because a^3 produces four data while a^4 consumes three. Thus, $q_5 = \lfloor \frac{4}{3} \rfloor = 1$ and $D_{4,5} = \frac{3}{4}$.

For $i = 5$ and a^4, no further conditions apply.

Finally we get $q = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 2 \\ 1 \end{bmatrix}$ and $D = \begin{bmatrix} 1 & \frac{1}{2} & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & \frac{1}{2} & 1 \\ 1 & 1 & 1 & 1 & \frac{3}{4} \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$.

It can be noticed that this solution is not a minimum in terms of decimated inputs. Indeed, it is possible to merge the decimation between a^2 and a^3 with the one between a^3 and a^4. It implies to execute a^3 four times instead of two, and to update $D_{4,5} = \frac{3}{8}$. Such a merge is very easy to compute since it corresponds to decimations placed before and after an actor that has a single input and a single output. Nevertheless, in the present example, it leads to decimate more than the half of the data and it is not sure that it allows a^4 to compute satisfying results.

4.3.4/ COMPATIBILITY CHECKING

According to the procedure shown in Figure 4.5, as soon as the graph has consistent sample rates (with or without resampling), the next step is to check the pattern compatibility. This is achieved by using a new pattern build from CP, called the admittance pattern.

4.3.4.1/ ADMITTANCE PATTERN GENERATION

An IP is considered to be **compatible** with a consumption pattern CP if all the actor executions implied by IP produce correct results. For testing the compatibility, we define the notion of **admittance pattern** (AP) that is built from CP and matched with IP. Let us detail its principles of construction and the matching process.

Due to the structure of CP and the value of Δ, checking the compatibility of IP may turn out to be a non-trivial task in practice. The simplest case is when there are no concurrent consumptions. Then, AP is simply constituted of several concatenations of CP. If removing some of the null columns of IP leads to AP, then IP is compatible with CP. The following Example 8 illustrates this process.

Example 8: For $P_I = 2$, $L_{CP} = 4$, $C = 3$, and $\Delta = 3$, the pattern $IP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$ and $CP = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$ are compatible.

It is obviously that, removing columns $1, 2, 4, 6, 9, 12$ leads to the concatenation of two CPs.

In case of concurrent consumptions, checking the compatibility requires a similar process of removing null columns and matching with a reference. Nevertheless, this problem is more complex because it raises up some questions about the number of compatible patterns and their structures.

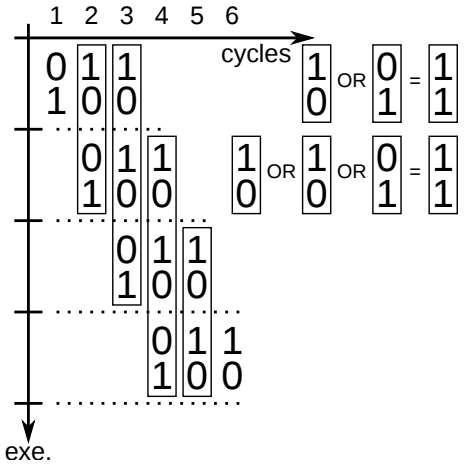

Figure 4.9: Building admittance pattern for Example 9.

Assuming that CP and Δ are consistent, for a given number of complete executions n_{exe}, there exists at least one admittance pattern. For a better understanding, the principles of its construction are firstly described with a basic case in Figure 4.9 and Example 9.

Example 9: $CP = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, $P_I = 2$, $L_{CP} = 3$, $C = 3$, $\Delta = 1$, and $n_{exe} = 4$.

In Example 9, the admittance pattern is built by copying CP n_{exe} times in the graph where columns represent the clock cycles and the rows represent the sequence of executions. Assuming that the first execution starts at clock cycle 1 and that the data groups are available as fast as possible, we can copy CP as in the first row, first column. Since $\Delta = 1$, data groups trigger a new execution at each clock cycle 2 and 3. Thus, we copy

CP in the second and third rows, respectively at columns 2 and 3. This copy process is done n_{exe} times by determining for each execution when it must (or can) start to compute correct results, taking the data groups needed by the previous executions into account. For example, the third execution needs the second data group at clock cycle 4, which triggers the fourth execution.

Even if there are different consumption policies at each clock cycle for concurrent consumptions, they can be unified by doing a logical OR element by element of the associated columns of CP. For example, at clock cycle 3, the first execution consumption is specified by $CP_{*,3}$, the second by $CP_{*,2}$ and the third by $CP_{*,1}$. The logical OR makes the input pattern compatible with the three executions. Doing this operation for each clock cycle yields the admittance pattern. For Example 1, it produces $AP = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$.

As in the simple case without concurrent consumptions, AP is just CP repeated n_{exe} times. If removing some null columns of an IP leads to AP, the IP is compatible with the corresponding CP.

In some cases, a problem arises. There may be more than one admittance pattern that can be built with this process. When CP contains null columns, for a given execution, there may be several choices of column to copy CP in the graph. Moreover, for $n_{exe} = \infty$, there may be an infinite number of admittance patterns if the same choices occur repeatedly. Such a case is illustrated by Figure 4.10 and Example 10.

Example 10: $CP = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$, $P_I = 2$, $L_{CP} = 5$, $C = 3$, $\Delta = 2$, and $n_{exe} = 4$.

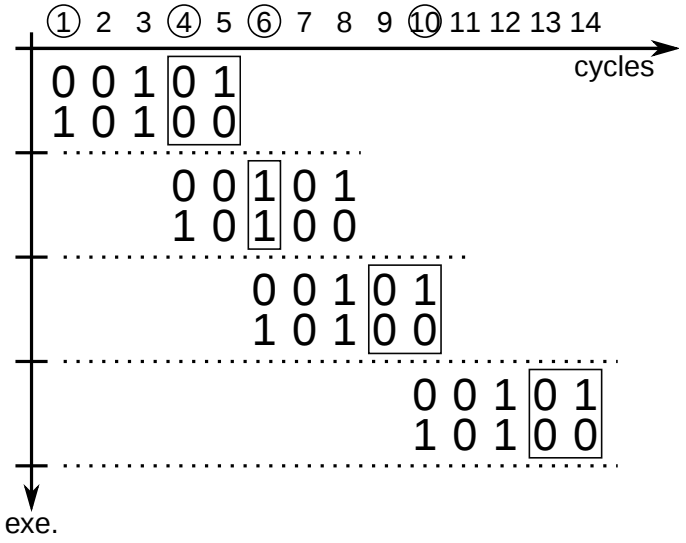

Figure 4.10: Infinite number of choices when building an admittance pattern.

Boxes in the graph represent the choices in different clock cycles to start the next execution. For example, since $\Delta = 2$, the second execution must start as soon as the third data group arrives. In the simplest case, it is at clock cycle 5. Nevertheless, since $CP_{*,4}$ is a null column, it could be at clock cycle 4, without causing any problem of compatibility. Thus, we have the choice to start the second execution at clock cycle 4 or 5. The former is taken in this example and it implies that there is a single choice for the third execution.

But for the fourth execution, we have once again two choices, and if it starts at clock cycle 10, it leads to two choices for execution 5. Thus, whatever the choice is done for a given execution, there will be more than one choice for some future executions.

Obviously, if n_{exe} is bounded, we can explore all combinations of choice to compute all the possible admittance patterns. But their number can become huge quickly as n_{exe} increases and checking IP compatibility with each of them is not a satisfying solution. Nevertheless, considering that "real" actors generally consume a data group at each clock cycle or are totally unavailable for that, null columns in CP is a very particular case. This is why we only consider actors with a consumption pattern without null columns.

If CP does not contain null columns (but may contain columns of $\times$) there is a single AP that contains a cyclic part repeated a number of times depending on n_{exe}. For the absence of null column, the latency between two executions is the same depending on Δ. If the given CP and Δ are consistent, the overlaps between two CPs are the same. That leads to the cyclic part consisting of some repeated patterns. The number of repetitions is $n_{exe} - 1$. Thus, the corresponding AP exists and is unique.

Under this assumption, the general process of building AP is given by Algorithm 7. For convenience, the algorithm starts by allocating and initializing an array with the maximum theoretical size of AP, i.e. $L_{CP} \times n_{exe}$ (line 1 to 4). As explained in Example 1, we assume that the first execution starts at clock cycle 1, thus CP is copied at the beginning of AP (line 2). For each following execution i, the algorithm searches for its triggering, which corresponds to the next insertion point of CP, named t. This is simply done by counting Δ valid groups (line 8 to 13). If there are columns with only $\times$, they are skipped until finding a valid group (line 14 to 16).

Then, each column of CP must be combined with a column of AP, starting at $t' = t$ (line 17 to 36). For each couple of columns, there are four possible cases:

1. The insertion point t' is after the current end of AP, thus the column of CP is just copied (line 19 to 22).

2. The columns of CP and AP can be combined directly by a logical OR (line 23 to 25), as in Figure 4.9 and example 1. This occurs when there are no operations like $\times$ OR 1.

3. The column of CP is only composed of $\times$ (line 26 to 31). If AP is not a $\times$-column, it implies to shift to the next column of AP, then copy the column of CP in the empty space.

4. The column of AP is only composed of $\times$ (line 32 to 35). The insertion point is incremented by one and no combination is needed.

Algorithm 7: Admittance generation.

for $i = 1$ to $L_{CP} \times n_{exe}$ **do**
  if $i \leq L_{CP}$ **then** $AP_{*,i} \leftarrow CP_{*,i}$;
  else $AP_{*,i} \leftarrow$ null column ;
end
$t \leftarrow 1$
$L_{AP} \leftarrow L_{CP}$
for $i = 2$ to n_{exe} **do**
  // search start of i^{th} exe.
  $count \leftarrow 0$
  while $count < \Delta$ **do**
    if $AP_{*,index}$ is a valid group **then**
      $count \leftarrow count + 1$
    $t \leftarrow t + 1$
  end
  while $AP_{*,index}$ is an $\times$ column **do**
    $t \leftarrow t + 1$
  end
  // combine CP with AP, from t
  $t' \leftarrow t$
  for $j = 1$ to L_{CP} **do**
    if $t' > L_{AP}$ **then**
      copy $CP_{*,j}$ at $AP_{*,t'}$
      $L_{AP} \leftarrow L_{AP} + 1$
      $t' \leftarrow t' + 1$
    else if $AP_{*,t'}$ and $CP_{*,j}$ can combine **then**
      $AP_{*,t'} \leftarrow AP_{*,t'}$ OR $CP_{*,j}$
      $t' \leftarrow t' + 1$
    else if $CP_{*,j}$ is an $\times$ column **then**
      if $AP_{*,t'}$ is a valid group **then**
        shift to the next column of AP from t'
        $L_{AP} \leftarrow L_{AP} + 1$
      copy $CP_{*,j}$ at $AP_{*,t'}$
      $t' \leftarrow t' + 1$
    else if $AP_{*,t'}$ is an $\times$ column **then**
      $j \leftarrow j - 1$
      $t' \leftarrow t' + 1$
    end
  end
end
remove existing columns of AP after L_{AP}.

Algorithm 7 and these four cases are illustrated by Example 11 and Figure 4.11.

Example 11: $CP = \begin{bmatrix} 0 & 1 & \times & 1 & 1 \\ 1 & 0 & \times & 1 & 1 \end{bmatrix}$, $P_I = 2$, $L_{CP} = 5$, $V_{CP} = 4$, $\Delta = 1$, and $n_{exe} = 3$.

The first row in Figure 4.11 is the initial copy of CP that corresponds to the first execu-

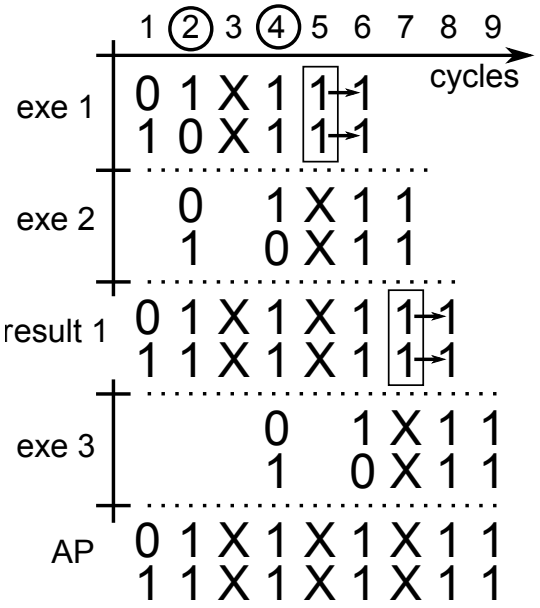

Figure 4.11: Building admittance pattern for Example 11.

tion. Then, the algorithm enters into the main loop (line 7) and searches (line 9 to 16) for the triggering of the second execution ($i = 2$), which is at $t = 2$. The new insertion point t' has been found and thus, CP can be combined with current AP (line 18 to 36) with the four possibilities mentioned above:

- At $t' = 2$, case 2 applies: $CP_{*,1}$ and $AP_{*,2}$ can be combined with a logical OR directly.
- At $t' = 3$, case 4 occurs. This leads to search for a possible insertion of $CP_{*,2}$ in the next several indexes, and it is effectively possible at $t' = 4$, where case 2 applies.
- At $t' = 5$, case 3 occurs . Since $AP_{*,5}$ is a valid group, AP is shifted right from t' (the next column figured by the small arrows), leaving an empty space where $CP_{*,3}$ can be copied.
- At $t' = 6$, case 2 occurs once again.
- At $t' = 7$, case 1 occurs.

At the end of the j-loop, columns 1 to 7 of AP have been modified, as shown in the row of Figure 4.11 entitled result 1. The bottom rows illustrates the same principles for the third execution that starts at $t = 4$. It yields $AP = \begin{bmatrix} 0 & 1 & \times & 1 & \times & 1 & \times & 1 & 1 \\ 1 & 1 & \times & 1 & \times & 1 & \times & 1 & 1 \end{bmatrix}$.

By construction, AP is unique and contains a cyclic part repeated a number of times depending on n_{exe}. Since the latency between two executions is fixed and driven by Δ, there is a unique AP because there are never several choices for the insertion point to copy CP. Moreover, there is a cyclic sequence of overlapping executions, except at the beginning and/or the end of AP. The length of this sequence is simply equal to the increment of t. In Example 11, the increment is 2 and this corresponds to the length of the cyclic part of AP, which is $\begin{bmatrix} \times & 1 \\ \times & 1 \end{bmatrix}$. For the same example, $\Delta = 2$ would result in an

increment of 3, and a cyclic part is $\begin{bmatrix} \times & 1 & 1 \\ \times & 1 & 1 \end{bmatrix}$.

4.3.4.2/ PATTERN COMPATIBILITY CHECKING

Algorithm 8: Compatibility checking.

```
1  for i = 1 to length(AP) do
2      if AP_{*,i} is a × column then
3          AP_{*,i} ← null column (i.e. only 0)
4      end
5  end
6  t ← 1
7  while IP_{*,t} contains only 0 do
8      t ← t + 1
9  end
10 i ← 1
11 while t ≤ length(IP) do
12     if IP_{*,t} != AP_{*,i} then
13         if AP_{*,i} contains some 1 then
14             while t ≤ length(IP) and IP_{*,t} contains only 0 do  /* search for next valid
                data group */
15                 t ← t + 1
16             end
17             if IP_{*,t} != AP_{*,i} then                          /* incompatible */
18                 return false
19             else                  /* compatible,IP and AP go one column further */
20                 t ← t + 1
21                 i ← i + 1
22             end
23         else                                                      /* incompatible */
24             return false
25         end
26     else                          /* compatible,IP and AP go one column further */
27         t ← t + 1
28         i ← i + 1
29     end
30 end
31 return true
```

We can check whether IP is compatible with AP by using Algorithm 8. The process is similar to the one described in Example 8, but CP should be replaced by AP obtained by Algorithm 7. Where there are $\times$ in the CP, they are turned into 0s (line 1 to 3).

In practice, there are usually some invalid data groups in the front of inputs. Thus, the algorithm firstly searches for the first valid data group (line 5 to 7). Then, if IP is the same as AP or if it just contains additional null columns between two valid data groups compared with AP, they are compatible (returning $true$). Otherwise, they are incompatible (returning $false$).

When analyzing the whole graph, the order of evaluation is given by the order vector O. For each actor, APs are built using the number of executions given by the repetition vector found during the sample rates checking phase. Its input pattern IP is deduced from the output pattern OP of its precursors. If the actor passes the compatibility check successfully, we compute its output pattern OP with Algorithm 4 and evaluate the next actor. If the compatibility test fails, a modification of the input pattern is investigated to ensure the compatibility, which is discussed in the next section. This principle repeats until all actors have passed the test successfully.

4.3.5/ Pattern Modification

4.3.5.1/ Synthesis on an Example Case

In practice, some very simple designs may lead to incompatible IPs. Such a design is given in Figure 4.12. It is intended to process a stereo signal with different filters (clipper, average and compressor) on left and right channels, before joining them to produce a mono signal. For each actor, CP, PP, PC, and Δ are given. These characteristics are perfectly plausible for the clipper and the average filter but have been chosen for the sake of illustration for the compressor.

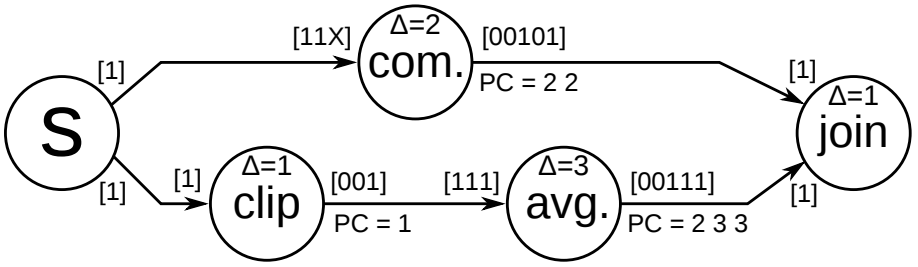

Figure 4.12: An ASAP design: filtering a stereo signal.

Firstly, if we assume that the source S produces a data at each clock cycle, it is obvious that the consumption pattern of compressor is incompatible with that, while those of the clipper and the average filter are compatible. If we want to avoid buffering with an infinite growth, a possible solution is to decimate the streams. Before the compressor, we keep the first two data out of three, and after the average filter, we keep the first and third data only. Under this assumptions, we have:

$$OP^{cmp.} = \left[\begin{array}{cccccccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \dots \end{array} \right]$$

$$OP^{avg.} = \left[\begin{array}{cccccccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & \dots \end{array} \right]$$

Then, a simple delay of two clock cycles may be put after the compressor to re-synchronize the streams before the junction. The main point is that decimators and delays are very easy to be generated automatically in VHDL, and use much less resources than FIFOs.

Secondly, we assume that the source S produces a data at every two clock cycles. In this case, all consuming patterns are compatible. We obtain:

$$OP^{cmp.} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \dots \end{bmatrix}$$
$$OP^{clip.} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \dots \end{bmatrix}$$
$$OP^{avg.} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \dots \end{bmatrix}$$

Then, there are several solutions to re-synchronize streams before the junction. One of them is to use a bi-state delay after the compressor. For every triplet in the compressor output, it adds a two clock cycles delay to the first two outputs, and a single one to the third. In fact, such a delay is also very easy to be generated automatically in VHDL.

We can observe that there is no need of FIFOs in this simple example. In fact, this is always true when merging branches in the design (like for the junction) are streaming the same volume of data for a given number of clock cycles. If it is not the case, some decimation must be done or a FIFO inserted, with the risk that it may overflow.

It is very easy for a human to find a solution for a re-synchronization of the streams in this example. Nevertheless, this is not always possible for complex designs. Thus, it is better to schedule the designed system under constraints to compute automatically a minimal set of components like delays, decimators and FIFOs, so that all actors can receive compatible input streams and execute correctly.

4.3.5.2/ PRINCIPLES OF PATTERN MODIFICATION

As discussed above, decimation is needed for a graph without consistent sample rates, which replace some 1 by 0 in patterns. Moreover, for a given actor, if IP and AP are incompatible, it is because some 1 in IP are misplaced compared with AP. Then a delay must be applied. This delay may concern only this 1 or maybe all. It totally depends on IP and AP. These two situations are considered as pattern modifications. They can be applied because patterns are stretchable. It makes the ASAP model more flexible than previous ones.

The whole process of pattern modifications is given in Algorithm 9, which applies delays and decimations at the same time for n_{exe} executions of an incompatible actor a. It uses IP, IS (the input schedule), AP and a decimator vector d of size P_I. For each input port i connected to an output port of an actor b, $d_i = n_{exe} \times CM_{a,b} \times D_{b,a}$. It also uses a matrix DS called **decimation schedule** to store for each input the clock cycle at which a 1 is turned into a 0.

Algorithm 9 starts to find the first valid group in IP (line 2) at clock cycle t. Then, for each column i of AP, it tries to obtain a match with IP (lines 3 to 35). For each input port j, there are three cases taken into consideration. If $AP_{j,i} = 0$ and $IP_{j,t} = 1$ (lines 8 and 9), it means that this 1 comes too early. Thus, it is possible to decimate (lines 9 to 12), or delay for a certain amount of clock cycles (line 14). If $AP_{j,i} = 1$ and $IP_{j,t} = 0$, it may indicates that a 1 comes too late. Nevertheless, it depends on the other inputs, thus the decision is postponed and $flag1$ is assigned to 1 (line 17). If $AP_{j,i} = IP_{j,t} = 1$, there is at least a match and $flag2$ is assigned to 1.

At the end of this loop, if only $flag1$ is equal to 1, it means that $IP_{*,t}$ has been modified to contain a null column, which is not a problem for the actor. If only $flag2$ is equal to 1, it means that there is a complete match between AP and IP. Finally, if $flag1$ and $flag2$ are both equal to 1 (lines 23 to 32), it means that there is a partial match. In this case,

another decimation (lines 25 to 27) or exchange (line 29) is made when $IP_{j,t} = 1$.

After these two loops, IP is either a null column or matches AP. In the second case, i is incremented to evaluate the next column of AP.

Algorithm 9: Pattern modification and decimation.

```
1  t ← 1; i ← 1
2  while IP_{*,t} contains only 0 do t ← t + 1;
3  while i ≤ length(AP) do
4      flag1 ← 0 // 1 when AP = 1 and IP = 0
5      flag2 ← 0 // 1 when AP = 1 and IP = 1
6      for j = 1 to P_I do // for each input port
7          if IP_{j,t} != AP_{j,i} then
8              if IP_{j,t} = 1 then
9                  if d_j ! = 0 then                    /* decimation needed */
10                     IP_{j,t} ← 0 // turn the 1 into 0
11                     d_j ← d_j − 1
12                     delete t in IS_{j,*} and store it in DS_j
13                 else
14                     exchange the 1 in IP_{j,t} with the first following 0
15                 end
16             else
17                 flag1 ← 1
18             end
19         else
20             if AP_{j,i} = 1 then flag2 ← 1 ;
21         end
22     end
23     if flag1 = 1 AND flag2 = 1 then
24         for j = 1 to P_I do // for each input port
25             if IP_{j,t} = 1 then
26                 if d_j ! = 0 then
27                     IP_{j,t} ← 0; d_j ← d_j − 1
28                     delete t in the corresponding line of IS and store it in DS
29                 else
30                     exchange the 1 in IP_{j,t} with the first following 0s
31                 end
32         end
33     end
34     if IP_{*,t} = AP_{*,i} then i ← i + 1 ;    /* all the corresponding bits are equal */
35     t ← t + 1
36 end
37 turn the extra 1 in the following of IP into 0 and move corresponding data in IS to DS
   /* omit the extra data at the end */
```

If we remove the values stored in DS from IS, we obtain the original input schedule without the clock cycles corresponding to decimations. It is noted IS^d. The modified input pattern produced by Algorithm 9 is noted IP^m and its corresponding schedule is noted IS^m.

To get enough information for VHDL code generation, Algorithm 10 is used to compute the exact numbers of the delays that should be put on each data. It is simply done by subtracting IS^d to IS^m. The result is the **delay matrix** DM. If an element in DM is equal to 0, it means that no delay is needed for the corresponding data. Otherwise a positive integer represents the length of the delay.

Algorithm 10: Delays calculation.

```
for i = 1 to P_I do
    for j = 1 to length(IS_i^m) do
        DM_{i,j} ← IS^m_{i,j} − IS^d_{i,j}
    end
end
```

4.4/ EXPERIMENTS AND ANALYSIS

In this section, a realistic design is tested to show the improvements of our ASAP model and the efficiency to implement IPs that match its constraints. The experiments are carried out based on the principles of both ASAP model and SDF-AP model with a comparison of resources consumption and latency on a real FPGA. This case comes from a project originally implemented using a Raspberry Pi board and that we have partially adapted to the APF23+SP Vision development board from the Armadeus company. This board hosts an iMX processor, physically linked to a Spartan 3, which can be used as it is, and also as a bridge to a Spartan 6 (in LX100 version) hosted on the SP Vision extension board. Both FPGAs are fed with an external signal at 100MHz, which is used to clock the design. Several jumper banks are linked to IO pins of the FPGAs, allowing to bind peripherals.

The project is based on robot cars equipped with a CMOS camera to provide video frames to a computation unit, which must recognize elliptic shapes of a particular color to detect the wheels and identify other cars. Indeed, cars have wheels painted in different colors. This pattern recognition was based on a first phase using basic image manipulations followed by a Canny filter and an ellipse detector. The proof of concept presented in this section is only based on the first phase because it is sufficient to clearly exhibit the advantages of our model.

Figure 4.13 describes this first phase, slightly modified to take the constraints of an FPGA processing into account. Its goal is to provide exploitable data to the Canny filter mainly by converting frames into grayscale and keeping pixels that only correspond to the wheels' color.

This process starts with a camera controller that grabs frames from the camera. It outputs pixels in RGB24 format as a sequence of three 8 bit values (one for each component). In order to explore any desired camera configuration easily, we have replaced this controller by a component that generates frames of a given size at a given rate and camera clock. Since the camera clock may differ from the external clock, the controller is followed by a FIFO to change the clock domain when necessary and ensure the rest parts of the design to work at 100MHz.

After the FIFO, pixels are converted in parallel into grayscale and YCbCr format. The

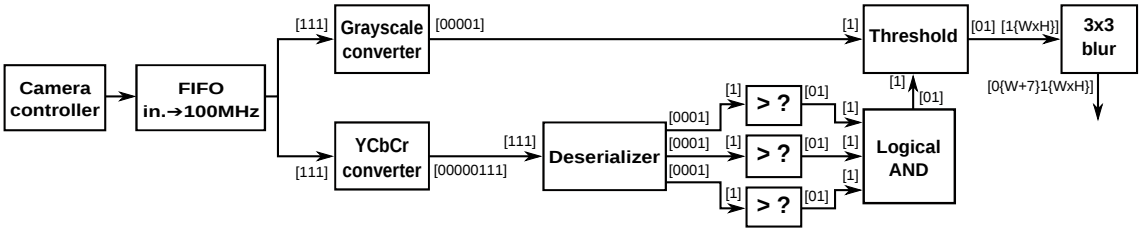

Figure 4.13: Demonstration case: a graph of blocks for real-time image processing on an FPGA.

latter is used as the first step to select pixels in a range of color and luminosity. After the conversion, a deserializer outputs the three components in parallel. Each component is sent to a block that just tests if the input is greater or lesser than a parametric value and outputs true/false depending on the result of the test. The boolean values are combined by a logical AND. A threshold receives the grayscale pixels and keeps their values as they are or sets them to 0, depending on the logical AND result. Finally, a blur filter with a 3×3 mask is done to avoid an incorrect behavior of the Canny filter.

Each block in Figure 4.13 represents an actor that has been implemented in VHDL "by hand" except the FIFO that has been generated using CoreGen from Xilinx. Since the output pattern of the FIFO is directly linked to the ratio between the camera clock and the external clock, the camera controller is not taken into account in the following experimentations and the FIFO is considered to be the true source actor. Each block has been implemented in **two versions**, one that is based on our ASAP model, and the other one that corresponds to the SDF-AP model.

In compliance with our model, an actor with stretchable access patterns is represented by a VHDL component with inputs and outputs constituted by a couple of signals `(data,validity)`. The following VHDL code gives the entity description of the threshold block.

```
entity threshold is
  generic(in_w : natural := 8; def_val : natural := 0 );
  port(
    clk          : in  std_logic;
    reset        : in  std_logic;
    data_in      : in  std_logic_vector(in_w-1 downto 0);
    data_in_enb  : in  std_logic;
    keep_in      : in  std_logic;
    keep_in_enb  : in  std_logic;
    data_out     : out std_logic_vector(in_w-1 downto 0);
    data_out_enb : out std_logic );
end threshold;
```

As mentioned above, this block reports an input value on `data_out` only if `keep_in` is equal to 1 at the same clock cycle. Otherwise, the output is set to the default value. As stated in the model, `data_in_enb` and `keep_in_enb` correspond to the validity signals that determine when `data_in` and `keep_in` carry valid values. For a correct behavior, these signals must be equal to 1 at the same time. It is very easy to translate such a condition to VHDL code as emphasized in the following code extracted from the architecture description of the threshold block.

```
threshold_process : process (clk, reset)
begin
  if reset = '1' then
    data_out_enb <= '0';
    data_out     <= (others => '0');
  elsif rising_edge(clk) then
    data_out_enb <= '0';
    if data_in_enb = '1' and keep_in_enb = '1' then
      if keep_in = '1' then
        data_out <= data_in;
      else
        data_out <= std_logic_vector(def_val);
      end if;
      data_out_enb <= '1';
    end if;
  end if;
end process threshold_process;
```

Starting from the above implementation, it is very easy to produce a version for the SDF-AP model. It merely consists in removing the validity signals and all the tests associated to them. The same principles apply for other blocks. Nevertheless, blocks that operate on a sequence of data (format conversions and blur filter) must also have a control input that indicates the beginning of the sequence. This is why there are no significant differences in terms of resources consumption between the two versions when synthesizing a single block.

Figure 4.13 gives the patterns of the different actors. W and H correspond to the width and height of the frames. CP and PP are the same for both versions. It denotes that they are based on the same code and have the same behavior when IP corresponds to CP. However, if it is not the case, the SDF-AP version is not able to produce correct results, implying to use buffers, as shown in the following.

Table 4.1 gives the production counters of different actors for ASAP version. They can be deduced directly from their behavior and/or simulations. For example, the deserializer takes three serial inputs to produce three parallel outputs. Thus, it is logical that $PC = 3$. For the blur filter, outputs starts as soon as $W + 2$ inputs have been consumed. This is why PC starts with $W+2$. But it also implies that when $W \times H$ pixels have been consumed, the filter must still produce $W + 2$ outputs. This is reflected by the end of PC that is $(W \times H)\{W + 2\}$.

Table 4.1: Production counters of blocks

Blocks	***PC***
Grayscale conv.	3
YCbCr conv.	$3, 3, 3$
Deserializer	3
Checker	1
AND	1
Threshold	1
Blur filter	$W + 2, W + 3, \ldots, W \times H - 1, (W \times H)\{W + 2\}$

Tests have been conducted using the following frame sizes: 128×128, 256×256, 512×512 and 1024×1024. We also considered five different camera clocks with five

different PPs for the FIFO (for a single frame of size $W \times H$) summarized in Table 4.2.

Table 4.2: Production patterns for different camera clocks

Camera clocks	*PP*s of FIFOs
50MHz	$(10)\{W \times H \times 3\}$
66MHz	$(101)\{W \times H \times 3/2\}$
75MHz	$(1011)\{W \times H\}$
80MHz	$(10111)\{W \times H \times 3/4\}$
100MHz	$1\{W \times H \times 3\}$

Taking into account that the camera controller produces pixel components (R,G and B) in a sequence of three (not in parallel), the camera clock corresponds to the component rate and not the pixel rate. For example, at 50MHz, a new component is produced every 20ns, so the FIFO outputs a component every two clock cycles. This gives a PP equal to 10, repeated $W \times H \times 3$ times for a whole frame. At 75MHz, the FIFO outputs three components every four clock cycles (with the second idle). Since the output pattern of the FIFO is directly linked to the ratio between the camera clock and the external clock, the camera controller is not taken into account in the following experiments and the FIFO is considered to be the true source actor.

The most important result is that the version with stretchable access patterns does not need any extra FIFO whatever the camera rate and frame size are. Nevertheless Algorithm 8 reports an incompatibility for the threshold block because its two inputs are not synchronous. In fact, the synchronization is obtained by inserting a simple delay of 6 clock cycles after the grayscale converter, which is totally negligible in term of resources consumption. After this light modification, all the blocks become compatible with their input streams. After synthesis and routing the design with Xilinx ISE, we obtained the results of the ASAP version shown in Table 4.3.

Table 4.3: Resources consumption with ASAP model

ASAP version	
FPGA resources	**Any camera clock/image size**
Slice registers	283 out of 126576 ($< 1\%$)
Slice LUTs	296 out of 63288 ($< 1\%$)
8Kbits RAM blocks	4 out of 536 ($< 1\%$)
Best achievable clock period	6.886ns

In fact, three RAM blocks are used by the blur filter to store image rows and one for the FIFO to make the clock domain conversion.

Results are totally different for the SDF-AP version. Indeed, for all camera rates except 100MHz, a FIFO is needed before the image conversions to ensure that pixel components are streamed in three consecutive clock cycles, which is not the case just after the clock domain conversion. Furthermore, another FIFO is needed before the blur filter because the grayscale conversion leads to a pixel every three cycles.

The minimum size of these two FIFOs is directly linked to the image size (in bytes) and camera rate. For example, assuming a camera rate at 75MHz within four clock cycles, there is an idle cycle without valid data, which has to be “removed”. With an image size of $1024 \times 1024 \times 3$, it implies a FIFO of size $(1024 \times 1024 \times 3)/4 = 786432$ bytes. Since

the Spartan 6 has only 536 RAM blocks of 1Kb, it means that it is impossible to process such an image. Moreover, since the second FIFO (before blur filter) is needed even with a camera rate at 100MHz, the same type of computation leads to a need of 684 RAM blocks, which is also too many. Table 4.4 summarizes the metrics given by ISE for the minimal and maximal combination of test parameters.

Table 4.4: Min. and max. combination of test parameters with SDF-AP model.

SDF-AP version		
FPGA resources	**128 × 128, 100MHz**	**512 × 512, 75MHz**
Slice registers	390/126576 (< 1%)	609/126576 (< 1%)
Slice LUTs	558/63288 (< 1%)	2170/63288 (3%)
8Kbits RAM blocks	16/536 (3%)	368/536 (68%)
Best achiev. clock per.	7.66ns	9.85ns

Even in the minimal configuration, SDF-AP version consumes more resources than the ASAP version. Moreover, maximum configuration leads to a very stressed design with a lot of RAM blocks used and the maximum clock path (9.85ns) very close to the clock period (10ns). Thus, it is not sure that a complete design including the Canny filter and ellipse detector would still match this constraint.

In terms of latency, the two versions are equivalent within a few clock cycles, which is totally consistent. Even if FIFOs greatly delay the pixels, they also allow the format converters and blur filter to consume them at each clock cycle. In the ASAP version, there are no FIFO, but the blocks consume pixels slower when they are available. Thus, in both cases, the last filtered pixels are produced at nearly the same clock cycle. Taking the behavior of the blur filter into account, the global latency is roughly equal to $(W \times H \times 3 + W) \times (100/camera\_rate)$. Table 4.5 gives two examples of timings in elapsed clock cycles to process an image (as reported by a simulation in ISim).

Table 4.5: Test results of two examples of timings

	128 × 128		**512 × 512**	
	ASAP	**SDF based**	**ASAP**	**SDF based**
75MHz	65679	65687	1049103	1049111
100MHz	49296	49301	786960	786965

These results are consistent with the above remarks. The gap between the two versions is constant and very small (5 cycles for 100MHz). The formula given above is also verified. For example, for 512×512 and 75MHz, it gives 1049258 cycles, which is very close from the simulation result.

4.5/ CONCLUSION

In this chapter, we illustrate assumptions and constraints imposed by the principles of the proposed ASAP model and provide the whole strategies for system analysis based on the model.

Besides the graph traversal and sample rate checking, the core parts of the strategies are the compatibility checking of the IPs and CPs/APs, the decimation by implementing

the ratio checking and resampling, and the pattern modification. All the approaches are explained in details with different examples and analysis. In the meanwhile, the algorithms of all the approaches are also provided. According the proposed principles and strategies, if the parameters of the designed system can pass the ratio checking and the pattern compatibility checking, the inputs can be processed directly with correct results. Otherwise, more designed systems can be processed correctly after some treatments of decimations and modifications.

At the end of the chapter, an experiment based on a real FPGA implementation is shown. In accordance to our expectations, the results show that the system can be analyzed correctly by our model, while the use of SDF-AP model yields more buffer consumption with the similar latency. Therefore, the proposed ASAP model and analysis strategies are proved to outperform the former SDF-AP model. Indeed, working with the decimations and modifications derived from our strategies, broaden the spectrum of feasible designs and reduce resource consumption.

5

A Block Assembly Tool to Build FPGA Designs (BLAsT)

5.1/ Introduction

Hardware design has been a tough work since it came into being, especially since the emerging of FPGAs and their continuous evolution in terms of size and capacities. Thus, it is necessary to study how to make the hardware programming easier. Apart from the complexity to write the VHDL code of components, the main problem is to produce the VHDL code for the whole design. Indeed, it is a fastidious task to write it “by-hand” and little modifications in the design may lead to a lot of changes in the code. This is why solutions to produce VHDL code more or less automatically are needed.

Among the current solutions, the most user-friendly tools rely on graphical interfaces, as Simulink+HDL coder or Labview. The design is built by laying functional blocks with interfaces on a panel, and connecting them. A block is the representation of a VHDL component and its interfaces are the ports of the block. Connections between blocks match the signals that link VHDL components. Generally, there is a main block that contains all others, which represents the top component of the design. The main block interfaces correspond to I/O of the FPGA. Thus, the design is abstracted by a graph that can process input data streams presented on the main block interface. In some tools, blocks are associated to behavioral models that allow to simulate the result of the process. Finally, they can generate the VHDL code of the whole design for a target architecture.

As said in the general introduction, such tools really speed up the development of designs but often suffer from some flaws. The first one is the time wasted to debug a graph that does not produce correct results. Problems can be either detected at the graph level using behavioral model of blocks, or at the VHDL level using a tool like ISim that simulates the execution. In both cases, it is quite hard to identify the source of the problem, as when debugging C code without the help of a debugger. Moreover, adding blocks to solve a particular problem may introduce other problems. Other flaws are related to the applicability of the generated VHDL code. Firstly, it is not always possible to synthesize and thus, only usable for simulations. Secondly, even if synthesis is possible, it may change the behavior noticed during simulations. For example, a multiplication is considered to be achieved during a single clock cycle. But, depending on the size of the operands and the target FPGA, it may take several clock cycles. Finally, routing and placement may lead to a minimum achievable clock period that does not match the desired constraints. It can be noticed that these VHDL problems (but not the debugging

one) may be solved by tools that can manipulate block libraries from FPGA vendors. For example, Xilinx sells such a solution (called System generator for DSPs) to be used within Simulink.

In order to address these flaws, we are currently working on a tool written in C++ and using the Qt library, based on the same principles as Simulink. It is called BlAsT (Block Assembly Tool). One of its goals is to check whether the graph can produce a correct result for a given input stream. If it is not the case, it proposes modifications of the graph to ensure the correctness. To reach that goal, BlAsT integrates the principles of ASAP model presented in Chapter 3 and 4. The second goal is to generate the VHDL code of the whole design automatically. This generation relies on patterns of VHDL code that are merely written to match the constraints of a given target architecture. It allows the code to pass the synthesis phase without modifying the expected behavior, and to maximize the chance to obtain a reasonable minimum clock period. Nevertheless, BlAsT neither ensures that this clock period will fit with the desired one nor proposes automatic modifications to enforce it.

Developing BlAsT is a long term task and it is presently in a prototype state. Some parts are partially implemented or are lacking of ergonomics. Nevertheless, the main core is operational. In the following, Section 5.2 introduces the expected software functionalities and more especially the principles of VHDL generation. Section 5.3 presents an example based on a design similar to the one given in Section 4.4 to show how to use BlAsT. Finally, a conclusion and perspectives are drawn in Section 5.4.

5.2/ BLAST FUNCTIONALITIES

The flow chart followed to work with BlAsT is shown in Figure 5.1.

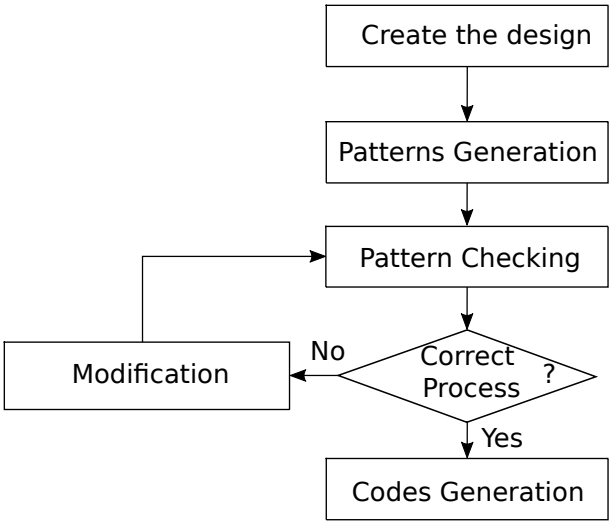

Figure 5.1: The flow chart of working process in BlAsT.

The first step is to create (or to load) the design via a graphical interface, laying blocks on a panel and connecting them. During this step, BlAsT constructs a graph that is the counterpart model of what is drawn on the screen. When the design is achieved, the user

tells BlAsT to analyze the graph to generate the input/output patterns of each block. It checks if the input and consumption patterns are compatible. If it is not the case, BlAsT proposes some modifications to enforce this compatibility for all blocks. These operations are achieved using algorithms presented in Chapters 3 and 4. Then, BlAsT can generate the VHDL code for the whole design.

The functionalities associated to this chart are presented with more details in the following sections.

5.2.1/ Project Management and Design Creation

This first group of functionalities is very similar to that proposed by Simulink or LabView. It is illustrated by Figure 5.2 that shows a demo design composed of three blocks, scattered over two groups.

The main window at the top represents the top group of the design, while the window at the bottom represents the content of a subgroup included in the top group. Both windows provide a tool bar with icons to change the creation mode, or to create/destroy subgroups. Thus, it is possible to create a design with a hierarchy of groups. For a given window, the first creation mode allows to lay blocks within the group frame. Blocks must be chosen in a hierarchical library shown at the middle right of the figure. This mode also allows to move, resize, rename, ... the blocks and their interfaces. The second mode allows to link interfaces of blocks and groups.

It can be noticed that clock and reset interfaces are not visible. Indeed, the present version of BlAsT assumes a single clock signal that comes from outside the FPGA. This input clock will be distributed automatically to all blocks during the VHDL generation.

A special category of blocks called “generator” can be used as source blocks for the design. They only have outputs and they can be outside the top group to represent a peripheral that will feed the design with data. Obviously, these blocks are only useful during the graph analysis. No VHDL is generated for them. Such a block is shown in greenish blue in the top window, feeding the design with a constant value that can be set by the user.

In fact, the main difference for the design creation between BlAsT and existing tools is that it is thought to be open, allowing the scientific community to make it evolving. Indeed, all configuration files, block representations, designs, ... are stored in XML files with open specifications. Thus, it is possible to write third-party tools that manipulates such files. For example, blocks are represented by two XML files (detailed in Section 5.2.2.3) and they can be generated from existing VHDL (NB: BlAsT provides such a tool but in a very basic version).

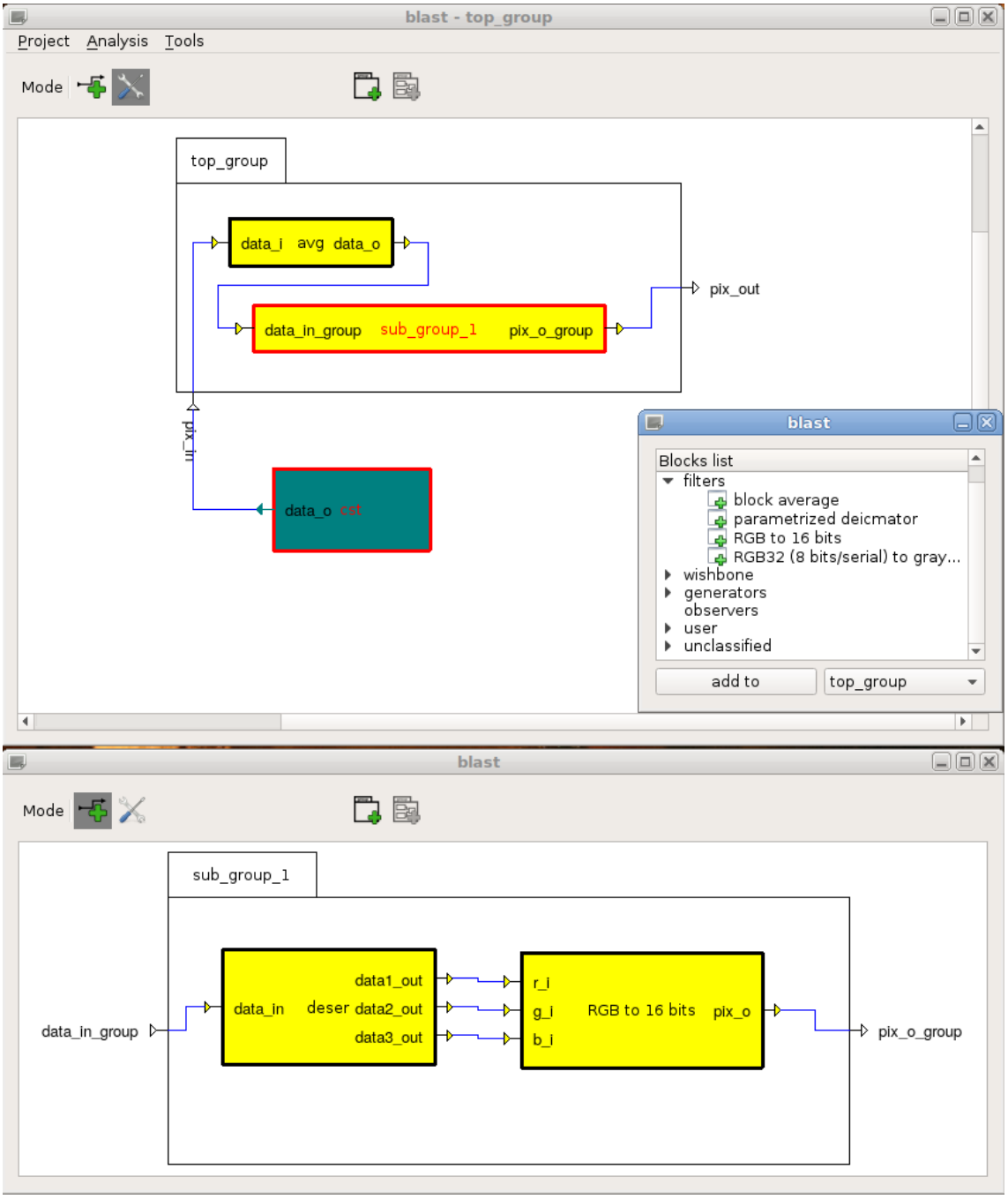

Figure 5.2: A demo design in BlAsT.

5.2.2/ Graph Analysis and VHDL Generation

For a given functional block, BlAsT separates what is needed to generate the VHDL entity from what is needed to generate the architecture. Both parts are stored in an XML file called the **reference** and the **implementation** respectively.

5.2.2.1/ THE REFERENCE FILE

The reference file is structured in three parts. The first one contains comments on the block and its function. The second one defines parameters that will modify the behavior or the structure of the block. For example, it is possible to represent the width of inputs/outputs with the same variable, instead of giving them an individual width. The value of the variable will be set by the designer before generating the VHDL code. The last parts contains the interfaces definition.

Since the syntax for defining parameters and interfaces is quite complex, it is only given in the technical documentation of BlAsT but not in this dissertation. Nevertheless, in simple cases, it is quite easy to understand. Figure 5.3 gives the reference file of the "checker" used in Section 4.4.

```
<block version="0.1">
  <informations>
    <name>checker</name>
    <category ids="9"/>
    <description>
      check if a value if >X or <X or within [X,Y]
    </description>
  </informations>
  <parameters>
    <parameter value="8" type="natural" context="generic" name="in_width"/>
    <parameter value="1" type="natural" context="generic" name="check_type"/>
    <parameter value="0" type="natural" context="user" name="inf_value"/>
    <parameter value="255" type="natural" context="user" name="sup_value"/>
  </parameters>
  <interfaces>
    <inputs>
      <input type="boolean" purpose="clock" width="1" name="clk"/>
      <input type="boolean" purpose="reset" width="1" name="reset"/>
      <input type="expression" purpose="data" width="$in_width" name="data_in"/>
      <control iface="data_in"/>
    </inputs>
    <outputs>
      <output type="expression" purpose="data" width="$in_width" name="data_out"/>
      <output type="boolean" purpose="data" width="1" name="check_out"/>
      <control iface="data_out"/>
      <control iface="check_out"/>
    </outputs>
  </interfaces>
</block>
```

Figure 5.3: An example of reference file.

All parameters have at least a name, a default value, the type of this value and at last a context. Generic means that the parameter will be "translated" into an entry in the generic block of the VHDL component. User parameters can be used within patterns or during VHDL generation. BlAsT will replace their value as soon as it encounters a special sequence that represents the parameter. For the designer, a simple right-click on a block reveals a menu with different options, one of them being to popup the block parameters windows. Figure 5.4 shows such a window for the checker block, positioned on the `$in_width` parameter and its current value that can be changed via a text field.

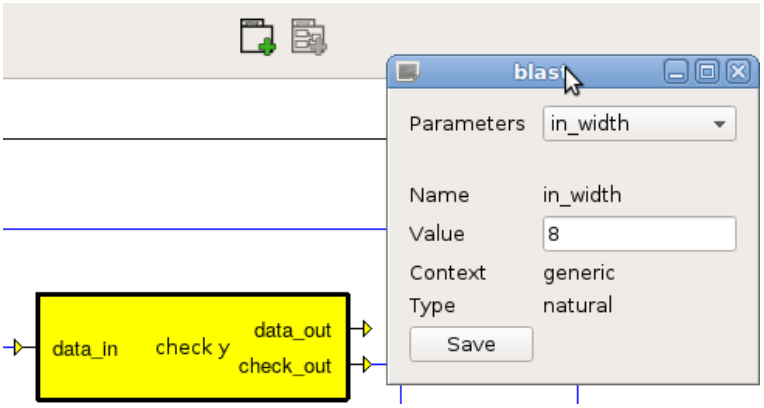

Figure 5.4: Parameter setting in BlAsT.

Interfaces have at least a name, a width, the type of this width and a purpose. When the type of the width is boolean or natural, the width itself corresponds to a numeric value. But it is also possible to specify a width with an arithmetic expression, containing variables, classical operators and common function names (floor, round, log, ...). A variable must correspond to a parameter name defined in the previous part, preceded by a $ (e.g. `$in_width`).

The purpose is quite explicit and allows to differentiate data signals from clock and reset. Finally, when an interface is used during the graph analysis, it must be associated to a control interface that will receive the validity signal.

5.2.2.2/ The Implementation File

It is worth noting that ASAP patterns are not defined in the reference file. There are two main reasons. Firstly, the reference describes the structure and not the behavior of the block which determines the ASAP patterns. Secondly, for a given structure, there are several ways to implement the function done by the block, which yields several behaviors and thus different patterns. For example, there may be different implementations for different target architectures, or because they do not use the same amount of dedicated logic resources (like multiplier, RAM blocks, ...). In conclusion, there may be several implementations files that are linked to a single reference file, and they all contain dedicated patterns.

An implementation file is structured in four parts. The first one contains general informations. The second one describes the libraries and packages needed by the implementation. The third one contains a pattern of VHDL that will be parsed by BlAsT to produce the architecture of the component. The last one defines the ASAP patterns. Figure 5.5 presents an extract from the implementation file of the checker block. The comments and libraries parts and some lines in the VHDL pattern have been replaced by "..." to shorten the code and to focus on what is the most interesting.

```
<block_impl ref_name="checker.xml" ref_md5="">
  ...
  <architecture>
begin

check_process : process (@{clk}, @{reset})
begin
  if @{reset} = '1' then
    ...
  elsif rising_edge(@{clk}) then
    ...
    if @{data_in_enb} = '1' then

      @{data_out} &lt;= @{data_in};
      @{data_out_enb} &lt;= '1';
      @{check_out_enb} &lt;= '1';

      if check_type = 1 then
        if unsigned(@{data_in}) &lt;= @{inf_value} then
          @{check_out} &lt;= '1';
        end if;
      ...
    end if;
  end if;
end process check_process;
  </architecture>

  <patterns>
    <delta value="1"/>
    <consumption>
      <input pattern="1" name="data_in_enb"/>
    </consumption>
    <production counter="1">
      <output pattern="01" name="data_out_enb"/>
      <output pattern="01" name="check_out_enb"/>
    </production>
  </patterns>
</block_impl>
```

Figure 5.5: An example of implementation file.

As shown in the architecture part, signal, port and parameter names have been replaced by escape sequences like `@{name}`. The name corresponds to the one that is used in the reference file. For signals and ports, the designer can change the reference name. In this case, the generated VHDL will use these real names instead of the reference. The escape sequences allow to detect easily where to make such changes.

BlAsT also defines more complex escape sequences that are roughly instructions. For example:

- `@eval{expr}`: will be replaced by the result of the evaluation of an arithmetic expression `expr`. Obviously, `expr` may use parameter names defined in the reference file.
- `@for{name=nb_iter:start:inc} instructions @endfor`: instructions will be generated as many times as the value of `nb_iter`. The iterator is an integer that is

initialized at `start` and that is incremented by `inc` at each iteration. Its value can be accessed via `@{name}`.

`@eval` is useful to generate a value that is based on a parameter value, like a port width. `@for` facilitates the generation of several identical components and/or signals. Figure 5.6 shows an example with three instances of a component named `compo`. A selector `sel` allows to choose what instance output is assigned to `dout`.

```
signal sel : unsigned(2 downto 0);
signal dout : std_logic_vector(@{in_width} downto 0)
@for{nums=3:1:1}
signal dout_@{nums} : std_logic_vector(@{in_width} downto 0);
@endfor
...
begin
@for{nums=3:1:1}
   compo_@{nums} : compo
      port map (
         clk => clk,
         reset => reset,
         din => din_@{nums},
         dout => dout_@{nums}
      );
@endfor
...
dout &lt;= @for{nums=3:1:1} dout_@{nums} when (sel = @{nums} ) else @endfor
           else (others => '0');
...
```

Figure 5.6: An example of `@for` instruction in an implementation file.

All the instructions are presented in details in the technical manual.

The patterns part contains all informations needed by the analysis. As stated by the model, consumption and production patterns are specified with tags for each control interface of the block, giving its name as a tag attribute. Since the production counter and Δ concern the whole block, there are no link to a particular interface. All this information can be expressed with numeric values or with expressions similar to those used in Chapter 3. Figure 5.7 shows the pattern part for the blur filter used in Section 4.4.

```
<patterns>
  <delta value="$img_width*$img_height"/>
  <consumption>
    <input pattern="1{$img_width*$img_height}" name="pix_in_enb"/>
  </consumption>
  <production counter="{$img_width+2:$img_width*$img_height-($img_width+2):1},
                       {$img_width*$img_height:$img_width+2:0}">
    <output pattern="0{$img_width+7}1{$img_width*$img_height}" name="pix_out_enb"/>
  </production>
</patterns>
```

Figure 5.7: The patterns definition for a blur filter.

5.2.2.3/ ANALYSIS

The graph analysis is conducted as presented in Figure 4.5 in Chapter 4. Nevertheless, some parts are not yet integrated in BlAsT or partly implemented:

- Resampling via decimation,
- Complex modifications based on multiple state delays or FIFOs.

Indeed, the results of Algorithm 9 are not straightforward to translate into a process that modify the input stream received on an interface. It results in quite complex operations in BlAsT that are not described here for concision. Nevertheless, some general principles can be given.

Firstly, BlAsT tries to synchronize all the input streams received by a block. It leads to add simple delays on inputs that are in advance compared with the latest one. The following example illustrates this principle.

$$AP = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & \dots \\ 1 & 1 & 1 & 1 & 1 & 1 & \dots \\ 0 & 1 & 0 & 1 & 0 & 1 & \dots \end{bmatrix} \text{ and } IP = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & \dots \end{bmatrix}.$$

In this case, a delay of three clock cycles is set on input 1, and a delay of one on input 2. It yields $IP = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \dots \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \dots \end{bmatrix}$ which is effectively compatible with AP.

Assuming now that the input pattern of the first input is:

$IP_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \dots \end{bmatrix}$, then Algorithm 9 would compute the following sequence $\{0, 1, 0, 1, \dots\}$, meaning that the first encountered 1 must not be delayed, but the second must be delayed by one. The third is not delayed, but the fourth is, and so on. In this case, if BlAsT detects a cycle, the input can be modified by a multi-state delay, which can be generated quite easily.

Finally, assuming that the algorithm computes a sequence like $\{0, 1, 2, \dots, N, 0, 1, 2, \dots, N, \dots\}$, it means that inputs are received too early compared with the consumption capacity. In this case, BlAsT use a FIFO as an input modifier. Nevertheless, it is quite complex to determine the size of the FIFO and its polling policy because it also depends on the input pattern received on other inputs. This is why such a case is not yet treated by BlAsT.

5.2.2.4/ VHDL GENERATION

VHDL generation is a recursive process that starts with the top group. From its interfaces, BlAsT generates the entity part, from the inner blocks/subgroups, it generates the components and their instances, and from the connections, it generates the signals that links the blocks/subgroups. Then, the VHDL code of blocks is generated and if there are subgroups, the same process is applied to them.

```
...
entity myproj_top_group is
  port (
    clk : in std_logic;
    reset : in std_logic;
    pix_in : in std_logic_vector(7 downto 0);
    pix_out : out std_logic_vector(7 downto 0)
  );
end entity myproj_top_group;

architecture rtl of myproj_top_group is

   component avg is
     generic (
       data_width : natural := 1
     );
     port (
       clk : in std_logic;
       reset : in std_logic;
       data_i : in std_logic_vector(data_width-1 downto 0);
       data_o : out std_logic_vector(data_width-1 downto 0)
     );
    end component;

   component sub_group_1 is
     generic (
       data_width : natural := 1
     );
     port (
       clk : in std_logic;
       reset : in std_logic;
       data_in_group : in std_logic_vector(data_width-1 downto 0);
       pix_o_group : out std_logic_vector(15 downto 0)
     );
    end component;

   signal top_group_data_i_TO_avg_data_i : std_logic_vector(7 downto 0);
   signal avg_data_o_TO_sub_group_1_data_in_group : std_logic_vector(7 downto 0);
   ...
   avg_inst : avg
     generic map (
       data_width => 8
     )
     port map (
       clk => clk,
       reset => reset,
       data_i => top_group_data_i_TO_avg_data_i,
       data_o => avg_data_o_TO_sub_group_1_data_in_group
     );
   ...

end architecture rtl;
```

Figure 5.8: An example of generated VHDL code for a top group.

Figure 5.8 shows an extract from what is produced by the generator for the top group of Figure 5.2. Some remarks can be done:

- Thanks to the purpose parameter of interfaces, BIAsT automatically connects the default clock/reset signals between blocks.
- When an input of a group is connected to a block output with a size depending on a generic value, the group inherits from the generic parameter. For example, `sub_group_1` inherits from the generic `data_width` defined in `avg`.
- Signal sizes are deduced from the sizes of the interfaces they are linked to. Obviously, these sizes must be equal.
- Component and signal names are quite verbose because they always refer to the name of the block and their interfaces. Nevertheless, they are not intended to be read by a human but just by a synthesizer.

5.3/ EXAMPLE CASE

This section presents a simplified version of the wheels detector design used in Section 4.4. Indeed, the FIFO that does the clock domain conversion has been replaced by a generator with a configurable output pattern. In the following, its output pattern corresponds to streaming a 4×3-24 bits image (i.e. RGB pixels), with a camera clock at 50MHz, which can be expressed by $(10)\{36\}$.

Apart from the FIFO, all VHDL components constituting the design have been "translated" into reference and implementation files, and integrated in the BIAsT library of blocks. The BIAsT version of the design is shown in Figure 5.9.

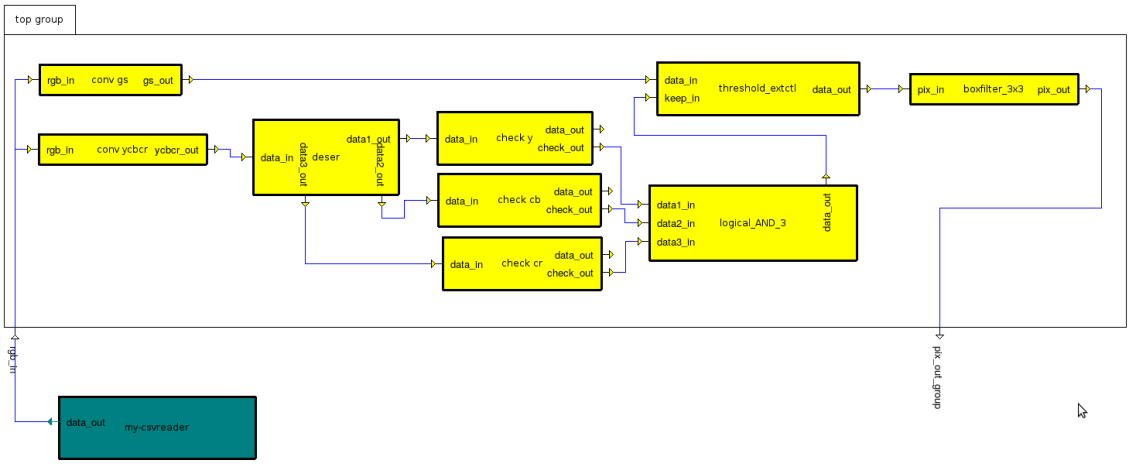

Figure 5.9: An example of wheels detector design in BIAsT.

Some interfaces and block names have been changed, notably the checker because the design uses three instances of this block. Different names avoids to generate VHDL components with the same name leading to a failing synthesis.

At the end of the creation process, we have launched the graph analysis. As shown in Figure 5.10, BIAsT has correctly detected that the threshold block has a compatibility

problem. In this case, it proposes to compute the modifications to apply on the inputs to ensure the compatibility.

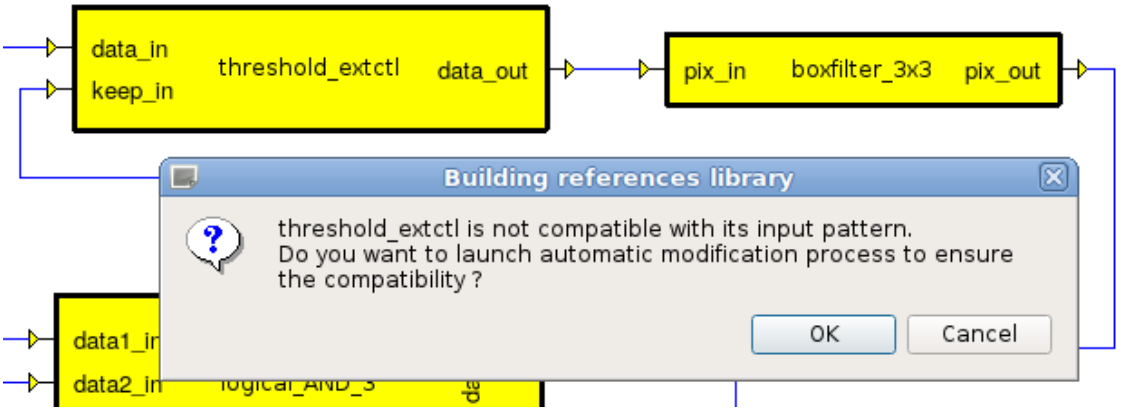

Figure 5.10: Detection of an incompatible case.

If the designer refuses this proposition, the graph analysis does not go further. Then, he has the possibility to investigate the problem by showing the input patterns of the faulty block as shown in Figure 5.11. The beginning of the patterns has been emphasized with red frames to point out that the first 1 of `keep_in` is six clock cycles after the first 1 of `data_in`. In fact, this gap remains for the following 1 as marked with the slanted red lines that are parallel. According to threshold consumption pattern, the 1 must appear at the same clock cycle. The simplest solution to obtain a synchronization of the two inputs is to delay `data_in` by six clock cycles.

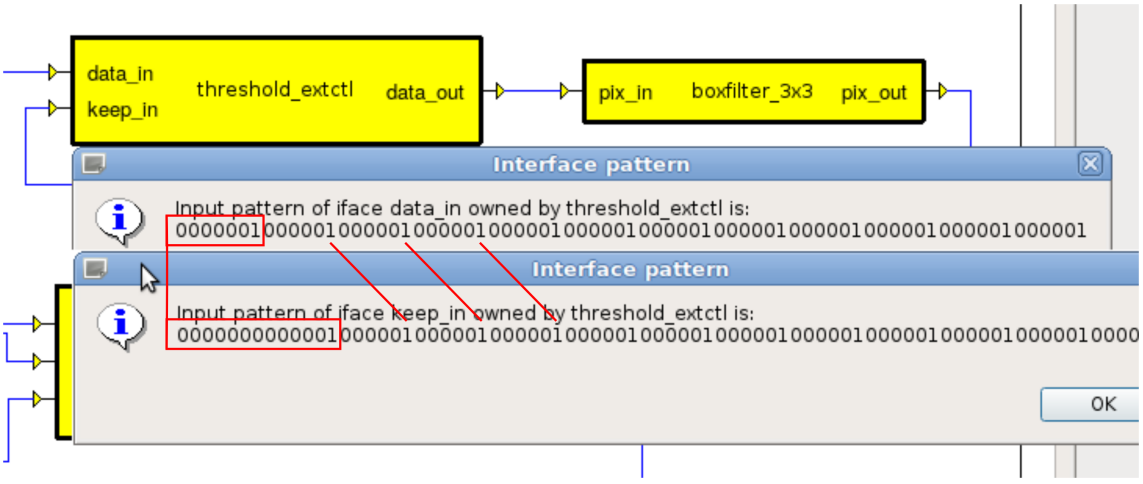

Figure 5.11: Investigating an incompatible case.

At this point, the designer has the choice to add an appropriate delay block himself, or to let BIAsT correct the problem by inserting input modifiers with the good characteristics. In this case, the designer launches the graph analysis and accepts the proposition to modify the graph automatically. For each input that must be modified, BIAsT computes the

type of the modifier (simple delay, multi-state delay, decimator, ...) and its parameters. In this example, it determines that a delay of 6 clock cycles must be put on `data_in` as shown in Figure 5.12. A small square is added on the modified interface and input patterns are now equal because six 0s have been prepend to `data_in` pattern.

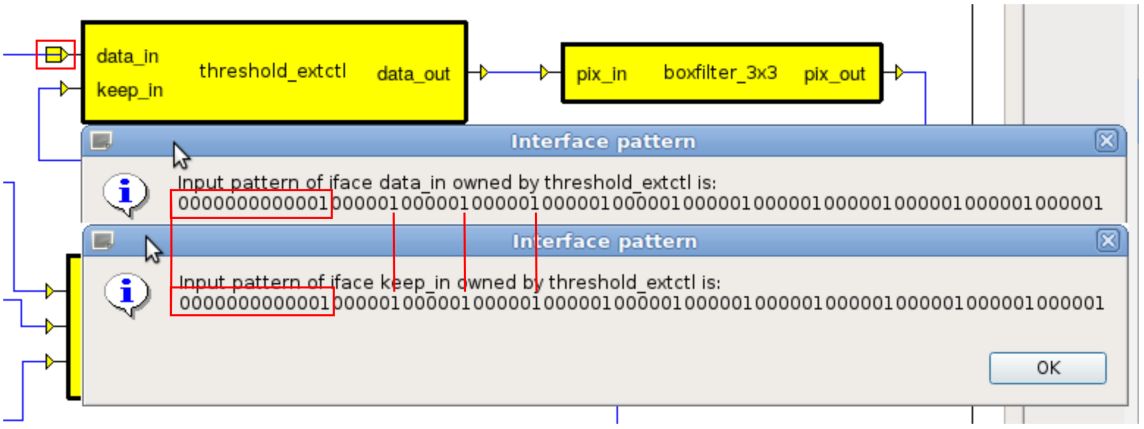

Figure 5.12: Solving an incompatible case.

5.4/ CONCLUSION AND PERSPECTIVES

BIAsT constitutes an ambitious project to provide an open-source environment to create FPGA designs without a great expertise of these architectures. It is intended to any researcher or engineer that would normally not use (or think to) an FPGA for his applications because of the complexity to use them. Moreover, it is also intended to skilled experts that want to speed-up the design process and to avoid a lot of benchmarks.

Because of the scale of this project, BIAsT is only at the beginning of its development and cannot be compared with professional software tools in terms of ergonomics and functionalities achievement. Firstly, only a few blocks are presently integrated in the block library. Secondly, a lot of things must be improved to prepare the VHDL generation. For example, setting parameter values for all blocks is quite fastidious and should be summarized in a single dialog instead of processing a block after another. For interface sizes, an automatic inheritance between connected blocks could be implemented. Thirdly, BIAsT lacks the support in simulation management. For instance, it could propose classic types of data generators (constant value, incremental, pulse, ...) to be used as source blocks. The benchmark files could be generated automatically, using the VHDL code of these generators to feed the design to simulate. BIAsT could also compile benchmark and design code before calling a simulator like ISim. Finally, BIAsT does not manage the type of target architecture and its particular logic resources. It allows to forbid the usage of some blocks in the design to select implementations of blocks that are adapted to available resources, and to create more easily “ucf” files to retrieve a bitstream.

Even if these functionalities are necessary to bring BIAsT at a level of professional criteria of quality, they have no interest from the research point of view and just need an expertise in C++/Qt development. But BIAsT relies on the ASAP concepts, so it is actually limited by these concepts. For example, there are still problems to be addressed

with the automatic modifications of the graph. There are also assumptions (implicit or not) in the model and its algorithms that limit feasible designs. For example, it is not possible to analyze graphs with cycles because of the principles of graph traversal and output pattern computation. It is also impossible to use blocks with a behavior that depends on the input values. Thus, it is necessary to study how to overcome these limitations so that the development of BlAsT carries on with feeding research works.

III

Conclusion and Perspectives

6

Conclusion and Perspectives

6.1/ Conclusion

In this dissertation, we concentrate on model based design for FPGA. We have analyzed existing models and tools and addressed their inherent problems and shortcomings. We proposed a model called Actors with Stretchable Access Patterns (ASAP) to solve these problems. It constitutes the base of framework to complete a static analysis of designs. This framework is integrated in an EDA software called BlAsT (Block Assembly Tool) that can be used by non-experts to make FPGA designs.

The first part of the dissertation presents the scientific background. Chapter 1 gives a general introduction of the subject, with the main motivations and objectives. Chapter 2 starts with a brief survey about literature on FPGA concepts to present the general context. Then, it describes existing models and tools that can help a designer in the tough process of producing and analyzing an FPGA design. A comparison of different models is presented with some examples. The advantages and disadvantages of the widely used models, such as SDF, CSDF and SDF-AP, are pointed out, together with their abilities to match more or less the behavior of a real hardware design. EDA tools that are based on these models inherit from their problems. In fact, even if a lot of tools can generate VHDL code automatically, the result is seldom applicable on a real FPGA. Therefore, our research aims to overcome these drawbacks in order to ease the creation of FPGA designs.

The contributions of our research are stated in the second part of the dissertation. A novel model ASAP and related approaches for system analysis are proposed. In addition, the software tool BlAsT is introduced.

In Chapter 3, we analyze the limitations of the SDF-AP model yield by the auto-concurrency property, the fact that patterns must be strictly matched, and the mandatory buffering. Based on this fact, the principles of ASAP model are given with the definitions and assumptions to overcome the shortcomings of existing models. The basic algorithms for patterns and schedules transformation and generation are also provided. The effectiveness of the ASAP model is confirmed by comparing the patterns generated according to the principles and the signals of real FIRs generated by CoreGen.

Chapter 4 describes the strategies to analyze a graph of actors based on the ASAP model. In order to ensure the compliance of input and consumption, the sample rates of the graph, the ratios of consumption rate on each port and the pattern compatibility are checked. If the tested graph is found to be inconsistent, decimations should be used

to resample the production rates. Pattern modifications are provided for the incompatible patterns so that the graph can produce correct results. The experimental results show that the ASAP model solves the limitations of concurrent executions and buffering problems. It also overwhelms other models in terms of resources saving without any impact on the global latency.

In Chapter 5, the newly developed EDA tool BlAsT is introduced. It integrates the concept of functional block assembly and the ASAP model. BlAsT is aimed at providing an open-source environment to create FPGA designs both for researchers and engineers, even for users without a great expertise of these architectures. The expected functions based on the principles of the ASAP model and VHDL generation are integrated in the current version of the BlAsT. The example cases shows the related functions in a direct way. Moreover, the ease of transposition of the model into VHDL code are verified.

6.2/ PERSPECTIVES

The purpose of our research is to provide an applicable solution to help the FPGA designers to develop implementations. We focus on the characteristics of FPGA design and the limitations of existing system analysis models. We proposed a novel model to describe the behaviors of functional blocks and develop an original version of the EDA tool BlAsT based on our theoretical approaches. Although we reached the goal we set for this dissertation, it is still a small part of the project. Many progresses can be done in many aspects in the future.

The proposed ASAP model can reach the goal to make a faithful description of the hardware behavior. The basic algorithms are provided for operations such as pattern generation, compatibility checking and pattern modification, but they still need to be improved. Firstly, the algorithms can fulfill the expected functions, but some of them need to be improved in efficiency. For instance, the pattern modifications should be more flexible. Secondly, some assumptions prevent the ASAP model to be applied to more practical cases, so further studies are required. The first track is to explore the case where admittance patterns contain null columns. The second track is more complex since it concerns cycles in the graph. In some practical designs, there are actors with a direct feedback (i.e. output connected to an input), and even cycles encompassing several actors. The last track is maybe the most complex and addresses the problem of an actor's behavior that depends on the input values. In this case, we need to modify the basic principles of the proposed model or even to develop a new one.

Finally, some changes should be made in BlAsT to fully employ the proposed approaches. As talked in the description of BlAsT, some of the algorithms have not been implemented. There is also a lot of development work to expand the block library. It will be essential for the VHDL generation. And if the research tracks mentioned above start to bear fruits, some drastic modifications will certainly occur. Since the tool is intended to be an open source software available online, we also hope that user's feedback will help to improve it or even that its development will be shared.

In our research, we have crossed all the range of the problem to create FPGA designs. We started from defining a novel theoretical model and came with the development of an EDA tool. Nevertheless, it is not the end but just the beginning of a long path and we are confident in the possibility of going further.

PUBLICATIONS

JOURNAL ARTICLES

[**1**] Ke Du, Stéphane Domas and Michel Lenczner. Actors with Stretchable Access Patterns. *Integration, the VLSI Journal, Elsevier, 2018, (Accepted, Major revision)*.

CONFERENCE PAPERS

[**1**] Ke Du, Stéphane Domas, and Michel Lenczner. Techniques for System Analysis Based on ASAP Model. In: ACM SIGMETRICS 2018, the International Conference on Measurement and Modeling of Computer Systems, ACM, June 2018 (Submitted).

[**2**] Ke Du, Jinlong Liu, Xingrui Zhang, Jianying Feng, Yudong Guan, and Stéphane Domas. A Graph-based Algorithm on Semi-supervised Image Classification. In ICCS 2018, the 18th International Conference on Computational Science, Lecture Notes on Computer Science (LNCS), Springer, June 2018 (Accepted).

[**3**] Ke Du, Stéphane Domas, and Michel Lenczner. A Solution to Overcome some Limitations of SDF Based Models. In ICIT 2018, the 19th IEEE International Conference on Industrial Technology, IEEE, February 2018.

[**4**] Ke Du, Stéphane Domas, Mengdie Wu, and Michel Lenczner. A Hole-Filling Framework Based on DIBR and Improved Criminisi's Algorithm for 3D Video. In ICCBDC 2017, the International Conference on Cloud and Big Data Computing, ACM, pages 119-124, September 2017.

[**5**] Ke Du, Stéphane Domas, Michel Lenczner, and Guangjin Zhang. An Improved Algorithm Based on SURF for MR Infant Brain Image Registration. In ICIC 2016, the 12th International Conference on Intelligent Computation, Intelligent Computing Theories and Application, Lecture Notes on Computer Science (LNCS), Springer, pages 458–470, August 2016.

[**6**] Y. D. Guan, R. F. Zhu, J. Y. Feng, K. Du, and X. R. Zhang. Research on Algorithm of Human Gait Recognition Based on Sparse Representation. In IMCCC 2016, the 6th International Conference on Instrumentation & Measurement, Computer, Communication and Control, IEEE, pages 405-410, July 2016.

[**7**] M. Lenczner, B. Yang, S. Cogan, S. Domas, D. Ke, R. Couturier, D. Renault, B. Koehler, and P. Janus. Temperature control of an SThM micro-probe with a heat source estimator and a lock-in measurement. In EuroSimE 2016, The 17th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems, IEEE, pages 1–8, April 2016.

[**8**] M. Lenczner, B. Yang, R. Couturier, S. Domas, K. Du, S. Cogan, P. Janus, and A. Bontempi. Two-scale modeling and model-based control law of temperature in an SThM probe. In Eurotherm Seminar No 109, Numerical Heat Transfer(NHT), September 2015.

BIBLIOGRAPHY

[1] Open Cores for FPGA and ASIC Development. http://www.opencores.org.

[2] Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, and Thierry Michel. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pages 11–20. IEEE, 2010.

[3] Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, and Pascal Urard. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *ACS/IEEE International Conference on Computer Systems and Applications-AICCSA 2010*, pages 1–8. IEEE, 2010.

[4] Albert Benveniste and Paul Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE transactions on Automatic Control*, 35(5):535–546, 1990.

[5] Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, 1992.

[6] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.

[7] SS Bhattacharyya, PK Murthy, and EA Lee. Software synthesis from dataflow graphs, volume 360 of the kluwer international series in engineering and computer science, 1996.

[8] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[9] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258. IEEE, 1995.

[10] Alessio Bonfietti, Luca Benini, Michele Lombardi, and Michela Milano. An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 897–902. IEEE, 2010.

[11] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994.

[12] Joseph Tobin Buck and Edward A Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432. IEEE, 1993.

[13] BM Cook et al. Legacy of the transputer. In *in BM Cook (editor), Architectures, Languages and Techniques, IOS*. Citeseer, 1999.

[14] Martyn Edwards and Peter Green. The implementation of synchronous dataflow graphs using reconfigurable hardware. In *International Workshop on Field Programmable Logic and Applications*, pages 739–748. Springer, 2000.

[15] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[16] Marc Engels, Greet Bilson, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow: model and implementation. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 503–507. IEEE, 1994.

[17] Paul Feautrier. Fine-grain scheduling under resource constraints. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 1994.

[18] Dirk Fimmel and Jan Müller. Optimal software pipelining under resource constraints. *International Journal of Foundations of Computer Science*, 12(06):697–718, 2001.

[19] Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens, Santiago González Pestana, and Edwin Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the æthereal network on chip. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 1–36. Springer, 2005.

[20] Kang Su Gatlin. Trials and tribulations of debugging concurrency. *Queue*, 2(7):66–73, 2004.

[21] Marc Geilen and Twan Basten. Requirements on the execution of kahn process networks. In *European Symposium on Programming*, pages 319–334. Springer, 2003.

[22] Marc Geilen and Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 137–146. ACM, 2004.

[23] Marc Geilen, Stavros Tripakis, and Maarten Wiggers. The earlier the better: A theory of timed actor interfaces. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 23–32. ACM, 2011.

[24] Amir Hossein Ghamarian, MCW Geilen, Twan Basten, and Sander Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 116–121. IEEE, 2008.

[25] Amir Hossein Ghamarian, MCW Geilen, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, and Sander Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 68–75. IEEE, 2006.

[26] Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, AJM Moonen, and MJG Bekooij. Throughput analysis of synchronous data flow graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36. IEEE, 2006.

[27] Amir Hossein Ghamarian, Sander Stuijk, Twan Basten, MCW Geilen, and Bart D Theelen. Latency minimization for synchronous data flow graphs. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196. IEEE, 2007.

[28] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, et al. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):3, 2009.

[29] Arkadeb Ghosal, Rhishikesh Limaye, Kaushik Ravindran, Stavros Tripakis, Ankita Prasad, Guoqiang Wang, Trung N Tran, and Hugo Andrade. Static dataflow with access patterns: semantics and analysis. In *Proceedings of the 49th Annual Design Automation Conference*, pages 656–663. ACM, 2012.

[30] Arthur Gill. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, New York, 1962.

[31] Alain Girault, Bilung Lee, and Edward A Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 18(6):742–760, 1999.

[32] Gwenhael Goavec-Merou. *Générateur de coprocesseur pour le traitement de données en flux (vidéo ou similaire) sur FPGA.* PhD thesis, 2014. Thèse de doctorat dirigée par Lenczner, Michel et Couturier, Raphaël Sciences pour l'ingénieur Besançon 2014.

[33] Steve Goddard and Kevin Jeffay. The synthesis of real-time systems from processing graphs. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 177–186. IEEE, 2000.

[34] Richard Goering. Multicore design strives for balance... but programming, debug tools complicate adoption. *Electronics Engineering Times*, 2006.

[35] Ramaswamy Govindarajan, Guang R Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI signal processing systems for signal, image and video technology*, 31(3):207–229, 2002.

[36] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Multiprocessor soc software design flows. *IEEE Signal Processing Magazine*, 26(6), 2009.

[37] Jens Horstmannshoff and Heinrich Meyr. Optimized system synthesis of complex rt level building blocks from multirate dataflow graphs. In *Proceedings of the 12th international symposium on System synthesis*, pages 38–43. IEEE Computer Society, 1999.

[38] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S Bhattacharyya. Dif: An interchange format for dataflow-based design tools. In *International Workshop on Embedded Computer Systems*, pages 423–432. Springer, 2004.

[39] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 37–49. ACM, 2005.

[40] Xilinx Inc. Xilinx Core Generator. Xilinx Inc., ISE Design Suite 12.1 edition, 2010.

[41] Accellera Systems Initiative. SPIRIT 1.5. http://www.spiritconsortium.org.

[42] National instruments Corp. LabVIEW FPGA. http://www.ni.com/fpga.

[43] Jörn W Janneck, Ian D Miller, David B Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, 2009.

[44] Ahmed Jerraya and Wayne Wolf. *Multiprocessor systems-on-chips*. Elsevier, 2004.

[45] Hyunuk Jung, Hoeseok Yang, and Soonhoi Ha. Optimized rtl code generation from coarse-grain dataflow specification for fast hw/sw cosynthesis. *Journal of Signal Processing Systems*, 52(1):13–34, 2008.

[46] G Kahn. The semantics of a simple language for parallel programming,"information processing'74: Proceedings of the ifip congress," 471–475, 1974.

[47] Lina Karam, Ismail AlKamal, Alan Gatherer, Gene A Frantz, David V Anderson, and Brian L Evans. Trends in multicore dsp platforms. *IEEE Signal Processing Magazine*, 26(6), 2009.

[48] Hojin Kee, Shuvra S Bhattacharyya, and Jacob Kornerup. Efficient static buffering to guarantee throughput-optimal fpga implementation of synchronous dataflow graphs. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 136–143. IEEE, 2010.

[49] Rudy Lauwereins, Marc Engels, Marleen Adé, and JA Peperstraete. Grape-ii: A system-level prototyping environment for dsp applications. *Computer*, 28(2):35–43, 1995.

[50] Paul Le Guernic and Thierry Gautier. *Data-flow to von Neumann: the SIGNAL approach*. PhD thesis, INRIA, 1990.

[51] Edward A Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed systems*, 2(2):223–235, 1991.

[52] Edward A Lee. Finite state machines and modal models in ptolemy ii. Technical report, DTIC Document, 2009.

[53] Edward A Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time dsp. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1279–1283. IEEE, 1989.

[54] Edward A Lee and II John. Overview of the ptolemy project, 1999.

[55] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[56] Edward A Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. *IEE Proceedings-Computers and Digital Techniques*, 152(2):239–250, 2005.

[57] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[58] Edward A Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123. ACM, 2007.

[59] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[60] Man-Kit Leung, Thomas Mandl, Edward A Lee, Elizabeth Latronico, Charles Shelton, Stavros Tripakis, and Ben Lickly. Scalable semantic annotation using lattice-based ontologies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 393–407. Springer, 2009.

[61] Weichen Liu, Mingxuan Yuan, Xiuqiang He, Zonghua Gu, and Xue Liu. Efficient sat-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *Real-Time Systems Symposium, 2008*, pages 492–504. IEEE, 2008.

[62] Grant Martin. Esl requirements for configurable processor-based embedded system design. *IP-SoC 2005*, pages 15–20, 2005.

[63] Grant Martin. Overview of the mpsoc design challenge. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 274–279. IEEE, 2006.

[64] Inc. MathWorks. HDL Coder. https://www.mathworks.com/products/hdl-coder.html.

[65] Inc. MathWorks. Simulink. https://www.mathworks.com/products/simulink.html.

[66] Marco Mattavelli, Shuvra S Bhattacharyya, Johan Eker, Carl von Platen, Gordon Brebner, Jorn W Janneck, and Mickael Raulet. Opendf–a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News, Special Issue: MCC08–Multicore Computing 2008*, 36(GR-LSM-ARTICLE-2010-001):29–35, 2008.

[67] Orlando Moreira, J-D Mol, Marco Bekooij, and Jef Van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 332–341. IEEE, 2005.

[68] Orlando M Moreira and Marco JG Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1):1–14, 2007.

[69] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[70] Thomas M Parks. *Bounded scheduling of process networks*. PhD thesis, University of California. Berkeley, California, 1995.

[71] Thomas M Parks, José Luis Pino, and Edward A Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210. IEEE, 1995.

[72] Andy D Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *International Journal of Embedded Systems*, 3(3):181–196, 2008.

[73] José Luis Pino, Shuvra S Bhattacharyya, and Edward A Lee. *A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs*. Electronics Research Laboratory, College of Engineering, University of California, 1995.

[74] José Luis Pino, Soonhoi Ha, Edward A Lee, and Joseph T Buck. Software synthesis for dsp using ptolemy. *Journal of VLSI signal processing systems for signal, image and video technology*, 9(1-2):7–21, 1995.

[75] Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 63–72. ACM, 2003.

[76] Kaushik Ravindran, Arkadeb Ghosal, Rhishikesh Limaye, Guoqiang Wang, Guang Yang, and Hugo Andrade. Analysis techniques for static dataflow models with access patterns. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–8. IEEE, 2012.

[77] Ghislain Roquier, Matthieu Wipliez, Mickaël Raulet, Jorn W Janneck, Ian D Miller, and David B Parlour. Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286. IEEE, 2008.

[78] Chris Rowen. *Engineering the complex SOC: fast, flexible design with configurable processors*. Pearson Education, 2008.

[79] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.

[80] Sun-Inn Shih. Code generation for vsp software tool in ptolemy. *MS Report, Plan II, ERL Technical Report UCB/ERL M*, 94, 1994.

[81] Sundararajan Sriram and Shuvra S Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2009.

[82] Sander Stuijk, Twan Basten, MCW Geilen, and Henk Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, pages 777–782. ACM, 2007.

[83] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf3: Sdf for free. In *ACSD*, volume 6, pages 276–278, 2006.

[84] Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[85] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411. IEEE, 2011.

[86] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings.*, pages 185–194. IEEE Computer Society, 2006.

[87] Bart D Theelen, MCW Geilen, Sander Stuijk, Stefan Valentin Gheorghita, Twan Basten, JPM Voeten, and AH Ghamarian. Scenario-aware dataflow. *TU Eindhoven, Tech. Rep. ESR-2008–08*, 2008.

[88] Stavros Tripakis, Hugo Andrade, Arkadeb Ghosal, Rhishikesh Limaye, Kaushik Ravindran, Guoqiang Wang, Guang Yang, Jacob Kormerup, and Ian Wong. Correct and non-defensive glue design using abstract models. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 59–68, 2011.

[89] Stavros Tripakis, Rhishikesh Limaye, Kaushik Ravindran, Guoqiang Wang, Hugo Andrade, and Arkadeb Ghosal. Tokens vs. signals: On conformance between formal models of dataflow and hardware. *Journal of Signal Processing Systems*, pages 1–21, 2015.

[90] Parishwad P Vaidyanathan. *Multirate systems and filter banks*. Pearson Education India, 1993.

[91] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.

[92] Carl von Platen and Johan Eker. Efficient realization of a cal video decoder on a mobile terminal (position paper). In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 176–181. IEEE, 2008.

[93] GQ Wang, Randy Allen, Hugo A Andrade, and A Sangiovanni-Vincentelli. Communication storage optimization for static dataflow with access patterns under periodic scheduling and throughput constraint. *Computers & Electrical Engineering*, 40(6):1858–1873, 2014.

[94] Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 10–15. ACM, 2006.

[95] Maarten H Wiggers, Marco JG Bekooij, Pierre G Jansen, and Gerard JM Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with backpressure. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 281–292. IEEE, 2007.

[96] Maarten H Wiggers, Marco JG Bekooij, and Gerard JM Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 658–663. IEEE, 2007.

[97] Maarten H Wiggers, Marco JG Bekooij, and Gerard JM Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 183–194. IEEE, 2008.

[98] Maarten H Wiggers, Marco JG Bekooij, and Gerard JM Smit. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):17, 2010.

[99] Michael C Williamson and Edward A Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *Signals, Systems and Computers, 1996. Conference Record of the Thirtieth Asilomar Conference on*, pages 1340–1343. IEEE, 1996.

[100] Haiyang Zheng and Edward A Lee. *Operational semantics of hybrid systems*. PhD thesis, University of California, Berkeley, 2007.

Document generated with LaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

Abstract:

In recent years, embedded systems has been widely used in both scientific environment and daily lives of common people. For some applications with strong real-time constraints, FPGA chips constitute a good choice. Their sizes and capacities are increasing continuously, allowing to build more and more complex applications. Thus, it is harder and harder to manage the application designs manually. This problem has been addressed through several ways. One is to use a model that is a more or less realistic abstraction of the behavior of the design. Nevertheless, it introduces another problem, which is the efficient implementation of the model on real architectures, like FPGAs. For example, some model characteristics may lead to a waste of resources, which can even make a design infeasible for a particular target architecture.
In this dissertation, we focus on overcoming some limitations yield by unfaithful descriptions of hardware behaviors for some existing models and the drawbacks of available tools. The Static/Synchronous Data Flow (SDF) based models, especially the version with Access Patterns (SDF-AP), are investigated. From the analysis of the problems of the existing models and EDA tools, our researches yield a new model: Actors with Stretchable Access Patterns (ASAP), and a new EDA tool called BlAsT (Block Assembly Tool). The model shares some basic principles of SDF-AP model but with other semantics and goals, which allows to model a wider range of behaviors and to obtain greater analysis capacities. Indeed, we propose a complete framework to check whether a design processes the input data streams correctly and if it is not the case, to modify the graph automatically to obtain this correctness. It is verified by experiments carried out on the realistic cases that clearly point out the advantages of ASAP over SDF-AP model, notably in terms of resources consumption. The BlAsT proposes a graphical interface to create designs by putting functional blocks on a panel and connecting them. It integrates the analysis principles defined by ASAP. It is also able to produce the VHDL code for the whole design. Thus, BlAsT offers the possibility for users without any knowledge in VHDL to create designs for FPGAs and with the insurance that it will produce correct results.

Keywords: Field Programmable Gate Arrays (FPGAs), Embedded Systems, System on Chips (SoC), Static Analysis and Scheduling, Synchronous Data Flow (SDF), Model Based Design, Electronic Design Automation (EDA).

Résumé :

Ces dernières années, les systèmes embarqués ont envahit tant les environnements scientifiques que la vie quotidienne. Pour les applications avec des fortes contraintes temps réel, les FPGA sont un choix pertinent. Leur taille et leurs capacités évoluent constamment, ce qui permet de créer des applications de plus en plus complexe. Cependant, cette augmentation va de pair avec une difficulté croissante à créer le design de ces application à la main. Ce problème a été abordé de diverses façons. L'un d'entre elles consiste à élaborer un modèle qui abstrait le comportement d'un design de façon plus ou moins réaliste. Cependant, cela conduit à un autre problème, qui est la transposition du modèle sur une architecture réelle telle qu'un FPGA. Par exemple, certaines caractéristiques du modèle peuvent entraîner un gâchis de ressources logiques, au point de rendre le design inapplicable sur certaines architectures.
Dans cette thèse, nous nous intéressons à comment dépasser les limitations de certains modèles en terme d'expressivité de comportement. Nous abordons également celles des outils d'aide au développement de designs. Les modèles basés sur les flux de données synchrones (SDF) et plus spécialement la version avec patrons d'accès (SDF-AP) ont été pris comme référence. A partir de l'étude des limitations de ces modèles, nous avons produit un nouveau modèle nommé Acteurs avec patrons d'accès extensibles (ASAP), ainsi qu'un nouvel environnement d'aide au développement nommé BlAsT. Ce modèle a des caractéristiques communes avec SDF-AP mais en leur donnant des nouvelles définitions afin d'élargir le nombre de comportements modélisés et les possibilité d'analyse du design. En effet, nous proposons un cadre d'analyse complet qui vérifie si le design traite correctement les flux de données entrants et si ce n'est pas le cas, qui fait automatiquement les modifications minimales pour assurer des résultats corrects. Ce cadre a été testé sur une application réelle qui montre clairement les avantages que procure notre modèle comparé à SDF-AP, notamment en terme de consommation de ressources logiques. Quant au logiciel BlAsT, il propose une interface graphique pour créer un design, simplement en posant des blocs fonctionnels sur un panneau et en les connectant. Il intègre les principes d'analyse tels que définis par ASAP. Enfin, il permet de générer automatique le code VHDL d'un design. En conclusion, il offre la possibilité de créer des designs FPGA sans aucun connaissance sur VHDL, tout en ayant l'assurance d'obtenir un code fonctionnel.

Mots-clés : Réseaux de portes logiques programmables (FPGAs), Systèmes embarqués, Système sur copeaux (SoC), Analyse et ordonnancement statique, Flux de données synchrones (SDF), Modèle basé sur la conception, Automatisation de la conception électronique (EDA).

SPIM

■ École doctorale SPIM - Université de Technologie Belfort-Montbéliard
F - 90010 Belfort Cedex ■ tél. +33 (0)3 84 58 31 39
■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

utbm
université de technologie
Belfort-Montbéliard