Converting Weakly Dynamic Programs to Equivalent Process Network Specifications

Todor Stefanov

Converting Weakly Dynamic Programs to Equivalent Process Network Specifications

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 14 December 2004
te klokke 15.15 uur

door

Todor Stefanov
geboren te Samokov, Bulgaria
in 1974

Samenstelling promotiecommissie:

promotor	Prof.dr.ir. Ed Deprettere	
co-promotor	Dr.ir. Bart Kienhuis	
referent	Prof.dr. Shuvra S. Bhattacharyya	(University of Maryland, USA)
overige leden:	Prof.dr. Stamatis Vassiliadis	(Technische Universiteit Delft)
	Prof.dr.-ing. Jürgen Teich	(Universität Erlangen-Nürnberg, Duitsland)
	Prof.dr.ir. Angel Popov	(Technical University of Sofia, Bulgaria)
	Prof.dr. Doug DeGroot	
	Prof.dr. S.M. Verduyn Lunel	
	Dr.ir. Erwin de Kock	(Philips Research, Eindhoven)
	Dr. Andy Pimentel	(Universiteit van Amsterdam)

The work in this thesis was carried out in the Artemis project supported by PROGRESS/STW.

Converting Weakly Dynamic Programs to Equivalent Process Network Specifications
Todor Plamenov Stefanov. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN 90-9018629-8

Printed in the Netherlands

To my wife Stefka, and my lovely daughter Eva

Contents

Acknowledgments

The research presented in this dissertation has been supported by PROGRESS, the embedded systems and software research program of the Dutch Technology Foundation STW, under the project ARTEMIS (Project number AES 5021).

This dissertation is the result of work conducted at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University in co-operation with researchers from Delft University of Technology, University of Amsterdam, and Philips Research. I would like to thank all the people who guided and supported me at these various places.

I would like to thank all the people with whom I worked in the context of the ARTEMIS project: Georgi Kuzmanov and Stamatis Vassiliadis from TU-Delft; Andy Pimentel, Simon Polstra, Cagkan Erbas, Joe Coffland, Barry van Halderen, and Frank Terpstra from University of Amsterdam; Paul Lieverse, Pieter van der Wolf, and Erwin de Kock from Philips Research. Many thanks to all of you for the successful co-operation and interesting scientific and non-scientific discussions we had during the course of the project.

I would like to thank the following fellow Ph.D. students from LIACS at Leiden University with most of whom I have shared room 122: Alexandru Turjan, Claudiu Zissulescu, Vladimir Zivkovic, Laurentiu Nicolae, Sylvain Alliot, Ioan Cimpian, Hristo Nikolov, Dmitry Cheresiz, and Mihai Cristea. You have made my four years at Leiden University a very pleasant experience. I always will remember the long and very interesting discussions we had about our research work and social life in The Netherlands as well as the social events we have organized together in order to relax. I also want to thank the secretaries of the LERC group at LIACS for helping me and my wife to settle in The Netherlands and for their excellent administrative assistance.

Several people with whom I worked in Bulgaria have contributed for building my knowledge and experience at a level which gave me the confidence to start a Ph.D. research. I would like to thank them for this. In particular, I want to thank Angel Popov and Peter Manoilov from TU-Sofia for encouraging me to do a Ph.D. research. With them, as my mentors, I made my first steps in the research field as a Master student and I wrote and published my first research paper. Also, I want to thank all my former colleagues at Innovative MicroSystems

Ltd. (currently Fabless Ltd.) for the engineering experience they shared with me when I was working there. This experience helped me a lot during my Ph.D. research.

Finally, I would like to thank my family and my close relatives and friends for supporting me in my Ph.D. Especially, I want to express my greatest gratitude to my wife Stefka for here love and patience. She always helped me when I had difficult moments in the past four years by constantly showing interest in my work and by showing understanding when again I had to come late at home.

Todor Stefanov
Leiden, September 30, 2004

Chapter 1

Introduction

In the recent years as well as in the future, Embedded Systems-on-Chip (SoC) can/will be found in many small, mobile, and ergonomic devices that provide information, entertainment, and communication capabilities to consumer electronics, industrial automation, retail automation, and medical markets. These Embedded SoCs require complex electronic design and system integration delivered in short time frames because of the time-to-market pressure. Currently, we are experiencing major complexity problems in Embedded Systems-on-Chip design because today's design approaches follow the traditional path of low-level design, simulation, and prototyping using the well known tools provided by electronic design automation (EDA) companies like Cadance and Synopsis. Prototyping may lead to high non-recurrent engineering costs that cannot be recovered from expected product sales. Moreover, there is a tendency that the users are getting more and more - and faster - very demanding in terms of applications diversity, complexity, and services that the Embedded Systems-on-Chip have to support.

Existing design methodologies and tools can no longer keep up with this trend because they cannot deal with such complex and highly flexible systems. These design methodologies and tools have been conceived and delivered in the past when applications were survivable and long-lasting, and the implementations - whether in programmable processors or in dedicated hardware - were not too complex. All this is changing rapidly: transistor densities grow exponentially whereas the traditional processors are not scalable, and applications grow more complex whereas the way in which they are specified is not at all impartial to traditional implementations. We believe that there is a way to overcome these problems though, be it that it implies completely new design methodology concepts and approaches:

- First, the on-chip architectures have to become heterogeneous networked multiprocessors. The current state-of-the-art is a first step in that direction: a CPU with one or more co-processors. Next will come a set of autonomous processing units that interact asynchronously over some sort of interconnection network. Then will come the chip containing mainly (possibly identical) copies of this architecture all embedded in what

has become known as a Network-on-Chip (NoC) [1].

- Second, the sequential imperative languages widely used to specify applications will no longer match the increasing amount of parallelism that will be enforced by the emerging architectures. There is thus a need for a parallel language and/or a translator to take sequential specifications to parallel specifications.

- Third, to reduce non-recurrent engineering costs and time-to-market delays, architectures have to be flexible enough that a variety of related applications can be mapped onto them. The implication of this constraint is that these architectures have to possess highly desirable properties, among which are *re-usability* (IPs), *separation of concerns* (computation vs. communication), *standardization* (interfaces), and last but not least *scalability*. The architectures will have to be instances of architecture templates (parameterized architectures, that is), which in turn will have to be versions of platforms. This has become know as *Platform-based Design* [2]. A platform is application-domain specific and has to be defined through domain analysis. Roughly speaking, a platform consists of two parts: One that concerns processing elements, and one that encompasses a communication and storage infrastructure. This partitioning is compliant with the computation vs. communication separation of concerns rule [3]. The processing elements are taken from a library - often as intellectual property (IP) components - and the communication and storage infrastructure is obeying certain predefined construction and operation rules. Specifying a platform is presently still more an art than a science.

- Fourth, System-level design methodologies have to be developed to master the complexities of the emerging applications and platforms. *System-level Design* is a radically new concept that is challenging many researchers all over the world [2–7].

The combination of *System-level Design* and *Platform-based Design* is a promising new approach to master the ever growing complexity of Embedded Systems-on-Chip. Although it is still not clear how this is to be materialized, we believe that the following is agreed upon in the system design community:

- **element 1:** *Applications have to be specified in some parallel language and modeled at a high level of abstraction*. Currently, applications are specified using sequential programming languages like C or Matlab. The lack of appropriate methodology and tool support for extracting and modeling of concurrency in its various forms is an essential limiting factor in commonly used programming languages to express design complexity and to exploit parallelism available in applications.

- **element 2:** *Architectures have to be specified in a parameterized form and modeled at a high level of abstraction*. Today, designers are familiar with working at levels of abstraction that are too close to implementation. So, sharing design components and verifying designs before prototypes are built is nearly impossible. For most designers the highest level of abstraction of their design (architecture) is the register transfer level (RTL). The RTL level is clearly too low for complex architecture design.

- **element 3:** *Methods have to be provided to map application models onto architecture models*. This includes techniques and tools to explore alternative mappings at a high level of abstraction in order to find optimal system solutions in terms of system performance and cost in relatively short amount of time. Also, techniques and tools have to be developed to gradually refine the optimal mappings to implementations.

The three main **elements** of the emerging *System-level Platform-based Design* approach, presented above, are closely related and equally important. Each and every element has its own specific problems that have to be solved. The problems further discussed in this dissertation and the solutions we propose are related to **element 1**. The dissertation focuses on methods, techniques, and tools to derive a set of parallel (concurrent) [1] specifications for an application (i.e. alternative application instances) in order to allow exploration and transparent and systematic mapping of these specifications onto heterogeneous multiprocessor architectures.

This chapter is further organized as follows. In Section 1.1, we provide the motivation behind this work by stating the actual problem we want to solve. A high-level sketch of the approach and the techniques we have developed to solve this problem is given in Section 1.2. Section 1.3 gives a brief overview of related work and Section 1.4 summarizes the main contributions of this dissertation. Section 1.5 describes the organization of this dissertation.

1.1 Problem Statement

The emerging Embedded Systems-on-Chip platforms are increasingly becoming *heterogeneous* multi-processor architectures. An example of a heterogeneous architecture is shown in the bottom part of Figure 1.1. This architecture is composed of fully programmable components (CPUs), reconfigurable components (RPUs), and dedicated hardware blocks (IP cores). Typically, these components are linked via some kind of communication structure, e.g., high-speed bus, multiple buses, or programmable network. To satisfy the performance needs of applications, these emerging platforms must be programmed in such a way that all the components that comprise the multi-processor architecture execute as concurrently as possible. This implies that the *task-level* parallelism available in an application must be revealed and exploited efficiently.

We believe that programming multi-processor architectures efficiently is a key challenge in the emerging system-level and platform-based design methodologies. System designers experience significant difficulties because the way an application is specified by the application developer does not match the way multi-processor architectures operate. We observe today that most of the applications are typically specified by application developers as sequential programs using imperative programming languages like C/C++ or Matlab - see Figure 1.1. Specifying an application as a sequential program is relatively easy and convenient for application developers because they have been doing this for years and they understand very well the imperative model of computation. Moreover, there exist very mature tools for building, testing, and debugging applications specified as sequential programs.

[1] In this dissertation we use the terms parallelism and concurrency interchangeably. However, we are aware that there is a slight difference.

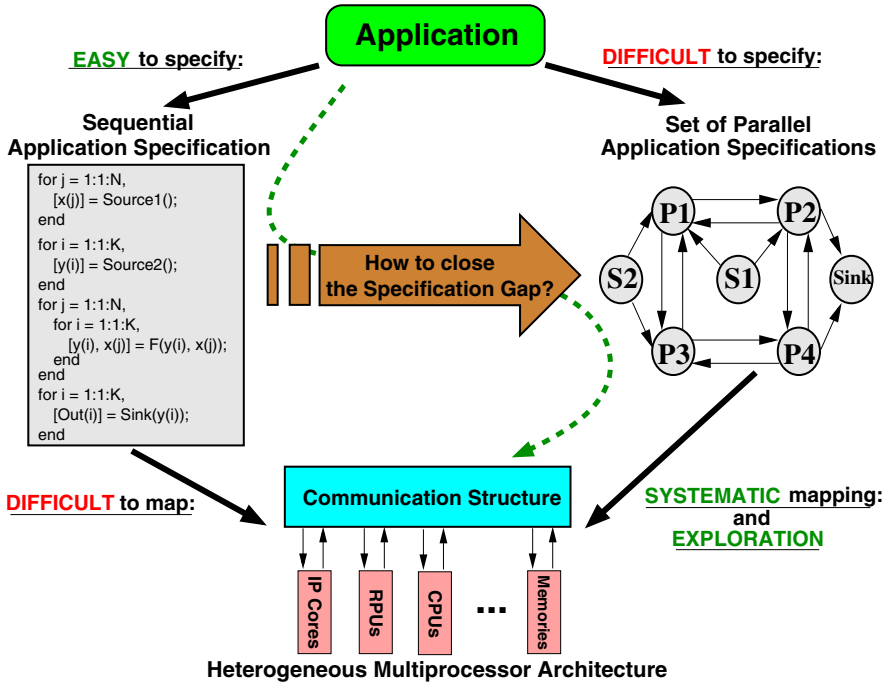

Figure 1.1: The main problem.

Although specifying an application as a sequential program is convenient and relatively easy, such specification does not reveal parallelism due to its inherent sequential nature. This fact makes the mapping of an application onto a parallel multi-processor architecture very difficult. Let us consider as a simple example the sequential program and the multi-processor architecture shown in Figure 1.1. In the program, the order of execution of the function calls is sequential, i.e., we have a single thread of control. Also, the function calls communicate data via shared variables located in a single global memory. On the other hand, the multi-processor architecture has components that run concurrently, i.e, *the control is distributed* over the components, and the architecture has several memory banks, i.e., *distributed memory*. So, the single memory and the single thread of control in the sequential program used for application specification are contradictory to the need of distributed control and memory for the architecture. Precisely this contradiction makes the mapping of sequential application specification onto multi-processor architecture very difficult and in most cases inefficient.

Instead, we believe that a much more appropriate way of specifying an application is to use a parallel model of computation (MoC). If an application is specified using a parallel MoC then the mapping of this application can be done in a systematic and transparent way using a disciplined approach [8]. An example of a parallel application specification is shown in the right part of Figure 1.1. This specification consists of several concurrent tasks making the *task-level* parallelism available in the application explicit. Also, the data dependencies and the communication between the tasks is explicit via distributed memory buffers. So,

these properties of the parallel model of computation match very well the need of distributed control and distributed memory in order to map an application onto a parallel multi-processor architecture in a systematic and efficient way.

Although a parallel model of computation is very suitable for multi-processor architecture mapping, specifying an application using a parallel MoC is difficult, not well understood by application developers, and a time consuming process. The application developers have to study an application in order to identify possible *task-level* parallelism that is available and to reveal it. Moreover, testing and debugging of a parallel application specification is notoriously difficult because several threads of control, one for every concurrent task, have to be considered and synchronized properly.

The facts discussed above and visualized in Figure 1.1 reveal that there is a gap between the way applications are currently specified (as sequential programs) and the way they should be specified (using a parallel model of computation) when a parallel multi-processor system has to be designed. This gap we call `Specification Gap`. Indeed, on the one hand application developers still prefer to specify an application as a sequential program using imperative programming languages because it is easy. On the other hand, system designers need an application specified using a parallel model of computation because this model is suitable for a systematic mapping of an application onto a parallel multi-processor architecture. Therefore, the `Specification Gap` has to be closed when designing a parallel multi-processor system by migrating from a sequential application specification (given by an application developer) to a parallel application specification (needed by a system designer). A challenging problem is *How to close the Specification Gap in a systematic and automated way*. By solving this problem, the following three issues have to be addressed:

- **issue1:** What should the parallel model of computation (MoC) be for parallel application specification? Many parallel MoCs exist [9], each of them with its own specific characteristics. The choice of a parallel MoC depends on the targeted application domain and architecture. There is not a universal parallel MoC which fits nicely to whatever application domain and architecture. Therefore, a parallel MoC is selected after detailed analysis of the application domain under consideration and after the architecture is selected and characterized;
- **issue2:** After selecting a parallel MoC for parallel application specification, a systematic approach is needed that allows automatic derivation of *task-level* parallel application specification from a sequential application specification using the selected MoC. A systematic approach will lead to a correct-by-construction derivation process of a parallel specification that can be easily automated. The automation will reduce significantly the design time of a system as well as possible errors in deriving parallel specifications will be eliminated;
- **issue3:** Techniques which allow automatic derivation of a *set* of alternative parallel specifications for the same sequential application specification at *task-level* are needed to extend the systematic approach. It is very important to research and develop such techniques because many parallel specifications for an application exist that are functionally equivalent but the degree of exploited parallelism is different. A set of alternative parallel specifications for an application gives a system designer an opportunity

to select a parallel specification from the set that maps best onto the designed system architecture.

In this dissertation we give a particular solution of closing the `Specification Gap` by addressing the three issues above in a specific way. An overview of our solution is presented in the next section.

1.2 Solution Approach

In this section we give an overview of the solution approach and the techniques we have developed to close the `Specification Gap` described in Section 1.1. Figure 1.2 shows a high-level sketch of our approach.

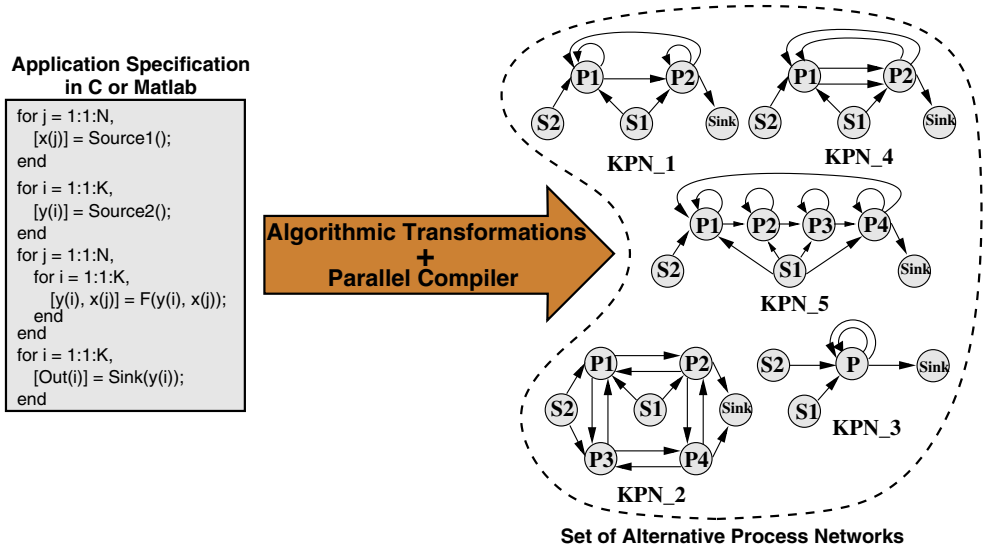

Figure 1.2: Our solution.

Our approach is based on *Algorithmic Transformation* techniques and *Parallel Compiler* techniques. These techniques form an application transformation layer. This layer derives, in a systematic and automated way, a set of alternative *task-level* parallel specifications called Kahn Process Networks (KPN) [10] for an application specified as a *weakly dynamic* sequential program in languages like C or Matlab. Our approach, depicted in Figure 1.2, addresses and solves the three **issues** described at the end of Section 1.1 in the following specific way:

- Our approach uses the Kahn Process Network (KPN) model of computation for parallel application specification. Our choice of this model is motivated in Section 1.2.1;

- We have developed a systematic and automated approach for deriving a KPN specification from an application specified as a weakly dynamic program. The term *weakly dynamic* program and the approach are explained in Section 1.2.2. Also, in this section we give an introduction to the main *Parallel Compiler* techniques that underlie this approach;

- We have developed *Algorithmic Transformation* techniques that in combination with the *Parallel Compiler* techniques allow automatic derivation of a ***set*** of KPN specifications from a weakly dynamic program. These KPN specifications are functionally equivalent to the input weakly dynamic program but the degree of exploited parallelism is different. We introduce the *Algorithmic Transformation* techniques in Section 1.2.3.

1.2.1 Kahn Process Network model of computation

As discussed in Section 1.1, the choice of parallel model of computation (MoC) to be used for parallel application specification depends on the targeted application domain. There is no universal parallel MoC which fits nicely to arbitrary application domains. The research described in this dissertation was performed in the context of the Artemis project [11]. The application domains targeted by this project were multimedia and signal processing applications such as JPEG codecs, MPEG codecs, Adaptive Digital Beam-forming, Smart Camera, Software Radio, etc. The main characteristic of these applications is that they are data-flow oriented applications, i.e., large streams of data have to be processed.

Our choice of parallel MoC is based on the application domains described above. Although, many parallel models of computation exist [9] [12], we have chosen the Kahn Process Network model of computation [10] [13] because its operational semantics are simple, yet general enough, to specify conveniently *stream-oriented* data processing that fits nicely with the application domain we are interested in - multimedia and signal processing applications. Moreover, for this application domain research work described in [14] [15] [16] [17] reinforces further the applicability of the Kahn Process Network (KPN) mode for specifying and mapping systematically and efficiently applications onto ”large-grain parallelism” multiprocessor architectures.

The KPN model of computation assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels (buffers), using a *blocking-read* synchronization primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes. The key characteristic of the KPN model is that it expresses an application in terms of *distributed control* and *distributed memory*. These key characteristics allow us to take advantage of the parallel resources available in multi-processor architectures. A KPN has the following favorable characteristics:

- The KPN model is *deterministic*, which means that irrespective of the schedule chosen to evaluate the network, the same input/output relation always exists. This gives a lot of scheduling freedom that can be exploited when mapping process networks onto multi-processor architectures;

- The processes in a KPN are *self-scheduling*. The inter-process synchronization is done by a blocking read. This is a very simple synchronization protocol that can be realized easily and efficiently in both hardware and software;
- The control in a KPN is completely distributed to the individual processes. Therefore, there is no global scheduler present. As a consequence, partitioning a KPN over multiple components is a simple task;
- The exchange of data among processes in a KPN is distributed over the FIFOs. There is no notion of a single global memory that has to be accessed by multiple processes. Therefore, resource contention does not occur.

1.2.2 Parallel Compiler Techniques (COMPAANDYN approach)

In this section we present our systematic approach to derive a KPN specification from an application specified as a *weakly dynamic program*. We call our approach COMPAANDYN. Also, we give a high-level introduction of the parallel compiler techniques integrated in COMPAANDYN as well as we explain the main novelties in the COMPAANDYN approach.

The problem of deriving a KPN specification for an application in a systematic and automated way has been addressed in the Compaan research work. The Compaan work presented in [18] [19] [20] [21] [22] [23] reports techniques for automatic derivation of Kahn Process Networks from applications specified as *static affine nested loop programs* (SANLP). Such programs are important in Scientific, Matrix Computation and Adaptive Signal Processing applications. The main property of such programs is that everything about the program execution is known at compile time. This property has been exploited extensively in Compaan to derive KPN specifications. A simple example of a SANLP is shown in Figure 1.3. This example shows that the bounds of loops and the outcome of conditions are known at compile time, i.e., the program execution is known.

```
1  for j = 1:1:4,
2    [x(j)] = F1(...);
3  end
4
5  for j = 1:1:4,
6    if j <= 2,
7      [x(j)] = F2( x(j) );
8   end
9   [...] = F3( x(j) );
10 end
```

Figure 1.3: Pseudo code of simple Static Affine Program.

```
1  for j = 1:1:4,
2    [ x(j)] = F1(...);
3  end
4
5  for j = 1:1:4,
6    if  x(j) <= 0,
7      [x(j)] = F2( x(j) );
8    end
9    [...] = F3( x(j) );
10 end
```

Figure 1.4: Pseudo code of simple Weakly Dynamic Program.

Many Scientific, Matrix Computation, and Adaptive Signal Processing applications can be specified as *static affine nested loop programs*, and the techniques developed in Compaan can be used to derive KPN specifications for such applications. However, many media applications such as JPEG codecs, MPEG codecs, Smart Cameras, Software Radio, etc. have dynamic (data-dependent) behavior which can not be expressed as static affine nested loop

programs. The Artemis project, the context in which our research has been performed, targets exactly such applications with dynamic (data-dependent) behavior. Therefore, we have developed the COMPAANDYN approach that supports automatic derivation of Kahn Process Networks from applications specified as *weakly dynamic programs*. COMPAANDYN is an *extension/generalization* of the Compaan research work mentioned above. As a consequence, the techniques in COMPAANDYN extend significantly the class of applications for which KPN specifications can be derived automatically. For example, COMPAANDYN can handle not only Scientific, Matrix Computation and Adaptive Signal Processing applications but also media applications with dynamic (data-dependent) behavior such as JPEG codecs, MPEG codecs, etc.

Weakly Dynamic Programs

The input of COMPAANDYN is an application specified as a weakly dynamic program (WDP). We define a WDP as a **task-level** sequential program where:

- the function calls in the program execute tasks.
- the control structures in the program are: 1) *for-loops* with upper and lower bounds as affine functions of iterators of other loops and parameters; 2) *if-then-else* constructs with **no restrictions on the condition** - the condition of the *if* can be an arbitrary function of loop iterators and/or data variables.
- the indexing of data variables (arrays) must be an affine function of *for-loop* iterators and possible parameters.

The weak dynamics in the above defined program come from the fact that the condition of *if-then-else* constructs can be an arbitrary function of data variables which values may be unknown at compile time, i.e., outcome of conditions in the program may be unknown at compile time. Notice that if we constrain the condition of *if-then-else* constructs to be an affine function of loop iterators and parameters then our WDP reduces to a static affine nested loop program.

As a simple illustrative, yet non-trivial example consider the WDP shown in Figure 1.4. This program consists of three function calls[2] $F1$, $F2$ and $F3$. These function calls execute tasks that communicate data via the array $x(j)$. Every element of the array can be anything from a scalar variable to a very complicated data structure. In this simple example we assume that every $x(j)$ is a scalar variable. The execution of $F2$ depends on the condition at line 6. This condition is data-dependent because it depends on the content of variable $x(j)$. The values of this variable are not known at compile time, i.e., they are not known before the actual execution of the program. This fact makes the program to have dynamic behavior, unpredictable at compile time. A challenging problem is how to analyze and transform this kind of dynamic programs at compile time in order to derive automatically executable Kahn Process Network specifications. Below, we present COMPAANDYN as a systematic solution approach to this problem.

[2] At some places in this dissertation we use the word *function* instead of *function call* for the sake of brevity.

The COMPAANDYN approach

Our COMPAANDYN approach is depicted in Figure 1.5. It consists of three main steps: 1) *Dependence Analysis*; 2) *Transformations*; 3) *Process Network (PN) Synthesis*. These steps are similar to the steps in the Compaan research work but our steps include more general models and techniques compared to Compaan in order to deal with weakly dynamic programs. The techniques involved in our three main steps are described in detail in Chapter 2. Below, we give only an introductory overview.

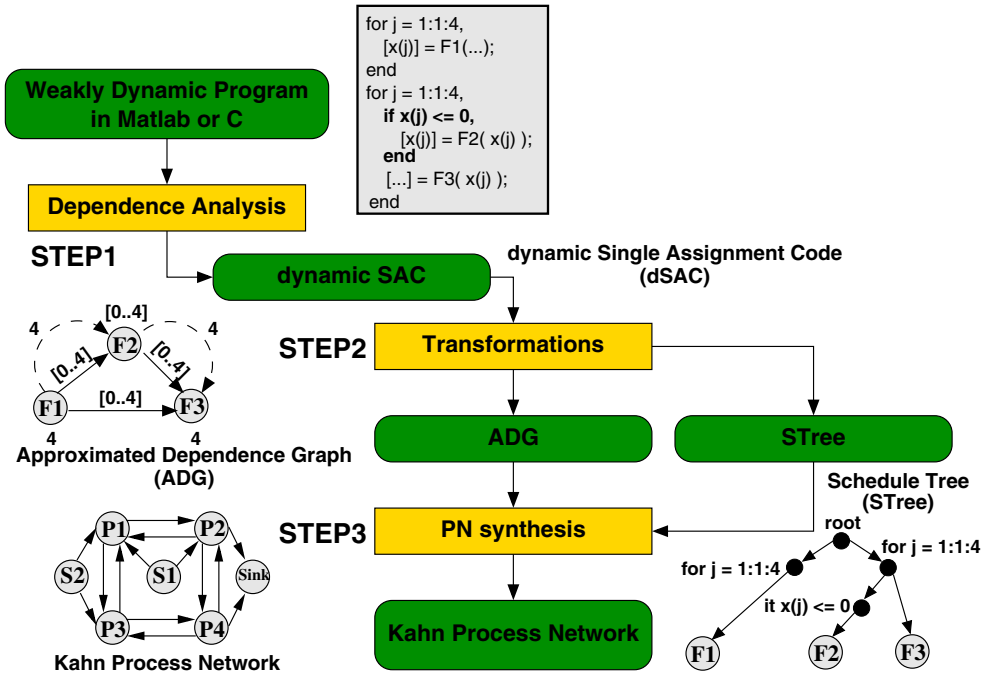

Figure 1.5: COMPAANDYN: a systematic approach to automatically derive an executable Kahn Process Network specification from a Weakly Dynamic Program.

In the COMPAANDYN approach we start with an application specified as a weakly dynamic program (WDP) in Matlab or C, and we convert this WDP into a *dynamic Single Assignment Code* (dSAC) representation by doing dependence analysis - STEP1 in Figure 1.5. The dSAC is a program that is functionally equivalent to the input WDP and has the property that: 1) every variable in the dSAC is written *at most* once because of the dynamics in the WDP; 2) for some variables, it is not known whether or not they will be written or read before the actual execution of the dSAC. Because of the property mentioned above, the dSAC reveals all possible data-dependencies between the functions in the WDP. Some of the data dependencies in the dSAC are not exactly defined, i.e., they depend on variables having values that are not known at compile time.

The dSAC generated in STEP1 is a very large and complex data structure to operate on.

Therefore, in the second step of our approach (STEP2) we transform the dSAC into a more compact representation that is a formal model. This model consists of two annotated graph structures, namely *Approximated Dependence Graph* (ADG) and *Schedule Tree* (STree). The ADG and the STree capture all the information that is present in the dSAC in a formal way. As a consequence, formal operations can be easily defined and applied on the ADG and the STree instead of the dSAC.

The ADG contains all the information that is related to the data dependencies between the functions in the dSAC. The data dependencies are approximated, i.e., the exact data dependencies are not known at compile time. The STree contains all the information about the execution order between the functions in the dSAC. The STree represents one valid schedule between all these functions that we call the global schedule. From the STree a local schedule between any arbitrary set of the functions in the dSAC can be obtained by pruning operations on the STree.

In the final step (STEP 3 in Figure 1.5) of our COMPAANDYN approach, a Kahn Process Network (KPN) is synthesized. A KPN consists of concurrent processes that communicate with each other over unbounded FIFO channels. Every process is specified as a sequential program. The synthesis of this program is based on information derived from the ADG and the STree. The synchronization between the processes is accomplished by blocking reads. By default, the computational and communicational workloads are distributed over the processes and the channels in accordance with the following partitioning rule: 1) for every node in the ADG a process is generated; 2) for every edge in the ADG a channel is generated. The workloads distribution can be changed by applying some techniques that will be introduced in Section 1.2.3. The process network synthesis in STEP3 is not limited to the generation of Kahn Process Networks only. With small modifications other process networks that have different inter-process communication and synchronization mechanisms can be generated.

As said before, our COMPAANDYN approach is an extension/generalization of the previous Compaan work. Below, we explain the main novelties in COMPAANDYN and we give simple examples (where necessary) for the sake of clarity.

Novelties in the COMPAANDYN approach

The COMPAANDYN approach deals with weakly dynamic programs (WDP). The definition of a WDP given earlier in this section shows that WDPs are more general class of programs than the static affine nested loop programs (SANLP) targeted by the Compaan work. Actually, this definition suggests that a SANLP is a special case of a WDP. Therefore, the COMPAANDYN approach relies on the techniques of Compaan when dealing with this special case of WDPs. For other cases of WDPs we have developed and integrated in COMPAANDYN techniques that extend or generalize the techniques used in Compaan. An overview of the novel techniques in the COMPAANDYN approach follows:

1) Advanced Dependence Analysis:

In order to derive a dSAC from a WDP (STEP1 in Figure 1.5) we have to find all possible data dependencies between the functions in the WDP. A well known and widely used dependence analysis technique is the exact array dataflow analysis (EADA) [24] [25] [26]. However,

EADA can not be performed to find data dependencies in WDPs because of dynamic (data-dependent) control structures that can be present in WDPs. Why dynamic (data-dependent) control structures are problem for EADA we explain with a simple example.

Let us consider the WDP shown in Figure 1.4 and try to find at which iterations j there is a data dependence between functions $F2$ and $F3$ through variable $x(j)$. Applying EADA for this example requires that we have to build a linear system of inequalities and/or equalities and to solve it using parametric integer programming (PIP) [27]. The solution will show exactly at which iterations j there is a data dependency between $F2$ and $F3$. For our example, EADA requires that the linear system must capture the following information: 1) the exact iterations j at which variable $x(j)$ is written by $F2$; 2) the exact iterations j at which variable $x(j)$ is read by $F3$; 3) the exact iterations j at which variable $x(j)$ at line 7 in Figure 1.4 and the same variable $x(j)$ at line 9 have equal indexes.

We can not build the linear system described above because information about the exact iterations j at which variable $x(j)$ is written by $F2$ can not be obtained from the WDP in Figure 1.4. The reason for this is the data-dependent if-condition at line 6. The outcome of this condition is unknown because the values of variable $x(j)$ are unknown at compile time.

The example given above shows clearly that exact array dataflow analysis (EADA) can not be performed to find data dependencies in WDPs because the dynamic (data-dependent) control structures in WDPs make some information unknown at compile time. Therefore, in COMPAANDYN we perform more advanced dependence analysis using a technique called *Fuzzy Array Dataflow Analysis* (FADA) [28] [29]. The main idea is to substitute the unknown information in a specific parametric way. By doing this we can build linear systems with parameters and we can solve these systems by PIP. As a result we find approximated data dependencies because of the specific parameterization. Below, we give an example of how unknown information is substituted by parameters and captured in a linear system.

Again, let us consider the WDP shown in Figure 1.4. We showed above that an exact array dataflow analysis (EADA) can not be performed to find if there is data dependency between functions $F2$ and $F3$ because we can not build the linear system EADA requires. The exact iterations j at which variable $x(j)$ is written by $F2$ are unknown. The fuzzy array dataflow analysis (FADA) solves this problem by substituting this unknown information as shown in Figure 1.6 - see lines a) and b) in the linear system.

$$\begin{array}{ll} a) & 1 \leq j_w \leq 4 \\ b) & j_w = C \\ c) & 1 \leq j_r \leq 4 \\ d) & j_w = j_r \end{array}$$

Figure 1.6: Example of building a linear system in Fuzzy Array Dataflow Analysis.

The meaning of lines a) and b) is: variable $x(j)$ can be written by $F2$ in any iteration $j_w \in [1..4]$. We do not know exactly at which iterations j_w this happens but we assume that this happens for iterations $j_w = C$ where C is a free parameter which values have to be determined at run time. The rest of the lines in Figure 1.6 capture the information known at compile time: c) the exact iterations j_r at which variable $x(j)$ is read by $F3$; d) the exact

iterations j at which variable $x(j)$ at line 7 in Figure 1.4 and the same variable $x(j)$ at line 9 have equal indexes.

2) Dynamic Single Assignment Code:

At STEP1 in Figure 1.5 we use the approximated data dependencies found by FADA to generate a program that we call Dynamic Single Assignment Code (dSAC). Because of the approximated data dependencies our dSAC notion is different from the classical single assignment code (SAC or SSA) used in the compiler community and systolic array community. The classical SAC is defined as a program in which every variable is written only once whereas our dSAC has the property that every variable is written *at most once*. This property implies that some of the variables may not be written at all. This is because of the dynamic control structures in a WDP where the conditions are data-dependent, i.e., the outcome of the conditions is not known at compile time. As an example, in Figure 1.7 we give the dSAC we derive for the simple WDP shown in Figure 1.4.

```
1  for j = 1:1:4,
2      ctrl(j) = 5;
3  end
4
5  for j = 1:1:4,
6     [x_1(j)] = F1(...);
7  end
8
9  for j = 1:1:4,
10
11    if x_1(j) <= 0,
12      [ x_2(j) ] = F2( x_1(j) );
13      [ ctrl(j) ] = j;
14    end
15
16    C = ctrl(j);
17    if j = C,
18      [in_0] = x_2(C);
19    else
20      [in_0] = x_1(j);
21    end
22
23    [...] = F3(in_0);
24
25 end
```

Figure 1.7: Example of Dynamic Single Assignment Code.

We call the code in Figure 1.7 dynamic SAC because if we consider for example line 12 we do not know at compile time at which iteration the elements of the array $x\_2(j)$ will be written. The only thing known is that they will be written at most once. Moreover, for every execution of function $F3$ in line 23, its input is not known at compile time. The input has to be determined at run time by the code lines 17-21. This code lines we derive based on the solution of the linear system shown in Figure 1.6. Both cases described above never occur in the classical SAC cases.

Another new feature of our dSAC is the presence of parameters that originate from the data-dependent control constructs in a weakly dynamic program (WDP). In order to keep the functionality of the dSAC equivalent to the functionality of the original WDP, the values of these parameters have to be changed dynamically at run time. We have developed an

approach to accomplish the dynamic change by introducing, for every such parameter, a control variable that stores the correct value of the parameter for every iteration. For example, C in the dSAC shown in Figure 1.7 is a parameter emerging because of the *if*-statement in line 6 of the original program shown in Figure 1.4. This *if*-statement also appears in the dSAC in line 11. The dynamic change of the value of C is accomplished by the lines 13 and 16 in Figure 1.7. The control variable $ctrl(j)$ in line 13 stores the iterations for which the data-dependent condition that introduces C is true. Also, the variable $ctrl(j)$ is used in line 16 to assign the correct value to C for the current iteration. The control variable $ctrl(j)$ is initialized at the beginning in line 2.

3) Approximated Dependence Graph:

In order to capture, in a formal way, all the information about the data dependencies between the functions in a dSAC we have introduced our formal model called Approximated Dependence Graph (ADG) at STEP2 in Figure 1.5. We have developed the ADG model because the classical and widely used Dependence Graph (DG) or Polyhedral Reduced Dependence Graph (PRDG) [19] models are not general enough to capture approximated data dependencies occurring in a dSAC. We explain this observation with a simple example.

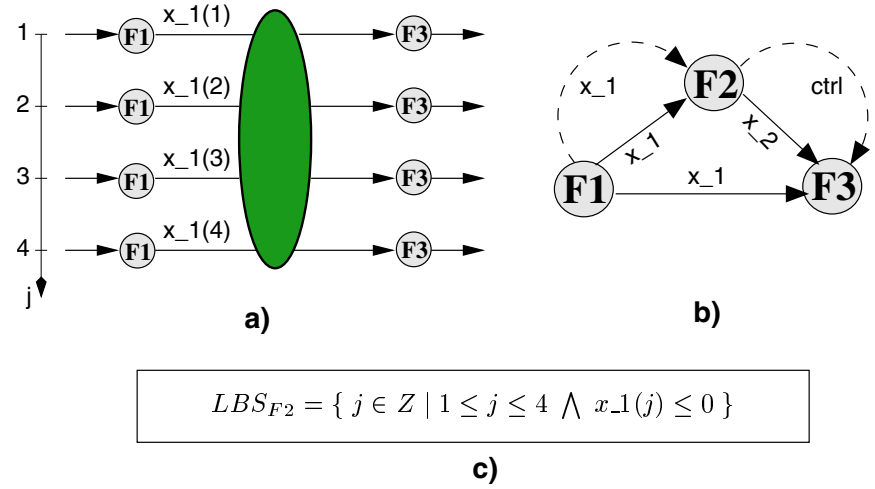

Figure 1.8: Examples of Approximated Dependence Graph and Linearly Bounded Set: a) Classical Dependence Graph cannot be obtained from dSAC; b) Approximated Dependence Graph; c) Linearly Bounded Set.

Let us consider the dSAC shown in Figure 1.7. If we try to derive a DG based on the information in the dSAC we will arrive at the incomplete graph shown in Figure 1.8-a). In this graph there is a dark spot in the middle indicating that some information is unknown at compile time. Because of the data-dependent *if*-statement at line 11 in Figure 1.7 we do not known at compile time how many times and at which iterations function $F2$ is executed and from where function $F3$ takes its input. Therefore, we conclude that a DG can not be derived at compile time from a dSAC. Moreover, a PRDG can not be derived as well because a PRDG

is a folded DG annotated with polyhedrons.

The example above clearly indicates the need for a model that can deal with the unknown information. The ADG model we have developed is a possible solution. Here, we present the ADG model by an example. The ADG corresponding to the dSAC shown in Figure 1.7 is depicted in Figure 1.8-b). For every function in the dSAC there is a node in the ADG. For every variable in the dSAC there is an edge in the ADG that indicates *possible* data dependency. As explained above, some information is not exactly known at compile time. Therefore, we annotate the nodes in the ADG with *Linearly Bounded Sets* (LBS). We have developed the notion of LBS in order to approximate the unknown information.

For example, the exact iterations j at which function $F2$ is executed in the dSAC are not known at compile time because of the dynamic condition at line 11 in the dSAC (Figure 1.7). The LBS shown in Figure 1.8-c) approximates the unknown iterations j. The linear bound of this LBS is the polytope $B = \{1 \leq j \leq 4\}$ that captures the information that we know at compile time about the bounds of the iterations j. The variable $x\_1(j)$ is interpreted as an unknown function of j called filtering function whose output is determined at run time and depends on $F1$ - see the dashed edge between $F1$ and $F2$ in Figure 1.8-b). Introducing the LBS notion in our ADG model to capture the dynamic behavior of the dSAC is to the best of our knowledge a novel approach.

Because of the semantics of the LBS, described above, we call the graph in Figure 1.8-b) approximated dependence graph (ADG). Another reason to call it that way is because the probability that some data dependencies exist can not be decided 100% at compile time. For example, the edge between $F1$ and $F3$ suggests that there might be a data dependency between $F1$ and $F3$ through the variable $x\_1(j)$ but this depends on the dynamic condition at line 11 of the dSAC in Figure 1.7. If this condition is always true at run time the data dependency between $F1$ and $F3$ does not exist.

1.2.3 Algorithmic Transformation Techniques (MATTRANSFORM)

In this section we introduce our algorithmic transformation techniques to derive a set of executable KPN specifications from an application specified as a *weakly dynamic program*. We call these techniques MATTRANSFORM for short. First, we motivate our work on the MATTRANSFORM techniques by explaining why deriving a set of alternative KPN specifications is important in system-level design. Then we give an overview of the MATTRANSFORM techniques. A more elaborate and detailed presentation of the MATTRANSFORM techniques is given in Chapter 3.

Why is a set of KPN specifications needed?

In Section 1.1 we argued that a system designer needs a parallel application specification in order to map an application onto a parallel multiprocessor architecture in a systematic way. Also, in Section 1.2.1 we motivated why in this dissertation we focus on the Kahn Process Network (KPN) model of computation for parallel application specification. A KPN specification corresponding to an application describes how this application is partitioned

into a composition of concurrent processes which communicate via unbounded FIFOs.

An application can be partitioned into a composition of concurrent processes in many different ways. Therefore, many KPN specifications that correspond to a single application exist. We call them *application instances*. For example, a set of alternative KPN instances of an application is shown in Figure 1.2. Each application instance differs from the others in the degree of exploited *task-level* parallelism.

When a system designer maps an application onto a parallel multiprocessor architecture the performance of the system (application + architecture) can significantly depend on the application instance (KPN specification) that is being mapped. So, a system designer needs support to derive a set of alternative KPN instances of an application in order to explore and evaluate the performance of the system and to choose an application partitioning that satisfies requirements the target system has to meet.

In general, a system designer is able to derive at most a few alternative KPN instances. This is so because no systematic way to derive a KPN instance, let alone alternatives, from an application is known, as a result of which heuristic and time consuming approaches are taken in practice. Nevertheless, many instances of a single application exist that are worth to be derived for exploration. In Section 1.2.2 we presented our systematic COMPAANDYN approach to derive an executable KPN specification from an application specified as weakly dynamic program. Below, we introduce algorithmic transformation techniques that we developed and implemented to extend the COMPAANDYN approach in order to help a system designer to derive systematically and quickly a set of alternative KPN specifications from an application. The algorithmic transformations that we present below are not generally applicable in the sense that the application has to be specified as a weakly dynamic program.

MATTRANSFORM

MATTRANSFORM is a transformation toolbox that consists of four algorithmic transformations, namely *Unfolding*, *Skewing*, *Plane Cutting*, and *Merging*. We developed these transformations in a specific way in order to efficiently exploit the parallel compiler techniques in COMPAANDYN when deriving alternative Kahn Process Network (KPN) specifications from an application specified as a weakly dynamic program (WDP). This means that the transformations are meaningful only when they are used in combination with COMPAANDYN to extract *task-level* parallelism and to increase or decrease the degree of *task-level* parallelism exploited in a KPN specification. In Figure 1.9 we present the transformations by separating them in two main groups depending on their ability to increase or decrease task-level parallelism.

If no transformation is selected to be applied then the COMPAANDYN approach derives an executable KPN specification from a WDP, where the computational workload of the WDP is distributed over several processes in accordance with the following partitioning rule: every function call (task) in the WDP is wrapped in a process that controls the execution of the function call. In some cases this partitioning rule gives sufficient degrees of task-level parallelism in the generated KPN specification. However, in many other cases the expressed task-level parallelism may not be the preferred one.

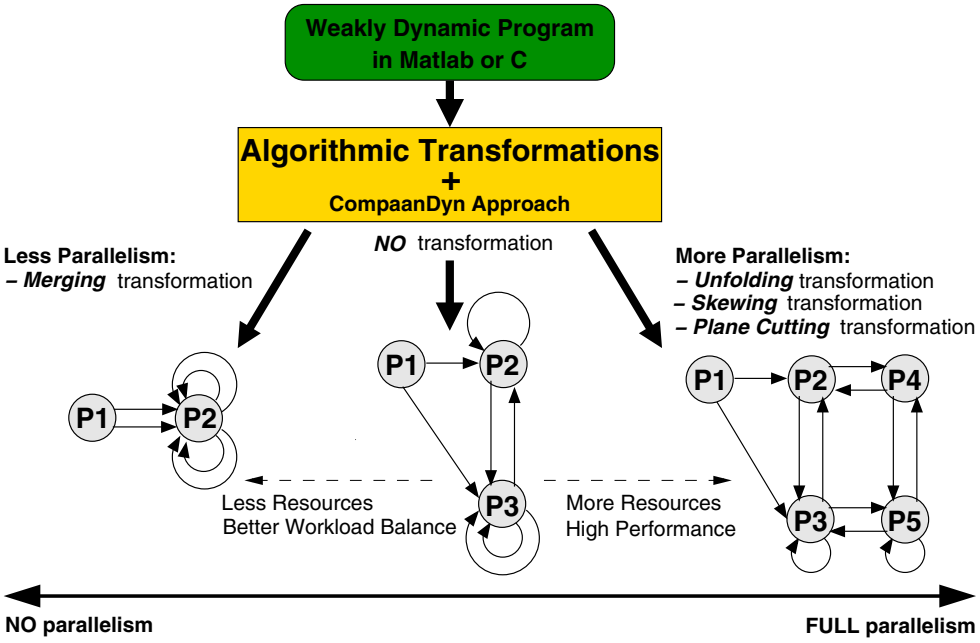

Figure 1.9: MATTRANSFORM: Algorithmic Transformation Techniques toolbox that allows systematic derivation of a set of alternative KPN specifications bounded by a KPN where NO parallelism is exploited to a KPN where FULL task-level parallelism is exploited.

For example, a system designer may want to increase the task-level parallelism in a KPN specification because his target multi-processor architecture has more parallel resources that can be exploited but the KPN specification does not have enough concurrent processes. Therefore, as shown in Figure 1.9, one of the transformations *Unfolding*, or *Skewing*, or *Plane Cutting* can be applied as well as any combination of them to increase the number of the concurrent processes in the KPN. This means more task-level parallelism in the generated KPN and higher performance when mapping this KPN onto the target architecture.

As another example, a situation may occur where a system designer wants to decrease the number of concurrent processes in a KPN specification. In Figure 1.9 we show that in such situation the *Merging* transformation can be applied. A reason to apply this transformation can be to generate a KPN with better workload balance between the processes by grouping N less computationally intensive functions (tasks) of the WDP in M concurrent processes, where $N > M$. Another reason can be that the multi-processor architecture onto which the KPN specification is to be mapped has less parallel resources than concurrent processes the KPN has.

In general, a large number of KPN specifications exist that are functionally equivalent to a single WDP. Two extreme KPN specifications bound this large number:

1. KPN specification that consists of one process where **NO** parallelism is exploited;

2. KPN specification that consists of M concurrent processes, $M > 1$, where **FULL** *task-level* parallelism available in the WDP is exploited.

Using our algorithmic transformations and any possible combination of them together with the COMPAANDYN approach, a set of KPN specifications can be derived. This set includes the two extreme KPNs described above and many other KPNs in between in which the degree of exploited task-level parallelism is different.

1.3 Related Work

The work presented in this dissertation is directly related to a previous work on automatic derivation of Process Networks initiated by Rijpkema et al. [19] [18] and further developed by Turjan et al. [30]. This previous work called COMPAAN focuses on deriving Kahn Process Network (KPN) specifications from applications specified as static parameterized affine nested loop programs. In contrast, our COMPAANDYN approach presented in this dissertation deals with a more general class of applications, i.e., applications described as weakly dynamic programs from which KPN specifications are derived in a systematic and automated way. The main novelties in our COMPAANDYN approach that make our approach more general than the COMPAAN work have been introduced in Section 1.2.2.

First, in the context of automatic parallelization of sequential programs research has been done on approaches to convert a nested loop program to an equivalent program which is in a single-assignment form, i.e., a program in which every memory cell is written at most once. Such program is easier to be analyzed and parallelized efficiently. The work of Knobe and Sarkar [31] and the work of Feautrier et al. [32] on this topic are directly related to the first step in our COMPAANDYN approach presented in this dissertation. This is because in this step we propose an approach to convert a weakly dynamic program (WDP) into a single-assignment form which we call dynamic Single Assignment Code (dSAC). The relations are explained below.

Knobe and Sarkar [31] proposed an approach to convert a nested loop program to a single-assignment form that they call Array Static Single Assignment (ASSA). Their approach is more general than our approach in the sense that the class of nested loop programs which they can convert to their ASSA includes our class of weakly dynamic programs (WDPs) which we can convert to our dSAC. However, when WDPs are considered, our approach is more efficient compared to their approach in the sense that our dSAC is a more efficient single-assignment form in terms of code lines and memory usage compared to their ASSA form. This is because in our approach a dependence analysis is performed at compile-time before the corresponding dSAC is generated. This compile-time dependence analysis, called fuzzy array data-flow analysis (FADA), allows an efficient code generation. The approach of Knobe and Sarkar does not perform any dependence analysis at compile-time. Instead, the dependence analysis is performed at run-time by placing a special code called ϕ functions and @ arrays in their ASSA, thereby making their approach more general. The ϕ functions and @ arrays introduce significant code overhead because in many cases unnecessary ϕ functions and @ arrays are placed in the ASSA, thereby making the ASSA form very inefficient in

terms of code lines and memory usage compared to our dSAC.

The work of Feautrier et al. in the context of the PAF parallelizer [32] describes an approach to convert nested loop programs similar to our WDPs into a single-assignment form called SA. Their approach is based on performing a fuzzy array data-flow analysis (FADA) at compile-time before generating the SA. The result of this FADA analysis is implemented by ϕ functions placed in their SA during code generation. The ϕ functions depend on parameters whose values have to be set dynamically at run-time in order to preserve the original data-flow when the control flow cannot be predicted at compile-time. The work of Feautrier et al. lacks a general approach to set the values of the parameters at run-time. The work described above relates to our approach for converting a WDP to a dSAC in the sense that we also perform a FADA analysis at compile-time and we also place a code with parameters in our dSAC similar to the ϕ functions but our code is more efficient. Also, the difference is that we have developed a very simple general approach to set the values of the parameters at run-time. This approach is presented in Section 2.1.2.

In the context of the IMEC's Data Transfer and Storage Exploration (DTSE) methodology a technical report by Vanbroekhoven et al. [33] sketches, in terms of examples only, two ways to transform a program to a single-assignment form that they call Dynamic Single Assignment (DSA). This work relates to our approach for converting a WDP to a dSAC in the sense that the programs they consider in their report are the same as our WDPs. However, from the examples they show it is not possible to determine how general, systematic, and efficient their approach is if data-dependent "if"-statements are present in a program. This fact does not allow us to make a real comparison between their approach and our general and systematic approach presented in this dissertation. The only thing which is clear about their approach is that they do not perform any array data-flow analysis, whereas in our approach we perform fuzzy array data-flow analysis at compile-time.

Substantial work has been done on the formal modeling of the behavior of nested loop programs in the area of regular array design. Although our work does not deal with regular array design, there is a relation in the sense that we also have developed and used formal models to capture and model the behavior of our weakly dynamic programs (WDPs). This is done in the second step of our CompaanDyn approach presented in this dissertation. Below we mention some of the formal models used for regular array design and compare them with our models which we translate to executable Kahn Process Network specifications.

In [34] the Reduced Dependence Graph (RDG) model and a system of uniform recurrence equations (URE) are introduced and used to model and specify affine nested loop programs in the context of automatic generation of systolic arrays. In [35] Thiele introduces piece-wise regular dependence graphs and reduced piece-wise regular dependence graphs annotated with linearly bounded lattices to represent the behavior of so-called piece-wise regular programs. Using such graphs Teich and Thiele have proposed in [36] [37] a systematic mapping of piece-wise regular algorithms onto processor arrays. In [38] Swaaij et al. introduce a model called Polyhedral Dependence Graph (PDG) where every node is annotated with conditional affine recurrence equations. They synthesize systolic arrays from the PDG model. All the work mentioned above is only capable of modeling the behavior of nested loop programs that are static, i.e., programs whose behavior is completely known at compile-time. In contrast, our *Approximated Dependence Graph* (ADG) model annotated with *Linearly Bounded Sets*

(LBS) has been introduced and developed in this dissertation in order to enable the capturing and modeling of our weakly dynamic programs, i.e., programs whose exact behavior is not known at compile-time.

In this dissertation we present our set of algorithmic transformations that we have developed to facilitate the systematic and automated derivation of alternative KPN specifications from a weakly dynamic program. Our set of transformations has been introduced in Section 1.2.3. Our *Unfolding* and *Skewing* transformations are related to the loop unrolling and loop skewing techniques used in compiler design [39] [40].

The relation between our unfolding transformation and the well known compiler transformation loop unrolling [39] is in that both transformations aim at enhancing parallelism in a sequential program. However, loop unrolling enhances instruction level parallelism by copying a loop body several times and re-indexing the variables in the body, thus creating more parallel instructions and reducing the loop control overhead. A prologue or epilogue is generated to guarantee that the unrolled version executes the correct number of iterations of the original loop. In contrast, our unfolding transformation enhances task-level parallelism by copying a loop body a number of times in such a way that these copies are mutually exclusive, thus these copies can be encapsulated in concurrent processes. Also, our unfolding transformation does not generate prologue or epilogue code.

The relation between our skewing transformation and the loop skewing transformation used in the classical high-performance compilers [40] is in that both transformations are enabling (auxiliary) transformations that are primarily useful in combination with other transformations to exploit parallelism. However, the classical loop skewing is used in combination with loop interchange to exploit fine-grain instruction level parallelism by handling so called wavefront computations. In contrast, our skewing transformation is used in combination with our unfolding transformation to expose and exploit task-level parallelism by deriving alternative KPNs.

In [41] Sriram and Bhattacharyya describe two techniques, namely unfolding and re-timing, that are used for improving block schedules for Homogeneous Synchronous Data Flow (HSDF) graphs by exploiting inter-iteration parallelism. These techniques are related to our transformations unfolding and skewing in the sense that our transformations also facilitate the exploitation of inter-iteration parallelism available in a weakly dynamic program (WDP) when such program is converted to a set of KPN specifications. The difference is that we have developed procedures to do these transformations on the source code of a WDP corresponding to an application, whereas in [41] the transformations are applied on the HSDF graph corresponding to an application. More general class of applications can be specified as WDPs compared to the class of applications that can be specified as HDFGs. Because of this our transformations have more general applicability compared to the transformations described in [41].

Our skewing transformation is similar to the re-timing transformation used in the signal-processing community [42] in that both transformations aim at improving the performance of an application by changing the time (iterations) at which some computations are executed. The difference is that re-timing involves manipulating delays in a Signal Flow Graph (SFG), thereby minimizing critical data paths in a SFG and maximizing clock rates. Our skewing

transformation involves manipulating loop bounds and variable indexes in a weakly dynamic program (WDP), thereby creating independent computations in a loop body that can be executed in parallel or in pipeline. As said earlier, we have developed a procedure to do the skewing transformation on the source code of a WDP. In [42] the re-timing transformation is applied to a SFG that can model static nested loop programs only. This fact and the fact that a WDP can specify a more general class of applications compared to a static nested loop program implies that our skewing transformation has more general applicability than the re-timing transformation.

In [43] Parhi and Messerschmitt describe an unfolding transformation developed to be applied on iterative data-flow programs. This transformation is similar to our unfolding in that both transformations increase the number of tasks in a program and unravel the hidden concurrency. However, iterative data-flow programs, as defined in [43], are static which limits the applicability of the Parhi's unfolding to static programs. In contrast, our unfolding transformation is developed to be applied on weakly dynamic programs which are more general than the iterative data-flow programs. This means that our unfolding transformation is more general than the Parhi's unfolding.

In [44] Teich and Thiele propose an approach to partition affine dependence algorithms for mapping onto reduced/fixed size processor arrays. Their approach is based on two transformations called EXPAND and REDUCE. Their work relates to our work presented in this dissertation in the sense that our approach to generate Kahn Process Networks (KPNs) using our *Unfolding* and *Plane cutting* transformations is also an approach to partition algorithms. However, there are some important differences. First, our approach deals with a more general class of algorithms, i.e., algorithms described as weakly dynamic programs. Second, the result of the partitioning, i.e., the generated KPNs are suitable for mapping onto heterogeneous multi-processor platforms. Third, by using our *Unfolding* and *Plane cutting* transformations to generate KPNs we do a *reverse* partitioning compared to the approach of Teich and Thiele. They start with a dependence graph (DG) representation of an algorithm which is the partitioning of an algorithm that exploits the maximum parallelism available in an algorithm. Then they apply tiling (grouping) on the DG representation to obtain a desired partitioning in which less parallelism is exploited. In contrast, we start with a WDP where no parallelism is exploited and by unfolding or plane cutting we partition the computational workload of the WDP onto several processes, thereby obtaining a desired partitioning in which more parallelism is exploited.

Kahn Process Networks are supported by the Ptolemy II framework [12] and the YAPI environment [45] for concurrent modeling and design of applications and systems. The designer has to specify manually the application as a Kahn Process Network and to give this network as an input to the Ptolemy II or YAPI simulation and verification engines. In many cases manually specifying an application as a Kahn Process Network is a very time consuming and an error prone process. Our work, presented in this dissertation, relates to Ptolemy II and YAPI in the sense that it can be used as a front-end tool by Ptolemy II or YAPI. This will significantly speedup the modeling effort when Kahn Process Networks are used, and modeling errors will be avoided because our techniques guarantee correct-by-construction generation of Kahn Process Networks.

The work presented in [14] [15] [16] [46] uses Kahn Process Networks to model applica-

tions and to explore the mapping of these applications onto multi-processor architectures. This work clearly indicates that the application modeling is done manually starting from a sequential C code and that significant amount of time (a few weeks) is spent by designers on correctly transforming the sequential C code into Kahn Process Networks. This fact slows down the design space exploration process. The work presented in this dissertation gives a solution for systematic and automatic derivation of Kahn Process Networks from sequential code that will contribute to faster design space exploration.

1.4 Research Contributions

The work presented in this dissertation focuses on the derivation of a set of executable Kahn Process Network specifications from an application specified as a weakly dynamic program. The main contributions are:

- **development of an approach (called COMPAANDYN) that allows derivation of a Kahn Process Network specification from a Weakly Dynamic Program in a systematic and automated way [47]:**
  Many system-level design flows and application modeling and exploration approaches reported in the literature use the Kahn Process Network (KPN) model of computation for a concurrent application specification [11] [12] [14] [15] [16] [45] [48] [49] [50]. The derivation of a KPN specification is based on heuristic and time consuming approaches because no systematic way to derive a KPN specification from an application has been known. Recently, the work presented in [19] proposed an approach to derive a KPN specification from a static affine nested loop program (SANLP). In contrast, this dissertation presents a systematic approach to derive a KPN specification from an application specified as a weakly dynamic program (WDP) that can be easily automated. A WDP is a more general program class where a SANLP is a special case of WDP. Therefore, the work presented in this dissertation extends significantly the class of applications from which KPN specifications can be derived in a systematic and automated way.

- **new notions such as Dynamic Single Assignment Code, Approximated Dependence Graph and Linearly Bounded Sets have been introduced in order to capture and model weakly dynamic behavior in a WDP [47]:**
  A lot of work has been done in capturing and modeling the behavior of nested loop programs in research fields such as regular array design, automatic parallelization, advanced parallel compilers. The main focus of this work is restricted to modeling static nested loop programs, thereby limiting the expressive power of the models. Such models are Static Single Assignment Code (SAC or SSA), Reduced Dependence Graph (RDG) annotated with uniform recurrence equations (URE) [34], Polyhedral Dependence Graph (PDG) annotated with conditional affine recurrence equations [38], Polyhedral Reduced Dependence Graph (PRDG) annotated with $\mathbb{Z}$-polyhedra or periodic lattice polyhedra [19], and Reduced Piece-wise Regular Dependence Graph (RPRDG) annotated with linearly bounded lattices [35]. The weakly dynamic programs (WDPs), this dissertation deals with, can be non-static nested loop programs that have dynamic

(data-dependent) behavior, unpredictable at compile time. The models described above can not be applied to model the behavior of WDPs. Therefore, we have developed our own models, namely Dynamic Single Assignment Code and Approximated Dependence Graph annotated with Linearly Bounded Sets - see Section 1.2.2 and Chapter 2.

- **task-level algorithmic transformations (called MATTRANSFORM) have been developed to derive a set of alternative KPN specifications from a WDP [51]:**
  Deriving a set of alternative KPN specifications from an application is very important in System-level design, because it gives system designers an opportunity to perform design space exploration and to select a KPN specification that meets best the system requirements. In this dissertation we present several algorithmic transformations which we have developed to facilitate a systematic derivation of alternative KPN specifications from a WDP. These KPN specifications are behaviorally equivalent to the input WDP but the degree of exploited task-level parallelism is different. To the best of our knowledge our algorithmic transformations together with our COMPAANDYN approach provide for the first time a systematic and fast approach to derive alternative KPNs from an application specified as a WDP.

- **validation of the presented approach and transformations with real-life industrial relevant applications [17] [47] [51]:**
  Case studies and experiments have been performed during the course of the research work presented in this dissertation to validate the techniques and methods we have developed. The case studies and experiments have shown that the result of our research work can be applied successfully in real-life applications. The case studies and some of the experiments as well as the obtained results are presented in this dissertation.

- **prototyping the presented approach (COMPAANDYN) and transformations (MATTRANSFORM) in software:**
  To automate and verify the approach and transformations presented in this dissertation the following software has been developed on top of the Compaan tool: 1) some of the techniques in COMPAANDYN have been prototyped and tested as software procedures. There is work in progress for the complete implementation of COMPAANDYN in software; 2) the presented algorithmic transformations are implemented in a software tool called MATTRANSFORM.

1.5 Dissertation Outline

The remaining part of this dissertation is organized as follows. Chapter 2 presents our 3-step approach for deriving an executable Kahn Process Network (KPN) specification from an application specified as a Weakly Dynamic Program (WDP). The chapter describes in great details the models, methods, and techniques we have developed and used in the approach. First, we describe the techniques and procedures involved in the conversion of a WDP to our dynamic Single Assignment Code (dSAC). Second, we define our two formal models, namely Approximated Dependence Graph (ADG) and Schedule Tree (STree) as well as the conversion from a dSAC to an ADG and a STree is described. Third, we present how the ADG and the STree models are translated to a Process Network model, thereby synthesizing

an executable KPN specification. The KPN synthesis is decomposed in several sub-steps that are described as well.

In Chapter 3 we present a set of four algorithmic transformations, namely unfolding, plane cutting, skewing, and merging that we have developed for a systematic derivation of alternative application instances (Kahn Process Networks) from an application specified as a weakly dynamic program. First, we explain how these transformations are encapsulated in an application transformation layer on the top of a Y-chart exploration environment in order to facilitate system designers in exploring alternative instances of an application mapped onto an architecture template. Next, we describe in detail each transformation in the set by explaining the general idea behind the transformation, and the formal procedure to do the transformation, and we give an example that illustrates the formal procedure.

In Chapter 4 we present two case studies that we conducted in order to validate and evaluate our approach presented in Chapter 2 and our algorithmic transformations presented in Chapter 3 on real-life, industrially relevant applications. We report and analyze the results we obtained in these case studies.

Finally, in Chapter 5 we conclude this dissertation with a summary of our research work presented in the dissertation, interlaced with some concluding remarks.

Chapter 2

Deriving Process Networks from Weakly Dynamic Programs

In Chapter 1 we argued that Kahn Process Networks are simple, yet powerful enough for the targeted application domain, parallel processing models which match the emerging multi-processor architectures in the following sense: 1) KPNs specify applications as a composition of concurrent processes where the computation, control, and memory are distributed; 2) the multi-processor architectures have components that run concurrently, i.e, the computation and control are distributed over the components, and the architecture has several memory banks, i.e., distributed memory. Because of 1) and 2), the mapping of Kahn Process Network (KPN) specifications of applications onto multi-processor architectures can be done in a systematic and transparent way. Also, we argued that a systematic approach to derive KPNs from sequential programs is needed because the application developers continue to specify applications as sequential programs.

In this chapter we present our systematic approach COMPAANDYN for deriving executable Kahn Process Network specifications from applications specified as Weakly Dynamic Programs (WDP). We have introduced briefly the main steps in COMPAANDYN in Chapter 1 - Section 1.2.2. Here, we elaborate, in more details, on the models and techniques we have developed and integrated in every step in COMPAANDYN. Figure 1.5 shows our 3-step COMPAANDYN approach. We start with a WDP and transform it into dynamic single assignment code (dSAC) - STEP1 in Figure 1.5. In Section 2.1, we describe our approach to derive dSAC by presenting in detail the main techniques and procedures involved in the derivation. An example is given as well.

The second step in our approach is to convert the dSAC into a formal model. This model consists of two annotated graph structures, namely *Approximated Dependence Graph* (ADG) and *Schedule Tree* (STree). The ADG and the STree capture all the information that is present in the dSAC in a formal way. As a consequence, formal operations can be easily defined and applied on the ADG and the STree instead of the dSAC. In Section 2.2, we give a formal

definition of the ADG. Also, we describe our procedure to derive an ADG from a dSAC and show an example. In Section 2.3 we define the STree and describe how it is obtained from a dSAC. Also, we give an example.

In the final step (STEP 3 in Figure 1.5) of our approach, a Kahn Process Network (KPN) is synthesized. Our synthesis approach is mainly a translation of the ADG model and the STree model into a Process Network model. This translation is done in several sub-steps that are presented in Section 2.4.

2.1 Dynamic Single Assignment Code

The Dynamic Single Assignment Code (dSAC) derived from a WDP program is a functionally equivalent program in which every variable is written *at most once*. This property suggests that some of the variables may not be written at all. This is because of the dynamic ”if”-statements in the input WDP where the conditions are data-dependent, i.e., the outcome of a condition is not known at compile time. Our dSAC notion is different from the classical single assignment code (SAC or SSA). The differences were outlined in Section 1.2.2. In order to derive a dSAC from a WDP we have to do the following: 1) find all possible data dependencies between the functions in the WDP. Below, in Section 2.1.1, we give the formal procedure we adapted from [28] to do the data-dependence analysis, and we illustrate it with an example; 2) The data-dependence analysis procedure is necessary but not sufficient to generate a complete dSAC suitable for our final goal of deriving KPNs. Therefore, we have developed an additional procedure in order to generate an executable dSAC which complies with our final goal. This procedure is explained in Section 2.1.2

2.1.1 Fuzzy Array Dataflow Analysis (FADA)

Because of the appearance of dynamic (data-dependent) ”if”-statements in a WDP, an exact array dataflow analysis [24] can not be performed to find data dependencies. This observation was illustrated in Section 1.2.2 with an example. The approach we follow to find the data dependencies in case of a WDP is based on parametric integer programming (PIP) [27]. We use the same technique as in the exact array dataflow analysis for building a PIP system and add to this system constraints with parameters for the dynamic ”if”-statements in the WDP. By introducing additional constraints with parameters in a PIP system we ”mask” the information that is not known at compile time. Because of this we find approximated data dependencies. The approach described above is known in the literature as *Fuzzy Array Dataflow Analysis* (FADA) [28].

Notations

Before we give our procedure to do dependence analysis based on FADA we give the notations we will use to describe the procedure:

- WDP denotes an arbitrary weakly dynamic program in accordance with the definition given in Section 1.2.2. An example of WDP is given in Figure 2.1;

- $\langle F, I_F \rangle$ denotes an arbitrary function call in WDP, where I_F is the iterator vector of F, i.e., the vector built from the iterators of the loops surrounding F. For example in Figure 2.1 the function call at code line 23 is denoted as $\langle F3, I_{F3} \rangle$;

- $I_F[k]$ denotes the k-th iterator from I_F. $I_F[k..l]$ denotes the sub-vector built from iterators k to l. If $k > l$ then this is by convention the vector of dimension 0;

- d denotes the *depth* of a construct in WDP. Or in other words, d is the number of loops surrounding a construct. A construct can be a for-loop, an if-statement, or a function call. For example, the for-loop at line 15 in Figure 2.1 has depth $d = 0$. The if-statement at line 21 has depth $d = 1$. The function call at line 22 has depth $d = 1$;

- $\langle F_i, I_{F_i} \rangle \prec \langle F_j, I_{F_j} \rangle \equiv \bigvee_{p=0}^{N} \langle F_i, I_{F_i} \rangle \prec_p \langle F_j, I_{F_j} \rangle$ defines when F_i precedes F_j lexicographically with respect to nest depths p, $0 \leq p \leq N$, where
  – N is the number of loops surrounding both F_i and F_j;
  – for $0 \leq p < N$:
  $\langle F_i, I_{F_i} \rangle \prec_p \langle F_j, I_{F_j} \rangle \Leftrightarrow (I_{F_i}[1..p] = I_{F_j}[1..p]) \wedge (I_{F_i}[p+1] < I_{F_j}[p+1])$;
  – for $p = N$:
  $\langle F_i, I_{F_i} \rangle \prec_N \langle F_j, I_{F_j} \rangle \Leftrightarrow (I_{F_i}[1..N] = I_{F_j}[1..N]) \wedge T$, where T is a boolean which is true iff F_i precedes F_j in the program code of WDP.

  For example, let us take function calls $F2$ and $F4$ in Figure 2.1. We say that $F2$ precedes $F4$ lexicographically, denoted as
  $\langle F2, I_{F2} \rangle \prec \langle F4, I_{F4} \rangle \equiv \bigvee_{p=0}^{1} \langle F2, I_{F2} \rangle \prec_p \langle F4, I_{F4} \rangle$, if one of the following hold:
  $\langle F2, I_{F2} \rangle \prec_0 \langle F4, I_{F4} \rangle \Leftrightarrow (0 = 0) \wedge (I_{F2}[1] < I_{F4}[1]) \Rightarrow I_{F2}[1] < I_{F4}[1]$
  $\langle F2, I_{F2} \rangle \prec_1 \langle F4, I_{F4} \rangle \Leftrightarrow (I_{F2}[1] = I_{F4}[1]) \wedge (T = true) \Rightarrow I_{F2}[1] = I_{F4}[1]$;

- $\mathbf{D}(\langle F, I_F \rangle)$ denotes the linear bound of the iteration domain of function call F. The linear bound of the iteration domain is the set of values that the iterator vector I_F can take, satisfying the following constraint $A.I_F \geq b$. A is an $m \times n$ integral matrix and b is an integral vector. The values of A and b are determined by: 1) the lower and upper bounds of iterators of the loops surrounding F in WDP; 2) the conditions of the "if"-statements surrounding F in WDP that are affine functions of loop iterators.

  For example, let us consider function call $F3$ in Figure 2.1. This function call is surrounded by only one "for"-loop (code line 15) with iterator i, thus the iterator vector $I_{F3} = i$. The lower and upper bounds of i are 1 and N, respectively. The "if"-statement at code line 21 also surrounds $F3$ but we ignore this statement because its condition is not an affine function of loop iterator i. So, the linear bound of the iteration domain of $F3$ is determined only by the "for"-loop bounds as follows:
  $\mathbf{D}(\langle F3, I_{F3} \rangle) = \{i \in \mathbb{Z} \mid A.i \geq b\}$ where $A = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $b = \begin{bmatrix} 1 \\ -N \end{bmatrix}$.

Procedure

Let an arbitrary weakly dynamic program WDP be given where a number of function calls are executed and these function calls communicate data through arrays of variables. If a variable is an input argument of a function call or it is used in an "if"-condition then we call this variable a right-hand-side (RHS) variable. If a variable appears as an output argument of a function call then we call this variable a left-hand-side (LHS) variable. A RHS variable and a LHS variable which have the same name form a RHS–LHS pair. Such pair may be a potential cause for a data dependency between the function call or the "if"-condition to which the RHS variable belongs and the function call to which the LHS variable belongs. Below, we give a 4-step procedure that finds approximated data dependencies between function calls or between function calls and "if"-conditions in WDP by analyzing every RHS-LHS pair that appears in WDP:

STEP1: Find all RHS-LHS pairs in WDP;

STEP2: For every RHS-LHS pair build a system of linear inequalities and equalities and solve it by parametric integer programming (PIP) [27]. If a solution exists than there is a data dependency between the function calls that correspond to the RHS-LHS pair. Moreover, the solution shows at which iterations the data dependency exists.

Let us take an arbitrary pair $< x(g(I_{F_n}))\ ,\ x(f(I_{F_m})) >$ from WDP. Variable $x(g(I_{F_n}))$ is the RHS variable where g is an affine index function. This RHS variable appears as an input argument of function call $\langle F_n, I_{F_n}\rangle$. Variable $x(f(I_{F_m}))$ is the LHS variable where f is an affine index function. This LHS variable appears as an output argument of function call $\langle F_m, I_{F_m}\rangle$. Function call $\langle F_n, I_{F_n}\rangle$ is data dependent on function call $\langle F_m, I_{F_m}\rangle$ if the following three conditions hold:

1. $g(I_{F_n}) = f(I_{F_m})$ - this condition says that the function calls must reference the same variable;
2. $I_{F_m} \prec I_{F_n}$ - this condition says that the iteration at which $\langle F_m, I_{F_m}\rangle$ writes to the variable must precede the iteration at which $\langle F_n, I_{F_n}\rangle$ reads the variable;
3. I_{F_m} is the lexicographical largest iteration satisfying the first two conditions.

To find at which iterations the above three conditions hold, a system that consists of the following four groups of inequalities and/or equalities has to be built and solved by PIP:

- **Existence Predicate group:** This group defines at which iterations the LHS variable $x(f(I_{F_m}))$ is written by the function call $\langle F_m, I_{F_m}\rangle$, i.e., at which iterations $\langle F_m, I_{F_m}\rangle$ is executed.

  If $\langle F_m, I_{F_m}\rangle$ is surrounded by dynamic (data-dependent) "if"-statements then this group consists of the following inequalities and equalities:

  $$\mathbf{D}(\langle F_m, I_{F_m}\rangle) \ \wedge \ I_{F_m}[1..d] = C$$

  where d is the depth of the *inner most* dynamic (data-dependent) "if"-statement which surrounds $\langle F_m, I_{F_m}\rangle$. The parameter vector C is introduced because we do not know

exactly, due to the dynamic (data-dependent) "if"-statements, at which iterations (for which values of the iterator vector I_{F_m}) variable $x(f(I_{F_m}))$ is written. The parameter vector C "masks" the unknown information.

If $\langle F_m, I_{F_m} \rangle$ is not surrounded by dynamic (data-dependent) "if"-statements then this group consists of:

$\mathbf{D}(\langle F_m, I_{F_m} \rangle)$

In this case we know exactly at which iterations variable $x(f(I_{F_m}))$ is written, therefore the parametric equality $I_{F_m}[1..d] = C$ is not necessary;

- **Conflicting Access group:** This group defines at which iterations the RHS variable $x(g(I_{F_n}))$ and the LHS variable $x(f(I_{F_m}))$ have equal indexes, i.e., at which iterations function calls $\langle F_n, I_{F_n} \rangle$ and $\langle F_m, I_{F_m} \rangle$ reference the same variable. This group consists of the equality:

  $g(I_{F_n}) = f(I_{F_m})$;

- **Sequencing Condition group:** This group defines at which iterations the LHS variable $x(f(I_{F_m}))$ is written before the RHS variable $x(g(I_{F_n}))$ is read, i.e., at which iterations function call $\langle F_m, I_{F_m} \rangle$ is executed before $\langle F_n, I_{F_n} \rangle$:

  $\langle F_m, I_{F_m} \rangle \prec \langle F_n, I_{F_n} \rangle$;

- **Environment group:** This group defines at which iterations the RHS variable $x(g(I_{F_n}))$ can be read, i.e., at which iterations the function call $\langle F_n, I_{F_n} \rangle$ can be executed. This group consists of the following inequalities and equalities :

  $\mathbf{D}(\langle F_n, I_{F_n} \rangle)$.

The four groups above define the following system which has to be solved with I_{F_n} and C as free variables:

$$\begin{aligned} \mathbf{Q}(\, I_{F_n}, C \,) = \{\, I_{F_m} \;\mid\; & I_{F_m} \in \mathbf{D}(\langle F_m, I_{F_m} \rangle) \;\wedge\; I_{F_m}[1..d] = C \;\wedge \\ & g(I_{F_n}) = f(I_{F_m}) \;\wedge \\ & \langle F_m, I_{F_m} \rangle \prec \langle F_n, I_{F_n} \rangle \;\wedge \\ & I_{F_n} \in \mathbf{D}(\langle F_n, I_{F_n} \rangle) \,\} \end{aligned} \tag{2.1}$$

The system above can not be solved by PIP because the sequencing condition $\langle F_m, I_{F_m} \rangle \prec \langle F_n, I_{F_n} \rangle$ is not an affine inequality or equality. However, from the notations given at the beginning of this section we know that $\langle F_m, I_{F_m} \rangle \prec \langle F_n, I_{F_n} \rangle \equiv \bigvee_{p=0}^{N} \langle F_m, I_{F_m} \rangle \prec_p \langle F_n, I_{F_n} \rangle$, so the system $\mathbf{Q}$ can be represented as the following set of systems with respect to possible depths p:

$$\begin{aligned} \mathbf{Q}^p(\, I_{F_n}, C \,) = \{\, I_{F_m} \;\mid\; & I_{F_m} \in \mathbf{D}(\langle F_m, I_{F_m} \rangle) \;\wedge\; I_{F_m}[1..d] = C \;\wedge \\ & g(I_{F_n}) = f(I_{F_m}) \;\wedge \\ & \langle F_m, I_{F_m} \rangle \prec_p \langle F_n, I_{F_n} \rangle \;\wedge \\ & I_{F_n} \in \mathbf{D}(\langle F_n, I_{F_n} \rangle) \,\} \end{aligned} \tag{2.2}$$

Now, every predicate $\prec_p$ is an affine inequality and/or equality (see the notation section above) thus every system $\mathbf{Q}^p$ is a polyhedron. Therefore, every system $\mathbf{Q}^p$ can be solved by

PIP. The PIP solution at depth p we denote as $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n}, C)$ and it gives the lexicographically largest iteration I_{F_m} that satisfies the system $\mathbf{Q}^p$, i.e., $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n}, C) = max\ \mathbf{Q}^p(\ I_{F_n}, C\ )$. For every possible depth p we obtain a solution $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n}, C)$ by PIP. The obtained solutions form a solution set that defines at which iterations the function call $\langle F_n, I_{F_n}\rangle$ is data dependent on $\langle F_m, I_{F_m}\rangle$ via the variables with a name x that form the RHS–LHS pair. The set of solutions depends on the parameter vector C which values are not known at compile time and have to be changed dynamically at run time. Therefore, the set of solutions is approximated (not exact) and we call the corresponding data dependency also *approximated*. If a solution is not found for all $\mathbf{Q}^p$, i.e., the set of solutions is empty then the function calls are independent with respect to the variables with a name x that form the RHS–LHS pair.

STEP3: Group the RHS-LHS pairs in sets where in every set the RHS-variables are the same, i.e, the RHS-variables are one and the same input argument of a function call or the RHS-variables are used in one and the same "if"-condition;

STEP4: For every set generated in STEP3 which has more than one RHS-LHS pair, combine the solutions found in STEP2 that correspond to the RHS-LHS pairs in the set. The combined solution for every set can be found by using the procedures implemented in MATPARSER [52] or by applying the rules defined in [28].

Example

Here, we illustrate the behavior of the procedure described above by an example. Let us take the weakly dynamic program shown in Figure 2.1. In this program, we have four arrays x, y, t, and z of variables that may define data dependencies. Now let us apply the four steps of the procedure to find these data dependencies:

STEP1: We have to find all RHS-LHS pairs of variables. Our example program has seven RHS-variables:

RHS_1: $x(i)$ which is the input argument of function call $F1$;
RHS_2: $y(i)$ which is the input argument of function call $F2$;
RHS_3: $t(i)$ which is the input argument of function call $F3$;
RHS_4: $y(i+1)$ which is the first input argument of function call $F4$;
RHS_5: $z(i)$ which is the second input argument of function call $F4$;
RHS_6: $z(i)$ which is in the if-condition at line 17;
RHS_7: $x(i)$ which is in the if-condition at line 21;

Also, the program has nine LHS-variables:

LHS_1: $x(i)$ which is the output argument of function call $\_Source\_x()$;
LHS_2: $y(i)$ which is the first output argument of function call $\_Source\_yt()$;
LHS_3: $t(i)$ which is the second output argument of function call $\_Source\_yt()$;
LHS_4: $z(i)$ which is the output argument of function call $\_Source\_z()$;
LHS_5: $x(i)$ which is the output argument of function call $F1$;
LHS_6: $y(i+1)$ which is the output argument of function call $F2$;
LHS_7: $t(i)$ which is the output argument of function call $F3$;
LHS_8: $y(i+1)$ which is the first output argument of function call $F4$;

```
1  %parameter N 8 16;
2
3  for i = 1:1:N,
4    [x(i)] = _Source_x( );
5  end
6
7  for i = 1:1:N+1,
8    [y(i), t(i)] = _Source_yt( );
9  end
10
11 for i = 1:1:N+2,
12   [z(i)] = _Source_z( );
13 end
14
15 for i = 1:1:N,
16
17   if z(i) = 0,
18     [ x(i) ] = F1( x(i) );
19   end
20
21   if x(i)*x(i) > 100,
22      [ y(i+1) ] = F2( y(i) );
23      [ t(i) ]   = F3( t(i) );
24   end
25
26   [ y(i+1), z(i+2) ] = F4( y(i+1), z(i) );
27
28 end
```

Figure 2.1: Pseudo code of a Weakly Dynamic Program.

LHS_9: $z(i+2)$ which is the second output argument of function call $F4$;

The RHS-variables and the LHS-variables form the following RHS-LHS pairs:
$pair1$: $< x(i)\ ,\ x(i) >$ defined by RHS_1 and LHS_1;
$pair2$: $< x(i)\ ,\ x(i) >$ defined by RHS_1 and LHS_5;
$pair3$: $< y(i)\ ,\ y(i) >$ defined by RHS_2 and LHS_2;
$pair4$: $< y(i)\ ,\ y(i+1) >$ defined by RHS_2 and LHS_6;
$pair5$: $< y(i)\ ,\ y(i+1) >$ defined by RHS_2 and LHS_8;
$pair6$: $< t(i)\ ,\ t(i) >$ defined by RHS_3 and LHS_3;
$pair7$: $< t(i)\ ,\ t(i) >$ defined by RHS_3 and LHS_7;
$pair8$: $< y(i+1)\ ,\ y(i) >$ defined by RHS_4 and LHS_2;
$pair9$: $< y(i+1)\ ,\ y(i+1) >$ defined by RHS_4 and LHS_6;
$pair10$: $< y(i+1)\ ,\ y(i+1) >$ defined by RHS_4 and LHS_8;
$pair11$: $< z(i)\ ,\ z(i) >$ defined by RHS_5 and LHS_4;
$pair12$: $< z(i)\ ,\ z(i+2) >$ defined by RHS_5 and LHS_9;
$pair13$: $< z(i)\ ,\ z(i) >$ defined by RHS_6 and LHS_4;
$pair14$: $< z(i)\ ,\ z(i+2) >$ defined by RHS_6 and LHS_4;
$pair15$: $< x(i)\ ,\ x(i) >$ defined by RHS_7 and LHS_1;
$pair16$: $< x(i)\ ,\ x(i) >$ defined by RHS_7 and LHS_5;

STEP2: For every RHS–LHS pair described above we have to build a set of systems as defined by Equation (2.2) and to solve it using PIP. Here as an example, we show how a set of systems is build for $pair9$ and what the final PIP solution for this set is.

For $pair9$ function call $F2$ writes in the LHS variable $y(i+1)$ and function call $F4$ reads

from the RHS variable $y(i+1)$ - see code lines 22 and 26, respectively. The iterations i at which $F4$ is data dependent on $F2$ via $pair9$ is determined by the set of systems given below:

$$\begin{aligned}\mathbf{Q}^p(\ I_{F4}, C\ ) = \{\ I_{F2}\ \mid\ & I_{F2} \in \mathbf{D}(\langle F2, I_{F2}\rangle)\ \wedge\ I_{F2}[1..d] = C\ \wedge \\ & g(I_{F4}) = f(I_{F2})\ \wedge \\ & \langle F2, I_{F2}\rangle\ \prec_p\ \langle F4, I_{F4}\rangle\ \wedge \\ & I_{F4} \in \mathbf{D}(\langle F4, I_{F4}\rangle)\ \}\end{aligned} \tag{2.3}$$

where:

- $0 \leq p \leq 1$ because $F2$ and $F4$ are surrounded by only one common "for"-loop which determines two possible depths p - see the program in Figure 2.1. The common "for"-loop is at depth $p = 0$. $F2$ and $F4$ are at depth $p = 1$;
- $I_{F4} = i_{F4}$ because $F4$ is surrounded by only one loop with iterator i;
- $I_{F2} = i_{F2}$ because $F2$ is surrounded by only one loop with iterator i;
- $I_{F2} \in \mathbf{D}(\langle F2, I_{F2}\rangle)\ \wedge\ I_{F2}[1..d] = C$ form the **existence predicate group** that defines at which iterations the LHS variable $y(i+1)$ is written by $F2$. The function $F2$ - line 22 in Figure 2.1 - writes the variable $y(i+1)$ in the following iterations:

  $$I_{F2} \in \mathbf{D}(\langle F2, I_{F2}\rangle)\ \Rightarrow\ 1\ \leq\ i_{F2}\ \leq\ N \qquad \text{and} \qquad I_{F2}[1..d] = C\ \Rightarrow\ i_{F2} = c$$

  NOTE: The function $F2$ is surrounded by the dynamic (data-dependent) "if"-statement at line 21 in Figure 2.1. It is not known at compile time at which iterations i_{F2} the condition of this "if"-statement is true implying that it is not known at which iterations variable $y(i+1)$ is written by $F2$. Therefore, the constraint $i_{F2} = c$ is needed to define the unknown iterations in a parametric way. The meaning is that we do not know exactly at which iterations the data dependent condition in line 21 is true but we assume that this happens for iterations $i_{F2} = c$ where c is a free parameter which values are not known at compile time.
- $g(I_{F4}) = f(I_{F2})$ is the **conflicting access group** that defines at which iterations the RHS variable $y(i+1)$ and the LHS variable $y(i+1)$ have equal indexes. For *pair9* we have:

  $$i_{F4} + 1 = i_{F2} + 1$$

- $\langle F2, I_{F2}\rangle\ \prec_p\ \langle F4, I_{F4}\rangle$ form the **sequencing conditions group** that defines at which iterations the LHS variable $y(i+1)$ is written before the RHS variable $y(i+1)$ is read. According to the lexicographical order given by the program in Figure 2.1 we have:

  $$\begin{gathered}\langle F2, I_{F2}\rangle\ \prec_0\ \langle F4, I_{F4}\rangle\ \Rightarrow\ i_{F2}\ <\ i_{F4} \\ \text{or} \\ \langle F2, I_{F2}\rangle\ \prec_1\ \langle F4, I_{F4}\rangle\ \Rightarrow\ i_{F2}\ =\ i_{F4}\end{gathered}$$

- $I_{F4} \in \mathbf{D}(\langle F4, I_{F4}\rangle)$ is the **environment group** that defines at which iterations the RHS variable $y(i+1)$ is read. Also, some constraints for the parameter N in the program in Figure 2.1 can be specified:

$$I_{F4} \in \mathbf{D}(\langle F4, I_{F4}\rangle) \;\Rightarrow\; 1 \le i_{F4} \le N \;\wedge\; 8 \le N \le 16$$

According to the specific information described above the Equation (2.3) can be represented as the following two PIP systems:

$$\begin{aligned}\mathbf{Q}^0(\, i_{F4}, c\,) = \{\, i_{F2} \mid\; & 1 \le i_{F2} \le N,\\ & i_{F2} = c,\\ & i_{F4}+1 = i_{F2}+1,\\ & i_{F2} < i_{F4},\\ & 1 \le i_{F4} \le N,\\ & 8 \le N \le 16\,\}\end{aligned}$$

$$\begin{aligned}\mathbf{Q}^1(\, i_{F4}, c\,) = \{\, i_{F2} \mid\; & 1 \le i_{F2} \le N,\\ & i_{F2} = c,\\ & i_{F4}+1 = i_{F2}+1,\\ & i_{F2} = i_{F4},\\ & 1 \le i_{F4} \le N,\\ & 8 \le N \le 16\,\}\end{aligned}$$

For the system $\mathbf{Q}^0(\, i_{F4}, c\,)$ a solution does not exist. We denote this as:

$$\mathbf{S}^0_{<F4,F2,y>}(i_{F4}, c) = \bot$$

For system $\mathbf{Q}^1(\, i_{F4}, c\,)$ PIP gives the following solution:

$$\mathbf{S}^1_{<F4,F2,y>}(i_{F4}, c) \;=\; (\,\mathbf{if}\;\; i_{F4} = c\;\; \mathbf{then}\;\; c\;\; \mathbf{else}\;\; \bot\,) \tag{2.4}$$

In the context of the program shown in Figure 2.1 this solution means that function $F4$ in line 26 is dependent on function $F2$ in line 22 via array y. This data dependency exists for iterations $i_{F4} = c$ where the first argument of function $F4$ is produced by function $F2$ and has to be taken from array element (variable) $y(c)$. We call this dependency *approximated* because: 1) the value of the parameter c is not a constant for every iteration i, i.e., c is a data dependent parameter; 2) how c changes its value is not known at compile time.

Above we showed how PIP solutions are found for $pair9$. In the same way solutions have to be found for the rest of the pairs specified in STEP1.

STEP3: We have to group the RHS-LHS pairs in sets such that in every set the RHS-variable is the same. For our example we have the following sets:
$set1$: consists of $pair1$, and $pair2$;
$set2$: consists of $pair3$, $pair4$, and $pair5$;
$set3$: consists of $pair6$, and $pair7$;
$set4$: consists of $pair8$, $pair9$, and $pair10$;
$set5$: consists of $pair11$, and $pair12$;
$set6$: consists of $pair13$, and $pair14$;
$set7$: consists of $pair15$, and $pair16$;

STEP4: For every set given above we have to combine the solutions corresponding to the pairs in the set because in every set the RHS variable is the same. Here, as an example, we show the combined solution for $set4$. This set consists of:

- $pair8$ - applying STEP2 for this pair the following set of solutions is found:
  $\mathbf{S}^0_{<F4,\_Source\_yt,y>}(i_{F4}) \ = \ (\ i_{F4}+1\ )$
- $pair9$ - applying STEP2 for this pair the following set of solutions is found:
  $\mathbf{S}^0_{<F4,F2,y>}(i_{F4},c) \ = \ \perp$
  $\mathbf{S}^1_{<F4,F2,y>}(i_{F4},c) \ = \ (\ \mathbf{if}\ \ i_{F4}=c\ \ \mathbf{then}\ \ c\ \ \mathbf{else}\ \ \perp\ )$
- $pair10$ - applying STEP2 for this pair the following set of solutions is found:
  $\mathbf{S}^0_{<F4,F4,y>}(i_{F4}) \ = \ \perp$
  $\mathbf{S}^1_{<F4,F4,y>}(i_{F4}) \ = \ \perp$

We combine the solutions above by applying the rules described in [28] or by applying the procedures Grafting and Pruning defined in [53]. The combined solution is:

$$\mathbf{S}_{<F4,y>}(i_{F4},c) \ = \ (\ \mathbf{if}\ \ i_{F4}=c\ \ \mathbf{then}\ \ c\ \ \mathbf{else}\ \ i_{F4}+1\ ) \tag{2.5}$$

The solution above defines the complete approximated data dependencies for function call $F4$ in Figure 2.1-line 26 via array y as follows: 1) $F4$ is dependent on $F2$ via array y. This data dependency exists for iterations $i_{F4} = c$ where the first argument of function $F4$ is produced by function $F2$ and has to be taken from array element (variable) $y(c)$; 2) $F4$ is dependent on $\_Source\_yt$ via array y. This data dependency exists for iterations $i_{F4} \neq c$ where the first argument of function $F4$ is produced by function $\_Source\_yt$ and has to be taken from array element (variable) $y(i_{F4}+1)$.

2.1.2 Dynamic Change of Values of Parameters introduced by FADA

In the previous section we showed how all possible data dependencies between function calls in a WDP can be found by FADA. For every RHS-LHS pair FADA finds the data dependencies as PIP solutions in one of the following two forms: 1) $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n},C)$ if F_m is surrounded by dynamic (data-dependent) "if"-statements; 2) $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n})$ if F_m is not surrounded by dynamic (data-dependent) "if"-statements. The PIP solutions can be easily converted to a corresponding program code, thereby generating non-executable single assignment code. This code will be non-executable because the PIP solutions does not contain information how the values of the parameter vectors C, appearing in some solutions, have to be set dynamically. Therefore, finding the data dependencies is not sufficient to generate complete executable dSAC which we need in our approach to derive executable KPN specifications. To overcome this problem we have developed a procedure which generates very simple and efficient additional code. This code sets the values of the parameters introduced by FADA at run time (dynamically) thereby making our dSAC code executable and functionally equivalent to the input WDP. The procedure is described below followed by an example.

General Procedure

Let us consider an arbitrary RHS-LHS pair $<\ x(g(I_{F_n}))\ ,\ \ x(f(I_{F_m}))\ >$ that appears in a WDP. Variable $x(g(I_{F_n}))$ is the RHS variable where g is an affine index function. This RHS

variable appears as an input argument of function call $\langle F_n, I_{F_n} \rangle$. Variable $x(f(I_{F_m}))$ is the LHS variable where f is an affine index function. This LHS variable appears as an output argument of function call $\langle F_m, I_{F_m} \rangle$. Let us assume that by applying the dependence analysis procedure described in Section 2.1.1 we obtain the PIP solutions $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n}, C)$ that depend on the unknown parameter vector C. The following code has to be generated in order to set the values of this vector correctly at run time:

1. For the given RHS-LHS pair $< x(g(I_{F_n}))\ ,\ x(f(I_{F_m})) >$ create an array BC of *control variables*. Every control variable $BC(f(I_{F_m}))$ of array BC has a corresponding variable $x(f(I_{F_m}))$ of array x. A control variable $BC(f(I_{F_m}))$ is used to identify the most recent iteration at which its corresponding variable $x(f(I_{F_m}))$ of array x is written by function call $\langle F_m, I_{F_m} \rangle$. At the beginning, every control variable $BC(f(I_{F_m}))$ of array BC is initialized with value $\perp$. The value $\perp$ is a unique value indicating that the corresponding variable $x(f(I_{F_m}))$ has not been written yet. Such value $\perp$ can be equal to $max(I_{F_m}) + 1$ because value $max(I_{F_m}) + 1$ is never written in any variable $BC(f(I_{F_m}))$ - see the next step. Place the initialization code for array BC at the beginning of the dSAC;

2. Place the code line $BC(f(I_{F_m})) = I_{F_m}$ immediately after the code line where the function call $\langle F_m, I_{F_m} \rangle$ is executed and a value is written in variable $x(f(I_{F_m}))$. By doing this the control variable $BC(f(I_{F_m}))$ always keeps the most recent iteration at which the LHS variable $x(f(I_{F_m}))$ is written by $\langle F_m, I_{F_m} \rangle$;

3. Place the code line $C = BC(g(I_{F_n}))$ immediately before the code corresponding to the PIP solution $\mathbf{S}^p_{<F_n,F_m,x>}(I_{F_n}, C)$. By doing this the correct value is assigned to the parameter vector C for the current iteration I_{F_n}. This value is the most recent iteration I_{F_m} at which the RHS variable $x(g(I_{F_n}))$ was written by function call $\langle F_m, I_{F_m} \rangle$. However, other functions may have written $x(g(I_{F_n}))$ in other iterations as well. Therefore, the PIP solution using the parameter vector C and iteration I_{F_n} will determine the function call which wrote most recently variable $x(g(I_{F_n}))$ and the iteration when this writing happened.

The three steps described above have to be applied on every RHS-LHS pair which has a PIP solution that depends on unknown parameters introduced by FADA.

Example

Let us consider $pair9$:$< y(i+1)\ ,\ y(i+1) >$ from the example in Section 2.1.1. This RHS-LHS pair belongs to the program shown in Figure 2.1. The PIP solution for the pair is Equation (2.4) which depends on the unknown parameter c. In Figure 2.2 we show a piece of the dSAC derived from the program in Figure 2.1 that is related to $pair9$. In this piece of the dSAC we have code lines 20 till 22 which are generated based on the Equation (2.4). Since we have unknown parameter c we have to use the procedure described in the previous section to generate additional code which sets the values of c at run time (dynamically).

```
1  %parameter N 8 16;
2
3
4
5
6  ...
7  for i = 1:1:N+1,
8    [y_1(i), ...] = _Source_yt( );
9  end
10 ...
11 for i = 1:1:N,
12   ...
13   if ...,
14      [ y_2(i) ] = F2( ... );
15
16      ...
17   end
18   ...
19
20  if i = c,
21    [ in_0 ] = y_2( c );
22  else
23    [ in_0 ] = y_1( i+1 ) );
24  end
25  ...
26  [ ... ] = F4( in_0, ... );
27
28 end
```

Figure 2.2: Piece of the dSAC related to $pair9$. The values of parameter c, introduced by the FADA analysis of $pair9$, are unknown.

```
1  %parameter N 8 16;
2
3   for i = 1:1:N,
4      [BC2(i+1)] = N+1;
5   end
6  ...
7  for i = 1:1:N+1,
8    [y_1(i), ...] = _Source_yt( );
9  end
10 ...
11 for i = 1:1:N,
12   ...
13   if ...,
14      [ y_2(i) ] = F2( ... );
15      BC2(i+1) = i;
16      ...
17   end
18   ...
19   c = BC2(i+1);
20  if i = c,
21    [ in_0 ] = y_2( c );
22  else
23    [ in_0 ] = y_1( i+1 ) );
24  end
25  ...
26  [ ... ] = F4( in_0, ... );
27
28 end
```

Figure 2.3: Piece of the dSAC related to $pair9$ with additional code which sets the values of parameter c at run time (dynamically).

1. For $pair9$ we create an array $BC2$ of control variables. Every control variable $BC2(i+1)$ of array $BC2$ is initialized with the value $N+1$ in accordance with the formula $max(I_{F2})+1$. The initialization code is placed at the beginning of the dSAC as shown in Figure 2.3 - code lines 3 till 5;

2. The code $BC2(i+1) = i$ is placed immediately after the function call $F2$ is executed - see code line 15 in Figure 2.3. Note that the indexing of variable $BC2(i+1)$ is the same as the indexing of the LHS variable $y(i+1)$;

3. The code $c = BC2(i+1)$ is placed just before the value of c is needed - see code line 19 in Figure 2.3. Note that the indexing of variable $BC2(i+1)$ is the same as the indexing of the RHS variable $y(i+1)$.

The example in Figure 2.3 shows that our general way of setting the unknown parameters at run time results in a very small and efficient additional code - see the lines in bold - which does not introduce significant overhead. In our example the array $BC2$ is indexed in the same way at code lines 15 and 19 just because the indexing functions of the RHS variable and the LHS variable in $pair9$ are the same. Because of this we may do further optimization by replacing the array $BC2$ with a scalar variable. In general, such optimization is not always possible because for other pairs the indexing functions may be different.

2.1.3 Generating Dynamic Single Assignment Code

Our procedure to generate the dSAC is an extended/modified version of the procedure implemented in MATPARSER [52]. The main extensions/modifications we make are:

- we replace the classical array dataflow analysis with the more advanced FADA analysis outlined in Section 2.1.1;
- we add and use our procedure, described in Section 2.1.2, for dynamic change of parameters introduced by FADA in order to make the functionality of the dSAC equivalent to the original WDP;
- the data dependent "if"-conditions are evaluated in separate functions in the dSAC. The outcome of these functions is substituted back as conditions in the "if"- constructs.

We explain the effect of the extensions/modifications described above by the dSAC shown in Figure 2.4. This dSAC corresponds to the program shown in Figure 2.1. First, because we use FADA for dependence analysis, two additional parameters appear in the dSAC that were not present in the original WDP. These parameters are u and c. The code lines that use these parameters are generated based on FADA as well. For example, consider the lines 76 to 85 in the dSAC. They are generated in accordance with the result obtained from our FADA example shown in Section 2.1.1 - see Equation (2.5).

Second, in order to keep the functionality of the dSAC equivalent to the functionality of the original WDP, the values of the parameters u and c have to be changed dynamically. Therefore, our procedure to accomplish the dynamic change introduces for every parameter an array of control variables that store the correct values of the parameter for every iteration. The dynamic change of the value of u is accomplished by the code lines 41 and 44 using the array $BC1$. The dynamic change of the value of c is accomplished by the code lines 67 and 75 using the array $BC2$. The two arrays $BC1$ and $BC2$ are initialized at the beginning of the dSAC. The initialization code for $BC1$ and $BC2$ is shown in lines 3-5 and 7-9, respectively.

Third, we explain how the data dependent conditions in the original WDP are represented in the dSAC. As an example we use the data dependent condition in line 21 of the original WDP shown in Figure 2.1. The corresponding code in the dSAC is given in Figure 2.4-lines 44 to 58. The data dependent condition $x(i) * x(i) > 100$ is transformed to $x(i) * x(i) - 100 > 0$ by moving all the terms from right to left. The expression $x(i) * x(i) - 100$ is computed in the dSAC by the function $if\_x(in\_0)$ in line 55 and the output of this function is checked in line 58. Lines 44-53 assign the correct value of $x(i)$ to in_0. In general, every data dependent condition is encapsulated and computed in a separate function in the dSAC.

Finally, the dSAC we generate contains functions called **ipd** and **opd**. These functions just propagate the value of its input to its output and they play a role in the conversion of a dSAC into an approximated dependence graph (ADG) presented in the next section.

```
%parameter N 8 16;

for i = 1:1:N,
  BC1(i) = N+1;
end

for i = 1:1:N+1,
  BC2(i) = N+1;
end

for i = 1 : 1 : N,
  [ out_0 ] = _Source_x( );
  [ x_1( i) ]  = opd( out_0 );
end

for i = 1 : 1 : N+1,
  [ out_0, out_1 ] = _Source_yt( );
  [ y_1( i) ]  = opd( out_0 );
  [ t_1( i) ]  = opd( out_1 );
end

for i = 1 : 1 : N+2,
  [ out_0 ] = _Source_z( );
  [ z_1( i) ]  = opd( out_0 );
end

for i = 1 : 1 : N,
 if i-3>= 0,
   [ in_0 ] = ipd( z_2( i-2 ) );
 else %% if -i+2 >= 0
   [ in_0 ] = ipd( z_1( i ) );
 end
 [ out_0 ] = if_z( in_0 );
 [ cond_1(i) ] = opd( out_0 );

 [ cond_1(i) ] = ipd( cond_1(i) );
 if cond_1(i) = 0,
   [ in_0 ] = ipd( x_1( i ) );
   [ out_0 ] = F1( in_0 );
   [ x_2( i) ]  = opd( out_0 );
   [ BC1(i) ] = opd( i );
 end

 [ u ] = ipd( BC1(i) );
 if -u+i>= 0,
   if u-i>= 0,
      [ in_0 ] = ipd( x_2( u ) );
   else %% if -u+i-1 >= 0
      [ in_0 ] = ipd( x_1( i ) );
   end
 else %% if u-i-1 >= 0
   [ in_0 ] = ipd( x_1( i ) );
 end
 [ out_0 ] = if_x( in_0 );
 [ cond_2(i) ] = opd( out_0 );

 [ cond_2(i) ] = ipd( cond_2(i) );
 if cond_2(i) > 0,
  if i-2>= 0,
    [ in_0 ] = ipd( y_3( i-1 ) );
  else %% if -i+1 >= 0
    [ in_0 ] = ipd( y_1( i ) );
  end

  [ out_0 ] = F2( in_0 );
  [ y_2( i) ]  = opd( out_0 );
  [ BC2(i+1) ] = opd( i );

  [ in_0 ] = ipd( t_1( i ) );

  [ out_0 ] = F3( in_0 );
  [ t_2( i) ]  = opd( out_0 );
 end

 [ c ] = ipd( BC2(i+1) );
 if -c+i>= 0,
   if c-i>= 0,
      [ in_0 ] = ipd( y_2( c ) );
   else %% if -c+i-1 >= 0
      [ in_0 ] = ipd( y_1( i+1 ) );
   end
 else %% if c-i-1 >= 0
   [ in_0 ] = ipd( y_1( i+1 ) );
 end

 if i-3>= 0,
   [ in_1 ] = ipd( z_2( i-2 ) );
 else %% if -i+2 >= 0
   [ in_1 ] = ipd( z_1( i ) );
 end
 [ out_0, out_1 ] = F4( in_0, in_1 );
 [ y_3( i) ]  = opd( out_0 );
 [ z_2( i) ]  = opd( out_1 );

end
```

Figure 2.4: Dynamic Single Assignment Code derived from the Weakly Dynamic Program shown in Figure 2.1.

2.2 Approximated Dependence Graph

In general, our dSAC notion presented in the previous section is a very complex data model to operate on especially when formal operations have to be defined and applied. Therefore, we have developed a more compact model called approximated dependence graph (ADG). The ADG captures, in a formal way, the information related to the data dependencies between the function calls in the dSAC.

2.2.1 Definitions

Below, we define our approximated dependence graph (ADG) model.

Definition 2.2.1 (approximated dependence graph)
An Approximated Dependence Graph (ADG) is given by a tuple $ADG = (Nodes, Edges)$, where

- $Nodes = \{N_i \mid i = 1,\ 2,\ ..., M\}$ is a set of nodes,
- $Edges = \{E_j \mid j = 1,\ 2,\ ..., P\}$ is a set of edges.

Definition 2.2.2 (node)
A node in the ADG is given by a tuple $N = (I_N, O_N, F_N, ND_N)$, where

- $I_N = \{p_k \mid k = 1,\ 2,\ ..., K\}$ is a set of input ports,
- $O_N = \{q_l \mid l = 1,\ 2,\ ..., L\}$ is a set of output ports,
- F_N is a tuple $F_N = (F, in, out)$, where in and out are sets of variables and $F : in \rightarrow out$ is a function,
- ND_N is the domain of N defined by a linearly bounded set (LBS - Definition 2.2.6).

Definition 2.2.3 (input port)
An input port is given by a tuple $p = (V_p, A_p, IPD_p)$, where

- V_p is an n-dimensional variable associated with the port,
- A_p is a variable that:
  - binds the port to the function in the node to which the port belongs if $A_p \in in$;
  - binds the port to filtering functions of a LBS (Definition 2.2.6) which defines the domain (ND_N) of the node to which the port belongs if $A_p \notin in \wedge A_p \equiv V_p$;
  - binds the port to filtering functions of LBSs which define the domains (IPDs) of other input ports if $A_p \notin in \wedge A_p \not\equiv V_p$;
- IPD_p is the domain of p defined by a LBS (Definition 2.2.6).

Definition 2.2.4 (output port)
An output port is given by a tuple $q = (V_q, A_q, OPD_q)$, where

- V_q is an m-dimensional variable associated with the port,
- A_q is a variable that:
  - binds the port to the function in the node to which the port belongs if $A_q \in out$;
  - is an integral iterator vector I pointing a point in the port domain (OPD_q) if $A_q \notin out$;

- OPD_q is the domain of q defined by a LBS (Definition 2.2.6).

Definition 2.2.5 (edge)
An edge in the ADG is a triple $E = (q, p, M)$, where

- $q = (V_q, A_q, OPD_q)$ is an output port,
- $p = (V_p, A_p, IPD_p)$ is an input port,
- $V_p = V_q$, i.e., variables V_p and V_q have same names and equal dimensions,
- $M : I_p \rightarrow I_q$ is an affine mapping where $I_p \in IPD_p$ and $I_q \in OPD_q$.

Definition 2.2.6 (linearly bounded set)
Let be given four sets of functions
$S1 = \{f_x^1(I) \mid x = 1..|S1|,\ I \in \mathbb{Z}^n\},\ S2 = \{f_x^2(I) \mid x = 1..|S2|,\ I \in \mathbb{Z}^n\}$
$S3 = \{f_x^3(I) \mid x = 1..|S3|,\ I \in \mathbb{Z}^n\},\ S4 = \{f_x^4(I) \mid x = 1..|S4|,\ I \in \mathbb{Z}^n\},$
an integral $m \times n$ matrix A and an integral n-vector b. A *linearly bounded set* (LBS) is a set of points

$$
\begin{aligned}
LBS = \{\ I \in \mathbb{Z}^n \mid A.I \geq b,\ & \\
& if\ S1 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S1|},\ f_x^1(I) \geq 0, \\
& if\ S2 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S2|},\ f_x^2(I) \leq 0, \\
& if\ S3 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S3|},\ f_x^3(I) > 0, \\
& if\ S4 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S4|},\ f_x^4(I) < 0\ \}.
\end{aligned}
$$

The set of points $B = \{\ I \in \mathbb{Z}^n \mid A.I \geq b\ \}$ is called *linear bound* of the LBS and the set of functions $S = S1 \cup S2 \cup S3 \cup S4$ is called *filtering set*. The functions $f_x^j(I) \in S$ are called *filtering functions*. Every $f_x^j(I) \in S$ can be an arbitrary function of I.

2.2.2 Deriving ADG from dSAC

This section deals with the conversion of a dynamic single assignment code into the approximated dependence graph defined in Section 2.2.1. The relation between the dSAC and the ADG is the following:

- for every function call $[\ arg\ ] = \mathbf{ipd}(\ var\ )$ in the dSAC there exists a corresponding input port $p = (V_p, A_p, IPD_p)$ in the AGD, where $V_p = var$, $A_p = arg$ and IPD_p is the set of iterations in which the function **ipd** is executed. The function **ipd** is an identity function;
- for every function call $[\ var\ ] = \mathbf{opd}(\ arg\ )$ in the dSAC there exists a corresponding output port $q = (V_q, A_q, OPD_q)$ in the AGD, where $V_q = var$, $A_q = arg$ and OPD_q is the set of iterations in which the function **opd** is executed. The function **opd** is an identity function;
- for every pair $< [\ arg_i\ ] = \mathbf{ipd}(\ var_i\ )\ , [\ var_o\ ] = \mathbf{opd}(\ arg_o\ ) >$ in the dSAC there exists an edge $E = (q, p, M)$ in the ADG if var_i and var_o have the same name and dimension. q is the output port in the ADG corresponding to **opd** and p is the input

port in the ADG corresponding to **ipd**. M is an affine mapping that relates iterations in which the **ipd** consumes a value to the corresponding iterations in which this value is produced by **opd**;

- for every function call $[\ oArguments\ ] = \langle \mathbf{name} \rangle (\ iArguments\ )$ in the dSAC with $\langle \mathbf{name} \rangle$ different than **ipd** or **opd** there exists a corresponding node $N = (I_N, O_N, F_N, ND_N) = (I_N, O_N, (F, in, out), ND_N)$ in the ADG, where:

  For every function $[\ arg\ ] = \mathbf{ipd}(\ var\ )$ in the dSAC if $arg \in iArguments$ then the corresponding input port p in the ADG belongs to I_N. For every function $[\ var\ ] = \mathbf{opd}(\ arg\ )$ in the dSAC if $arg \in oArguments$ then the corresponding output port q in the ADG belongs to O_N. The elements of F_N are related to the dSAC as follows: $F = \langle \mathbf{name} \rangle$, $in \equiv iArguments$ and $out \equiv oArguments$. ND_N is the set of iterations in which the function $\langle \mathbf{name} \rangle$ is executed.

The conversion of a dSAC to an ADG is done in two steps: 1) the dSAC is converted to a syntax tree [54]; 2) the syntax tree is parsed and all the elements of the ADG are created and specified in accordance with the relations given above. The syntax tree of the dSAC is similar to the parse tree defined in [19]. The difference is that the expressions in the syntax tree of a dSAC are not limited to affine expressions of loop iterators. In the dSAC the expressions can be an arbitrary functions of loop iterators and/or data variables. For more details about the parse tree and how it can be created we refer the reader to [19] [52].

2.2.3 Examples

Let us consider the dynamic single assignment code shown in Figure 2.4. The corresponding approximated dependence graph is shown in Figure 2.5. It consists of 9 nodes and 19 edges. Below, we show examples of how these nodes and edges are constructed in accordance with the definitions given in Section 2.2.1.

Example of Definition 2.2.1:
The ADG in Figure 2.5 is given by the tuple $ADG = (Nodes, Edges)$, where

- $Nodes = \{N1, N2, N3, N4, N5, N6, N7, N8, N9\}$ is the set of nodes,
- $Edges = \{ED1, ED2, ED3, ED4, ED5, ED6, ED7, ED8, ED9, ED10,$
  $ED11, ED12, ED13, ED14, ED15, ED16, ED17, ED18, ED19\}$ is the set of edges.

Every node in the ADG has a corresponding function call in the dSAC. The function call is shown in round brackets below the name of the node in Figure 2.5. Every edge in the ADG corresponds to a variable in the dSAC. The variable is shown in round brackets next to the name of the edge.

Example of Definition 2.2.2:
Consider node $N7$ in the ADG shown in Figure 2.5. $N7$ corresponds to function call $F2$ in the dSAC. This node is given by the tuple $N7 = (I_{N7}, O_{N7}, F_{N7}, ND_{N7})$, where

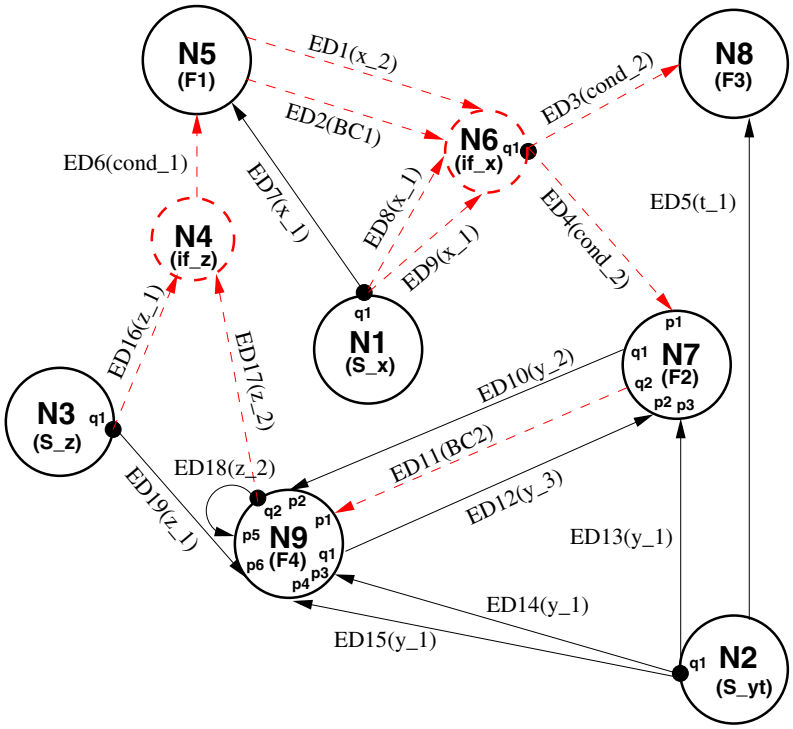

Figure 2.5: Approximated Dependence Graph derived from the dSAC shown in Figure 2.4.

- $I_{N7} = \{p1, p2, p3\}$ is the set of input ports,
- $O_{N7} = \{q1, q2\}$ is the set of output ports,
- $F_{N7} = (F2, \{in\_0\}, \{out\_0\})$, where $F2 : \{in\_0\} \rightarrow \{out\_0\}$,
- ND_{N7} is the domain of $N7$ defined by the linearly bounded set $LBS = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0 \,\}$.

  ND_{N7} represents the iterations i at which function call $F2$ is executed in the dSAC. These iterations are determined by code lines 1, 27, and 58 of the dSAC in Figure 2.4. The linear bound of the LBS is determined by code lines 1 and 27 and it is the set of points $B = \{\, I \in \mathbb{Z}^n \mid A.I \geq b \,\} = \{\, i \in \mathbb{Z}^1 \mid \begin{bmatrix} 1 \\ -1 \end{bmatrix} .i \geq \begin{bmatrix} 1 \\ -N \end{bmatrix} \wedge 8 \leq N \leq 16 \,\} = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16 \,\}$. The filtering set of the LBS is determined by the "if"-condition in code line 58 thus this set is $S = S1 \cup S2 \cup S3 \cup S4 = \emptyset \cup \emptyset \cup \{\, cond\_2(i) \,\} \cup \emptyset$.

  The exact iterations i at which function call $F2$ is executed are not known at compile time because of the dynamic (data-dependent) condition at code line 58 in the dSAC (Figure 2.4). That is why we introduce the notion of linearly bounded set (see Definition 2.2.6) by which we approximate the unknown iterations i. The linear bound B of

the LBS defining ND_{N7} captures the information that we know at compile time about the bounds of the iterations i. The variable $cond\_2(i)$ in code line 58 is interpreted as an unknown function of i called filtering function whose output is determined at run time.

Examples of Definition 2.2.3:
Consider the input port $p2$ of node $N7$ in the ADG shown in Figure 2.5. $p2$ corresponds to function call $[\ in\_0\ ] = \mathbf{ipd}(\ y\_3(i-1)\ )$ at line 60 in the dSAC. The input port $p2$ is given by the tuple $p2 = (V_{p2}, A_{p2}, IPD_{p2})$, where

- $V_{p2} = y\_3(i-1)$ is the variable associated with the port,
- $A_{p2} = in\_0$ is the variable binding the port to the function $F2$,
- IPD_{p2} is the domain of $p2$ defined by the linearly bounded set $LBS = \{\ i_{p2} \in \mathbb{Z} \mid 2 \leq i_{p2} \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i_{p2}) > 0\ \}$. This LBS approximates the iterations at which the function call $[\ in\_0\ ] = \mathbf{ipd}(\ y\_3(i-1)\ )$ is executed in the dSAC.

Consider the input port $p1$ of node $N7$ in the ADG shown in Figure 2.5. $p1$ corresponds to function call $[\ cond\_2(i)\ ] = \mathbf{ipd}(\ cond\_2(i)\ )$ at line 57 in the dSAC. The input port $p1$ is given by the tuple $p1 = (V_{p1}, A_{p1}, IPD_{p1})$, where

- $V_{p1} = cond\_2(i)$ is the variable associated with the port,
- $A_{p1} = cond\_2(i)$ is the variable binding the port to the filtering function $cond\_2(i)$ which defines the domain of node $N7$ - see Example of Definition 2.2.2 given above,
- IPD_{p1} is the domain of $p1$ defined by the linearly bounded set $LBS = \{\ i_{p1} \in \mathbb{Z} \mid 1 \leq i_{p1} \leq N \wedge 8 \leq N \leq 16\}$. This LBS defines the iterations at which the function call $[\ cond\_2(i)\ ] = \mathbf{ipd}(\ cond\_2(i)\ )$ is executed in the dSAC. Notice that these iterations are known at compile time. Therefore, in this case the LBS is actually a polytope.

Consider the input port $p2$ of node $N9$ in the ADG shown in Figure 2.5. $p2$ corresponds to function call $[\ in\_0\ ] = \mathbf{ipd}(\ y\_2(c)\ )$ at line 78 in the dSAC. The input port $p2$ is given by the tuple $p2 = (V_{p2}, A_{p2}, IPD_{p2})$, where

- $V_{p2} = y\_2(c)$ is the variable associated with the port,
- $A_{p2} = in\_0$ is the variable binding the port to the function $F4$,
- IPD_{p2} is the domain of $p2$ defined by the linearly bounded set $LBS = \{\ (i_{p2}, c) \in \mathbb{Z}^2 \mid 1 \leq i_{p2} \leq N \wedge 8 \leq N \leq 16,\ -c + i_{p2} \geq 0,\ c - i_{p2} \geq 0\ \}$. This LBS approximates the iterations at which function call $[\ in\_0\ ] = \mathbf{ipd}(\ y\_2(c)\ )$ is executed in the dSAC because the values of c are not known at compile time. NOTE: The LBS has two filtering functions, namely $f_1^1(i_{p2}, c) = -c + i_{p2}$ and $f_2^1(i_{p2}, c) = c - i_{p2}$ that depend on c which is to be determined at run time. These functions are binded via c to the input port described below.

Consider the input port $p1$ of node $N9$ in the ADG shown in Figure 2.5. $p1$ corresponds to function call $[\, c \,] = \mathbf{ipd}(\, BC2(i+1) \,)$ at line 75 in the dSAC. The input port $p1$ is given by the tuple $p1 = (V_{p1}, A_{p1}, IPD_{p1})$, where

- $V_{p1} = BC2(i+1)$ is the variable associated with the port,
- $A_{p1} = c$ is the variable binding this port ($p1$) to the filtering functions $f_1^1(i_{p2}, c)$ and $f_2^1(i_{p2}, c)$ that define the domain (IPD_{p2}) of port $p2$ in node $N9$ - see the example of port $p2$ given above,
- IPD_{p1} is the input port domain of $p1$ defined by the linearly bounded set $LBS = \{\, i_{p1} \in \mathbb{Z} \,|\, 1 \leq i_{p1} \leq N \wedge 8 \leq N \leq 16\}$.

Examples of Definition 2.2.4:
Consider the output port $q1$ of node $N9$ in the ADG shown in Figure 2.5. $q1$ corresponds to function call $[\, y\_3(i) \,] = \mathbf{opd}(\, out\_0 \,)$ at line 92 in the dSAC. The output port $q1$ is given by the tuple $q1 = (V_{q1}, A_{q1}, OPD_{q1})$, where

- $V_{q1} = y\_3(i)$ is the variable associated with the port,
- $A_{q1} = out\_0$ is the variable binding the port to the function $F4$,
- OPD_{q1} is the domain of $q1$ defined by the linearly bounded set $LBS = \{\, i_{q1} \in \mathbb{Z} \,|\, 1 \leq i_{q1} \leq N \wedge 8 \leq N \leq 16 \,\}$.
  NOTE: In this case the LBS is a polytope that defines exactly at which iterations the function call $[\, y\_3(i) \,] = \mathbf{opd}(\, out\_0 \,)$ is executed in the dSAC.

Consider the output port $q2$ of node $N7$ in the ADG shown in Figure 2.5. $q2$ corresponds to function call $[\, BC2(i+1) \,] = \mathbf{opd}(\, i \,)$ at line 67 in the dSAC. The output port $q2$ is given by the tuple $q2 = (V_{q2}, A_{q2}, OPD_{q2})$, where

- $V_{q2} = BC2(i+1)$ is the variable associated with the port,
- $A_{q2} = i$ is the iterator vector pointing to the points in OPD_{q2}, i.e., the vector that consists of the iterators of the loops surrounding the function call $[\, BC2(i+1) \,] = \mathbf{opd}(\, i \,)$,
- OPD_{q2} is the domain of $q2$ defined by the linearly bounded set $LBS = \{\, i_{q2} \in \mathbb{Z} \,|\, 1 \leq i_{q2} \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i_{q2}) > 0\}$.
  NOTE: In this case the LBS approximates the iterations at which function call $[\, BC2(i+1) \,] = \mathbf{opd}(\, i \,)$ is executed in the dSAC.

Example of Definition 2.2.5:
Consider edge $ED12$ in the ADG shown in Figure 2.5. $ED12$ corresponds to the variable y_3 that appears in function calls $[\, in\_0 \,] = \mathbf{ipd}(\, y\_3(i-1) \,)$ and $[\, y\_3(i) \,] = \mathbf{opd}(\, out\_0 \,)$ at lines 60 and 92 in the dSAC. The edge $ED12$ is given by the triple $E12 = (q1, p2, M)$, where

- $q1 = (\ y\_3(i),\ out\_0,\ \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N \wedge 8 \leq N \leq 16\ \}\ )$ is the output port,
- $p2 = (\ y\_3(i-1),\ in\_0,\ \{\ i_{p2} \in \mathbb{Z} \mid 2 \leq i_{p2} \leq N \wedge 8 \leq N \leq 16,\ cond2(i_{p2}) > 0\ \}\ )$ is the input port,
- the affine mapping M is given by $i_{q1} = i_{p2} - 1$.

2.3 Schedule Tree

The ADG model presented in the previous section does not capture the information about the execution order between the function calls that is available in the dSAC. This information is captured in a formal and compact way by our schedule tree (STree) model described in this section.

2.3.1 Definition

Below, we give the definition of our STree model.

Definition 2.3.1 (schedule tree)
A Schedule Tree $STree = (N_S, E_S)$ is a syntax tree $Tree = (N, E)$ [54], derived from a dSAC, where

- the set of nodes $N_S \subset N$,
- the set of edges $E_S \subset E$,
- the topology of the $STree$ has to represent a control structure of a program that executes the function calls $[\ ] = \langle \mathbf{name} \rangle (\ )$ of the dSAC in a correct order. $\langle \mathbf{name} \rangle$ is different from **ipd** and **opd**.

2.3.2 Deriving STree from dSAC

The procedure to obtain the STree from the dSAC is done in two steps: 1) the dSAC is converted to a syntax tree using a standard syntax parser [54]; 2) the STree is extracted from the syntax tree. According to Definition 2.3.1 the schedule tree (STree) is a pruned version of the syntax tree derived from the dSAC. The pruning procedure is given by the pseudo code below:

```
for all leafNode_j ∈ N do
  if leafNode_j ≠ ipd ∧ leafNode_j ≠ opd then
    MARK all nodes from leafNode_j to rootNode;
  endif
endfor
```

```
for all Node_j ∈ N do
  if Node_j is not marked then
    REMOVE Node_j;
  endif
endfor
```

First, this procedure visits every leaf node in the syntax tree and checks whether the leaf node represents a function call of the dSAC with name different from **ipd** and **opd**. If this is true then all the nodes in the path from the leaf node to the root node in the syntax tree are marked. After all the leaf nodes are visited, the marked syntax tree is traversed and all the nodes that are not marked are removed.

2.3.3 Example

Consider the dynamic single assignment code shown in Figure 2.4. The corresponding schedule tree (STree) is depicted in Figure 2.6. If we parse this tree top-down from left to right a

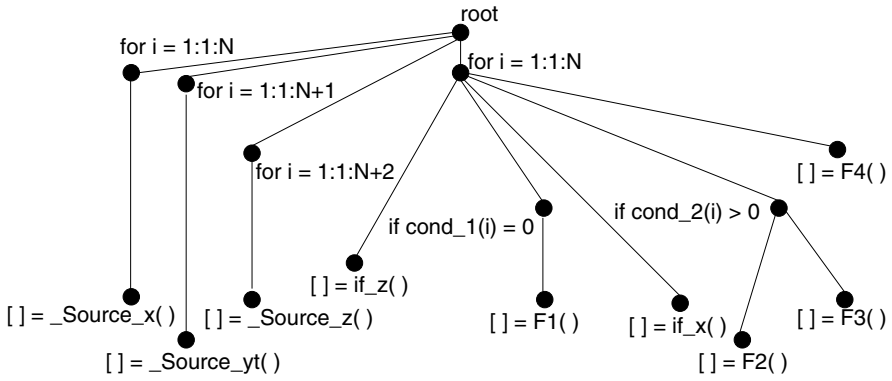

Figure 2.6: Schedule Tree derived from the dSAC shown in Figure 2.4.

program code can be generated that gives a valid sequential execution order (global sequential schedule) among function calls $\_Source\_x$, $\_Source\_yt$, $\_Source\_z$, if_z, if_x, $F1$, $F2$, $F3$, and $F4$. This sequential order is the original order given by the dSAC.

2.4 Process Network Synthesis

In this section, we present the final step of our approach in which we synthesize a process network - see STEP3 in Figure 1.5. Our synthesis approach is mainly a translation of the ADG model and STree model into a process network (PN) model. The structure of our PN model is defined in Section 2.4.3. The basic synthesis flow in STEP3 is depicted in Figure 2.7. The starting point for the synthesis is the information captured in the ADG and the STree.

This information is enough to generate a set of process networks with different topologies and degree of exploited parallelism. The dashed box in Figure 2.7 selects one possible topology of a network that has to be synthesized.

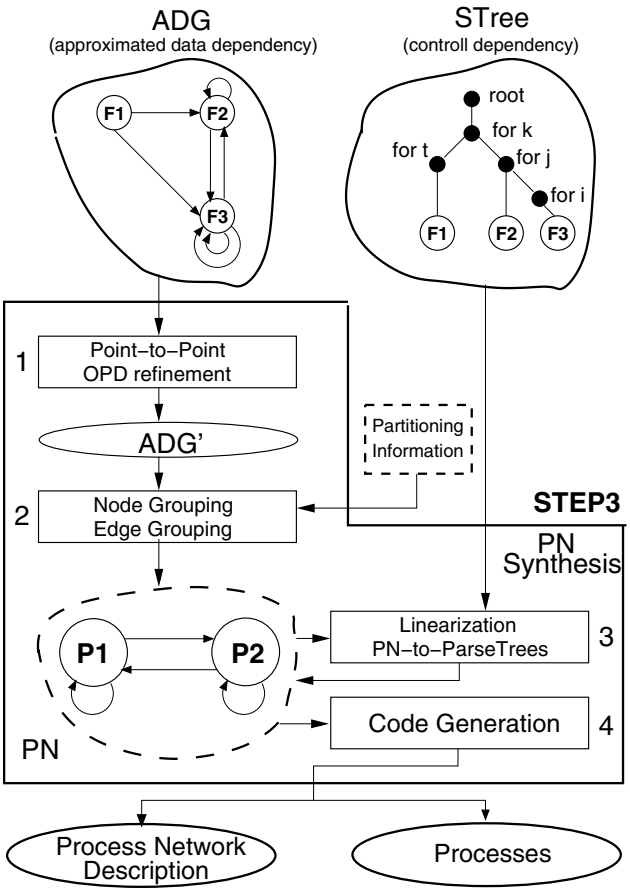

Figure 2.7: The Process Network Synthesis step in COMPAANDYN consists of four sub-steps.

The process network synthesis consists of four steps as shown in Figure 2.7. In step 1, we apply two transformations on the ADG, namely *Point-to-Point* and *OPD refinement*. This transformations have to be done because of two reasons:

1. On the one hand, our synthesis approach translates an ADG model into a PN model, where there is one-to-one correspondence between the input ports of the ADG and the input ports of the PN. The same is true for the output ports. The ADG may have several input ports connected to a single output port. On the other hand, we focus on the synthesis of a special class of process networks: Kahn Process Networks, where every input port is connected to only one unique output port. This fact requires that

we have to change the topology of the ADG such that every input is connected to only one unique output port, i.e., *point-to-point* connection. The change of the topology is always possible without changing the semantics of the ADG model. For more details see the transformation *Point-to-Point* presented in Section 2.4.2;

2. Every output port in the PN has to send a token in its corresponding output channel if it is possible that the token will be needed in the process that reads this channel. This is accomplished by the transformation *OPD refinement* presented in Section 2.4.2. This transformation is applied on every OPD that is associated with an output port in the ADG before the ADG model is translated to the PN model.

In steps 2 and 3 (Figure 2.7), the process network model (PN) is created gradually by creating the topology of the PN - step 2, followed by creating the behavior of the PN - step 3. The topology of the process network is created by grouping nodes and edges of the ADG into processes and channels in the PN. The grouping is based on the partitioning information delivered by the dashed box in Figure 2.7. The grouping procedures are given in Section 2.4.4.

In step 3 in Figure 2.7 the behavior of the PN is created. The procedures that create the behavior are presented in Section 2.4.5. These procedures operate on the PN model and use the information captured in the STree. A procedure called *Linearization* deals with the communication behavior of every process in the PN. This procedure extends the research work presented in [21] [22]. The procedure is built depending on the target class of process networks under synthesis - in our case Kahn Process Networks (KPN). In a KPN the processes communicate with each other over 1-dimensional unbounded FIFO channels. In Section 2.4.5 we define four possible ways to model this communication. Also, we give a procedure that selects for every channel the optimal communication model that preserves the correct behavior of the PN. The information needed to build this model is derived by the *Linearization* procedure as well.

Internally, a process in a KPN has by definition [10] a sequential behavior. This means that the function calls that have to be executed inside a process are executed in a sequential order. The procedure *PN-to-ParseTree*, given in Section 2.4.5, derives this order such that the PN execution is deadlock free and expresses it as a parse tree for every process in the PN.

The last step of the PN synthesis (Figure 2.7 - step 4) is called *Code Generation*. In this step an executable code of a Kahn process network is generated from the PN model. We use a software engineering technique called *Visitor* [55] to visit the PN model structure and to generate the executable code. This code can be expressed in any programming language on top of which an environment to execute Kahn process networks is built. For example, the YAPI environment [45] in C++, or the PtolemyII framework [12] in Java, or SystemC. The *Code Generation* step is presented in Section 2.4.6 by an example.

2.4.1 Notations

In the rest of the sections in this chapter, the following notations apply. Let $G = (S, T) = (\{s_j | j = 1..|S|\}, (E1, E2))$ be given. G is called a tuple with elements: set S and tuple T. The number of the elements of a given set S we denote by $|S|$. Adding a new element e to a

given set S is denoted by S.ADD(e). An empty set is denoted by $\emptyset$. The symbol "." is used to point hierarchically to an element of a given tuple. For example, if the tuple G is given then $G.T.E1$ points to the element $E1$. The symbol "()" is used to point hierarchically to the tuples to which a given element belongs. For example, if the element $E1$ is given then $(E1)^T$ points to the tuple T and $((E1)^T)^G$ points to the tuple G. Also, the combination of "." and "( )" is used. For example, if $E1$ is given then $(E1)^T.E2$ points to the element $E2$ of T. The symbol "$\perp$" has to be read as *not defined*. The symbol "$\leftarrow$" has to be read as *gets*. The symbol "||" denotes concatenation. The text followed by the symbol "$\triangleright$" has to be interpreted as a comment line or auxiliary explanation.

2.4.2 ADG transformations

Point-to-Point

The *Point-to-Point* transformation transforms an ADG such that every output port is associated with at most one edge as follows. Let an $ADG = (Nodes, Edges)$ be given. For every edge $E_i = (q_l, p_k, M) \in Edges$ create a new port $q_{l\_i} = q_l$, add this port to the node that contains q_l and modify E_i such that $E_i = (q_{l\_i}, p_k, M)$. Finally, for every node $N \in Nodes$ remove the original output ports q_l from O_N.

Example: Figure 2.8 illustrates the *point-to-point* transformation applied on port q_1 of node $N2$ in the ADG shown in Figure 2.5. Three edges start from port q_1. After the transformation node $N2$ has three output ports q_{1_13}, q_{1_14}, q_{1_15}, where
$q_{1\_13} = q_{1\_14} = q_{1\_15} = q_1 = (\ y\_1(i),\ out\_0,\ \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N + 1 \wedge 8 \leq N \leq 16\ \})$.

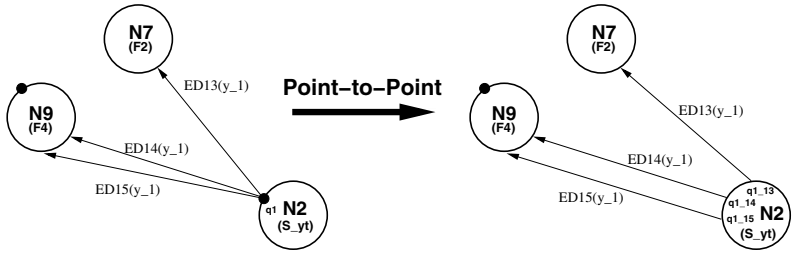

Figure 2.8: Example of Point-to-Point transformation for a part of the ADG shown in Figure 2.5.

OPD refinement

Let an $ADG = (Nodes, Edges)$ be given. According to Definition 2.2.5 in Section 2.2 every edge $E \in Edges$ is given by $E = ((V_q, A_q, OPD_q), (V_p, A_p, IPD_p), M)$, where OPD_q and IPD_p are linearly bounded sets with linear bounds $B_q = \{\ i_q \in \mathbb{Z}^n \mid A_q.i_q \geq b_q\ \}$

and $B_p = \{\ i_p \in \mathbb{Z}^m \mid A_p.i_p \geq b_p\ \}$, respectively. The transformation *OPD refinement* derives for every $E \in Edges$ a new linearly bounded set $OPD_q^{'} = OPD_q \cap M(B_p)$, where $M(B_p)$ is the image of B_p in $\mathbb{Z}^n$ defined by the affine mapping M. In order to find the image $M(B_p)$ we use the procedure described in [56]. Also, the Ehrhart test described in [19] can be used.

Example of OPD refinement:
Consider edge $ED12$ in the ADG shown in Figure 2.5. The edge $ED12$ is given by the triple $E12 = (q1, p2, M)$, where

- $q2 = (\ y\_3(i),\ out\_0,\ \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N \wedge 8 \leq N \leq 16\ \}\ )$ is the output port,
- $p2 = (\ y\_3(i-1),\ in\_0,\ \{\ i_{p2} \in \mathbb{Z} \mid 2 \leq i_{p2} \leq N \wedge 8 \leq N \leq 16, cond\_2(i_{p2}) > 0\ \}\ )$ is the input port,
- the affine mapping M is given by $i_{q1} = i_{p2} - 1$.

The OPD refinement of $OPD_{q2} = \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N \wedge 8 \leq N \leq 16\ \}$ using the procedure described in [56] gives the new $OPD_{q1}^{'} = \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N - 1 \wedge 8 \leq N \leq 16\ \}$. The Ehrhart test procedure described in [19] gives $OPD_{q1}^{'} = \{\ i_{q1} \in \mathbb{Z} \mid 1 \leq i_{q1} \leq N \wedge 8 \leq N \leq 16,\ Multiplicity(i_{q1}, N) - 1 \geq 0\ \}$, where

$$Multiplicity(i_{q1}, N) = \begin{cases} 1 & \text{if } 1 \leq i_{q1} \leq N-1 \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

2.4.3 The Process Network (PN) model

In this section we define our process network model.

Definition 2.4.1 (process network)
A process network (PN) is given by a tuple $PN = (P, C)$, where

- $P = \{P_i \mid i = 1,\ 2,\ ..., |P|\}$ is a set of processes,
- $C = \{C_j \mid j = 1,\ 2,\ ..., |C|\}$ is a set of channels.

Definition 2.4.2 (process)
A process in the PN is given by a tuple $P = (NP, IP, OP, ST)$, where

- $NP = \{N_m \mid m = 1,\ 2,\ ..., M\}$ is a set of nodes, where N_m is given by Definition 2.2.2,
- $IP = \{IG_k \mid k = 1,\ 2,\ ..., K\}$ is a set of input gates,
- $OP = \{OG_l \mid l = 1,\ 2,\ ..., L\}$ is a set of output gates,

- ST is a schedule tree that gives a valid execution order between the functions F associated with every N_m.

Definition 2.4.3 (input gate)
An input gate is given by a tuple $IG = (I_{IG}, IK_{IG})$, where

- $I_{IG} = \{p_k \mid k = 1,\ 2,\ ...,|I_{IG}|\}$ is a set of ports, where p_k is given by Definition 2.2.3,
- $IK_{IG} = \{InKey_{p_k} \mid k = 1,\ 2,\ ...,|I_{IG}|\}$ is a set of functions, where for every $p_k \in I_{IG}$ a function $InKey_{p_k} \in IK_{IG}$ is associated, $InKey_{p_k} : IPD_{p_k} \rightarrow \mathbb{Z}^1$.

  The functions $InKey_{p_k}$ are used for color (tag) matching when the input gate IG is connected to a channel that communicates colored (tagged) data. Also, these functions are used when the communicated data has to be re-ordered. The re-ordering is done by a special controller and a re-ordering memory located in the process to which IG belongs. The functions $InKey_{p_k}$ are used as address generators for the re-ordering memory. More details about when and why coloring and/or re-ordering of data is needed as well as how the functions $InKey_{p_k}$ are determined are given in Section 2.4.5 - linearization.

Definition 2.4.4 (output gate)
An output gate is given by a tuple $OG = (O_{OG}, OK_{IG})$, where

- $O_{OG} = \{q_k \mid k = 1,\ 2,\ ...,|O_{OG}|\}$ is a set of ports, where q_k is given by Definition 2.2.4,
- $OK_{OG} = \{OutKey_{q_k} \mid k = 1,\ 2,\ ...,|O_{OG}|\}$ is a set of functions, where for every $q_k \in O_{OG}$ a function $OutKey_{q_k} \in OK_{OG}$ is associated, $OutKey_{q_k} : OPD_{q_k} \rightarrow \mathbb{Z}^1$ is a one-to-one mapping.

  Every function $OutKey_{q_k}$ is used to generate unique integer numbers that are attached as colors (tags) to every data sent via the corresponding port of output gate OG. This is done only when the output gate OG is connected to a channel that communicates colored (tagged) data. Also, functions $OutKey_{q_k}$ are used when the sent data via output gate OG has to be re-ordered at the other side of the communication channel where we have an input gate IG. In this case functions $OutKey_{q_k}$ are used to determine the functions $InKey_{p_k}$ (Definition 2.4.3) of the input gate IG. More details about when and why coloring and/or re-ordering of data is needed as well as how the functions $OutKey_{q_k}$ are determined are given in Section 2.4.5 - linearization.

Definition 2.4.5 (channel)
A channel is given by a tuple $C = (OG, IG, E, CM)$, where

- $OG = (O_{OG}, OK_{IG})$ is an output gate,
- $IG = (I_{IG}, IK_{IG})$ is an input gate,
- $E = \{E_m \mid m = 1,\ 2,\ ...,|E|\}$ is a set of edges, where E_m is given by Definition 2.2.5,

- $CM \in \{1, 2, 3, 4\}$ is the communication model of the channel, where

$$CM = \begin{cases} 1 & \text{in-order communication without coloring of tokens} \\ 2 & \text{out-of-order communication without coloring of tokens} \\ 3 & \text{in-order communication with coloring of tokens} \\ 4 & \text{out-of-order communication with coloring of tokens} \end{cases} \tag{2.7}$$

  In a Kahn Process Network the processes communicate data with each other over 1-dimensional unbounded FIFO channels. Our PN model supports four communication models that can realize such communication. CM defined above specifies one of the four possible communication models for a given channel C. More details about the communication models as well as why we need four models are given in Section 2.4.5 - linearization. Also, in this section, a procedure that selects for every channel C the optimal communication model that preserves the correct behavior of the PN is given.

2.4.4 Creating the PN topology

Let an empty PN model $PN = (P, C) = (\emptyset, \emptyset)$ be given. Creating the topology of the PN means that the set of processes P and the set of channels C have to be created by grouping nodes and edges of the ADG obtained after the transformations presented in Section 2.4.2. The following elements of every process $P_i \in P$ and every channel $C_j \in C$ have to be determined (see Section 2.4.3 for precise definitions):

- $P_i.NP$ - nodes of the ADG that are grouped in process P_i;
- $P_i.IP$ - the set of input gates of P_i;
- $P_i.OP$ - the set of output gates of P_i;
- $C_j.E$ - edges of the ADG that are grouped in channel C_j;
- $C_j.OG.O_{OG}$ - output ports of the ADG that are grouped in the output gate of C_j;
- $C_j.IG.I_{IG}$ - input ports of the ADG that are grouped in the input gate of C_j.

In order to create the PN topology, i.e., to determine the elements of the PN model described above we use the following information:

- The $ADG = (Nodes, Edges) = (\ \{N_i \mid i = 1..|Nodes|\}, \{E_j \mid j = 1..|Edges|\}\ )$ that is obtained after the transformations presented in Section 2.4.2.
- A set $SP = \{SP_i \mid i = 1..|SP|\}$, where
  1. $\forall_{i=1..|SP|} \Rightarrow SP_i \subseteq Nodes$;
  2. $\forall_{i \neq j} \Rightarrow SP_i \cap SP_j \equiv \emptyset$.

The set SP specifies how the nodes in the ADG have to be grouped in processes. $|SP|$ specifies the total number of processes that have to be created. Every element $SP_i \in SP$ specifies which nodes of the ADG form a process. SP_i can be an arbitrary set of ADG nodes satisfying condition 2) above.

The set SP can be given manually by a system designer who wants to generate a network with a particular topology or SP can be given by the dashed box shown in Figure 2.7. This box may implement some design space exploration procedures and/or some optimization procedures. If SP is not given then we use the following grouping to define a default SP: for every node in the ADG a process has to be created, i.e., every $SP_i \in SP$ consists of only one ADG node.

- A set $SC = \{SC_i \mid i = 1..|SC|\}$, where
  1. $\forall_{i=1..|SC|} \Rightarrow SC_i \subseteq Edges$;
  2. $\forall_{i \neq j} \Rightarrow SC_i \cap SC_j \equiv \emptyset$.
  3. in every set SC_i all the edges begin from nodes that belong to a particular set SP_q and all the edges end to nodes that belong to a particular set SP_t.

  The set SC specifies how the edges in the ADG have to be grouped in channels. $|SC|$ specifies the total number of channels that have to be created. Every element $SC_i \in SC$ specifies which edges of the ADG form a channel. Grouping ADG edges into PN channels has to be specified after the grouping of ADG nodes into PN processes is specified because of the restriction given in 3) above. This restriction limits the edge grouping as follows: All the edges that we want to group in a channel have to start from the same process, say P_q, and have to end at the same process, say P_t. The restriction given in 3) preserves one of the KPN semantics, i.e., only one process can write to a channel and only one process can read from a channel.

  The set SC can be given manually or by the dashed box shown in Figure 2.7. If SC is not given then we use the following grouping to define a default SC: for every edge in the ADG a channel has to be created, i.e., every $SC_i \in SC$ consists of only one ADG edge.

We start with the empty process network $PN = (P, C) = (\emptyset, \emptyset)$ and create the PN topology in accordance with the information described above. First, we execute a procedure called *Node Grouping* which creates the processes of the PN. Second, we execute a procedure called *Edge Grouping* which creates the channels of the PN and determines the connections between the processes (created by the node grouping) via these channels. Both procedures are defined and explained below.

Node Grouping

```
1 for all SP_i ∈ SP do
2   CREATE P_i = (NP, IP, OP, ST); ▷ new process is created
3   P_i.NP ← SP_i;
4   P_i.IP ← ∅;
```

```
5    P_i.OP ← ∅;
6    P_i.ST ← ⊥;
7    P.ADD(P_i); ▷ the process is added to the set of processes
8  endfor
```

The node grouping procedure given above uses the set SP to create the set of processes P and to determine which nodes of the ADG belong to every process. For every element $SP_i \in SP$ a process P_i is created. The ADG nodes specified by SP_i become the set of nodes $P_i.NP$ of the process P_i. The created process P_i is added to the set of processes P of the process network PN.

Edge Grouping

```
1  for all SC_i ∈ SC do
2    CREATE C_i = (OG, IG, E, CM); ▷ new channel is created
3    C_i.OG ← (∅, ⊥);
4    C_i.IG ← (∅, ⊥);
5    C_i.E ← SC_i;
6    C_i.CM ← ⊥;
7     for all E_m ∈ C_i.E do
8        C_i.IG.I_IG.ADD(E_m.p);
9        C_i.OG.I_OG.ADD(E_m.q);
10    endfor
11   ((E_1.p)^N)^P.IP.ADD(C_i.IG); ▷connect channel to process via input gate
12   ((E_1.q)^N)^P.OP.ADD(C_i.OG); ▷connect channel to process via output gate
13   C.ADD(C_i); ▷ the channel is added to the set of channels
14  endfor
```

The edge grouping procedure given above uses the set SC to create the set of channels C - see lines 2 and 13. Also this procedure determines:

- which edges of the ADG belong to every channel. This is done at line 5 where the set $C_i.E$ of every channel C_i is determined. This set gets all the edges that belong to set SC_i;
- which input ports and output ports of the ADG belong to the input gate and output gate of every channel. This is done at lines 7 to 10 where: 1) the input port of every edge that belongs to the channel C_i is added to the set $C_i.IG.I_{IG}$ - see line 8; 2) the output port of every edge that belongs to the channel C_i is added to the set $C_i.OG.I_{OG}$ - see line 9;
- to which processes the input and output gates of every channel belong. This is done at lines 11 and 12 where:
  1) the input gate $C_i.IG$ of every channel C_i is added to the set of input gates IP of process $((E_1.p)^N)^P$ - see line 11. Notice that this process is found by starting from the

input port p of the first edge E_1 of channel C_i and going up hierarchically by finding the node N to which the port p belongs followed by finding the process P to which the node N belongs;
2) the output gate $C_i.OG$ of every channel C_i is added to the set of output gates OP of process $((E_1.q)^N)^P$ - see line 12. Notice that this process is found by starting from the output port q of the first edge E_1 of channel C_i and going up hierarchically in the same way as described in 1) above.

Examples

Assume that the transformations presented in Section 2.4.2 have been applied on the ADG model shown in Figure 2.5. Here, we give, by an example, how this transformed ADG model is translated to a PN model by the node grouping and edge grouping discussed above. The grouping is specified by the sets SP and SC. Let us assume that the sets SP and SC are defined as follows:

$$SP = \{SP_i \mid i = 1..7\} = \\ = \{\ \{N1\}_1, \{\mathbf{N2}, \mathbf{N7}\}_{\mathbf{2}}, \{N3\}_3, \{\mathbf{N4}, \mathbf{N5}\}_4, \{N6\}_5, \{N8\}_6, \{N9\}_7\ \};$$

$$SC = \{SC_i \mid i = 1..17\} = \\ = \{\ \{ED1\}_1, \{ED2\}_2, \{ED3\}_3, \{ED4\}_4, \{ED5\}_5, \{ED6\}_6, \{ED7\}_7, \\ \{\mathbf{ED8}, \mathbf{ED9}\}_{\mathbf{8}}, \{ED10\}_9, \{ED11\}_{10}, \{ED12\}_{11}, \{ED13\}_{12}, \\ \{\mathbf{ED14}, \mathbf{ED15}\}_{\mathbf{13}}, \{ED16\}_{14}, \{ED17\}_{15}, \{ED18\}_{16}, \{ED19\}_{17}\ \ \}$$

The set SP specifies that a PN model has to be created that consists of 7 processes $P1$ to $P7$ because $|SP| = 7$. Process $P2$ has to be created by grouping nodes $N2$ and $N7$ of the ADG. Process $P4$ has to be created by grouping $N4$ and $N5$. The rest of the processes have to be created as follows: $P1$ takes $N1$, $P3$ takes $N3$, $P5$ takes $N6$, $P6$ takes $N8$, and $P7$ takes $N9$.

Similarly, the set SC specifies that the PN model consists of 17 channels. Channel $C8$ has to be created by grouping edges $ED8$ and $ED9$. Channel $C13$ groups edges $ED14$ and $ED15$. To the rest of the channels one edge each has to be assigned.

The topology of the PN model created after applying the grouping procedures, described in the previous section, is depicted in Figure 2.9. This topology is in accordance with the specification given by the sets SP and SC described above. The elements of this PN model are determined by the grouping procedures as well. The PN elements comply with the definitions given in Section 2.4.3. Below, we give some examples.

Example of Definition 2.4.1:
The process network in Figure 2.9 is given by the tuple $PN = (P, C)$, where

- $P = \{\ P1, P2, P3, P4, P5, P6, P7\ \}$ is the set of processes,
- $C = \{\ C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17\ \}$ is the set of channels.

Examples of Definition 2.4.2:

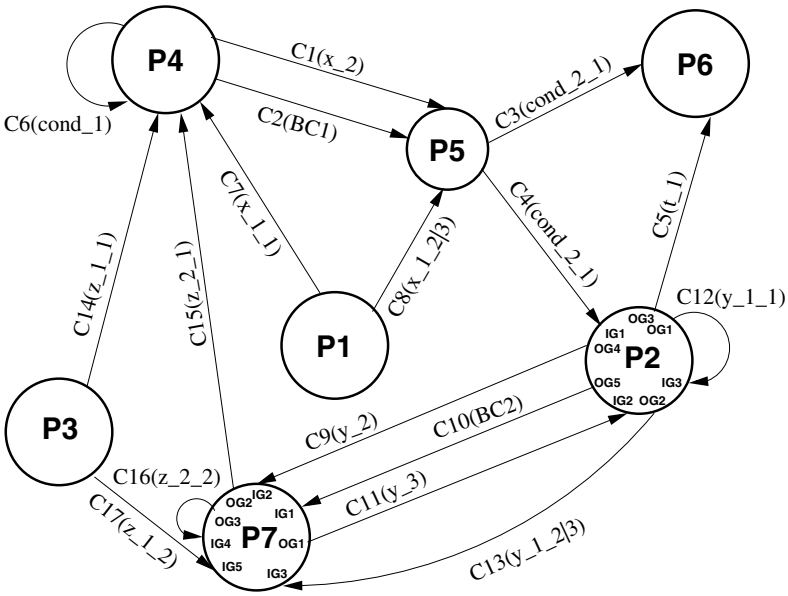

Figure 2.9: The Process Network (PN) model created after applying the procedures *Node Grouping* and *Edge Grouping* on the ADG model shown in Figure 2.5.

Consider process $P2$ of the PN in Figure 2.9. $P2$ is created by grouping two nodes $N2$ and $N7$ of the ADG. $P2$ is given by the tuple $P2 = (NP, IP, OP, ST)$, where

- $NP = \{N2, N7\}$ is the set of nodes. $N2$ and $N7$ are nodes of the ADG shown in Figure 2.5. These nodes are defined in accordance with Definition 2.2.2. An example of how node $N7$ is defined was given in Section 2.2.3;
- $IP = \{IG_1, IG_2, IG_3\} = \{\ (I_{IG_1}, IK_{IG_1}), (I_{IG_2}, IK_{IG_2}), (I_{IG_3}, IK_{IG_3})\ \} =$
  $= \{\ (\ \{N7.p_1\}, \emptyset\ ), (\ \{N7.p_2\}, \emptyset\ ), (\ \{N7.p_3\}, \emptyset\ )\ \}$
  IP is the set of input gates. Every input gate IG_i in the set IP is defined in accordance with Definition 2.4.3. Notice that here every input gate IG_i consists of one input port that belongs to node $N7$; An example of how input gate IG_2 is defined is given below - see **Examples of Definition 2.4.3**;
- $OP = \{OG_1, OG_2, OG_3, OG_4, OG_5\} =$
  $= \{\ (O_{OG_1}, OK_{OG_1}), (O_{OG_2}, OK_{OG_2}), (O_{OG_3}, OK_{OG_3}), (O_{OG_4}, OK_{OG_4}),$
  $(O_{OG_5}, OK_{OG_5})\ \} =$
  $= \{\ (\ \{N2.q_{1\_13}\}, \emptyset\ ), (\ \{N2.q_{1\_14}, N2.q_{1\_15}\}, \emptyset\ ), (\ \{N2.q_2\}, \emptyset\ ), (\ \{N7.q_1\}, \emptyset\ ),$
  $(\ \{N7.q_2\}, \emptyset\ )\ \}$
  OP is the set of output gates. Every output gate OG_i in the set OP is defined in accordance with Definition 2.4.4. Notice that the output gate OG_2 consists of two output ports that belong to node $N2$. An example of how output gate OG_2 is defined is given below - see **Examples of Definition 2.4.4**;

- $ST = \perp$ is the schedule tree of process $P2$. This tree is not defined yet. **NOTE:** This tree is not defined by the grouping procedures because ST is not related to the topology of the PN model. ST is related to the internal behavior of process $P2$, i.e., it gives a valid execution order between the functions $\_Source\_yt$ and $F2$ associated with the nodes $N2$ and $N7$ that are grouped in $P2$. In general, ST has to be defined when the behavior of the PN is created by the procedures described in Section 2.4.5 - **PN-to-ParseTrees**.

Consider process $P7$ of the PN in Figure 2.9. $P7$ is created by taking only node $N9$ of the ADG. $P7$ is given by the tuple $P7 = (NP, IP, OP, ST)$, where

- $NP = \{N9\}$ is the set of nodes. $N9$ is the node in the ADG shown in Figure 2.5 and it is defined in accordance with Definition 2.2.2;
- $IP = \{IG_1, IG_2, IG_3, IG_4, IG_5\} =$
  $= \{ (\{N9.p_1\}, \emptyset), (\{N9.p_2\}, \emptyset), (\{N9.p_3, N9.p_4\}, \emptyset), (\{N9.p_5\}, \emptyset), (\{N9.p_6\}, \emptyset) \}$
  IP is the set of input gates where every gate is defined in accordance with Definition 2.4.3. Notice that the input gate IG_3 consists of two input ports p_3 and p_4 of node $N9$. An example of how input gate IG_3 is defined is given below - see **Examples of Definition 2.4.3**;;
- $OP = \{OG_1, OG_2, OG_3\} = \{ (\{N9.q_1\}, \emptyset), (\{N9.q_{2\_17}\}, \emptyset), (\{N9.q_{2\_18}\}, \emptyset) \}$ is the set of output gates;
- $ST = \perp$ is the parse tree that gives the sequence of executions of function call $F4$ associated with node $N9$. This tree is not defined yet. **NOTE:** As we said above ST is defined when the behavior of the PN is created.

Examples of Definition 2.4.3:
Consider the input gate IG_2 of process $P2$ in Figure 2.9. IG_2 is given by the tuple $IG_2 = (I_{IG_2}, IK_{IG_2})$, where

- $I_{IG_2} = \{N7.p_2\}$ is the set of input ports. In this case the set consists of only one port $N7.p_2$. This port is input port p_2 of node $N7$ in the ADG shown in Figure 2.5. Port p_2 is defined in accordance with Definition 2.2.3. An example of how input port p_2 is defined was given in Section 2.2.3;
- $IK_{IG_2} = \{InKey_{N7.p_2}\} = \perp$ is the set of functions of the input gate. It consists of only one function $InKey_{N7.p_2}$ associated with the port $N7.p_2$ because the input gate consists of only one port. The function $InKey_{N7.p_2}$ is not defined yet. **NOTE:** This function is not defined by the grouping procedures because $InKey_{N7.p_2}$ is not related to the topology of the PN model. $InKey_{N7.p_2}$ is related to the communication behavior of process $P2$ via port $N7.p_2$ of input gate IG_2. The data coming from port $N7.p_2$ is colored (tagged). When a data is read from port $N7.p_2$ of gate IG_2 then function $InKey_{N7.p_2}$ generates an integer number. If this number matches the unique color (tag) of the read data then this data is processed inside $P2$. If there is not a match then the data is discarded because this data is not needed by process $P2$. In

general, $InKey_{N7.p_2}$ has to be defined when the behavior of the PN is created by the procedures described in Section 2.4.5 - **Linearization**.

Consider the input gate IG_3 of process $P7$ in Figure 2.9. IG_3 is given by the tuple $IG_3 = (I_{IG_3}, IK_{IG_3})$, where

- $I_{IG_3} = \{N9.p_3, N9.p_4\}$ is the set of input ports, where $N9.p_3$ and $N9.p_4$ are input ports of node $N9$ of the ADG shown in Figure 2.5. Notice that the input gate IG_3 consists of two input ports that belong to node $N9$ because this gate is connected to channel $C13$. This channel is created by grouping edges $ED14$ and $ED15$ of the ADG. These edges end up at ports p_3 and p_4 of node $N9$, respectively;
- $IK_{IG_3} = \{InKey_{N9.p_3}, InKey_{N9.p_4}\} = \{\perp, \perp\}$ is the set of functions, where $InKey_{N9.p_3}$ and $InKey_{N9.p_4}$ are associated with input ports $N9.p_3$ and $N9.p_4$, respectively. **NOTE:** $InKey_{N9.p_3}$ and $InKey_{N9.p_4}$ have to be defined when the behavior of the PN is created. In Section 2.4.5 - **Examples of Linearization**, we give an example of how these functions are defined.

Example of Definition 2.4.4:
Consider the output gate OG_2 of process $P2$ in Figure 2.9. OG_2 is given by the tuple $OG_2 = (O_{OG_2}, OK_{OG_2})$, where

- $O_{OG_2} = \{N2.q_{1\_14}, N2.q_{1\_15}\}$ is the set of output ports, where $N2.q_{1\_14}$ and $N2.q_{1\_15}$ are output ports of the node $N2$ of the ADG shown in Figure 2.5. Notice that the output gate OG_2 consists of two output ports that belong to node $N2$ because this gate is connected to channel $C13$ which is created by grouping edges $ED14$ and $ED15$ of the ADG. These edges start from ports q_{1_14} and q_{1_15} of node $N2$;
- $OK_{OG_2} = \{OutKey_{N2.q_{1\_14}}, OutKey_{N2.q_{1\_15}}\} = \{\perp, \perp\}$ is the set of functions of output gate OG_2, where $OutKey_{N2.q_{1\_14}}$ and $OutKey_{N2.q_{1\_15}}$ are associated with the output ports $N2.q_{1\_14}$ and $N2.q_{1\_15}$, respectively. This functions are not defined here by the grouping procedures because they are not related to the topology of the PN. The functions are related to the communication behavior of process $P2$ via output gate OG_2. When a data is sent via port $N2.q_{1\_14}$ or port $N2.q_{1\_15}$ of gate OG_2 the corresponding function generates a unique integer number which is attached as a color (tag) to the data. Details about when and why the tags are needed during data communication is given in Section 2.4.5 - **Linearization**. **NOTE:** $OutKey_{N2.q_{1\_14}}$ and $OutKey_{N2.q_{1\_15}}$ have to be defined when the behavior of the PN is created. In Section 2.4.5 - **Examples of Linearization**, we give an example of how these functions are defined.

Example of Definition 2.4.5:
Consider channel $C13$ of the PN shown in Figure 2.9. $C13$ is created by grouping edges $ED14$ and $ED15$ of the ADG. Channel $C13$ is given by the tuple $C13 = (OG, IG, E, CM)$, where

- $OG = OG_2$ is the output gate. This output gate was described in the examples above;

- $IG = IG_3$ is the input gate. Also, this gate was described above;
- $E = \{ED14, ED15\}$ is the set of edges, where $ED14$ and $ED15$ are edges in the ADG shown in Figure 2.5;
- $CM = \perp$ is the communication model of $C13$. CM has to be one of the four possible communication models that our PN model supports. CM is not defined here because it is related to the communication behavior of the channel. Therefore, CM is selected when the behavior of the PN is created. An example of how CM is selected is given in Section 2.4.5 - **Examples of Linearization**.

2.4.5 Creating the PN behavior

The starting point for creating the PN behavior is the process network model $PN = (P, C)$ created by the *Node Grouping* and *Edge Grouping* procedures described in Section 2.4.4. This PN model is not complete, i.e., not all of the elements of PN, as defined in Section 2.4.3, have been specified by the grouping procedures. Only the elements of PN related to its topology have been specified. Therefore, by creating the behavior of PN the rest of the elements are specified. This means that the following elements of every process $P_i \in P$ and every channel $C_j \in C$ have to be determined (see Section 2.4.3 for precise definitions):

- $P_i.ST$ - the schedule tree of process P_i that gives a valid execution order between the function calls which have to be executed inside process P_i;
- $C_j.CM$ - the communication model of channel C_j;
- $C_j.OG.OK_{OG}$ - the set of functions associated with the output gate OG of channel C_j. These functions are used to realize a correct communication behavior of C_j when the communication model of C_j requires coloring (tagging) or re-ordering of the communicated data;
- $C_j.IG.IK_{IG}$ - the set of functions associated with the input gate IG of channel C_j. Again, these functions are used to realize a correct communication behavior of C_j when the communication model of C_j requires coloring (tagging) or re-ordering of the communicated data.

The PN behavior is obtained by executing two procedures: 1) *Linearization* procedure that determines elements $C_j.CM$, $C_j.OG.OK_{OG}$, and $C_j.IG.IK_{IG}$ for every communication channel C_j; 2) *PN-to-ParseTrees* procedure that determines element $P_i.ST$ for every process P_i. Both procedures are defined and explained below:

Linearization

By definition [10], in a Kahn Process Network (KPN) the functions executed inside a process communicate data with functions inside other processes over 1-dimensional FIFO channels. However, our approach has to derive KPNs from weakly dynamic programs (WDP) in which

functions communicate data between each other via N-dimensional variables. In order to keep the functionality of the KPN the same as the functionality of the WDP, we apply the procedure *Linearization*, explained in this section, on the PN model. This procedure adds to the PN model information that is necessary to convert the WDP communication model to the KPN communication model. Our PN model supports four models of 1-dimensional FIFO communication. These models are:

1. **in-order communication without coloring of tokens** - this model is used when the order of the tokens (data) written in the FIFO channel by a producer process is the same as the order the tokens are read by a consumer process. The number of tokens that will be written or read to/from the channel is known at compile time;

2. **out-of-order communication without coloring of tokens** - this model is used when the order of the tokens (data) written in the FIFO channel by a producer process is different than the order the tokens are read by a consumer process. The number of tokens that will be written or read to/from the channel is known at compile time.

3. **in-order communication with coloring of tokens** - this model is used when the order of the tokens (data) written in the FIFO channel by a producer process is the same as the order the tokens are read by a consumer process. Also, the number of tokens that will be written or read to/from the channel is not known at compile time. Because of this, more tokens can be written to the channel than needed at run time. Every token is tagged by a unique number (color) that is used to remove the tokens that are not needed while reading them from the channel;

4. **out-of-order communication with coloring of tokens** - this model is used when the order of the tokens (data) written in the FIFO channel by a producer process is different than the order the tokens are read by a consumer process. Also, the number of tokens that will be written or read to/from the channel is not known at compile time. In order to keep the correct behavior of the KPN, every token is tagged by a unique number (color) that is used to re-order the tokens while reading them form the channel;

Our procedure LINEARIZATION(PN) is given below. This procedure takes as an input the process network model $PN = (P, C)$ created by the procedures described in Section 2.4.4.

```
Linearization( PN ) begin
1    for all C_j ∈ PN.C do
2      C_j.CM ← channelModel( C_j ); ▷select communication model for C_j
3      for all q_k ∈ C_j.OG.O_OG do ▷derive info to build the selected model
4          OutKey_{q_k} ← LBS2ColorGenerator( q_k.OPD_{q_k} );
5          InKey_{(q_k)^E.p} ← compose( OutKey_{q_k}, (q_k)^E.M );
6          C_j.OG.OK_OG.ADD( OutKey_{q_k} );
7          C_j.IG.IK_IG.ADD( InKey_{(q_k)^E.p} );
8      endfor
9    endfor
10 end
```

First, the procedure LINEARIZATION(PN) selects for every channel C_j in the network model PN the least expensive communication model that guarantees the correct behavior of the network - see line 2 where the element CM (Definition 2.4.5) of every C_j is determined. The four communication models, described above, have different cost of implementation in terms of required memory and complexity of the control. The *out-of-order communication with coloring of tokens* model is the most expensive model but it is the most general model. This model can be used for every channel and the correct behavior of the PN is guaranteed. However, depending on the edges that are grouped in a particular channel, in some cases a less expensive communication model can be selected. This model still guarantees the correct behavior of the PN. The selection is done by calling the procedure CHANNELMODEL(C) in line 2.

This procedure extends the work presented in [22] [57]. In this work only two communication models are considered, i.e., models 1) and 2) described above. This is because this work deals with deriving KPNs from static programs where everything is known at compile time, thus models 1) and 2) are sufficient for the communication. In our case we deal with deriving KPNs from weakly dynamic programs (WDP) where not everything is known at compile time. Therefore our procedure CHANNELMODEL(C) considers four communication models that are sufficient to realize correct inter-process communication. The procedure CHANNELMODEL(C) is given below:

```
channelModel( C ) begin
1    if |C.E| = 1 then
2      e ← C.E.E_1;
3      if isOutOfOrder( e.M ) = true then
4        if S ∈ e.p.IPD_p ≡ ∅ then
5          return( 2 ); ▷ out-of-order without coloring of tokens
6        else
7          return( 4 ); ▷ out-of-order with coloring of tokens
8        endif
9      else
10       if S ∈ e.p.IPD_p ≡ ∅ then
11         return( 1 ); ▷ in-order without coloring of tokens
12       else
13         return( 3 ); ▷ in-order with coloring of tokens
14       endif
15     endif
16   else
17     return( 4 ); ▷ out-of-order with coloring of tokens
18   endif
19 end
```

This procedure takes as an input a channel C and decides what the communication model for C has to be. In line 1 the procedure checks if the channel consists of only one edge. If this is not true then model 4) is selected (line 17) because many streams of data corresponding to every edge will be interlaced and communicated over the channel. This requires re-ordering and coloring of the data while reading it from the channel in order to identify and split the

multiple streams of data. If the condition in line 1 is true then we have only one edge, i.e., one stream of data in the channel and we use the procedure ISOUTOFORDER(M) in line 3 to decide whether re-ordering of data is needed. This procedure is actually the *Reordering Detection Decision Tree* presented in [22]. If we need re-ordering of data we execute code lines 4 till 8, else we execute code lines 10 till 14. In both cases we have to decide whether we need to color the data or not - lines 4 and 10. We use the domain IPD of the input port p (see Definition 2.2.3) of the edge e which belongs to the channel C. The IPD is a linearly bounded set (see Definition 2.2.6) and we check whether the set of filtering functions S of IPD is empty. If this is not true then this means that some information about the communication is not known at compile time (see models 3) and 4)), thus we have to color the data.

After the selection of communication model for channel C, our procedure LINEARIZATION(PN), given above, derives the information needed to realize the selected communication model - see code lines 3 till 8 where the elements $C.OG.OK_{OG}$ (Definition 2.4.4) and $C.IG.IK_{IG}$ (Definition 2.4.3) of C are determined. This is done by executing the procedure LBS2COLORGENERATOR in line 4 followed by the procedure COMPOSE in line 5. The procedure LBS2COLORGENERATOR($q_k.OPD_{q_k}$) derives the function $OutKey_{q_k}$ associated with every output port q_k in the output gate OG of channel C. Every output port q_k has a corresponding input port p_k in the input gate IG of channel C. Port p_k can be found using the edge $E = (q_k, p_k, M)$ (see Definition 2.2.5) that connects the ports, so we can denote port p_k as $(q_k)^E.p$ in accordance with the notations in Section 2.4.1. The procedure COMPOSE($OutKey_{q_k}$, $(q_k)^E.M$) derives the function $InKey_{(q_k)^E.p}$ associated with every input port p_k. So, for every pair of ports $< q_k , p_k >$, a pair of functions $< OutKey_{q_k} , InKey_{p_k} >$ is derived. Such pair of functions is used for generating colors (tags) attached to the data communicated over channel C via ports q_k and p_k in case the selected communication model for C is 3) or 4). Also, such pair of functions is used for data re-ordering in case the selected model is 2). More details about the use of $< OutKey_{q_k} , InKey_{p_k} >$ are given in Section 2.4.5-**Communication Models Realizations** where these functions are used in the controllers described in Figure 2.13, Figure 2.15, Figure 2.16, and Figure 2.17.

The procedure LBS2COLORGENERATOR($q_k.OPD_{q_k}$) derives function $OutKey_{q_k}$ as a polynomial. This procedure takes as an input the output port domain OPD_{q_k} (see Definition 2.2.4) of port q_k. This output port domain is a linearly bounded set (LBS - Definition 2.2.6) with a linear bound B which is a polytope. This polytope is defined in the n-dimensional space with dimension vector $\mathbf{i} = (i_1, ..., i_n)$. By using B the procedure finds a hypercube HC that bounds B. This cube is again a polytope. Then the function $OutKey_{q_k}(\mathbf{i})$ is the very simple *ranking polynomial* of HC that is obtained using the technique described in [19]. We can apply this technique directly on the linear bound B but in this case $OutKey_{q_k}(\mathbf{i})$ may be a pseudo-polynomial which is a much more complex function.

The procedure COMPOSE($OutKey_{q_k}(\mathbf{i})$, $(q_k)^E.M$) derives function $InKey_{p_k}$ as a polynomial. This procedure takes as an input the polynomial $OutKey_{q_k}(\mathbf{i})$ discussed above. Also, it takes the affine function $M : \mathbf{j} \rightarrow \mathbf{i}$ (see Definition 2.2.5) which maps the input port domain of port p_k to the output port domain of port q_k. This can be written as $\mathbf{i} = M(\mathbf{j})$. The procedure COMPOSE substitutes $\mathbf{i}$ with $M(\mathbf{j})$ in the polynomial $OutKey_{q_k}(\mathbf{i})$, so the derived function $InKey_{p_k}(\mathbf{j}) = OutKey_{q_k}(M(\mathbf{j}))$ is again a polynomial.

Examples of Linearization

In this section we give an example of how the procedure LINEARIZATION(PN), described in the previous section, works for one channel $C_j \in PN.C$. Consider channel $C13$ in the PN shown in Figure 2.9. We used the same channel as an example in Section 2.4.4-**Example of Definition 2.4.5**. Channel $C13$ is given by the tuple $C13 = (OG, IG, E, CM)$, where

- $OG = OG_2 = (\ \{N2.q_{1\_14}, N2.q_{1\_15}\}\ ,\ \{OutKey_{N2.q_{1\_14}}, OutKey_{N2.q_{1\_15}}\}\ ) =$
  $= (\ \{N2.q_{1\_14}, N2.q_{1\_15}\}\ ,\ \{\perp, \perp\}\ )$ is the output gate;
- $IG = IG_3 = (\ \{N9.p_3, N9.p_4\}\ ,\ \{InKey_{N9.p_3}, InKey_{N9.p_4}\}\ ) =$
  $= (\ \{N9.p_3, N9.p_4\}\ ,\ \{\perp, \perp\}\ )$ is the input gate;
- $E = \{ED14, ED15\}$ is the set of edges, where $ED14$ and $ED15$ are edges in the ADG shown in Figure 2.5;

  $ED14$ corresponds to the variable y_1 that appears in function calls $[\ in\_0\ ] = \mathbf{ipd}(\ y\_1(i+1)\ )$ and $[\ y\_1(i)\ ] = \mathbf{opd}(\ out\_0\ )$ at lines 80 and 18 in the dSAC shown in Figure 2.4. Therefore, edge $ED14$ is defined by the triple $E14 = (q_{1\_14}, p_3, M)$, where

  - $q_{1\_14} = (V_{q_{1\_14}}, A_{q_{1\_14}}, OPD_{q_{1\_14}}) =$
    $= (\ y\_1(i),\ out\_0,\ \{\ i_{q1\_14} \in \mathbb{Z} \mid 1 \leq i_{q1\_14} \leq N+1 \wedge 8 \leq N \leq 16\ \}\ )$
    is the output port;
  - $p_3 = (V_{p_3}, A_{p_3}, IPD_{p_3}) =$
    $= (\ y\_1(i+1),\ in\_0,\ \{\ i_{p3} \in \mathbb{Z} \mid 1 \leq i_{p3} \leq N \wedge 8 \leq N \leq 16,$
    $-c + i_{p3} \geq 0,\ -c + i_{p3} - 1 \geq 0\ \}\ )$ is the input port;
  - the affine mapping M is derived from the indexing of variable y_1 and it is given by the equation $i_{q1\_14} = i_{p3} + 1$.

  $ED15$ corresponds to the variable y_1 that appears in function calls $[\ in\_0\ ] = \mathbf{ipd}(\ y\_1(i+1)\ )$ and $[\ y\_1(i)\ ] = \mathbf{opd}(\ out\_0\ )$ at lines 83 and 18 in the dSAC shown in Figure 2.4. Therefore, edge $ED15$ is defined by the triple $E15 = (q_{1\_15}, p_4, M)$, where

  - $q_{1\_15} = (V_{q_{1\_15}}, A_{q_{1\_15}}, OPD_{q_{1\_15}}) =$
    $= (\ y\_1(i),\ out\_0,\ \{\ i_{q1\_15} \in \mathbb{Z} \mid 1 \leq i_{q1\_15} \leq N+1 \wedge 8 \leq N \leq 16\ \}\ )$
    is the output port;
  - $p_4 = (V_{p_4}, A_{p_4}, IPD_{p_4}) =$
    $= (\ y\_1(i+1),\ in\_0,\ \{\ i_{p4} \in \mathbb{Z} \mid 1 \leq i_{p4} \leq N \wedge 8 \leq N \leq 16,$
    $c - i_{p4} - 1 \geq 0\ \}\ )$ is the input port;
  - the affine mapping M is derived from the indexing of variable y_1 and it is given by the equation $i_{q1\_15} = i_{p4} + 1$.

- $CM = \perp$ is the communication model of $C13$.

The first step in the procedure LINEARIZATION(PN) is to specify the CM of channel $C13$ by calling the procedure CHANNELMODEL($C13$). Since the number of edges $|C13.E|$ that

belong to $C13$ is equal to 2, the procedure CHANNELMODEL($C13$) selects the communication model CM of $C13$ to be 4. This means *out-of-order communication with coloring of tokens*.

The next step in the procedure LINEARIZATION(PN) is the derivation of functions $OutKey_{N2.q_{1\_14}}$ and $OutKey_{N2.q_{1\_15}}$ by calling procedure LBS2COLORGENERATOR. We illustrate this procedure for function $OutKey_{N2.q_{1\_14}}$ associated with output port q_{1_14}. The linear bound of $OPD_{q_{1\_14}}$ is the polytope $1 \leq i_{q1\_14} \leq N+1$ defined in the one dimensional space i_{q1_14}. A hypercube that bounds this polytope is equal to the same polytope in this simple case. So, the technique described in [19] for deriving a ranking polynomial is applied on the polytope $1 \leq i_{q1\_14} \leq N+1$ which gives us the following function $OutKey_{N2.q_{1\_14}}$:

$$OutKey_{N2.q_{1\_14}}(i_{q1\_14}) = i_{q1\_14}$$

Similarly, the procedure LBS2COLORGENERATOR derives the function $OutKey_{N2.q_{1\_15}}$ as follows:

$$OutKey_{N2.q_{1\_15}}(i_{q1\_15}) = i_{q1\_15}$$

The final step in the procedure LINEARIZATION(PN) is the derivation of functions $InKey_{N9.p_3}$ and $InKey_{N9.p_4}$ by calling the procedure COMPOSE. The following mapping functions defined above are used: $i_{q1\_14} = i_{p3} + 1$ and $i_{q1\_15} = i_{p4} + 1$. So, the functions $InKey_{N9.p_3}$ and $InKey_{N9.p_4}$ are derived as follows:

$$InKey_{N9.p_3}(i_{p3}) = OutKey_{N2.q_{1\_14}}(i_{p3} + 1) = i_{p3} + 1$$
$$InKey_{N9.p_4}(i_{p4}) = OutKey_{N2.q_{1\_15}}(i_{p4} + 1) = i_{p4} + 1$$

Communication Models Realizations

As said before our PN model supports four models of communication. The procedure LINEARIZATION(PN) selects a communication model for every channel in the network and derives the information needed to realize the communication models. In this section, we explain how we realize the models using the simple process network shown in Figure 2.10. This network consists of two processes Pm and Pn and one communication channel. The components **(A)** and **(C)** in processes Pm and Pn, respectively, are the sequential code that describes the behavior of Pm and Pn, respectively. When Pm has to send some data to Pn one of the controllers -**putController(V,I)**- in component **(B)** of process Pm is activated. The **putController(V,I)** takes the data **V** and the current value of the iteration vector **I**, forms a token and puts the token in the channel using output gate **OG**. Similarly, when process Pn needs data from process Pm, then Pn activates a controller -**getController(J)**- in component **(D)**. The **getController(J)** gets a token from gate **IG** using the value of the current iteration vector **J** and returns the data **V** of the token.

In our PN model every FIFO channel supports two communication primitives for putting and getting tokens to/from the channel:

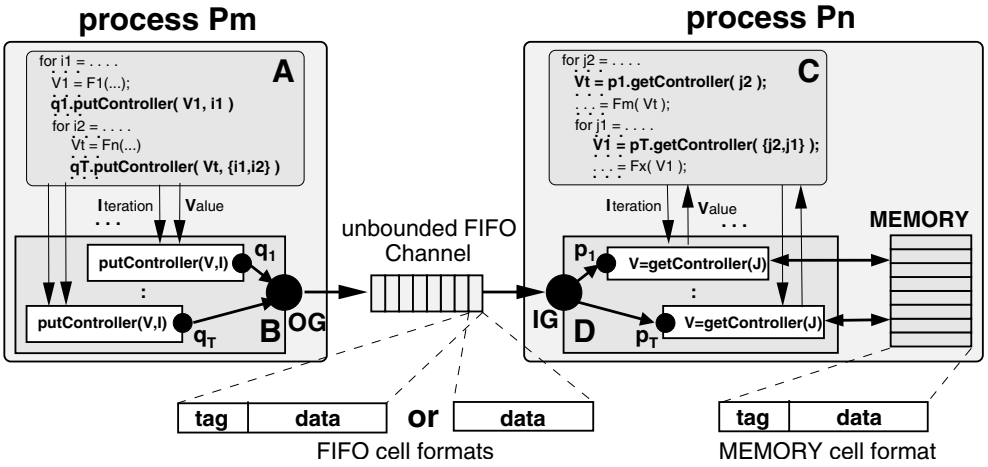

Figure 2.10: A simple Kahn Process Network that consists of one ”producer” process and one ”consumer” process communicating data over one FIFO channel.

1. **WRITE(** $gate$, $token$ **)** - this primitive is used in a **putController(V,I)** to put a $token$ into the FIFO channel that has $gate$ as an output gate.
2. **READ(** $gate$ **)** - this primitive is used in a **getController(J)** to get a token from the FIFO channel that has $gate$ as an input gate. When the primitive **READ** is called it returns the first token that is in the FIFO channel.

For every edge that belongs to a channel (see Definition 2.4.5) a pair of controllers **putController(V,I) – getController(J)** is created. The controllers are accessed via the output port q and the input port p of a given edge as shown in Figure 2.10. The values of the variable associated with q and p (see Definitions 2.2.4 and 2.2.3) are communicated using these controllers. Every value is encapsulated in a token that has a format depending on the format of the cells in the FIFO channel - see Figure 2.10. The ”data” component of a token is the value and the ”tag” component is a color of the token that makes the token unique among other tokens in the channel. As shown in Figure 2.10 our communication model supports also channels without coloring of tokens.

The tokens in the FIFO channel, shown in Figure 2.10, are written in a particular order. A communication problem appears when this order is not the same as the order the tokens are needed in the consumer process. In this case, the tokens have to be re-ordered in order to preserve the correct behavior of the network. This is accomplished by the controllers in component **(D)** of process Pn together with the component **MEMORY** of Pn. The component **MEMORY** is a content addressable memory (CAM) that is used as a temporary storage of tokens during the re-ordering procedure.

The components **(B)** of Pm, **(D)** of Pn, **MEMORY**, and the FIFO channel shown in Figure 2.10 realize the four communication models supported by our PN model. The difference between the communication models is the implementation of the controllers **putCon-**

troller(V,I) in component **(B)** and **getController(J)** in component **(D)**, the FIFO cell format, and the necessity of component **MEMORY** in process Pn. Below, we describe every communication model:

1) in-order communication without coloring of tokens - this model consists of components **(B)** and **(D)**. All controllers (**putController(V,I)**) in component **(B)** implement the procedure given in Figure 2.11. All controllers (**getController(J)**) in component **(D)** implement the procedure given in Figure 2.12. Every token and cell in the FIFO channel has the format: $(data)$.

```
1  q_l . putToken( value, I ) begin
2      WRITE( (q_l)^OG, value );
3
4  end
```

Figure 2.11: Implementation of the controller used in communication models 1) and 2) to write data into a channel.

```
1  p_k . getToken( J ) begin
2      value ← READ( (p_k)^IG );
3      return( value );
4  end
```

Figure 2.12: Implementation of the controller used in communication model 1) to read data from a channel.

The controller shown in Figure 2.11 takes the data stored in variable $value$ and uses the primitive **WRITE** to put this data, via output port q_l, in the channel connected to output gate $(q_l)^{OG}$. Similarly, the controller shown in Figure 2.12 uses the primitive **READ** to get the first token from the channel connected to input gate $(p_k)^{IG}$. Also, the controller returns the value of the token via input port p_k. Notice that the value of input argument I in the controller in Figure 2.11 is not used. The same is true for argument J of the controller shown in Figure 2.12. This is because we do not have coloring of tokens (data) when using these controllers.

2) out-of-order communication without coloring of tokens - this model consists of components **(B)**, **(D)**, and **MEMORY**. All controllers (**putController(V,I)**) in component **(B)** implement the procedure given in Figure 2.11. All controllers (**getController(J)**) in component **(D)** implement the procedure given in Figure 2.13. In this procedure the tokens that are received from the channel are not colored. However, to re-order these tokens, a unique tag is added to the tokens internally in the controller. The tags are generated by the procedure shown in Figure 2.14. Every token and cell in the FIFO channel has the format: $(data)$. Every cell in the re-ordering memory (component **MEMORY**) has the format: $(tag, data)$.

The procedure shown in Figure 2.13 does the re-ordering of the incoming tokens using a re-ordering memory as a temporary storage. The input of this procedure is the current iteration J at which the procedure is called by component **(C)** in Figure 2.10. The value of J is used in this procedure to generate an integer number called key - see code line 2 in Figure 2.13. J is put as an argument of function $(p_k)^{IG}.InKey_{p_k}(J)$ which corresponds to input port p_k of input gate IG (see Definition 2.4.3). The integer number key is a tag which is used to identify the needed token for the current iteration J. First, the procedure checks whether the needed token with tag key is in the re-ordering memory - see code line 3. If this is true, then the procedure returns the value of the token (code line 4) and releases the memory location occupied by this token (code line 5).

If the needed token is not in the re-ordering memory then the procedure takes the needed token from the channel. This is done by executing code lines 7 till 14. The first token from

```
1  p_k . getReorderToken( J ) begin
2    key ← (p_k)^IG.InKey_{p_k}( J );
3    if isInMEMORY( key ) = true then
4      return( MEMORY[key].value );
5      MEMORY.remove( key );
6    else
7      do
8        value ← READ( (p_k)^IG );
9        color ← recoverColor( p_k );
10       if color ≠ key then
11         MEMORY.insert( (color, value) );
12       endif
13     while color ≠ key
14     return( value );
15   endif
16 end
```

Figure 2.13: Implementation of the controller used in communication model 2) to read and re-order data coming from a channel.

```
1  recoverColor( p_k ) begin
2  outp ← (p_k)^E.q;
3  for n = start : 1 : |outp.OPD.B| do
4    if i_n ∈ outp.OPD then
5      start ← n + 1;
6      c ← (outp)^OG.OutKey_{outp}(i_n);
7      return( c );
8    endif
9  endfor
10 end
```

Figure 2.14: Procedure to generate colors for the incoming tokens in the controller shown in Figure 2.13.

the FIFO channel is read at line 8 and its value is stored in variable $value$. The tag of this token is determined by the procedure RECOVERCOLOR at line 9 and this tag is stored in variable $color$. At code line 10 tag key of the needed token is compared to tag $color$ of the read token from the channel. If the tags do not match then the read token is stored in the re-ordering memory (see code line 11) and another token is read from the channel by repeating the code lines 8 till 13. The procedure continues to read tokens from the channel, i.e., executing code lines 8 to 13, till the tags key and $color$ match. Then the value of the token read from the channel is returned because this is the needed token.

As said before, in communication model 2) the communicated tokens are not colored when they are written into the channel. However, we described above that in order to do re-ordering the tokens read from the channel in model 2) are tagged internally in the procedure shown in Figure 2.13 - see code line 9 where procedure RECOVERCOLOR is executed. This procedure takes as an input the input port p_k of input gate IG via which the tokens are read. First, the procedure RECOVERCOLOR takes the corresponding output port q - see code line 2. Second, the procedure starts to enumerate the points bounded by the linear bound of output port domain OPD (see Definition 2.2.4) of q using the loop at line 3. Initially, the value of variable $start$ is equal to 0. Every enumerated point i_n is checked whether it belongs to the output port domain OPD - see code line 4. If this is true then: 1) variable $start$ stores the number n of the next point - line 5; 2) function $(outp)^{OG}.OutKey_{outp}(i_n)$ that corresponds to output port q is used to generate an integer number - line 6. This number is returned as tag (color) by the procedure RECOVERCOLOR. NOTE: the next time procedure RECOVERCOLOR is called it starts to enumerate the OPD of q from the value stored in variable $start$.

3) in-order communication with coloring of tokens - this model consists of components **(B)** and **(D)**. All controllers (**putController(V,I)**) in component **(B)** implement the procedure given in Figure 2.15. All controllers (**getController(J)**) in component **(D)** implement the procedure given in Figure 2.16. Every token and cell in the FIFO channel has the format: $(tag, data)$.

The controller shown in Figure 2.15 takes as an input the data stored in variable $value$.

```
1 q_l . putColoredToken ( value, I ) begin
2   color ← (q_l)^E ||(q_l)^OG .OutKey_{q_l}( I );
3   WRITE( (q_l)^OG , (color, value) );
4 end
```

Figure 2.15: Implementation of the controller used in communication models 3) and 4) to write data into a channel.

```
1 p_k . getColoredToken ( J ) begin
2   key ← (p_k)^E ||(p_k)^IG .InKey_{p_k}( J );
3   do
4     (color, value) ← READ( (p_k)^IG );
5   while color ≠ key
6   return ( value );
7 end
```

Figure 2.16: Implementation of the controller used in communication model 3) to read data from a channel.

Also, it takes iteration vector I which points to the current iteration at which the controller is activated by component **(A)** in Figure 2.10. First, a unique color (tag) for the data stored in $value$ is generated by function $(q_l)^{OG}.OutKey_{q_l}(I)$ that corresponds to output port q_l - see code line 2. This function takes as an input iteration vector I and generates an integer number. To ensure the uniqueness of the color (tag) among other colors used in the channel and generated by other functions, the generated integer number is concatenated with the number of the edge to which port q_l belongs - see the notation '||' in code line 2. Second, the primitive **WRITE** is used to put, in the channel, the data stored in $value$ together with its tag. This is done via output port q_l of output gate $(q_l)^{OG}$ that is connected to the channel - see code line 3.

The controller shown in Figure 2.16 takes as an input iteration vector J which points to the current iteration at which the controller is activated by component **(C)** in Figure 2.10. The value of J is used to generate an integer number called key - see code line 2. J is put as an argument of function $(p_k)^{IG}.InKey_{p_k}(J)$ which correspond to input port p_k of input gate IG. The integer number key is a tag which is used to identify the needed token for the current iteration J. The controller reads tokens from the channel using the primitive **READ** until tag key of the needed token matches tag $color$ of the read token - see code lines 3 to 5. When there is a match the value of the read token is returned because this is the needed token (code line 6).

4) out-of-order communication with coloring of tokens - this model consists of components **(B)**, **(D)** and **MEMORY**. All controllers (**putController(V,I)**) in component **(B)** implement the procedure given in Figure 2.15. All controllers (**getController(J)**) in component **(D)** implement the procedure given in Figure 2.17. Every token, cell in the FIFO channel, and cell in the re-ordering memory (component **MEMORY**) has the format: $(tag, data)$.

The controller shown in Figure 2.17 behaves exactly the same way as the controller in Figure 2.13. However, there is a small difference. The controller in Figure 2.13 executes procedure RECOVERCOLOR in line 9 to get the tag of every incoming token. In contrast, the controller in Figure 2.17 does not use the procedure RECOVERCOLOR - see that code line 9 is empty. This is because the tokens in the channel are tagged (colored), thus the tag (color) of the incoming token is get when the token is read from the channel - see code line 8.

```
1  p_k . getReorderColoredToken( j ) begin
2    key ← (p_k)^E||(p_k)^IG.InKey_{p_k}( j );
3    if isInMEMORY( key ) = true then
4      return( MEMORY[key].value );
5      MEMORY.remove( key );
6    else
7      do
8        (color, value) ← READ( (p_k)^IG );
9
10       if color ≠ key then
11         MEMORY.insert( (color, value) );
12       endif
13     while color ≠ key
14     return( value );
15   endif
16 end
```

Figure 2.17: Implementation of the controller used in communication model 4) to read and re-order data coming from a channel.

PN-to-ParseTrees

In previous sections we discussed how the communication behavior of our PN model is determined by our procedure LINEARIZATION(PN). In this section we describe how the internal behavior of every process P_i that belongs to PN is determined. This means that we have to explain how the element $P_i.ST$ (see Definition 2.4.2) of every process P_i is determined. $P_i.ST$ is the schedule tree of process P_i that gives a valid execution order between the function calls which have to be executed inside process P_i.

We take the network model $PN = (P, C)$ after the procedure LINEARIZATION(PN) is applied on it. We execute the procedure PNTOPARSETREES(PN) which determines element $P_i.ST$ of every process $P_i \in P$. After this procedure, the PN model $PN = (P, C)$ is complete, i.e., all elements of PN given by the definitions in Section 2.4.3 are determined. The procedure PNTOPARSETREES(PN) is defined and explained below:

```
PNtoParseTrees( PN ) begin
1    for all P_i ∈ PN.P do
2      for all N_m ∈ P_i.NP do
3        tree ← processNode( N_m );
4        vectorOfTrees.ADD( (tree, N_m) );
5      endfor
6      tree ← scheduleNodes( P_i.NP, STree );
7      P_i.ST ← connectParseTrees( tree, vectorOfTrees );
8    endfor
9  end
```

For every process P_i in PN the procedure PNTOPARSETREES(PN) visits the following three steps:

1) for all nodes N (see Definition 2.4.2) in process P_i the procedure PROCESSNODE is exe-

cuted at code line 3. This procedure derives for a given node N a syntax tree that corresponds to a control structure of a program. This program specifies from which ports the function F (see Definition 2.2.2) associated with N gets input data and to which ports function F puts the output data for every iteration $i \in N.ND_N$. The syntax tree is stored in a data structure called $vectorOfTrees$ as shown in code line 4.

2) for every process P_i the procedure SCHEDULENODES is executed at code line 6. This procedure takes a schedule tree $STree$ defined in Section 2.3 and derives a valid sequential execution order between the functions that have to be executed inside P_i. These functions are functions F associated with the nodes N that belong to P_i. The sequential execution order is represented as a syntax tree of a sequential program that specifies the order. This syntax tree is denoted as $tree$ at code line 6.

3) for every process P_i the procedure CONNECTPARSETREES is executed at code line 7. This procedure connects the syntax tree ($tree$) derived in 2) with the syntax trees ($vectorOfTrees$) derived in 1). As a result a syntax tree of a sequential program that defines completely the internal behavior of P_i is generated. Actually, this tree is the element $P_i.ST$ of process P_i.

For the sake of clarity we illustrate the behavior of procedure PNTOPARSETREES(PN) by an example. Let us take process $P2$ of the network shown in Figure 2.9. $P2$ is created by grouping two nodes $N2$ and $N7$ of the ADG shown in Figure 2.5. We used process $P2$ as an example in Section 2.4.4-**Example of Definition 2.4.2**. So, $P2$ is given by the tuple $P2 = (NP, IP, OP, ST)$, where

- $NP = \{N2, N7\}$ is the set of nodes. $N2$ and $N7$ are nodes of the ADG shown in Figure 2.5. These nodes are defined in accordance with Definition 2.2.2 as follows:

  $N2$ corresponds to function call $\_Source\_yt$ in the dSAC shown in Figure 2.4. This node is given by the tuple $N2 = (I_{N2}, O_{N2}, F_{N2}, ND_{N2})$, where

  - $I_{N2} = \emptyset$ is the set of input ports which is empty,
  - $O_{N2} = \{q1\_13, q1\_14, q1\_15, q2\}$ is the set of output ports,
  - $F_{N2} = (\_Source\_yt, \{out\_0, out\_1\})$ is the function associated with the node,
  - ND_{N2} is the node domain of $N2$ defined by the linearly bounded set $LBS = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N+1 \wedge 8 \leq N \leq 16\}$. ND_{N2} represents the iterations i at which function call $\_Source\_yt$ is executed in the dSAC.

  $N7$ corresponds to function call $F2$ in the dSAC shown in Figure 2.4. This node is given by the tuple $N7 = (I_{N7}, O_{N7}, F_{N7}, ND_{N7})$, where

  - $I_{N7} = \{p1, p2, p3\}$ is the set of input ports,
  - $O_{N7} = \{q1, q2\}$ is the set of output ports,
  - $F_{N7} = (F2, \{in\_0\}, \{out\_0\})$, where $F2 : \{in\_0\} \rightarrow \{out\_0\}$,
  - ND_{N7} is the node domain of $N7$ defined by the linearly bounded set $LBS = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0\,\}$. ND_{N7} represents the iterations i at which function call $F2$ is executed in the dSAC.

- $IP = \{IG_1, IG_2, IG_3\}$ is the set of input gates of $P2$.

- $OP = \{OG_1, OG_2, OG_3, OG_4, OG_5\}$ is the set of output gates of $P2$.
- $ST = \perp$ is the schedule tree of process $P2$ that has to be determined by procedure PNTOPARSETREES.

Now, let us apply code lines 2 till 7 in procedure PNTOPARSETREES for process $P2$ given above. First, procedure PROCESSNODE at code line 3 is executed two times because process $P2$ has two nodes $N2$ and $N7$. This procedure generates the trees bounded by the dashed boxes in Figure 2.18 - the tree on the left corresponds to node $N2$ and the tree on the right corresponds to node $N7$. Both trees are stored in the data structure $vectorOfTrees$ - see code line 4.

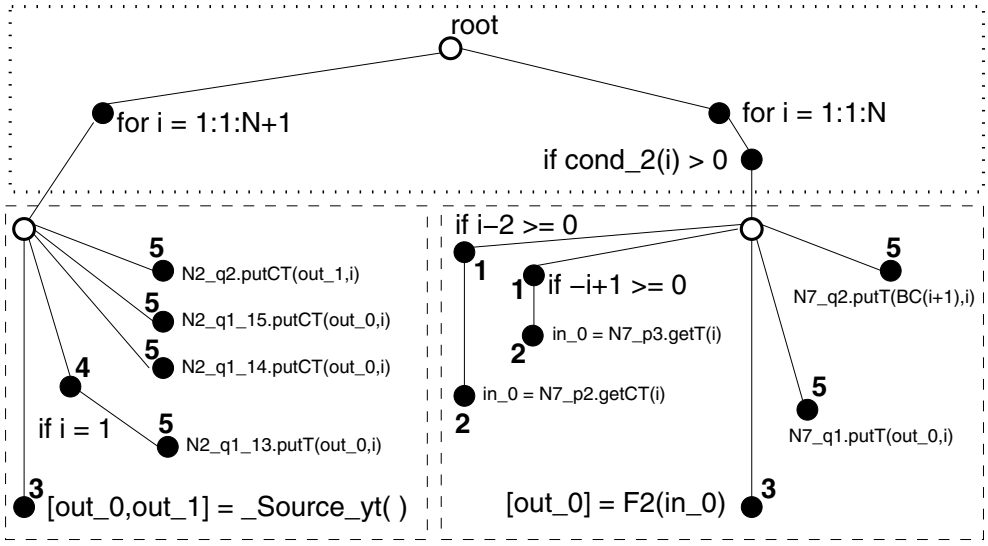

Figure 2.18: The schedule tree ST of process $P2$ that is derived by procedure PNTOPARSE-TREES.

Second, procedure SCHEDULENODES is executed at code line 6. This procedure uses the schedule tree $STree$ given in Section 2.3.3 and derives a valid sequential execution order between functions $\_Source\_yt$ and $F2$. These two functions correspond to nodes $N2$ and $N7$ in process $P2$. The sequential order is represented as a tree. In our case this is the tree bounded by the dotted box in Figure 2.18.

Finally, procedure CONNECTPARSETREES is executed at code line 7. This procedure connects the trees generated by procedure PROCESSNODE and the tree generated by procedure SCHEDULENODES. The resultant tree is shown in Figure 2.18. This tree is the complete schedule tree ST of process $P2$.

In the next three sections we describe the procedures PROCESSNODE, SCHEDULENODES, and CONNECTPARSETREES in more detail. Also, we give examples using process $P2$ and its nodes $N2$ and $N7$ described above.

Procedure PROCESSNODE - definition and example

The pseudo code below defines the procedure PROCESSNODE(N). The input of this procedure is a node N that belongs to a process. Node N (Definition 2.2.2) has a set of input ports denoted as $N.I_N$, a set of output ports $N.O_N$, a function $N.F_N$, and a node domain $N.ND_N$.

```
processNode( N ) begin
1    s ← CREATE RootNode;
2    for all p_k ∈ N.I_N do
3     st ← s;
4     vec ←InputPort2IfNodes(p_k); ▷nodes #1 in Fig.2.18
5     st ← addNodes( vec, st );
6     vec ←InputPort2CommunicateNode(p_k); ▷nodes #2 in Fig.2.18
7     st ← addNodes( vec, st );
8    endfor
9    vec ← Node2FunctionNode(N.F_N); ▷nodes #3 in Fig.2.18
10   st ← addNodes( vec, s );
11   for all q_l ∈ N.O_N do
12    st ← s;
13    vec ←OutputPort2IfNodes(q_l); ▷nodes #4 in Fig.2.18
14    st ← addNodes( vec, st );
15    vec ←OutputPort2CommunicateNode(q_l); ▷nodes #5 in Fig.2.18
16    st ← addNodes( vec, st );
17   endfor
18   return( s );
19  end
```

The output of procedure PROCESSNODE(N) is a syntax tree that corresponds to a control structure of a program. This program specifies from which ports the function $N.F_N$ gets input data and to which ports function $N.F_N$ puts the output data for every iteration $i \in N.ND_N$. First, the procedure creates an empty syntax tree, i.e, tree that has only root node - see code line 1.

Second, every input port $p_k \in N.I_N$ is converted to a number of "IfNodes" that are added to the syntax tree - code lines 4 and 5. The conversion is based on the input port domain IPD of p_k and the node domain ND of node N to which port p_k belongs. Always, IPD is included in ND or IPD is equal to ND. We explain the conversion by an example. Let us consider input port $p2$ of node $N7$ - see Section 2.2.3. The input port domain IPD of $p2$ is defined as follows:
$IPD_{p2} = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N \wedge i \geq 2 \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0 \,\}$
The node domain ND of node $N7$ is defined as follows:
$ND_{N7} = \{\, i \in \mathbb{Z} \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0 \,\}$
We see that ND_{N7} includes IPD_{p2} because IPD_{p2} has one additional condition $i \geq 2$. For every additional condition an "IfNode" is generated in the syntax tree. In our case we have only one additional condition, thus only one node is generated for input port $p2$ at code line 4. This node is denoted as "if i-2 >= 0" in the syntax tree depicted in Figure 2.18.

Third, for every input port $p_k \in N.I_N$ a communication node is generated and added to the syntax tree - code lines 6 and 7. The communication node specifies a communication controller that has to be used for port p_k. The possible controllers were described in Section 2.4.5-**Communication Models Realizations**. We know that port p_k belongs to an input gate, say IG. This gate is connected to a channel, say C. Channel C has an element CM that specifies the communication model of the channel. Depending on the value of CM one of the following communication nodes is generated:

$$communication\ node = \begin{cases} \mathbf{N\_p_k.getT} & \text{if } ((p_k)^{IG})^C.CM = 1 \\ \mathbf{N\_p_k.getRT} & \text{if } ((p_k)^{IG})^C.CM = 2 \\ \mathbf{N\_p_k.getCT} & \text{if } ((p_k)^{IG})^C.CM = 3 \\ \mathbf{N\_p_k.getRCT} & \text{if } ((p_k)^{IG})^C.CM = 4 \end{cases} \tag{2.8}$$

For example, let us consider again input port $p2$ of node $N7$. Port $p2$ belongs to input gate $IG2$ of process $P2$ - see Section 2.4.4-**Examples**. $IG2$ is connected to channel $C11$ which has communication model CM equal to 3, i.e., in-order communication with coloring of tokens. Therefore, the communication node generated for $p2$ and depicted in Figure 2.18 is "N7_p2.getCT".

Fourth, for function F that belongs to node N a function node is generated and added to the syntax tree - code lines 9 and 10. For example, the function that belongs to node $N7$ is defined as follows: $F_{N7} = (F2, \{in\_0\}, \{out\_0\})$. The generated function node is depicted in Figure 2.18 as "[out_0] = F2(in_0)".

Fifth, every output port $q_l \in N.O_N$ is converted to a number of "IfNodes" that are added to the syntax tree - code lines 13 and 14. The conversion is based on the output port domain OPD of q_l and the node domain ND of node N to which port q_l belongs. Always, OPD is included in ND or OPD is equal to ND. We explain the conversion by an example. Let us consider output port $q2$ of node $N7$ - see Section 2.2.3. The output port domain OPD of $q2$ is defined as follows:
$OPD_{q2} = \{\, i \in \mathbb{Z} \,|\, 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0 \,\}$
The node domain ND of node $N7$ is defined as follows:
$ND_{N7} = \{\, i \in \mathbb{Z} \,|\, 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ cond\_2(i) > 0 \,\}$
We see that ND_{N7} is equal to OPD_{q2}. This means that OPD_{q2} does not have additional conditions compared to ND_{N7} for which "IfNodes" have to be generated in the syntax tree. Therefore, for port $q2$ "IfNodes" are not generated in Figure 2.18.

Finally, for every output port $q_l \in N.O_N$ a communication node is generated and added to the syntax tree - code lines 15 and 16. The communication node specifies a communication controller that has to be used for port q_l. The possible controllers were described in Section 2.4.5-**Communication Models Realizations**. We know that port q_l belongs to an output gate, say OG. This gate is connected to a channel, say C. Channel C has an element CM that specifies the communication model of the channel. Depending on the value of CM one of the following communication nodes is generated:

$$communication\ node = \begin{cases} \mathbf{N\_q_l.putT} & \text{if } ((q_l)^{OG})^C.CM = 1 \text{ or } 2 \\ \mathbf{N\_q_l.putCT} & \text{if } ((q_l)^{OG})^C.CM = 3 \text{ or } 4 \end{cases} \tag{2.9}$$

For example, let us consider again output port $q2$ of node $N7$. Port $q2$ belongs to output gate $OG5$ of process $P2$ - see Section 2.4.4-**Examples**. $OG5$ is connected to channel $C10$ which has communication model CM equal to 1, i.e., in-order communication without coloring of tokens. Therefore, the communication node generated for $q2$ and depicted in Figure 2.18 is "N7_q2.putT".

Procedure SCHEDULENODES - definition and example

The pseudo code below defines the procedure SCHEDULENODES(NP, $Stree$). The first input argument of this procedure is a set of nodes $NP = \{N_1, N_2, ..., N_i\}$ that belongs to a process. To every node $N_m \in NP$ a function $N_m.N_F$ is associated (Definition 2.2.2). The name of the function we denote as $N_m.N_F.F$.

```
scheduleNodes( NP, Stree ) begin
1     s ← COPY Stree;
2     for all leafNode_j ∈ s do
3       for all N_m ∈ NP do
4         if N_m.N_F.F = leafNode_j.F then
5           MARK all nodes from leafNode_j to root;
6         endif
7       endfor
8     endfor
9     for all Node_j ∈ s do
10      if Node_j is not marked then
11        REMOVE Node_j;
12      endif
13    endfor
14    return( s );
15  end
```

The second argument of the procedure is a schedule tree $Stree$ as defined in Section 2.3. Every function $N_m.N_F$ described above has a corresponding leaf node $leafNode_j.F$ in $STree$. The output of procedure SCHEDULENODES is a sub-tree derived from $STree$. The sub-tree represents a control structure of a program that gives a valid sequential execution order between all functions $N_m.N_F$ associated with the set of nodes NP described above.

We explain the behavior of procedure SCHEDULENODES using the following example. Let assume the set of nodes $NP = \{N2, N7\}$ of process $P2$. Function $\_Source\_yt$ is associated with $N2$ and function $F2$ is associated with $N7$. We take the schedule tree $Stree$ depicted in Figure 2.6 where functions $\_Source\_yt$ and $F2$ are leaf nodes. First, procedure SCHEDULENODES finds these leaf nodes and traverses $Stree$ from these nodes up to the root node marking every node in the path - see code lines 2 till 8. The marked tree is shown in Figure 2.19-a). Next, the procedure prunes the marked tree by removing all nodes in this tree that are not marked - code lines 9 till 13. The resultant tree is depicted in Figure 2.19-b). If we parse the tree top-down from left to right we can generate a sequential program that specifies a sequential order between functions $\_Source\_yt$ and $F2$.

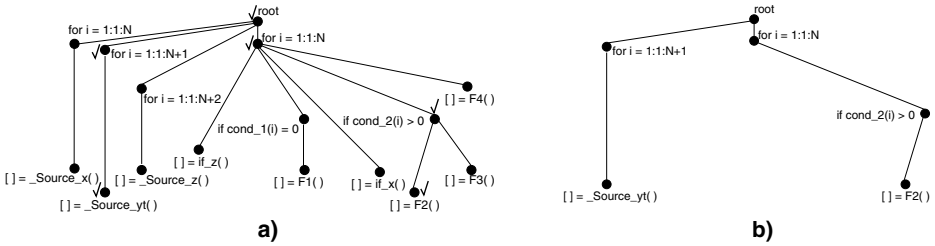

Figure 2.19: Example of applying procedure SCHEDULENODES on the schedule tree shown in Figure 2.6: a) Marking the schedule tree; b) Pruning the schedule tree.

Procedure CONNECTPARSETREES - definition and example

The pseudo code given below defines the procedure CONNECTPARSETREES($tree$, vec). The first input argument ($tree$) of this procedure is a tree generated by procedure SCHEDULENODES defined above. The second input argument is a set of tuples $vec = \{v_1, v_2, ..., v_i\} = \{ (tree_1, N_1), (tree_2, N_2), ..., (tree_i, N_i)\}$ where $tree_1$ to $tree_i$ are trees generated by procedure PROCESSNODE and N_1 to N_i are the nodes corresponding to these trees - see above the definition of procedure PROCESSNODE.

```
connectParseTrees( tree, vec ) begin
1     for all leafNode_j ∈ tree do
2        s ← PARENT of leafNode_j;
3        for all v_m ∈ vec do
4           if v_m.N_m.F = leafNode_j.F then
5              position ← s.removeChild( leafNode_j );
6              s.addChildAt( position, v_m.tree_m );
7           endif
8        endfor
9     endfor
10    return( tree );
11 end
```

The output of procedure CONNECTPARSETREES is a tree that is generated by connecting input argument $tree$ with the trees $tree_1$ to $tree_i$. This is done as follows: 1) for every leaf node $leafNode_j$ that belongs to $tree$ the parent node s is found - code line 2; 2) every node N_m of every tuple v_m in vec is checked whether it matches the current leaf node $leafNode_j$ - see code lines 3 and 4. When a match is found then tree $tree_m$ corresponding to node N_m is connected to $tree$ by replacing $leafNode_j$ and adding $tree_m$ as a "child" of node s - see code lines 5 and 6.

We illustrate the behavior of procedure CONNECTPARSETREES($tree$, vec) by an example. Let us assume that the first argument $tree$ is the tree shown in Figure 2.19-b). This tree is generated by procedure SCHEDULENODES for node $N2$ and $N7$ of process $P2$. Also, let us assume that the second argument vec is equal to $\{(tree_{N2}, N2), (tree_{N7}, N7)\}$ where

$tree_{N2}$ is the tree bounded by the dashed box in the left part of Figure 2.18 and $tree_{N7}$ is the tree bounded by the dashed box in the right part of the same figure. Both trees are generated by procedure PROCESSNODE applied on nodes $N2$ and $N7$. Connecting $tree_{N2}$ and $tree_{N7}$ to $tree$ by applying procedure CONNECTPARSETREES results in the tree shown in Figure 2.18.

2.4.6 Code Generation

In this section, we describe the last step of the PN synthesis (Figure 2.7 - step 4) called *Code Generation*. In this step an executable code of a Kahn process network is generated from a PN model. This PN model is completely defined by the procedures discussed in Section 2.4.4 and Section 2.4.5. We use a software engineering technique called *Visitor* [55] to traverse the PN model structure and to generate the executable code. This code can be expressed in any programming language on top of which an environment to execute Kahn process networks is built. For example, Philips Research has developed the YAPI environment [45] where a Kahn process network can be described in C++, run and simulated. Another example is the PtolemyII framework [12] developed at UC Berkeley where Java is used to describe and run a process network. Recently, the SystemC environment (see www.systemc.org) supports the description and execution of Kahn process networks in C++, as well.

Below, we give an example of how the code generation works for our example network (see Figure 2.9) which was gradually defined in Section 2.4.4 and Section 2.4.5. We generate the network as YAPI C++ code distributed in several header files. The topology of the network is described in header file "Example.h" given in Figure 2.22. The processes $P1$ to $P7$ are described in separate header files "P1.h" till "P7.h". In Figure 2.20 and Figure 2.21 the files "P2.h" and "P7.h" are given that describe the behavior of process $P2$ and process $P7$, respectively.

The procedure that generates the processes is called YAPIPROCESSVISITOR. For every process P_i in a PN model this procedure converts the schedule tree $P_i.ST$ (Definition 2.4.2) into a C++ class file which describes the behavior of process P_i. For process $P2$ in our example, the result of procedure YAPIPROCESSVISITOR is the C++ code in file "P2.h" shown in Figure 2.20.

In code lines 4–6 some base classes of the YAPI environment are included. Code lines 8–78 define the C++ class `P2`. Class `P2` is derived from the base YAPI class "Process" defined in file "process.h". In lines 10–21 gates and variables are declared that are used later in the code[1]. The process is constructed in code lines 24–40 by declaring the constructor of class `P2`. The real work is done in method `void P2::main()` defined at lines 43–76. This method executes a sequential program that describes the internal behavior of process $P2$. This program is generated mainly by parsing top-down from left to right the schedule tree shown in Figure 2.18. At lines 49 and 67 the function calls $\_Source\_yt$ and $F2$ are executed that are the main computation tasks of process $P2$. These tasks are defined in file "aux_func.h". The rest of the code in method `void P2::main()` controls the sequence

[1] At this place the port controllers used in code lines 52, 54, 55, 56, 60, 62, 64, 70, and 71 have to be declared as well. We omit these declarations for the sake of brevity

```
1   #ifndef P2_H
2   #define P2_H
3
4   #include "process.h"
5   #include "port.h"
6   #include "aux_func.h"
7
8   class P2 : public Process {
9
10    private:
11    //Input Gates declaration
12    InPort<Token> IG1; InPort<ColoredToken> IG2; InPort<Token> IG3;
13
14    // OutPut Gates declaration
15    OutPort<Token> OG1; OutPort<ColoredToken> OG2;
16    OutPort<ColoredToken> OG3;
17    OutPort<Token> OG4; OutPort<Token> OG5;
18
19    // Parameters Input Arguments, and Output Arguments
20    int N, *BC2;
21    double in_0, out_0, out_1;
22
23    public:
24      P2( Id n,
25          In<Token>& _IG1, In<ColoredToken>& _IG2, In<Token>& _IG3,
26          Out<Token>& _OG1, Out<ColoredToken>& _OG2,
27          Out<ColoredToken>& _OG3,
28          Out<Token> _OG4; Out<Token> _OG5,
29          int parm_N
30        ) :
31          Process(n),
32          IG1( id("IG1"), _IG1 ), IG2( id("IG2"), _IG2 ),
33          IG3( id("IG3"), _IG3 ),
34          OG1( id("OG1"), _OG1 ), OG2( id("OG2"), _OG2 ),
35          OG3( id("OG3"), _OG3 ),
36          OG4( id("OG4"), _OG4 ), OG5( id("OG5"), _OG5 ), N( parm_N )
37      {
38          int pBC[parm_N + 1];
39          BC2 = pBC;
40      };
41
42      //sequential code executed in process P2
43      void P2::main() {
44        for ( int i = 1; i <= N+1; i += 1 ) {
45           BC2[i] = N+1;
46        } //end for
47
48        for ( int i = 1; i <= N+1; i += 1 ) {
49           _Source_yt( out_0, out_1 );
50
51           if ( i = 1 ) {
52             N2_q1_13->putToken( out_0, i );
53           }
54           N2_q1_14->putColoredToken( out_0, i );
55           N2_q1_15->putColoredToken( out_0, i );
56           N2_q2->putColoredToken( out_1, i );
57        } //end for
58
59        for ( int i = 1; i <= N; i += 1 ) {
60           if ( N7_p1->getToken( i ) > 0 ) {
61              if ( i-2 >= 0 ) {
62                 in_0 = N7_p2->getColoredToken( i );
63              } else {
64                 in_0 = N7_p3->getToken( i );
65              }
66
67              F2( in_0, out_0 );
68              BC2[i+1] = i;
69
70              N7_q1->putToken( out_0, i );
71              N7_q2->putToken( BC2[i+1], i );
72           } else {
73              N7_q2->putToken( BC2[i+1], i );
74           }
75        } //end for
76      } //end main
77
78  }; //end class P2
79
80  #endif /* P2_H */
```

Figure 2.20: YAPI C++ code describing the behavior of process P2.

of executions of tasks $\_Source\_yt$ and $F2$. Also, the code controls from/to which ports the input/output arguments of $\_Source\_yt$ and $F2$ are read/written. For example, code lines 61–65 specify at which iterations i input argument in_0 of $F2$ is read from port $N7\_p2$ of input

gate $IG2$ and at which iterations - from port $N7\_p3$ of input gate $IG3$. The actual reading is done by activating the controllers at lines 62 and 64. These controllers implement the procedures described in Figure 2.16 and Figure 2.12 where the primitives `READ(`$IG2$`)` and `READ(`$IG3$`)` are used to read tokens from channels $C11$ and $C12$ connected to input gates $IG2$ and $IG3$, respectively.

In Figure 2.21 we show the code generated by procedure YAPIPROCESSVISITOR for process $P7$ of our example PN model. This code is the file "P7.h" where the class `P7` is defined at code lines 8 till 70. The structure of class `P7` is similar to the structure of class `P2` explained above.

The procedure that generates the topology of a network is called YAPINETWORKVISITOR. This procedure converts the structure of a PN model into a class file describing the topology. In our case the result of procedure YAPINETWORKVISITOR is the C++ code in file "Example.h" shown in Figure 2.22. In code lines 4–6 some base classes of the YAPI environment are included. The classes that describe the processes of the network are included at lines 8–14. For examples, code line 9 includes class `P2` given in Figure 2.20 and code line 14 includes class `P7` given in Figure 2.21.

Code lines 16–67 define the C++ class `Example` which corresponds to our example PN model. Class `Example` is derived from the base YAPI class "Process Network" defined in file "network.h". The structure of class `Example` begins with declaration of the communication channels of our PN model. At lines 20–36 the channels $C1$ to $C17$ are declared as instances of class `Fifo` which is a base YAPI template class. Also, the type of the data communicated over every channels is specified. For example, channel $C1$ at line 20 communicates data that is not tagged (colored), i.e., data of type `Token`. Channel $C13$ at line 32 communicates tagged (colored) data, i.e., data of type `ColoredToken`.

The processes named as $P1\_instance$ to $P7\_instance$ are declared as instances of classes `P1` to `P7`. This is done at code lines 39–45. Next, the process network is constructed by code lines 48–65 where the constructor of class `Example` is defined. The topology of the process network is specified at code lines 56–62 where the connections of processes $P1\_instance$ to $P7\_instance$ via channels $C1$ to $C17$ are given.

2.4.7 Discussion and Conclusions

In this chapter we presented a novel systematic approach that allows automatic derivation of executable Kahn Process Network (KPN) specifications from Weakly Dynamic Programs (WDP). The problem of deriving a KPN specification for an application in a systematic and automated way has been addressed in the past by fellow researchers in the work presented in [18] [19] [20] [21] [22] [23]. This work reports techniques for automatic derivation of Kahn Process Networks from applications specified as *static affine nested loop programs* (SANLP). The main property of such program is that everything about the program execution is known at compile time. In contrast, our approach presented in this chapter targets Weakly Dynamic Programs (WDP) where the program execution may not be known completely at compile time, thus making such programs more difficult for analysis and conversion to a KPN specifications.

```
1   #ifndef P7_H
2   #define P7_H
3
4   #include "process.h"
5   #include "port.h"
6   #include "aux_func.h"
7
8   class P7 : public Process {
9
10    private:
11    //Input Gates declaration
12    InPort<Token> IG1; InPort<Token> IG2; InPort<ColoredToken> IG3;
13    InPort<Token> IG4; InPort<Token> IG5;
14
15    // OutPut Gates declaration
16    OutPort<ColoredToken> OG1; OutPort<Token> OG2; OutPort<Token> OG3;
17
18    // Parameters Input Arguments, and Output Arguments
19    int N, c;
20    double in_0, in_1, out_0, out_1;
21
22    public:
23      P7( Id n,
24          In<Token>& _IG1, In<Token>& _IG2, In<ColoredToken>& _IG3,
25          In<Token>& _IG4, In<Token>& _IG5,
26          Out<ColoredToken>& _OG1, Out<Token>& _OG2, Out<Token>& _OG3,
27          int parm_N
28        ) :
29          Process(n),
30          IG1( id("IG1"), _IG1 ), IG2( id("IG2"), _IG2 ),
31          IG3( id("IG3"), _IG3 ),
32          IG4( id("IG4"), _IG4 ), IG5( id("IG5"), _IG5 ),
33          OG1( id("OG1"), _OG1 ), OG2( id("OG2"), _OG2 ),
34          OG3( id("OG3"), _OG3 ), N( parm_N )
35      {   };
36
37      //sequential code executed in process P7
38      void P7::main() {
39        for ( int i = 1; i <= N; i += 1 ) {
40           c = N9_p1->getToken( i );
41           if ( -c+i >= 0 ) {
42              if ( c-i >= 0 ) {
43                 in_0 = N9_p2->getToken( i, c );
44              } else {
45                 in_0 = N9_p3->getReorderColoredToken( i, c );
46              }
47           } else {
48              in_0 = N9_p4->getReorderColoredToken( i, c );
49           }
50
51           if ( i-3 >= 0 ) {
52              in_1 = N9_p5->getToken( i );
53           } else
54              in_1 = N9_p6->getToken( i );
55           }
56
57           F4( in_0, in_1, out_0, out_1 );
58
59           if ( -i+N-1 >= 0 ) {
60              N9_q1->putColoredToken( out_0, i );
61           }
62
63           if ( -i+N-2 >= 0 ) {
64              N9_q2_17->putToken( out_1, i );
65              N9_q2_18->putToken( out_1, i  );
66           }
67        } //end for
68      } //end main
69
70  }; //end class P7
71
72  #endif /* P7_H */
```

Figure 2.21: YAPI C++ code describing the behavior of process P7.

Although, the program execution of a WDP is not known completely at compile time, we have shown in this chapter that still a WDP can be analyzed and transformed into a KPN in a formal and structured way. To do this we used Fuzzy Array Dataflow Analysis techniques and we introduced the notions of Dynamic Single Assignment Code, Approximated Dependence Graph, and Linearly Bounded Sets. Our definition of a WDP (see Chapter 1) includes the static affine nested loop programs considered in the past as a special case. This means that our approach extends the range of applications where KPNs can be derived in a systematic

```
1   #ifndef Example_H
2   #define Example_H
3
4   #include "fifo.h"
5   #include "process.h"
6   #include "network.h"
7
8   #include "P1.h"
9   #include "P2.h"
10  #include "P3.h"
11  #include "P4.h"
12  #include "P5.h"
13  #include "P6.h"
14  #include "P7.h"
15
16  class Example : public ProcessNetwork {
17
18    private:
19      // Fifo Channels instantiation
20      Fifo<Token> C1;
21      Fifo<Token> C2;
22      Fifo<Token> C3;
23      Fifo<Token> C4;
24      Fifo<ColoredToken> C5;
25      Fifo<Token> C6;
26      Fifo<ColoredToken> C7;
27      Fifo<ColoredToken> C8;
28      Fifo<Token> C9;
29      Fifo<Token> C10;
30      Fifo<ColoredToken> C11;
31      Fifo<Token> C12;
32      Fifo<ColoredToken> C13;
33      Fifo<Token> C14;
34      Fifo<Token> C15;
35      Fifo<Token> C16;
36      Fifo<Token> C17;
37
38      // Processes instantiation
39      P1 P1_instance;
40      P2 P2_instance;
41      P3 P3_instance;
42      P4 P4_instance;
43      P5 P5_instance;
44      P6 P6_instance;
45      P7 P7_instance;
46
47    public:
48      Example(Id n, int parm_N) :
49      ProcessNetwork(n),
50      C1( id("C1") ), C2( id("C2") ),
51      C3( id("C3") ), C4( id("C4") ), C5( id("C5") ),
52      C6( id("C6") ), C7( id("C7") ), C8( id("C8") ),
53      C9( id("C9") ), C10( id("C10") ), C11( id("C11") ),
54      C12( id("C12") ), C13( id("C13") ), C14( id("C14") ),
55      C15( id("C15") ), C16( id("C16") ), C17( id("C17") ),
56      P1_instance( id("P1"), C7, C8, parm_N ),
57      P2_instance( id("P2"), C4, C11, C12, C12, C13, C5, C9, C10, parm_N ),
58      P3_instance( id("P3"), C14, C17, parm_N ),
59      P4_instance( id("P4"), C6, C7, C14, C15, C1, C2, C6, parm_N ),
60      P5_instance( id("P5"), C1, C2, C8, C3, C4, parm_N ),
61      P6_instance( id("P6"), C3, C5, parm_N ),
62      P7_instance( id("P7"), C10, C9, C13, C16, C17, C11, C15, C16, parm_N )
63      {
64
65      };
66
67  }; //end class Example
68
69  #endif /* Example_H */
```

Figure 2.22: YAPI C++ code describing the topology of the process network defined in Section 2.4.4-**Examples** and visualized in Figure 2.9.

and automated way.

In a KPN derived from a WDP we distinguish two types of communication FIFO channels depending on the purpose of the communicated data: 1) *Data FIFO channels* where computational data used/generated by function calls (tasks) executed inside processes is communicated; 2) *Control FIFO channels* where data that controls the internal sequential behavior of processes is communicated. By sequential behavior of a process we mean the sequential

order of execution of function calls inside the process.

The control FIFO channels appear in a KPN derived from a WDP because the behavior of the WDP is not know completely at compile time. The unknown behavior has to be resolved at run time in the KPN and the control FIFO channels are used to communicate the necessary data to do this. Control FIFO channels do not appear in case a KPN is derived from a static program. This means that the presence of control FIFO channels, i.e., extra communication workload is the ”price” we have to pay when deriving KPNs from WDPs.

Most of the methods and techniques of our approach presented in this chapter have been prototyped as software procedures and tested on a small set of sample weakly dynamic programs (WDP). The running example used in this chapter was also completely generated by our prototype implementation. Moreover, the approach and the prototype software have been applied and validated successfully on a real-life application, namely a Motion JPEG encoder (MJPEG) - see Chapter 4 where we present a case study in which we apply our approach to derive a KPN specification for the MJPEG encoder application. Although, our approach is not fully automated yet, the MJPEG KPN was derived in a semi-automatic way in four days. For comparison, a KPN specification of an MJPEG encoder was derived by hand in [16] that took four weeks. The facts above prove the feasibility of using our approach to derive KPNs from large industrial relevant applications in a relatively short amount of time - currently, a few days. When our approach is fully automated this time will be reduced to several minutes.

The MJPEG KPN was generated as C++ code in YAPI [45] format which allowed us to run the network and to verify its functional correctness. The computational workload of the MJPEG program was partitioned by our approach into 9 concurrent processes and the communication workload was distributed over 19 FIFO channels. We analyzed the network and concluded that the computational workload was very well balanced over the 9 concurrent processes. However, the distribution of the communication workload was not optimal. We found that the number of communication FIFO channels can be reduced from 19 to 14 FIFOs by merging some FIFOs without obstructing the exploited parallelism.

The presented approach includes the basic techniques that we have developed to derive automatically KPNs from WDPs. The results, we have obtained for the MJPEG application, indicated that as a future work some optimization techniques have to be added to the approach that will help the improving of the quality of the generated KPNs in terms of optimal partitioning of the computation and communication workloads of a WDP over processes and channels in the KPN.

Chapter 3

Algorithmic Transformation Techniques

Today, most of the system-level design methodologies are based on the Y-chart paradigm [58] [4]. Following the Y-chart for designing a system, an application and an architecture are modeled separately and mapped onto each other in an explicit design step. Next, a performance analysis for alternative application instances, architecture instances and mappings has to be done, thereby exploring the design space of the target system. Deriving alternative application instances is not trivially done. Nevertheless, many instances of a single application exist that are worth to be derived for exploration. In this chapter, we present algorithmic transformation techniques that we have developed for systematic and fast derivation of alternative application instances that express task-level concurrency hidden in an application in some degree of explicitness. These techniques facilitate a system designer in the design space exploration process of application instances.

This chapter is organized as follows. In Section 3.1 we introduce the problem of deriving alternative application instances in the context of system-level design and the importance of finding a systematic approach to derive them. Section 3.2 introduces our set of four transformations, namely *unfolding*, *plane cutting*, *skewing*, and *merging* that we have developed to derive alternative application instances. Also, these transformations are shown as a part of an *Application Transformation Layer* positioned in the Y-chart exploration paradigm. Next, in Section 3.3 - unfolding, Section 3.4 - plane cutting, Section 3.5 - skewing, and Section 3.6 - merging, we present the transformations by explaining the general idea behind the transformation, the formal procedure to do the transformation, and we give an example which illustrates the formal procedure. Finally, we conclude the chapter with a discussion and concluding remarks in Section 3.7.

3.1 Introduction

In system-level design of embedded multi-processor systems, a system designer sees the target system as the pair *Application(s) specification - Architecture template*. An example of such a pair is shown in the left part of Figure 3.1. The application specification provides the

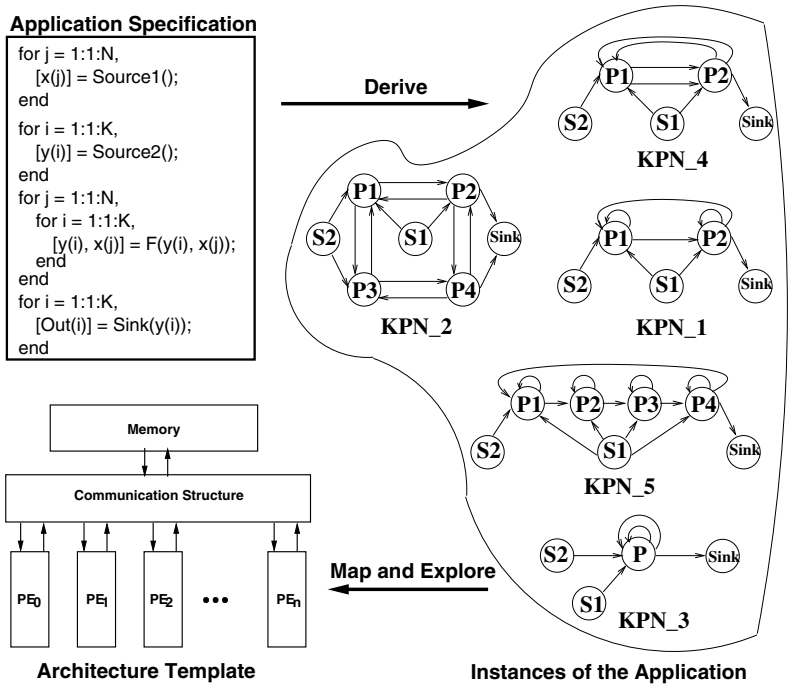

Figure 3.1: Alternative instances of the application have to be derived, mapped onto the architecture template and explored in order to evaluate the performance/cost of the *Application-Architecture* pair.

functional behavior of the system. The architecture template specifies the organization of the resources of the system onto which the functional behavior is to be mapped. In this stage, a designer has to make some design decisions, for example, how to partition the application into tasks, how to map the tasks onto the architecture template, what kind of communication structure to use in the architecture template, etc. In order to evaluate different design decisions, a system designer uses a model of the target system and does performance analysis for alternative application instances, architecture instances and mappings, thereby exploring the design space of the *Application - Architecture* pair.

A general scheme for a design space exploration is the Y-chart paradigm [58]. Tools like POLIS [4], SPADE [48] [16], ORAS [59], Archer [60] [61], Sesame [62] [46] implement techniques that support the Y-chart paradigm but they focus only on the exploration of alternative architecture instances and mappings. In this chapter, however, we focus on techniques that support efficient exploration of alternative application instances in system-level design.

An application instance is any feasible partitioning of an application into a composition of concurrent tasks. We use the Kahn Process Network (KPN) model of computation to describe application instances. In the Kahn model, concurrent processes communicate via unbounded FIFO channels. In Figure 3.1, we show a simple application and a set of alternative KPN instances of this application (KPN_1 to KPN_5). Each application instance differs from the others in the degree of exploited *task-level* parallelism. The performance of the *Application - Architecture* pair can significantly depend on the application instance. So, a system designer needs support to derive a set of instances of an application in order to explore and evaluate the performance of the system and to select an application partitioning that satisfies requirements the target system has to meet.

In general, a system designer is only able to derive at most a few alternative application instances. This is so because no systematic way to derive an application instance, let alone alternatives, from an application specification is known, as a result of which heuristic and time consuming approaches are taken in practice. Nevertheless, many instances of a single application exist that are worth to be derived for exploration. We present in this chapter algorithmic transformations that we have developed and implemented in order to help a system designer to derive systematically and fast alternative application instances. These transformations together with the COMPAANDYN approach presented in Chapter 2 are encapsulated in an *Application Transformation Layer* that derives a set of alternative KPN instances from an application specified as a weakly dynamic program (WDP). Below, we show the position of the *Application Transformation Layer* in the Y-chart paradigm.

3.2 Application Transformation Layer

In this section, we discuss the application transformation layer in the context of the design space exploration process. We use this layer as an extension to the *Y-chart environment* [58]. The positioning of the transformation layer is shown in Figure 3.2.

We start with an application specified as a WDP and written in an imperative language like Matlab or C. Our objective is to derive and explore a set of instances (Kahn Process Networks) functionally equivalent to the application. First, algorithmic transformations are applied to the application specification. The transformations are controlled by a set of parameters. At the beginning some initial values are assigned to the parameters depending on the available resources in the architecture template. With these values, the original code of the application is automatically transformed and structured in a particular way in order to make the parallelism that is inherently available in the application explicit and to increase or decrease the *task-level* parallelism in the application. Second, the transformed code is converted in a systematic and automated way to a KPN description using the COMPAANDYN approach described in Chapter 2. Third, we use a Y-chart environment to map the KPN onto an architecture template and do performance analysis. The result of this performance analysis can be used to change the values of the parameters (step 4 in Figure 3.2) if the system performance is not satisfactory. Then, we repeat the procedure described above resulting in a *design space exploration* of alternative instances of the application. This is shown in Figure 3.2 as a feed-back arrow to the transformation layer.

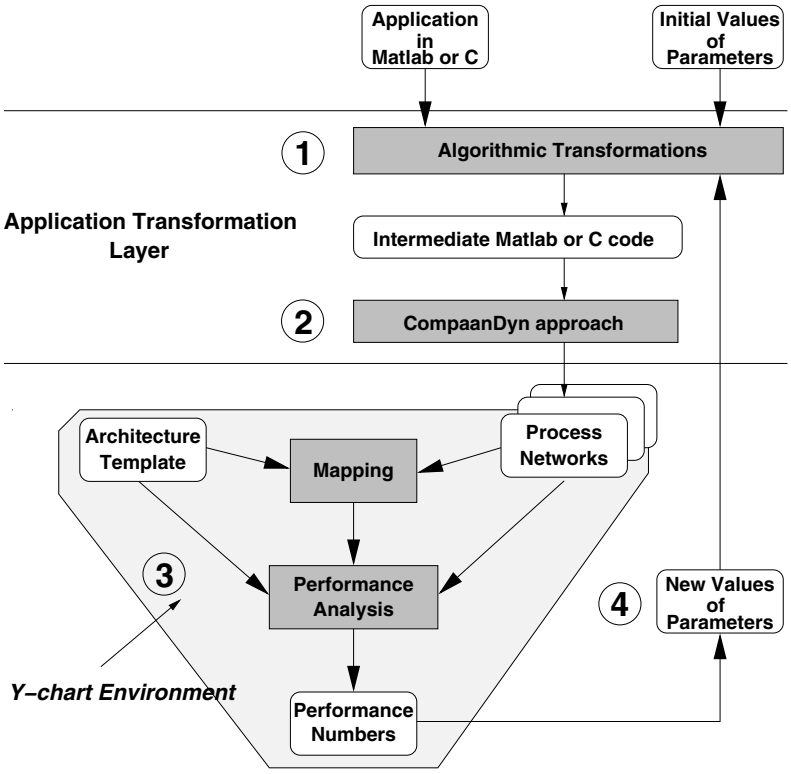

Figure 3.2: The Y-chart extended with the *Application Transformation Layer*.

By changing the values of the parameters, the application transformation layer systematically derives a set of KPN specifications corresponding to a single application specified as a WDP. The difference among the KPNs is the degree of the task-level parallelism that is exploited. In the sections that follow we describe in detail the algorithmic transformations we have developed and incorporated in the application transformation layer. We present four algorithmic transformations, namely *Unfolding*, *Plane Cutting*, *Skewing*, and *Merging*. These transformations take as an input a weakly dynamic program (WDP) and a set of parameters. The output of the unfolding and plane cutting transformations is a weakly dynamic program which is functionally equivalent to the input program but with increased task-level parallelism. The skewing transformation makes the potential parallelism in the input weakly dynamic program explicit. The output of the merging transformation is the input WDP annotated with special pragma symbols that specify how the function calls in the WDP have to be grouped in processes by COMPAANDYN, thereby decreasing the available task-level parallelism.

The unfolding, plane cutting, and skewing transformations operate directly on the WDP source code without using complex intermediate representations like dependence graphs, signal-flow graphs or data-flow graphs corresponding to the WDP. Therefore, we have implemented these transformations in a source-to-source tool box called MATTRANSFORM. The merging transformation operates on the approximated dependence graph (ADG) and sched-

ule tree (STree) that are obtained from the WDP. This transformation is a part of STEP3 in our COMPAANDYN approach.

3.3 Unfolding Transformation

First, we explain the general idea behind our *unfolding* transformation in Section 3.3.1 by an illustrative example. To keep the example simple and clear we apply the unfolding transformation on a special case of a WDP which is a static affine nested loop program. Next, in Section 3.3.2, we define the unfolding transformation as a formal procedure which operates on a general WDP. Finally, we explain the formal procedure by going through an example in Section 3.3.3.

3.3.1 General Idea

Let us consider the nested loop program (NLP) shown in Figure 3.3-a). The NLP has two

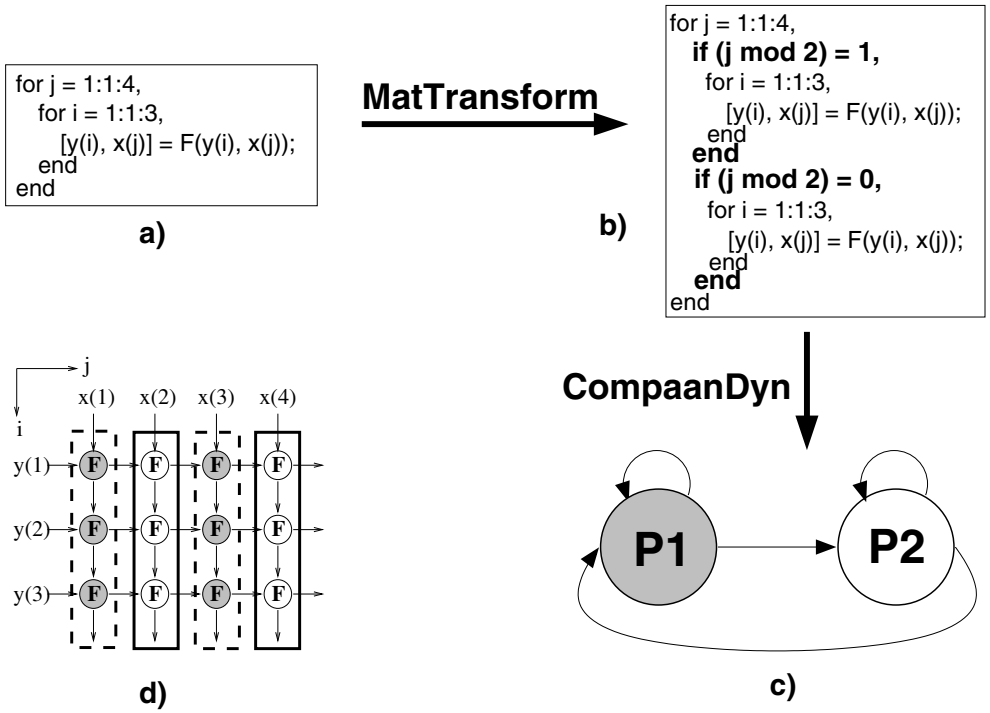

Figure 3.3: Simple example illustrating the unfolding transformation.

loops with iterators j and i. The body of the loop nest executes a single task represented by the function call F for every iteration (j, i), thereby creating the computational workload of the NLP. This workload can be distributed over several processes in order to exploit some

degree of parallelism. Our new approach to get the desired degree of parallelism - at the task level - is to copy a loop body a number of times in such a way that these copies are *mutually exclusive*. We call this new approach *unfolding* and we have implemented it in MATTRANSFORM as a source-to-source *unfolding* transformation.

An example of applying our unfolding transformation is shown in Figure 3.3-b), where the j-loop of the NLP is unfolded by a factor of two. The body of loop j is copied two times. Every copy is bounded by an "if" statement where the condition is a *modulo* function of the loop iterator j as shown in Figure 3.3-b). The modulo functions in the conditions are used in a particular way to make the execution of the two pieces of code bounded by the "if" statements mutually exclusive. The mutually exclusiveness is exploited in our COMPAANDYN approach to distribute the computational workload of the program given in Figure 3.3-b) over two concurrent processes. These two processes form a Kahn Process Network with a topology depicted in Figure 3.3-c).

In the example given above our unfolding transformation is used to unfold loop j by a factor of two, thereby distributing the workload over two processes. This transformation can be used to unfold loop j by any factor u_j where $u_j \in \mathbb{N} \bigwedge u_j \in [1..4]$ as well as to unfold loop i by any factor u_i where $u_i \in \mathbb{N} \bigwedge u_i \in [1..3]$ [1]. This means that our unfolding transformation can be used to distribute the computational workload of the NLP in Figure 3.3-a) over N processes where $N = u_j \times u_i$. If loop j is unfolded by a factor $u_j = 4$ and loop i is unfolded by a factor $u_i = 3$ then the workload of the NLP will be distributed over 12 concurrent processes in the process network shown in Figure 3.3-d). In this network, every process executes the function call F only once, thereby exploiting the maximum task-level parallelism available in the NLP. The topology of the network is equal to the dependence graph (DG) that corresponds to the NLP [2].

The process network in Figure 3.3-c) is functionally equivalent to the network in Figure 3.3-d) but it exploits in less degree the task-level parallelism available in the NLP. This is because the computational workload of the NLP is distributed over two concurrent processes $P1$ and $P2$. $P1$ executes in a sequential order the function calls bounded by the dashed boxes depicted in Figure 3.3-d). Process $P2$ executes in a sequential order the function calls bounded by the solid boxes.

In general, our unfolding transformation can be used to derive a set of alternative Kahn Process Networks from a weakly dynamic program (WDP) by unfolding the loops in the WDP with different factors. We have developed this transformation in a specific way to exploit task-level parallelism available in a WDP. There is a relation between our unfolding transformation and the well known compiler transformation loop unrolling [39] in the sense that both transformations aim at enhancing parallelism in a sequential program. However, the loop unrolling enhances instruction level parallelism by copying a loop body several times and re-indexing the variables in the body, thus creating more parallel instructions and reducing the loops control overhead. In contrast, our unfolding transformation enhances task-level parallelism by copying a loop body a number of times in such a way that these copies are mutually exclusive, thus these copies can be encapsulated in concurrent processes.

[1] Unfolding factor of one means that the corresponding loop is not unfolded.

[2] The DG is a graphical representation of the NLP. The nodes in the DG represent the NLP function calls that are executed in each loop iteration and the edges represent the data dependencies between the function calls.

3.3.2 Formal Procedure

Let WDP be a weakly dynamic program and let us assume that WDP has N loops. The iterators of these loops form an iterator vector $I = \{i_1, i_2, ..., i_N\}$. For every loop iterator $i_k \in I \mid k = 1, 2, .., N$ a parameter $u_k \in \mathbb{N}$ is associated which is the unfolding factor of the loop i_k. All parameters u_k form a parameter vector $U = \{u_1, u_2, ..., u_N\}$ which we call *unfolding vector*. We define a transformation `UNFOLD(WDP,U,I)` which is described below:

- **STEP1 -** Convert WDP to a parse tree representation, called $parseTree$ for short. The $parseTree$ representation is the syntax tree [54] corresponding to the WDP where to every "For"-statement node in the tree an unfolding factor is assigned based on the information captured in the iterator vector I and the unfolding vector U. There are very well known procedures used in the compiler community to convert a program to a syntax tree representation [54]. These procedures can be used to convert our WDP to a parse tree representation.

- **STEP2 -** Transform the parse tree representation obtained in STEP1 by applying the procedure **treeUNFOLD** given in Figure 3.4. This procedure takes as an input the root node of $parseTree$. Starting from the root node the procedure visits every node in $parseTree$ by traversing it from top to bottom and from left to right. This is accomplished by calling the procedure **treeUNFOLD** recursively - see code lines 32 till 37. When visiting a node, **treeUNFOLD** checks if this node is a "For"-statement node and if the unfolding factor is greater than one - code lines 3 and 9. If both conditions are true then: 1) the children nodes of the current node together with the sub-trees starting from them are stored and removed from the current node - code lines 11 and 13; 2) several "If"-statement nodes are created and these nodes become children of the current node - see code lines 15 till 25. The number of "If"-statement nodes is equal to the unfolding factor of the current node. For every "If"-statement node a special condition is assigned as shown in code line 19 as well as the children of the current node become children of the "If"-statement node - code line 22; 3) the current "For"-statement node is marked as processed by assigning a zero value to the unfolding factor - code line 28.

- **STEP3 -** Convert the transformed $parseTree$ in STEP2 to a weakly dynamic program. This step is the inverse of STEP1. Every node in $parseTree$ is visited by traversing the tree top-down from left to right. For every visited node corresponding lines of code are generated, thereby creating the weakly dynamic program.

3.3.3 Example

In order to explain the behavior of the procedure `UNFOLD(WDP,U,I)` described above we consider the following example. Let WDP be the weakly dynamic program shown in Figure 3.5-a). WDP has only two for-loops. The outer loop has an iterator i_1 with lower bound lb_1, upper bound ub_1, and iterator step $step_1$. The body of this loop consists of three parts; two code segments $< code1 >$ and $< code2 >$, and a for-loop. $< code1 >$ and $< code2 >$ are arbitrary pieces of code that comply with our definition of weakly dynamic

```
1  treeUNFOLD( node ) {
2   // Check this Statement
3   if (node is "For"-statement) {
4     // take the unfolding factor of the "for"-statement
5      factor = node.unfoldFactor;
6     // take the iterator of the "for"-statement
7      iterator = node.loopIterator;
8
9     if ( factor > 1 ) {
10       // store the children nodes of the "for"-statement
11        tempList = node.listOfChildren;
12       // make the list of children empty
13       node.listOfChildren =  NULL;
14
15       for (i = 1; i <=  factor; i = i+1) {
16         // create a node which is "if"-statement
17          tempNode =  create "If"-statement;
18         // set a string which is the condition of the "if"-statement
19          tempNode.condition = "("+iterator+" mod "+factor+") = "+factor-i;
20
21         // the children of the "for"-statement become children of the "if"-statement
22          tempNode.listOfChildren = tempList;
23         // add the "if"-statement as a child of the "for"-statement
24         node.listOfChildren[i-1] = tempNode;
25       }
26
27       // indicate that this "for"-statement has been processed
28       node.unfoldFactor = 0;
29    }
30  }
31
32  // get the number of children of the current node
33   numElements = getNumberOfElemnts( node.listOfChildren );
34  // go down recursively to check and unfold the children
35  for (j = 0; j < numElements; j = j+1) {
36      treeUNFOLD( node.listOfChildren[j] );
37  }
38
39  return;
40 }
```

Figure 3.4: Pseudo code describing the **treeUNFOLD** procedure.

program. The for-loop has an iterator i_2 with lower bound lb_2, upper bound ub_2, and iterator step $step_2$. Its loop body $< code3 >$ is an arbitrary piece of code.

The iterator vector I corresponding to WDP has two elements $I = \{i_1, i_2\}$ because WDP has only two loops. We assign to every element of I a number which is the unfolding factor of the corresponding loop. In our example we unfold the outer loop by a factor of two and the inner loop by a factor of three. In this case the unfolding vector $U = \{u_1, u_2\} = \{2, 3\}$. Now, let us apply the procedure `UNFOLD(WDP,U,I)` step-by-step on our example program given in Figure 3.5-a).

STEP1

First, we have to convert WDP to a parse tree representation. The resulting parse tree is shown in Figure 3.5-b). The parse tree consists of five nodes. Every program statement in WDP has a corresponding node in the parse tree. In our example, $node1$ and $node3$ correspond to the outer for-loop and inner for-loop statements, respectively. We call $node1$

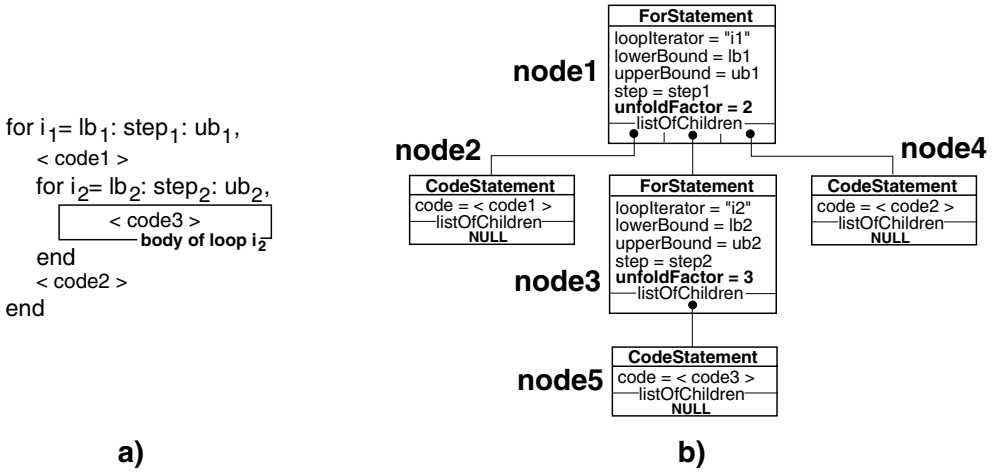

Figure 3.5: Example of: a) Weakly Dynamic Program and b) its parse tree representation.

and $node3$ "For"-statement nodes. These nodes contain the information about the loops present in the WDP. Similarly, $node2$, $node4$, and $node5$ correspond to the code segments $< code1 >$, $< code2 >$, and $< code3 >$, respectively. These nodes we call "Code"-statement nodes.

Every node in the parse tree has a list of pointers, called $listOfChildren$, that connect the node with several other nodes. These connections are shown as edges in Figure 3.5-b). Every edge defines a "father-child" relationship between two nodes. For example, $node1$ is a "father" node of its three children $node2$, $node3$, and $node4$. Also, $node3$ is a "father" node of $node5$. The existence of a "father-child" relationship between two nodes in the parse tree depends on the position of the corresponding program statements in WDP. For example, $node2$, $node3$, and $node4$ are "children" of $node1$ because: 1) the outer for-loop in WDP corresponding to $node1$ has a body in which the code segment $< code1 >$, the inner for-loop, and the code segment $< code2 >$ are located and they correspond to $node2$, $node3$, and $node4$ in the parse tree; 2) $< code1 >$, the inner for-loop, and $< code2 >$ are not nested in other program statements.

At the end of STEP1, the procedure `UNFOLD(WDP,U,I)` assigns to every "For"-statement node in the parse tree a number which is the unfolding factor. In our example, $node1$ and $node3$ are "For"-statement nodes and they have the special field $unfoldFactor$ which has to be set with the corresponding number taken from the unfolding vector $U = \{u_1, u_2\} = \{2, 3\}$. So, $node1$ has $loopIterator = i_1$, therefore $unfoldFactor = u_1 = 2$. Similarly, $node3$ has $loopIterator = i_2$, therefore $unfoldFactor = u_2 = 3$.

STEP2

Here, the recursive procedure **treeUNFOLD** given in Figure 3.4 is applied on the parse tree depicted in Figure 3.5-b). The procedure transforms this parse tree into the parse tree shown in Figure 3.6.

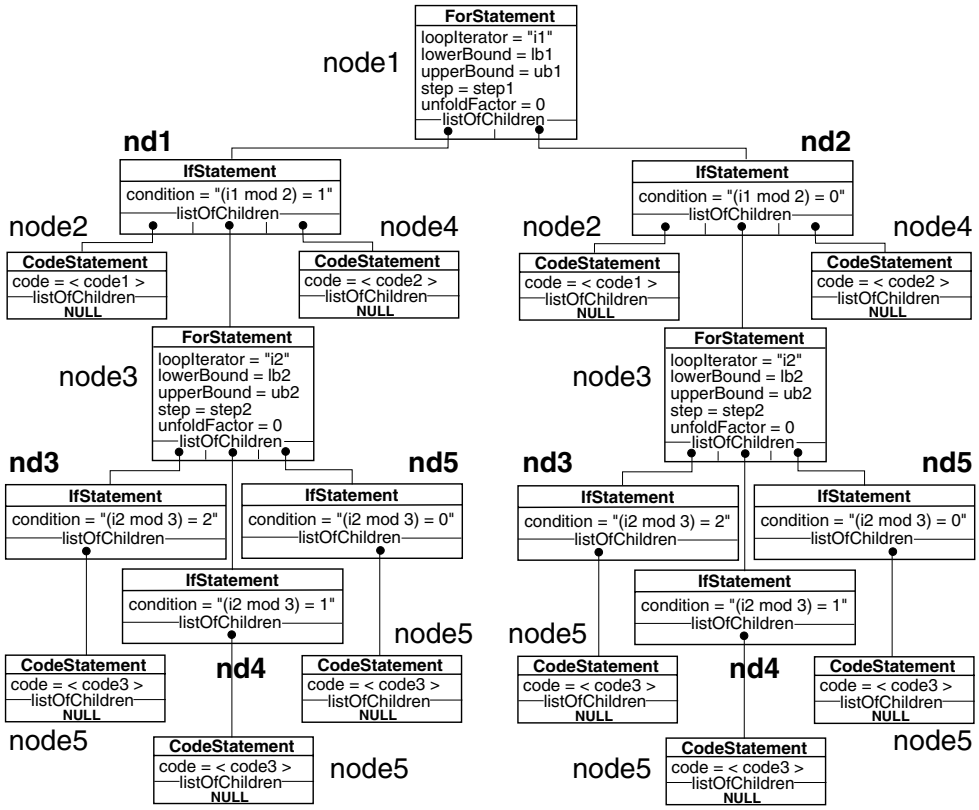

Figure 3.6: The parse tree in Figure 3.5-b) transformed by the procedure **treeUNFOLD**.

The procedure **treeUNFOLD** takes as an input the root node of the parse tree. In our example this is $node1$. Then, **treeUNFOLD** checks whether $node1$ is a "For"-statement node - see code line 3 in Figure 3.4. In our case this is true and **treeUNFOLD** checks whether the unfolding factor $unfoldFactor$ of $node1$ is greater than one - code line 9. Since $unfoldFactor = 2$, the code lines 10 till 29 are executed. As a result, the two "If"-statement nodes **nd1** and **nd2** in Figure 3.6 are created. The field $condition$ of **nd1** and **nd2** is generated by code line 19. The children of $node1$ ($node2$, $node3$, and $node4$) become children of **nd1** and **nd2** - code line 22. Also, **nd1** and **nd2** become children of $node1$ - code line 24. $Node1$ is marked that it has been processed by setting its $unfoldFactor = 0$ - code line 28.

The procedure **treeUNFOLD** continues with executing code line 33. At this point, $node1$ has two children **nd1** and **nd2**. Therefore, code line 36 is executed two times by calling the procedure **treeUNFOLD** recursively. The input node of the first recursive call is **nd1**. The second recursive call has as an input **nd2**. These recursive calls complete the parse tree transformation by creating the nodes **nd3**, **nd4**, and **nd5**. The final transformed parse tree is shown in Figure 3.6.

STEP3

In this final step of the procedure `UNFOLD(WDP,U,I)` the transformed parse tree depicted in Figure 3.6 is converted to the weakly dynamic program shown in Figure 3.7.

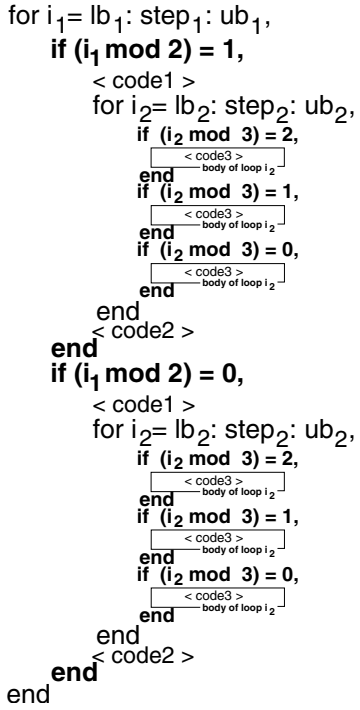

Figure 3.7: The weakly dynamic program in Figure 3.5-a) unfolded by the procedure `UNFOLD(WDP,U,I)`. $I = \{i_1, i_2\}$ and $U = \{2, 3\}$.

3.4 Plane Cutting Transformation

The general idea of our *plane cutting* transformation is explained in Section 3.4.1. We explain the idea with the same illustrative example used in Section 3.3.1. Next, in Section 3.4.2, we define the plane cutting transformation as a formal procedure which operates on a general WDP. Finally, we explain the formal procedure by going through an example in Section 3.4.3.

3.4.1 General Idea

Let us consider the nested loop program (NLP) shown in Figure 3.8-a). This NLP program was used as an example in Section 3.3.1 where we distributed its workload over two processes

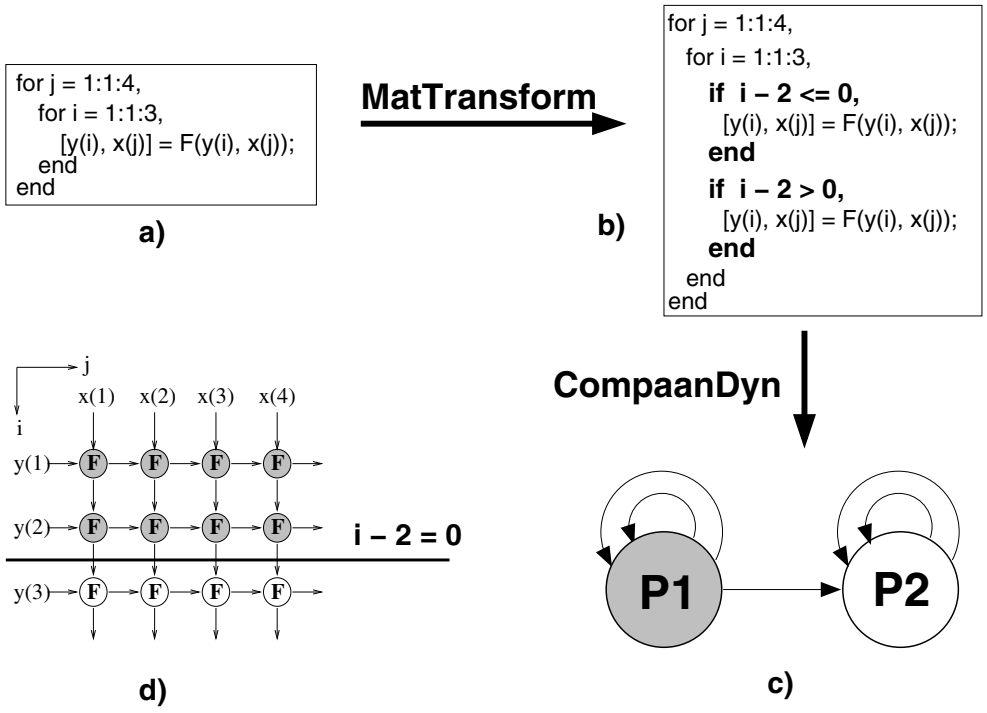

Figure 3.8: Simple example illustrating the plane cutting transformation.

by applying the unfolding transformation. The distribution was done by partitioning the iteration space (j, i) of the NLP using "if" statements with modulo conditions. Here, in contrast, we present another approach to distribute the workload of the NLP by partitioning the iteration space using *hyper planes*. By choosing different planes we can exploit different degrees of task-level parallelism. This approach we call *plane cutting* and we have implemented it as a source-to-source transformation in MATTRANSFORM.

An example of applying our plane cutting transformation is shown in Figure 3.8-b). We use the plane $i - 2 = 0$ to split the body of the inner loop i in two parts [3]. This is accomplished by copying the body of loop i two times. The two copies are bounded by "if" statements with conditions $i - 2 \leq 0$ and $i - 2 > 0$ which keep the transformed program in Figure 3.8-b) functionally equivalent to the initial program given in Figure 3.8-a). The splitting of the loop body in the particular way described above is exploited in our COMPAANDYN approach to distribute the computational workload of the transformed program given in Figure 3.8-b) over two concurrent processes in the Kahn Process Network (KPN) depicted in Figure 3.8-c).

If we compare this KPN with the KPN in Figure 3.3-c) (obtained by unfolding) we see that both KPNs have two processes, i.e., they exploit similar degree of task-level parallelism. However, these KPNs have different topologies. On the one hand, the KPN depicted in

[3] In the two dimensional iteration space (j, i) of the example, the plane equation $i - 2 = 0$ defines a line.

Figure 3.8-c) has more communication FIFO channels than the KPN in Figure 3.3-c). On the other hand, most of the channels of the KPN in Figure 3.8-c) are local for the processes (see the self-loop channels). The facts mentioned above can be taken into account to select the most appropriate transformation (unfolding or plane cutting) in order to get some degree of task-level parallelism with appropriate communication between the processes.

In the example given above our plane cutting transformation splits the body of loop i in two parts, thereby distributing the workload over two processes. This is because we use only one plane $i - 2 = 0$ for splitting. We would like to mention that our plane cutting transformation can use a set of planes S to split the body of loop i. Each plane in S has to be of the format $p_1(j, i) = 0$ or $p_2(j) = 0$ or $p_3(i) = 0$ where p_1, p_2, and p_3 are arbitrary affine functions. Applying the plane cutting transformation on the NLP in Figure 3.8-a) with a set of planes leads to distribution of the computational workload over N processes. Depending on the set that is used, N can be in the following range: $1 < N \leq 12$. For example, if we consider the set of planes $S = \{\ i - 1 = 0,\ i - 2 = 0,\ j - 1 = 0,\ j - 2 = 0,\ j - 3 = 0\ \}$ then the workload of the NLP will be distributed over 12 concurrent processes in the process network shown in Figure 3.8-d). In this network, every process executes the function call F only once, thereby exploiting the maximum task-level parallelism available in the NLP. The topology of the network is equal to the dependence graph (DG) that corresponds to the NLP.

The process network in Figure 3.8-c) is functionally equivalent to the network in Figure 3.8-d) but it exploits in less degree the task-level parallelism available in the NLP. This is because the computational workload of the NLP is distributed over two concurrent processes $P1$ and $P2$. $P1$ executes in a sequential order the function calls located above the plane $i - 2 = 0$ depicted in Figure 3.8-d). Process $P2$ executes in a sequential order the function calls located below this plane.

In general, our plane cutting transformation can be used to derive a set of alternative Kahn Process Networks from a weakly dynamic program (WDP) by splitting the bodies of the loops in the WDP with different sets of planes. There is a similarity between our plane cutting transformation and our unfolding transformation presented in Section 3.3. Both transformations aim at exploiting the task-level parallelism available in a WDP by distributing the computational workload over several concurrent processes. However, with the plane cutting transformation, more unbalanced (irregular) workload distributions can be achieved compared to the distributions achievable by the unfolding transformation. By using both transformations in combination, very complex computational workload distributions can be achieved.

3.4.2 Formal Procedure

Let WDP be a weakly dynamic program and let us assume that WDP has N loops. The iterators of these loops form an iterator vector $I = \{i_1, i_2, ..., i_N\}$. For every loop iterator $i_k \in I \mid k = 1, 2, .., N$ a parameter p_k is associated which is the cutting plane of the body of loop i_k. All parameters p_k form a parameter vector $P = \{p_1, p_2, ..., p_N\}$ which we call *set of cutting planes*. Every plane p_k in the set P has the following format: $f(I') = 0$, where f is an arbitrary affine function of $I' \subseteq I$. I' includes loop iterator i_k and loop iterators that belong to loops which are outer in respect to loop i_k. We define a transformation `PLANECUT(WDP,P,I)` which is described below:

- **STEP1 -** Convert WDP to a parse tree representation, called $parseTree$ for short. The $parseTree$ representation is the syntax tree corresponding to the WDP where to every "For"-statement node in the tree a cutting plane is assigned based on the information captured in the iterator vector I and the set of cutting planes P. As we indicated in Section 3.3.2, there are very well known procedures used in the compiler community to convert a program to a syntax tree that can be used to convert our WDP to a parse tree representation.

- **STEP2 -** Transform the parse tree representation obtained in STEP1 by applying the procedure **treePLANECUT** given in Figure 3.9. This procedure takes as an input the root node of $parseTree$. Starting from the root node the procedure visits every node in $parseTree$ by traversing it from top to bottom and from left to right. This is accomplished by calling the procedure **treePLANECUT** recursively - see code lines 36 till 41. When visiting a node, **treePLANECUT** checks if this node is a "For"-statement node and if there is a cutting plane assigned to it - code lines 3 and 7. If both conditions are true then: 1) the children nodes of the current node together with the sub-trees starting from them are stored temporarily (code line 9) and removed from the current node - code line 11; 2) Two "If"-statement nodes are created - see code lines 14 and 32. For the first "If"-statement node a special condition is assigned as shown in code line 16 as well as the children of the current node become children of this "If"-statement node - code line 18. The "If"-statement node becomes the first child of the current node (code line 20). Similarly, for the second "If"-statement node a special condition is assigned by code line 25 and the "father-child" relationship of this node is established by code lines 27 and 29; 3) the current "For"-statement node is marked as processed by assigning a "-" symbol as a cutting plane - code line 32.

- **STEP3 -** Convert the transformed $parseTree$ in STEP2 to a weakly dynamic program.

3.4.3 Example

To illustrate the behavior of the procedure `PLANECUT(WDP,P,I)` described above we apply it step-by-step on the weakly dynamic program shown in Figure 3.5-a). This program we call WDP for short. WDP was used to illustrate the procedure `UNFOLD(WDP,U,I)` in Section 3.3.3, so a detailed description of WDP can be found there.

WDP has two loops and, therefore, the iterator vector I corresponding to WDP has two elements $I = \{i_1, i_2\}$. We assign to every element of I a cutting plane which is used to split the corresponding loop body. In our example, described here, we choose to split the body of the outer loop by a cutting plane $2 * i_1 - ub_1 = 0$ and the body of the inner loop by a cutting plane $i_1 - 3 * i_2 + 2 = 0$. In this case the set of cutting planes $P = \{p_1, p_2\} = \{\ 2 * i_1 - ub_1 = 0,\ i_1 - 3 * i_2 + 2 = 0\ \}$. Now, let us apply the procedure `PLANECUT(WDP,P,I)` step-by-step.

```
1  treePLANECUT( node ) {
2   // Check this Statement
3   if (node is "For"-statement) {
4     // take the cutting plane of the "for"-statement
5      plane = node.cuttingPlane;
6
7     if ( plane != "-" ) {
8        // store the children nodes of the "for"-statement
9         tempList = node.listOfChildren;
10       // make the list of children empty
11       node.listOfChildren =  NULL;
12
13       // create a node which is "if"-statement
14        tempNode = create "If"-statement;
15       // set a string which is the condition of the "if"-statement
16        tempNode.condition = plane + " <= 0";
17       // the children of the "for"-statement become children of the "if"-statement
18        tempNode.listOfChildren = tempList;
19       // add the "if"-statement as a child of the "for"-statement
20       node.listOfChildren[0] = tempNode;
21
22       // create a node which is "if"-statement
23        tempNode = create "If"-statement;
24       // set a string which is the condition of the "if"-statement
25        tempNode.condition = plane + " > 0";
26       // the children of the "for"-statement become children of the "if"-statement
27        tempNode.listOfChildren = tempList;
28       // add the "if"-statement as a child of the "for"-statement
29       node.listOfChildren[1] = tempNode;
30
31       // indicate that this "for"-statement has been processed
32       node.cuttingPlane = "-";
33    }
34  }
35
36  // get the number of children of the current node
37   numElements = getNumberOfElemnts( node.listOfChildren );
38  // go down recursively to check and cut the children
39  for (j = 0; j < numElements; j = j+1) {
40      treePLANECUT( node.listOfChildren[j] );
41  }
42
43   return;
44 }
```

Figure 3.9: Pseudo code describing the **treePLANECUT** procedure.

STEP1

First, we have to convert WDP to a parse tree representation. The resulting parse tree is shown in Figure 3.10. This parse tree consists of five nodes. Every program statement in WDP has a corresponding node in the parse tree. Every edge in the parse tree defines "father-child" relationship between two nodes in the parse tree. This parse tree has the same topology and semantics as the parse tree given in Figure 3.5-b) and described in detail in Section 3.3.3 - STEP1. However, there is a small difference between a "For"-statement node in Figure 3.10 and a "For"-statement node in Figure 3.5-b). The former has field $cuttingPlane$ whereas the latter has field $unfoldFactor$.

At the end of STEP1, the procedure `PLANECUT(WDP,P,I)` assigns a cutting plane to every "For"-statement node of the parse tree in Figure 3.10. In our example, $node1$ and $node3$ are

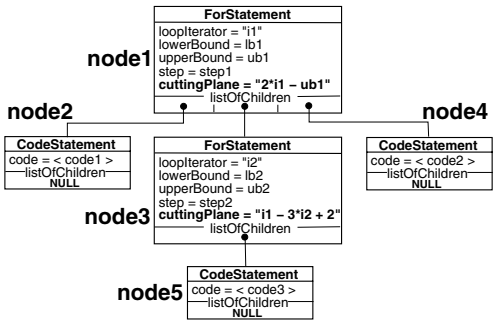

Figure 3.10: Parse tree representation of the program shown in Figure 3.5-a).

"For"-statement nodes and they have the special field $cuttingPlane$ which has to be set with the corresponding plane taken from the set of planes $P = \{p_1, p_2\} = \{\ 2 * i_1 - ub_1 = 0,\ i_1 - 3 * i_2 + 2 = 0\ \}$. So, $node1$ has $loopIterator = i_1$ therefore the corresponding plane is p_1 which implies that $cuttingPlane = 2 * i_1 - ub_1$. Similarly, for $node3$ the corresponding plane is p_2, therefore $cuttingPlane = i_1 - 3 * i_2 + 2$.

STEP2

Here, the recursive procedure **treePLANECUT** given in Figure 3.9 is applied on the parse tree depicted in Figure 3.10. The procedure transforms this parse tree into the parse tree shown in Figure 3.11.

The procedure **treePLANECUT** takes as an input the root node of the parse tree. In our example this is $node1$. Then, **treePLANECUT** checks whether $node1$ is a "For"-statement node - see code line 3 in Figure 3.9. In our case this is true and **treePLANECUT** checks whether there is a cutting plane assigned to $node1$, i.e., $cuttingPlane$ of $node1$ is different than the symbol "-" (see code line 7). Since $cuttingPlane = 2 * i_1 - ub_1$, the code lines 9 till 32 are executed. As a result, the two "If"-statement nodes **nd1** and **nd2** in Figure 3.11 are created, their fields $condition$ are set, and their "father-child" relationship is established.

For **nd1**: the field $condition$ is set to $2 * i_1 - ub_1 \leq 0$ by code line 16. The children of $node1$ ($node2$, $node3$, and $node4$) become children of **nd1** - code line 18. Also, **nd1** becomes the first child of $node1$ - code line 20.

For **nd2**: the field $condition$ is set to $2 * i_1 - ub_1 > 0$ by code line 25. The children of $node1$ ($node2$, $node3$, and $node4$) become children of **nd2** - code line 27. Also, **nd2** becomes the second child of $node1$ - code line 29.

$Node1$ is marked that it has been processed (code line 32) by setting its field $cuttingPlane$ with the symbol "-".

The procedure **treePLANECUT** continues with executing code line 37. At this point, $node1$ has two children **nd1** and **nd2**. Therefore, code line 40 is executed two times by calling the

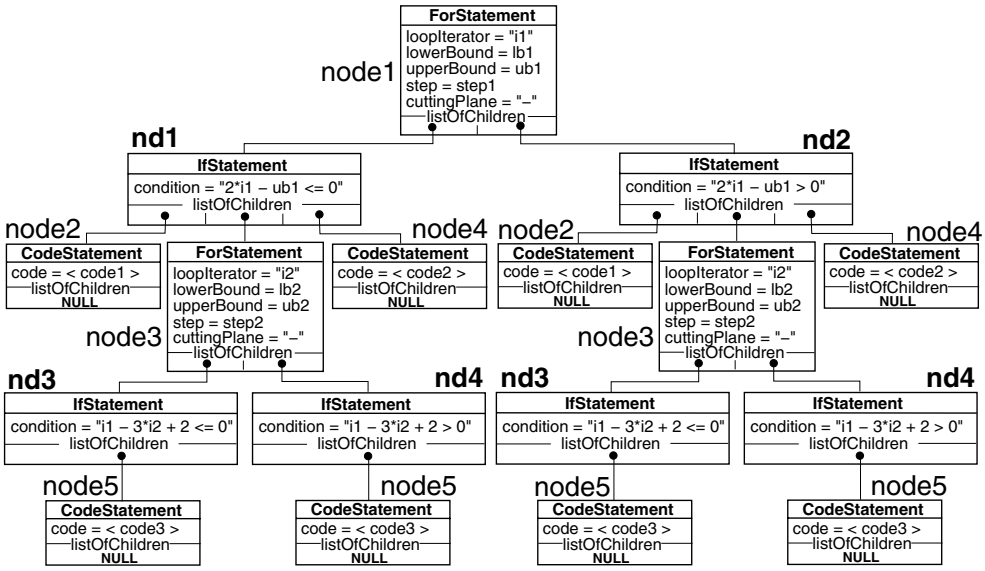

Figure 3.11: The parse tree in Figure 3.10 transformed by the procedure **treePLANECUT**.

procedure **treePLANECUT** recursively. The input node of the first recursive call is **nd1**. The second recursive call has as an input **nd2**. These recursive calls complete the parse tree transformation by creating the nodes **nd3** and **nd4**. The final transformed parse tree is shown in Figure 3.11.

STEP3

In this final step of the procedure `PLANECUT(WDP,P,I)` the transformed parse tree depicted in Figure 3.11 is converted to the weakly dynamic program shown in Figure 3.12.

3.5 Skewing Transformation

First, we explain the general idea behind our *skewing* transformation in Section 3.5.1 by applying the transformation on the example program we used in Section 3.3.1 and Section 3.4.1. The main reason for using the same example program is to show clearly the effect of the skewing transformation compared to the previously discussed unfolding and plane cutting transformations. Next, in Section 3.5.2, we define the skewing transformation as a formal procedure which operates on an N-deep nested loop WDP. Finally, we explain the formal procedure by going through an example in Section 3.5.3.

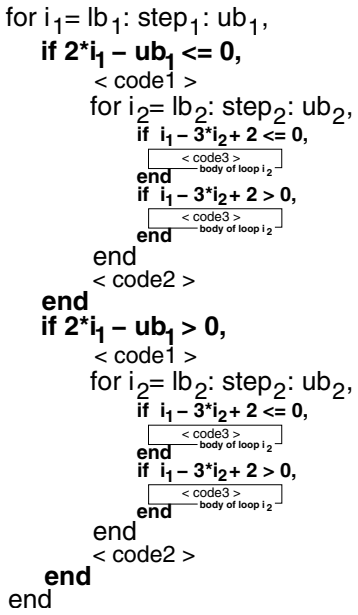

Figure 3.12: The weakly dynamic program in Figure 3.5-a) planecut by the procedure `PLANECUT(WDP,P,I)`. $I = \{i_1, i_2\}$ and $P = \{\ 2 * i_1 - ub_1 = 0,\ i_1 - 3 * i_2 + 2 = 0\ \}$.

3.5.1 General Idea

Let us consider the nested loop program (NLP) shown in Figure 3.13-a). The goal of our skewing transformation is to transform this initial NLP into a new program in which the bounds of the loops and the indexes of some variables are changed so as to make the potential parallelism between the function calls F of the initial NLP explicit in the transformed program.

For example, skewing the inner loop i of the program in Figure 3.13-a) leads to the NLP shown in Figure 3.13-b). Both programs are functionally equivalent. However, the program in Figure 3.13-b) has the following very important property: for every loop iteration $j \in [2..7]$ the function calls F executed in the body of loop j are data independent of each other. This property reveals the independent function calls that can be executed in parallel. Therefore, we say that the potential parallelism between the function calls F of the program in Figure 3.13-b) is explicit.

The property, mentioned above, is visualized by showing, in Figure 3.13-e), the dependence graph (DG) which corresponds to the program in Figure 3.13-b). In the DG the function calls F which are executed in every iteration j are bounded by dashed or solid box. The DG explicitly shows that the function calls inside a box can be executed in parallel because there are no data dependences between these function calls. For comparison, the initial NLP in Figure 3.13-a) does not have that property. This can be seen from the corresponding

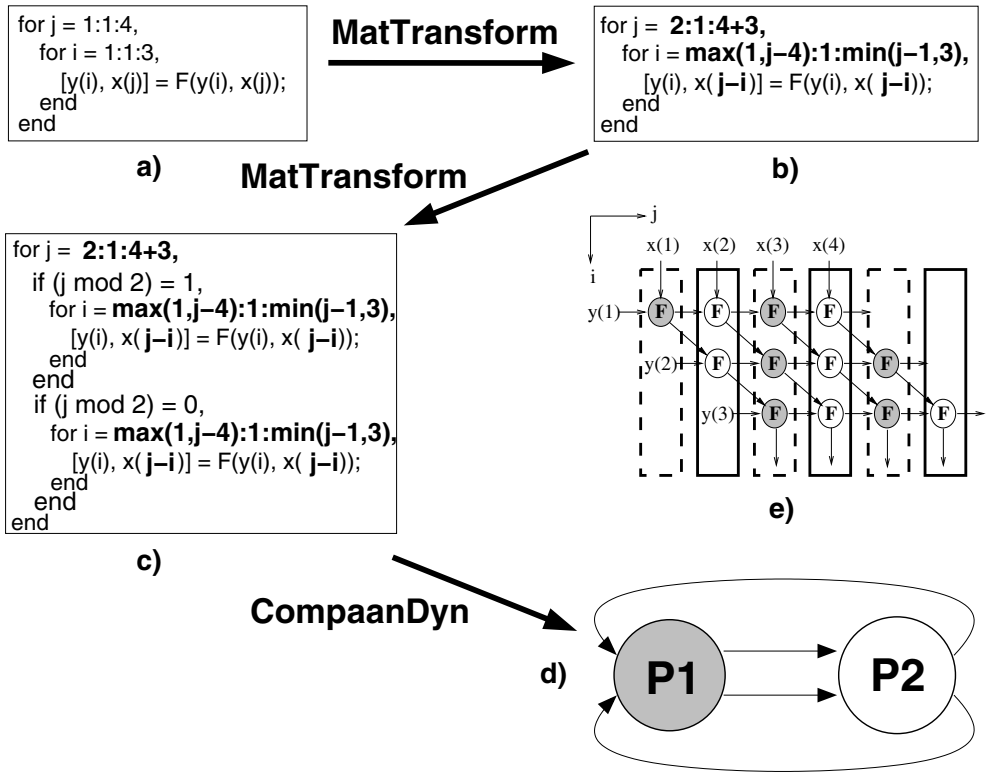

Figure 3.13: Simple example illustrating the skewing transformation.

DG depicted in Figure 3.3-d). The function calls F in every dashed or solid box are data dependent.

Above, we showed how the skewing transformation makes the potential parallelism between function calls F explicit. This parallelism can be exploited by our COMPAANDYN approach in combination with the unfolding or plane cutting transformations to distribute the computational workload of the skewed NLP in Figure 3.13-b) over several concurrent processes. For example, if we unfold loop j of this skewed NLP by a factor of two we get the program depicted in Figure 3.13-c). Taking this program, the COMPAANDYN approach generates a Kahn Process Network (KPN) that consists of two processes and has the topology shown in Figure 3.13-d). Process $P1$ executes the function calls F bounded by the dashed boxes in Figure 3.13-e). Process $P2$ executes the function calls F bounded by the solid boxes. Moreover, inside these processes some function calls can be executed in parallel or in a pipeline fashion because of the skewing transformation.

Our skewing transformation is meaningful if it is applied together with the unfolding or plane cutting transformations to obtain a KPN. In such case, the degree of exploited task-level parallelism in the KPN, achievable by the unfolding or plane cutting transformations alone,

is increased additionally by the skewing transformation. The price for this increase is more inter-process communication in the KPN. This can be seen from the following two examples.

First, the KPN shown in Figure 3.3-c) is derived from the NLP given in Figure 3.3-a) by applying our unfolding transformation alone (loop j is unfolded by a factor of two). This KPN has two processes and two inter-process communication channels. The function calls F executed inside every process are directly data dependent, i.e., these function calls can not be executed in parallel. This is visualized in Figure 3.3-d) and explained in Section 3.3.1.

Second, let us consider the KPN shown in Figure 3.13-d) which is derived from the same NLP as considered above by applying the skewing transformation together with the same unfolding transformation as described above. Now, the KPN has again two processes but two more inter-process communication channels compared to the KPN in Figure 3.3-c). On the other hand, the exploited parallelism is increased because the function calls F executed inside every process are data independent, i.e., these function calls can be executed in parallel or in very efficient pipeline.

In general, our skewing transformation can be applied on an N-deep nested loop WDP. There is a relation between our skewing transformation and the loop skewing transformation used in the classical high-performance compilers [40] in the sense that both transformations are an enabling (auxiliary) transformations that are primarily useful in combination with other transformation to exploit parallelism. However, the classical loop skewing is used in combination with loop interchange to exploit fine-grain instruction level parallelism by handling so called wavefront computations. In contrast, our skewing transformation is used in combination with our unfolding transformation to enhance and exploit task-level parallelism by deriving alternative KPNs. Our skewing transformation is similar to the re-timing transformation used in the signal-processing community [42]. However, we have developed a procedure to do the skewing transformation on the source code of a weakly dynamic program (WDP), whereas in [42] the re-timing transformation is applied on signal-flow graphs or dependence graphs corresponding to a static nested loop program because only for static programs such graphs can be derived. This means that our skewing transformation has more general applicability than the re-timing transformation.

3.5.2 Formal Procedure

Let WDP be a weakly dynamic program and let us assume that WDP has N loops. All loops in WDP have to be nested in each other at different levels, i.e., there are no loops at a same level of nest. Such program we call an N-deep nested loop WDP. The iterators of these loops form an iterator vector $I = \{i_1, i_2, ..., i_N\}$. For every loop iterator $i_k \in I \mid k = 1, 2, .., N$ a parameter vector $D_k = \{m_1, m_2, ..., m_N\}$ is associated, where $m_p \in \mathbb{N} \mid p = 1, 2, .., N$. This vector is the *skewing vector* of loop i_k in the N-dimensional iteration space of WDP. All parameter vectors D_k form a parameter matrix

$$M = \{D_1^T, D_2^T, ..., D_N^T\} = \begin{bmatrix} m_{11} & .. & m_{1N} \\ .. & .. & .. \\ m_{N1} & .. & m_{NN} \end{bmatrix}$$

which we call *skewing matrix*. We require M to be unimodular. We define a transformation `SKEW(WDP,M,I)` which is described below:

- **STEP1 -** Represent the iteration space of WDP by the polytope $P = \{I \in \mathbb{Z}^n \mid A.I \geq b\}$, where A is an integral matrix and b is an integral vector. The values of A and b are determined by the lower and upper bounds of the loop iterators in WDP and the conditions of the "if"-statements in WDP that are affine functions of loop iterators. The conditions that are not affine functions of loop iterators are not taken into account. Therefore, the polytope P represents only the linear bound of the iteration space of WDP;
- **STEP2 -** Use the *skewing matrix* M to transform P as follows: $A.M^{-1}.M.I \geq b \implies A^{'}.I^{'} \geq b$, where $A^{'} = A.M^{-1}$ and $I^{'} = M.I$;
- **STEP3 -** Use the Fourier-Motzkin (FM) procedure [63] to represent the iteration space, described by $A^{'}.I^{'} \geq b$, in terms of nested loops. This is the new iteration space of WDP with iterator vector $I^{'}$;
- **STEP4 -** Change all indexes of the variables in WDP in accordance with the equation $I = M^{-1}.I^{'}$.

3.5.3 Example

We illustrate the four steps of procedure `SKEW(WDP,M,I)` described above by applying it on the weakly dynamic program shown in Figure 3.14. Here, we will refer to this program

```
1  for j = 1:1:N,
2     for i = 1:1:K,
3        if y(i) = x(j),
4          [ y(i), x(j) ] = F( y(i), x(j) );
5        end
6     end
7  end
```

Figure 3.14: Example of a weakly dynamic program.

as WDP for short. WDP has two loops j and i which are nested, i.e., WDP is a 2-deep nested loop program. The upper bounds of the loops are the parameters N and K which can take any positive integer value. WDP is weakly dynamic because the outcome of the "if"-condition (code line 3) depends on the values of variables $y(i)$ and $x(j)$ which can not be determined at compile time.

The loop iterators j and i of WDP form an iterator vector $I = \{j, i\}$ which defines a two dimensional iteration space. In this space, with every loop iterator a skewing vector is associated. By default, $D_j = \{1, 0\}$ is the initial skewing vector of loop j. $D_i = \{0, 1\}$ is the initial skewing vector of loop i. These two vectors are orthogonal and they form the skewing matrix $M = \{{D_j}^T, {D_i}^T\} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ which is the identity matrix. Applying our skewing transformation `SKEW(WDP,M,I)` with the default identity matrix M does not change the WDP. Therefore, the skewing is interesting if we use some non-identity unimodular skewing

matrix M. For our example, let us use the skewing matrix $M = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$. These means that we skew only the inner loop i in the direction of loop j by a factor of three because we change only the skewing vector of loop i as follows $D_i = \{3, 1\}$. Now, let us apply the procedure `SKEW(WDP,M,I)` step by step.

STEP1

In this step, the linear bound of the iteration space of WDP has to be represented by a polytope. From Figure 3.14 we see that the linear bound of WDP is determined by the bounds of loop iterators j and i. The following linear inequalities can be derived from the loop's code in line 1 and line 2: $j \geq 1,\ j \leq N,\ i \geq 1$, and $i \leq K$. These four inequalities can be rewritten in the form $j \geq 1,\ -j \geq -N,\ i \geq 1$, and $-i \geq -K$. These linear inequalities define a polytope $P = \{(j, i) \in \mathbb{Z}^2 \mid A.(j, i)^T \geq b\}$ where $A.(j, i)^T \geq b$ is the following matrix inequality:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} j \\ i \end{bmatrix} \geq \begin{bmatrix} 1 \\ -N \\ 1 \\ -K \end{bmatrix}$$

STEP2

We use the skewing matrix $M = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$ and its inverse matrix $M^{-1} = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix}$ to do the mathematical manipulation $A.M^{-1}.M.(j, i)^T \geq b$. This manipulation does not change the polytope P because $M^{-1}.M$ is equal to the identity matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. However, the matrix inequality $A.M^{-1}.M.(j, i)^T \geq b$ can be written in the form $A'.(j', i')^T \geq b$ where $A' = A.M^{-1} = \begin{bmatrix} 1 & -3 \\ -1 & 3 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}$ and $(j', i')^T = M.(j, i)^T = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix} \cdot (j, i)^T$. The matrix inequality $A'.(j', i')^T \geq b$ is used to define a new polytope $P' = \{(j', i') \in \mathbb{Z}^2 \mid A'.(j', i')^T \geq b\}$ which is the linear bound of a new iteration space for our input WDP with new loop iterators j' and i'.

STEP3

The Fourier-Motzkin (FM) procedure [63] is used to represent the new iteration space, defined by the polytope P', as nested loops. We briefly describe how this procedure works for our example. So, in STEP2, the polytope P' is defined by the matrix inequality $A'.(j', i')^T \geq b$

which is $\begin{bmatrix} 1 & -3 \\ -1 & 3 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} j' \\ i' \end{bmatrix} \geq \begin{bmatrix} 1 \\ -N \\ 1 \\ -K \end{bmatrix}$. This matrix inequality is equivalent to the system of linear inequalities shown in Figure 3.15. This system can not be expressed directly

$$\begin{aligned} a) \quad & j' - 3 * i' \geq 1 \\ b) \quad & -j' + 3 * i' \geq -N \\ c) \quad & i' \geq 1 \\ d) \quad & -i' \geq -K \end{aligned}$$

Figure 3.15: The new iteration space of WDP obtained in STEP2.

$$\begin{aligned} a) \quad & j' - 3 * i' \geq 1 \\ b) \quad & -j' + 3 * i' \geq -N \\ c) \quad & i' \geq 1 \\ d) \quad & -i' \geq -K \\ e) \quad & 3 * i' \geq 3 \\ f) \quad & -3 * i' \geq -3 * K \end{aligned}$$

Figure 3.16: Adding two redundant inequalities e) and f) to the system in Figure 3.15.

as nested loops (j' - outer loop and i' - inner loop) because the inequalities a) and b) in j' are dependent on i'. Therefore, the system in Figure 3.15 is transformed in a few steps into an equivalent system with some inequalities in j' that are independent on i'. First, we add to the system two redundant inequalities as shown in Figure 3.16. Inequalities e) and f) are redundant because they are equivalent to c) and d), respectively.

Second, the system in Figure 3.16 is transformed into the system shown in Figure 3.17. In this system, the new inequality e) is obtained by summing a) and e) from the system in Figure 3.16. Similarly, the new inequality f) is obtained by summing b) and f) from the system in Figure 3.16.

$$\begin{aligned} a) \quad & j' - 3 * i' \geq 1 \\ b) \quad & -j' + 3 * i' \geq -N \\ c) \quad & i' \geq 1 \\ d) \quad & -i' \geq -K \\ e) \quad & j' \geq 4 \\ f) \quad & -j' \geq -(N + 3 * K) \end{aligned}$$

Figure 3.17: System equivalent to the system in Figure 3.16 with two inequalities e) and f) independent on i'.

$$\begin{aligned} a) \quad & i' \geq max(1, ceil((j' - N)/3)) \\ b) \quad & i' \leq min(floor((j' - 1)/3), K) \\ c) \quad & j' \geq 4 \\ d) \quad & -j' \geq -(N + 3 * K) \end{aligned}$$

Figure 3.18: The system in Figure 3.17 represented by using functions *max*, *min*, *floor*, and *ceil*.

Finally, we arrive at the system shown in Figure 3.18. In this system the inequality a) is equal to the inequalities b) and c) taken together from Figure 3.17. Also, inequality b) is equal to the inequalities a) and d) taken together from Figure 3.17.

Now, the system of linear inequalities given in Figure 3.18 represents the linear bound of the new iteration space of our input WDP. Converting this system to nested loops with j' as outer loop and i' as inner loop we get the transformed WDP shown in Figure 3.19.

```
1  for j' = 4:1:N+3*K,
2     for i' = max(1,ceil((j'-N)/3)) : 1 : min(floor((j'-1)/3),K),
3        if y(i) = x(j),
4          [ y(i), x(j) ] = F( y(i), x(j) );
5        end
6     end
7  end
```

Figure 3.19: The program in Figure 3.14 after the transformations applied in STEP3.

STEP4

After STEP3 all variables inside the loops of the program shown in Figure 3.19 are still indexed by the old iterators j and i. We have to replace them with the new iterators j' and i'. In order to do this we know from STEP2 that $(j', i')^T = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix} \cdot (j, i)^T$, which implies that $(j, i)^T = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix} \cdot (j', i')^T$. So, we have to replace index j with $j' - 3 * i'$ and index i with i' in all variables. This gives us the final program shown in Figure 3.20 as a result of applying our skewing transformation. This program is functionally equivalent to the initial

```
1  for j' = 4:1:N+3*K,
2     for i' = max(1,ceil((j'-N)/3)) : 1 : min(floor((j'-1)/3),K),
3        if y(i') = x(j'-3*i'),
4          [ y(i'), x(j'-3*i') ] = F( y(i'), x(j'-3*i') );
5        end
6     end
7  end
```

Figure 3.20: The weakly dynamic program in Figure 3.14 skewed by `SKEW(WDP,M,I)`.

WDP shown in Figure 3.14. Although, the bounds of the loops have been changed in STEP2 and STEP3 (see code lines 1 and 2 in Figure 3.20), the initial functional behavior of WDP is preserved by changing the indexes of all variables in STEP4.

3.6 Merging Transformation

In previous sections we have presented our three source-to-source transformations (unfolding, plane cutting, and skewing) which in combination with our COMPAANDYN approach (Chapter 2) can be used to derive alternative Kahn Process Network (KPN) specifications from a weakly dynamic program (WDP). These transformations enhance the task-level parallelism available in the WDP, thereby allowing COMPAANDYN to generate KPNs in which a high degree of concurrency is exploited. However, a lot of task-level parallelism requires a lot of computational resources which in some cases may not be available. Therefore, in such cases the task-level parallelism available in a WDP has to be reduced, thereby allowing COMPAANDYN to generate KPNs with a lower degree of parallelism.

In this section, we present our *merging* transformation as a means of reducing task-level

parallelism. We start by explaining the general idea of the transformation in Section 3.6.1. The formal procedures which define the merging transformation are not given here. They are given in Chapter 2 because the merging is part of STEP3 in our COMPAANDYN approach. Finally, in Section 3.6.2, we explain and show how these formal procedures work by going through an example.

3.6.1 General Idea

Let us consider the nested loop program (NLP) shown in Figure 3.21-a). This program has

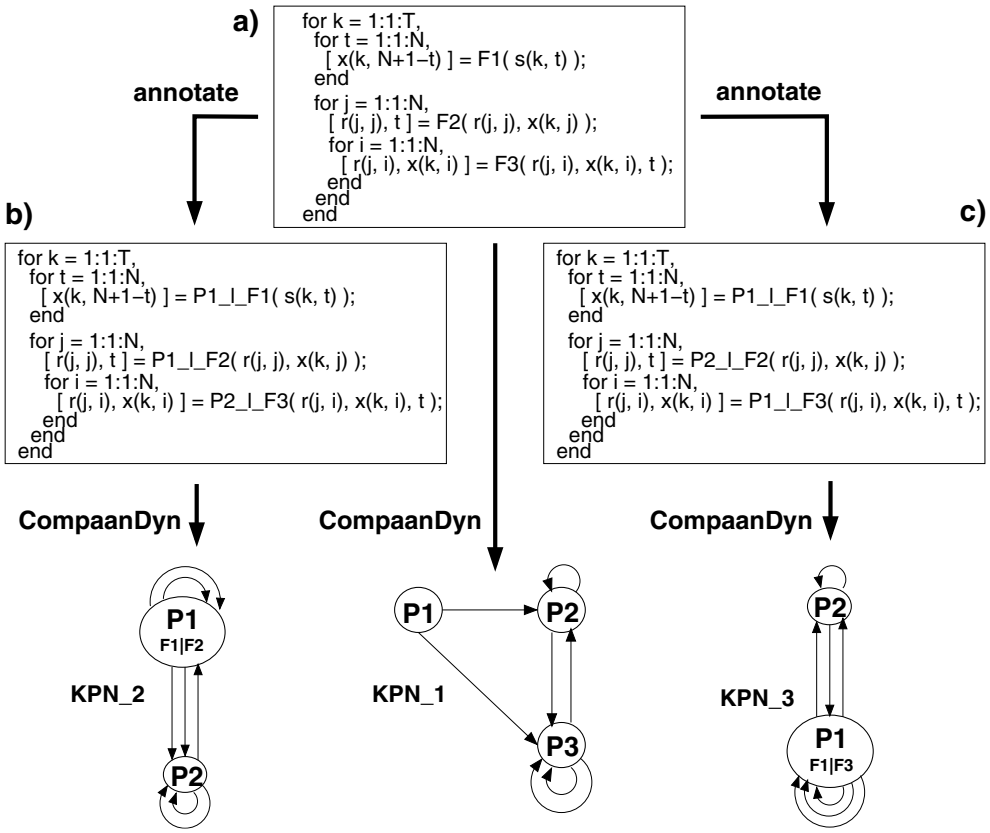

Figure 3.21: Simple example illustrating the merging transformation.

three function calls $F1$, $F2$, and $F3$ which are executed several times in a particular sequence defined by the loops. This multiple execution determines the computational workload of the NLP. Applying our COMPAANDYN approach on this NLP results in the process network KPN_1 depicted in Figure 3.21. The number of processes in KPN_1 is three and it is equal to the number of function calls in the input NLP. This is because COMPAANDYN uses the following default rule for distributing the computational workload: for every function

call that appears in the input NLP a process is generated. This distribution rule we call *oneFunction-in-oneProcess*. So, in KPN_1 process $P1$ executes several times function call $F1$. Similarly, process $P2$ executes $F2$ and process $P3$ executes $F3$.

In some cases, the distribution rule *oneFunction-in-oneProcess* can lead to a KPN specification with a high degree of concurrency which can not be exploited. For example, if we have to map KPN_1 in Figure 3.21 onto an architecture which consists of only two components then we do not have enough components to map every process onto a single component, i.e., the concurrency in KPN_1 can not be exploited in full scale. In such case, a better KPN specification will be one that have only two concurrent processes because it matches better the available architecture resources. Examples of such KPN specifications are KPN_2 and KPN_3 in Figure 3.21 where process $P1$ in KPN_2 executes $F1$ and $F2$ and process $P1$ in KPN_3 executes $F1$ and $F3$.

In order to derive KPN_2 or KPN_3 from the program in Figure 3.21-a), we need a flexible mechanism to specify and execute alternative distributions which do not obey the default distribution rule *oneFunction-in-oneProcess*. In our COMPAANDYN approach such alternative distributions can be specified in the code of the input NLP and executed by our merging transformation which takes place in STEP3 of COMPAANDYN. For example, let us assume that we want to distribute the computational workload of the NLP shown in Figure 3.21-a) over two processes $P1$ and $P2$ where $P1$ executes function calls $F1$ and $F2$ and $P2$ executes function call $F3$. This distribution has to be specified in the NLP by annotating it as shown in Figure 3.21-b). We see that labels are attached to the function call names[4]. $F1$ and $F2$ have the same label `P1_l_`, therefore our merging transformation will place these function calls in one process with name $P1$. The resulting network is KPN_2 shown in Figure 3.21.

According to the Kahn Process Network semantics process $P1$ has to execute the function calls $F1$ and $F2$ in a sequential order. Therefore, the main step in our merging transformation is to find a valid sequential order, in our case an order between $F1$ and $F2$ such that the network KPN_2 is deadlock free. To get such sequential order a knowledge of a global schedule between all functions $F1$, $F2$, and $F3$ is required. Then our merging transformation extracts a valid sequential order between $F1$ and $F2$ from the global schedule. Our merging transformation uses the global schedule defined by the execution order of the function calls in the input NLP.

Our merging transformation is not a source-to-source transformation compared to the previously presented unfolding, plane cutting, and skewing transformations. This is because in many cases an input weakly dynamic program (WDP) can not be re-written into a functionally equivalent WDP where selected function calls of the input WDP are merged in one function call. For example, let us take the program in Figure 3.21-a) which is a special simplified case of a WDP. This program can not be re-written such that function calls $F1$ and $F3$ are merged in a single function call because there is function call $F2$ in between, and function $F3$ is data dependent on $F2$ and vice versa. So, we can not perform the merging on the source code of the input WDP in order to get a transformed WDP and then give the transformed WDP to COMPAANDYN to derive a KPN.

[4]The general format of a label is `<processName>_l_` where `_l_` is a special terminal symbol used as a separator between `<processName>` and the name of the function call. Function calls with the same labels are placed in one process.

Because of the fact given above our merging transformation is part of the COMPAANDYN approach. The transformation operates on the approximated dependence graph (ADG) and the schedule tree (STree), both corresponding to an input weakly dynamic program (WDP). Our merging operation consists of two main steps. First, the nodes of the ADG are grouped in processes thereby creating the topology of the KPN. The grouping complies with the workload distribution specified in the input WDP. Second, a valid intra-process sequential order is derived from the STree for the processes that have to execute more than one function call, i.e., the processes that have been created by grouping nodes of the ADG. To make the two main steps of our merging transformation more concrete we illustrate them with the example below.

3.6.2 Example

Let us consider the weakly dynamic program shown in Figure 3.22. The functional behavior

```
1  for j = 1:1:4,
2    [x(j)] = P2_1_F1(...);
3  end
4
5  for j = 1:1:4,
6    if x(j) <= 0,
7      [x(j)] = P1_1_F2( x(j) );
8    end
9    [...] = P2_1_F3( x(j) );
10 end
```

Figure 3.22: A weakly dynamic program annotated with labels `P1_1_` and `P2_1_` that specify a particular computational workload distribution.

of this program is discussed in Section 1.2.2. Here, we annotate the program with the labels `P1_1_` and `P2_1_`, thereby specifying how our merging transformation has to distribute the computational workload of the program over two processes $P1$ and $P2$. Process $P1$ has to execute several times function call $F2$ and process $P2$ has to execute several times function calls $F1$ and $F3$ in a sequential order.

To perform the distribution specified above our merging transformation operates on the approximated dependence graph (ADG) and the schedule tree (STree) derived from the program in Figure 3.22. General formal definitions of an ADG and an STree are given in Chapter 2. Formal procedures how to derive them from a weakly dynamic program can also be found there. For the program in Figure 3.22 the ADG and the STree are depicted in Figure 3.23-a) and Figure 3.24-a), respectively.

The ADG consists of three nodes and five edges. Nodes $N1$, $N2$, and $N3$ correspond to function calls $F1$, $F2$, and $F3$, respectively. Edges $ED1$, $ED2$, $ED3$, $ED4$, and $ED5$ correspond to possible data dependencies between $F1$, $F2$, and $F3$ through variable $x(j)$. Every node in the ADG has input ports p_i and output ports q_j.

The STree is a syntax tree with leaf nodes that correspond to function calls $F1$, $F2$, and $F3$. The rest of the nodes correspond to control statements of the program in Figure 3.22. The STree represents in a compact form one valid sequential execution order (global schedule)

among functions $F1$, $F2$, and $F3$ that is the original execution order of the program in Figure 3.22. If we parse the STree top-down from left to right this order can be obtained.

Having the ADG and the STree together with the distribution specified in the input program, the merging transformation performs the two steps described at the end of the previous section:

STEP1

The ADG is transformed to a process network (PN) data structure. A formal definition of a general PN and a formal procedure called *Node Grouping* to transform an ADG to a PN are given in Chapter 2. For our example, the nodes of the ADG in Figure 3.23-a) are structurally grouped in two processes in a PN data structure with the topology shown in Figure 3.23-b).

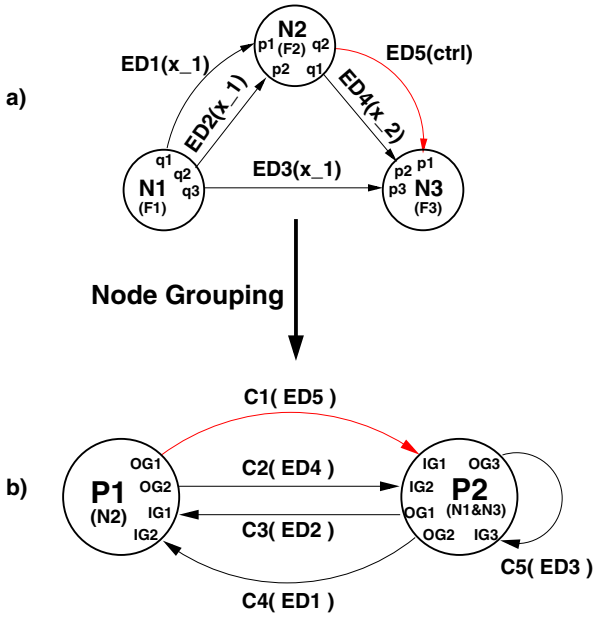

Figure 3.23: Example of Node Grouping performed by our merging transformation to transform the Approximated Dependence Graph (a) into the Process Network (b).

Process $P1$ consists of node $N2$. Process $P2$ consists of nodes $N1$ and $N3$. For every edge ED_i in the ADG a corresponding channel C_j in the PN is created. Every channel has an input gate IG and output gate OG which are related to the ports in the ADG. For example, the OG and IG of channel $C2$ are related to port $q1$ of node $N2$ and port $p2$ of node $N3$, respectively.

STEP2

After STEP1 of our merging transformation the topology of the network is defined in the PN data structure but the sequential order of execution of the function calls inside every process is not defined yet. Therefore, in this step the merging transformation extracts from the STree a valid sequential order for every process that guarantees deadlock free process network. A formal procedure called *Schedule Nodes* is performed. This procedure was given in Chapter 2.

In Figure 3.24 we show how the procedure *Schedule Nodes* works for process $P2$ in our example. Since $P2$ consists of nodes $N1$ and $N3$, $P2$ has to execute sequentially function

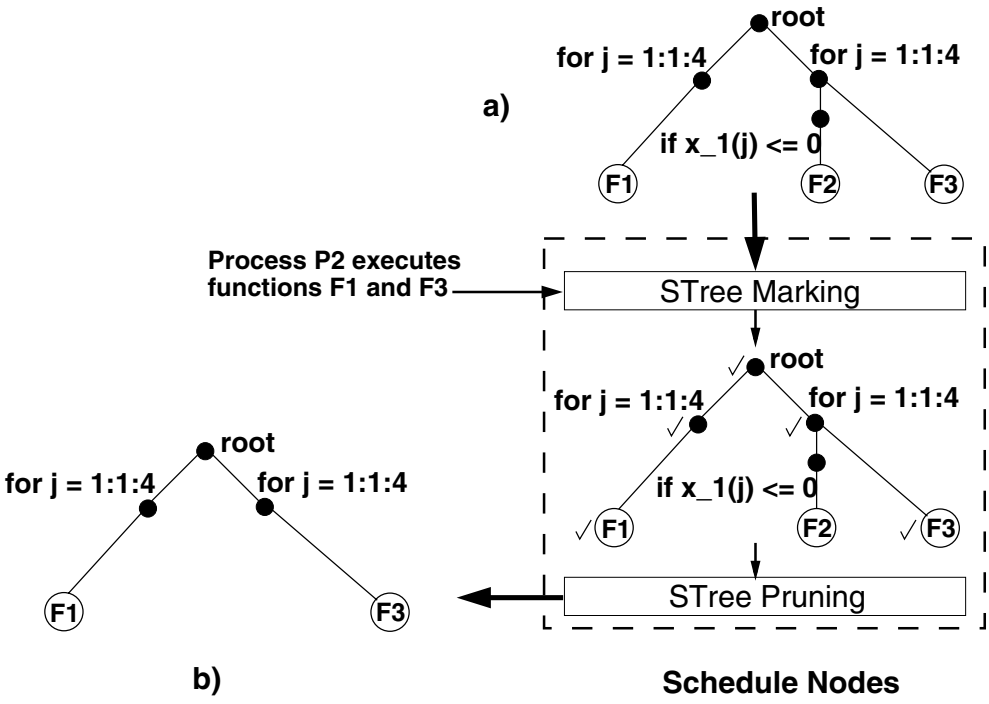

Figure 3.24: Example of applying the *Schedule Node* procedure to obtain a valid intra-process sequential order.

calls $F1$ and $F3$. This information and the STree in Figure 3.24-a) are given as an input to the procedure. First, the operation STree Marking is applied. This operation finds the $F1$ and $F3$ leaf nodes in the STree and parses the tree from these nodes up to the tree root, marking all the nodes in the path - see Figure 3.24. Second, the STree is pruned by removing all the unmarked nodes in the tree. The resultant three shown in Figure 3.24-b) can be converted to a program by traversing it top-down from left to right. This program gives a valid sequential order between $F1$ and $F3$ for process $P2$.

3.7 Discussion and Conclusions

In this chapter, we presented a set of transformations for a systematic derivation of alternative application instances (Kahn Process Networks) from an application specified as a weakly dynamic program. These transformations are encapsulated in an application transformation layer on the top of a Y-chart exploration environment in order to facilitate system designers in exploring alternative instances of an application mapped onto an architecture template.

Our set consists of four transformations, namely unfolding, plane cutting, skewing, and merging. Although, this is a very small set of transformations its transformation power is very large when the transformations are applied in combination on a weakly dynamic program (WDP). By applying the unfolding or plane cutting transformation in combination with the merging transformation one can get *whatever* task-level distribution of the computational workload of a WDP over concurrent processes. This statement is true because: 1) in Section 3.3 and Section 3.4 we showed that our unfolding or plane cutting transformations can be used to transform a WDP such that the maximum task-level parallelism is revealed; 2) On this transformed WDP, our merging transformation can be applied in order to get an arbitrary distribution. We showed in Section 3.6 that this is possible by specifying whatever merging of function calls of a WDP in concurrent processes.

Our transformations have been defined as formal procedures which accept a set of parameters. The behavior of every transformation depends on the values of these parameters. This means that the way an input program is transformed by a transformation depends on the parameter values. For an arbitrary parameter value, the unfolding, plane cutting, and merging transformations always transform an input program to a functionally equivalent program, i.e., these transformations are always legal. The skewing transformation is not always legal because it changes the order of execution of function calls in an input program. The values of the skewing parameters have to be set such that the changed execution order does not violate the data dependencies between the function calls.

Our unfolding, plane cutting, and skewing transformations add extra control structures and operations in the transformed program such as ”if”-statements, and ”mod”, ”div”, ”max”, ”min”, ”ceil”, and ”floor” operations. By converting the transformed program into a process network a lot of optimizations are done to minimize the effect of the additional control and operations on the performance of the network. These extra control and operations are distributed over the concurrent processes of the network and they are executed in parallel. Moreover, the effect of these extra control and operations on the network performance gets lower when the granularity of the function calls executed inside the processes gets higher. This means that our transformations are very efficient when the function calls in an input program represent relatively big and computational intensive tasks (not several simple instructions). We have developed our transformations to exploit task-level parallelism and to be used when the input program is a composition of task-level function calls. Therefore, the extra control and operations added by our transformations do not play a significant role on the performance.

Chapter 4

Case Studies

4.1 System Design Flow Using Kahn Process Networks: an M-JPEG Case Study

New emerging embedded system platforms in the realm of high-throughput multimedia, imaging, and signal processing consist of multiple microprocessors and reconfigurable components. In Chapter 1 we argued that the Kahn Process Network (KPN) model of computation is a suitable parallel model for specifying stream-oriented applications because the specific characteristics of this model match the characteristics of the new emerging embedded system platforms. This allows a system designer to perform systematic mapping of an application onto these platforms.

In this section we present a case study which supports the statement above. The main objective of this case study is to show that our COMPAANDYN approach presented in Chapter 2 can be used and integrated successfully in a system design flow which relies on the KPN model of computation to map efficiently real-life applications onto real hardware platforms in a systematic and automated way. In the case study we show how for an application written as a weakly dynamic program in Matlab, a Kahn Process Network specification is derived in a systematic and automated way using the techniques presented in Chapter 2. This specification is systematically mapped onto a real hardware platform composed of a microprocessor and an FPGA using the COMPAAN/LAURA tools [18] [64]. In the case study we use a real-life application, namely an M-JPEG encoder.

4.1.1 Introduction

New emerging embedded system platforms in the realm of high-throughput multimedia, imaging, and signal processing consist of multiple microprocessors and reconfigurable components. To satisfy the performance needs of tomorrow's applications, these emerging plat-

forms leverage task-level parallelism, i.e., the microprocessors and the reconfigurable components run concurrently. To execute an application on these platforms, the platforms have to be programmed, which implies writing software for the microprocessors using languages like C and writing hardware descriptions using languages like VHDL to configure the reconfigurable components.

To use the concurrency available in the platforms, we need to program them in a way that we exploit *distributed control* and *distributed memory*. Distributed control means that the individual components on a platform can proceed autonomously in time without much interference from other components. Distributed memory means that the exchange of data is contained in the communication structure between individual components and not pooled in a large global memory. Although distributed memory and control are key requirements to take advantage of the new emerging platforms, we observe that sequential programs written in imperative programming languages like C, Java, or Matlab are still the preferred way to specify applications that execute on these platforms. The sequential nature of these specifications makes it easy to reason about an application as only a single thread of control needs to be considered. Also, memory is global and all the data comes from the same memory source. But precisely the single memory and single thread of control in a sequential program are contradictory to the need for distributed control and memory. Therefore, programming these new platform is a very tedious, error prone, and time consuming process.

Instead, we believe that a much more appropriate model of computation is the Kahn Process Network model as it inherently expresses applications in terms of distributed control and memory. As said before, most applications are written in an imperative model of computation. To facilitate the migration from an imperative application to a KPN specification and the mapping of this specification onto a platform, we have developed a top-down system design flow which integrates our COMPAANDYN approach presented in Chapter 2, the COMPAAN tool chain, the LAURA tool, and other tools. We have been involved in the development of parts of the COMPAAN/LAURA tools together with the other main contributors Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Our system design flow allows an application written as a weakly dynamic program in Matlab to be converted in a systematic and automated way to a KPN using the COMPAANDYN approach. This conversion is correct by construction. The obtained processes in the KPN can subsequently be mapped either in software using standard compilers or on hardware using the COMPAAN/LAURA tools.

Our system design flow is centered around exploiting the Kahn Process Network model characteristics. We present the flow by illustrating how we map an M-JPEG application written as a weakly dynamic program in Matlab onto a target architecture that consists of a CPU and an FPGA. The design flow consists of two major steps. In the first step, we convert the Matlab specification of the M-JPEG to a KPN specification. In the second step, we map one process in hardware on the FPGA whilst the remaining processes are mapped in software on the CPU. Before we explain our flow in more detail, we first look at the KPN model and its specific characteristics.

Kahn Process Networks

The KPN model of computation [10] [13] assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded FIFO channels, using a *blocking-read* synchronization primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes. A KPN has the following favorable characteristics:

- The KPN model is deterministic, which means that irrespective of the schedule chosen to evaluate the network, always the same input/output relation exists. This gives us a lot of scheduling freedom that we can exploit when mapping processes to hardware or software.
- The inter-process synchronization is done by a blocking read. This is a very simple synchronization protocol that can be realized easily and efficiently in hardware and software.
- Processes run autonomously and synchronize via the blocking read. When mapping processes on hardware like an FPGA, you get autonomous islands on the FPGA that are only synchronized via blocking reads.
- As control is completely distributed to the individual processes, there is no global scheduler present. As a consequence, partitioning a KPN over a number of reconfigurable components or microprocessors is a simple task.
- As the exchange of data has been distributed over the FIFOs, there is no notion of a global memory that has to be accessed by multiple processes. Therefore, resource contention does not occur.

COMPAANDYN approach and Compaan/Laura tools

In our system design flow the COMPAANDYN approach and the COMPAAN and LAURA tools play a crucial role as described above. The COMPAANDYN approach was introduced in Chapter 1 and elaborated in Chapter 2. We refer the reader there for details. Here, we briefly describe only the COMPAAN/LAURA tools which we use in our flow as well. The COMPAAN compiler, introduced in [18] and further developed in [22, 23], fully automates the transformation of Matlab code into Kahn Process Network (KPN) specifications. The applications COMPAAN can handle, have to be specified as parameterized static affine nested loop programs, which is a subset of the Matlab language. COMPAAN consists of three tools. The first tool transforms the initial Matlab code into single assignment code (SAC), which resembles the *dependence graph* (DG) of the initial nested loop program. The second tool converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure, which is a compact mathematical representation of the DG in terms of polyhedra. The third tool converts the PRDG into a process network by associating a process with each node of the PRDG. The parallel processes communicate with each other according to the data-dependency given in the DG.

LAURA [64] maps a KPN specification onto hardware, for example, FPGAs. The LAURA tool operates as a back-end for the COMPAAN compiler. First, the KPN specification is converted into a functionally equivalent network of virtual processors, called *hardware model*. This is a platform independent step as no information on the target platform is taken into account. Second, platform specific information is added as well as IP cores to this hardware model leading to a network of synthesizable processors. Finally, the hardware model is converted into synthesizable VHDL code.

Our System Design Flow

We demonstrate and evaluate our design flow in the context of a case study in which we map an M-JPEG application onto a platform that consists of a microprocessor and an FPGA running in parallel and communicating with each other via shared memory banks. We give a brief description of the M-JPEG application and the target platform in Section 4.1.2. This is followed by a step-by-step description of our system design flow in Section 4.1.3. In Section 4.1.4, we present some results that we have obtained. In Section 4.1.5 we draw some conclusions and compare our system design flow with other design flows.

4.1.2 M-JPEG and the Platform Architecture

The application we consider is a modified Motion JPEG (M-JPEG) encoder. We have chosen this application because it is a real-life application that is not overly complex, but has enough features to illustrate the use and usefulness of our design flow. Like traditional M-JPEG encoders, the modified M-JPEG encoder compresses a sequence of video frames, applying JPEG [65] [66] compression to each frame in the video sequence. M-JPEG is used for motion pictures compression like MPEG [67] but without inter-frame predictive coding. Our modified M-JPEG encoder, which we further refer to as M-JPEG*, operates on video data in both 4:2:2 YUV and RGB formats on a per-frame basis. This implies that the behavior of M-JPEG* is dependent on the incoming video data. The M-JPEG* encoder application is depicted as a block diagram in Figure 4.1-a).

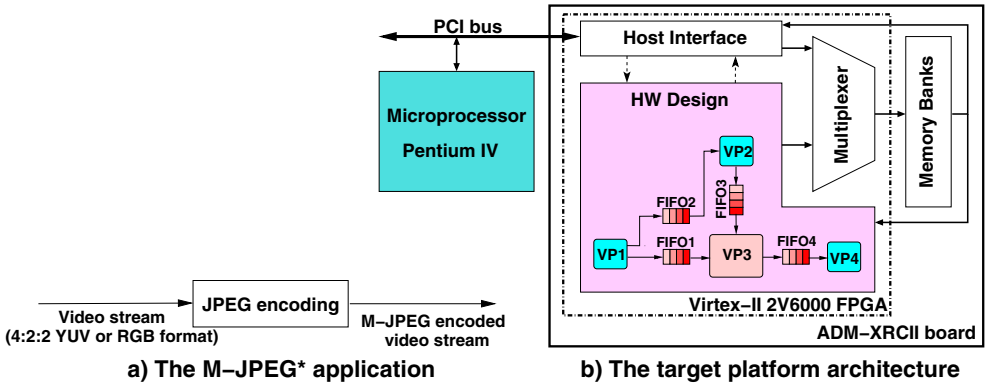

Figure 4.1: Block diagrams: a) M-JPEG*; b) target platform

We map and run the M-JPEG* application on our target platform architecture which is depicted in Figure 4.1-b). The platform architecture consists of a microprocessor (i.e., a Pentium IV) running WindowsNT and connected via PCI bus to the ADM-XRCII board manufactured by Alpha Data Parallel Systems, Ltd [68]. The ADM-XRC-II board is a high performance PCI Card, designed for supporting development of applications using the Xilinx Virtex-II series of FPGAs. The board consists of a Virtex-II 2V6000 FPGA and six ZBT memory banks of size 256k $\times$ 32 bit.

The platform described above allows some parts of the application to run on the *Pentium IV* microprocessor and other parts to run on the FPGA in the *HW Design* block in Figure 4.1-b). To facilitate the communication between the microprocessor and the FPGA, we designed a special interface, labeled as the *Host Interface* block in Figure 4.1-b). This block is responsible for uploading/downloading data to/from the *Memory Banks*. It also controls the *HW Design* block. When the *Microprocessor* has to communicate data with the *HW Design* block, it loads some of the memory banks with data through the *Host Interface*. Next, the *Microprocessor* sends a control signal to the *HW Design* to start execution. The microprocessor also sends a control signal to the *Multiplexer* block to give the *HW Design* access to the memory banks. The *Microprocessor* continues to run in parallel with the *HW Design* and it can access memory banks that are not accessed by the *HW Design*. When the *HW Design* finishes with the execution, it notifies the *Microprocessor*. Then, the *Microprocessor* can change the state of the *Multiplexer* in order to read the data produced by the *HW Design*.

4.1.3 The Mapping

Our system design flow maps an application onto a target platform in a systematic and automated way in a number of steps. We illustrate these steps by mapping the M-JPEG* application onto our platform. Central to our system design flow are the COMPAANDYN approach and the COMPAAN/LAURA tools as shown in Figure 4.2. The figure shows that an application written as a weakly dynamic program (WDP) in Matlab is converted to a KPN specification using the COMPAANDYN approach. By doing workload analysis, candidate processes of this specification are selected for mapping on hardware (FPGA). The remaining processes are mapped on the CPU as software. The KPN is written in a particular format in C++ called YAPI [45]. Using a standard C++ compiler, the processes are compiled to run on the CPU on top of a lightweight multi-threading package. The processes that need to be mapped onto the hardware are further processed by the COMPAAN tool to obtain a hierarchical subnetwork. This subnetwork, which is again a KPN, is compiled into hardware using the LAURA tool. Using commercial synthesizers, we obtain the bitstream to map one or more processes onto the FPGA. The communication between the FPGA and the CPU is automatically generated by LAURA.

We now look at the various steps in more detail and see how they apply to our M-JPEG* application.

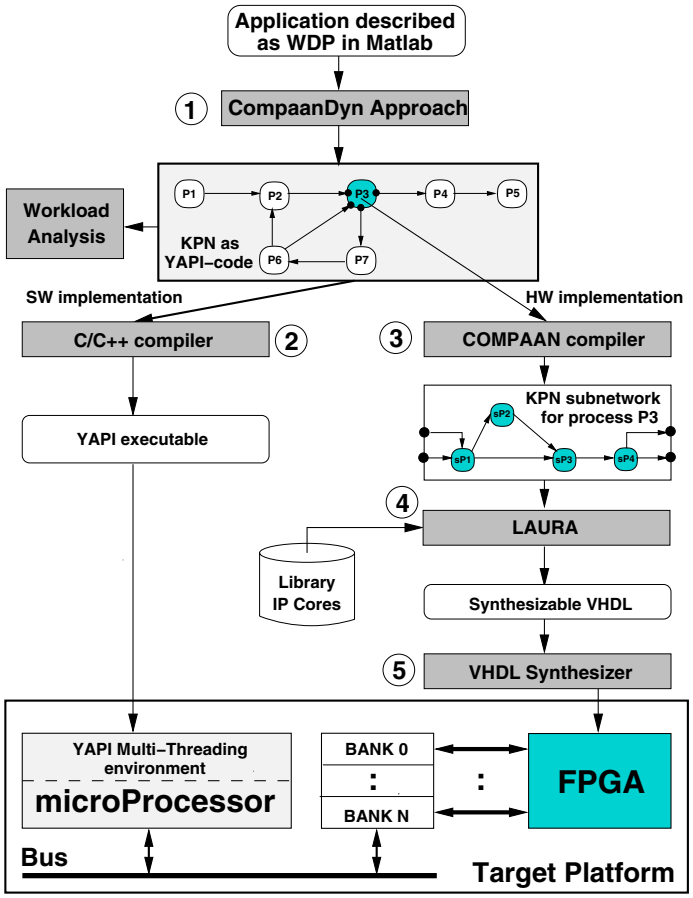

Figure 4.2: The System Design Flow.

STEP 1

The input of our design flow is an application described as a weakly dynamic program in Matlab. It can be debugged easily and the functional correctness of the application can easily be verified. For the M-JPEG* application, we started from a public domain JPEG codec implementation in C [69]. First, we extracted the encoder part from the implementation and modified it to obtain the M-JPEG* application. For example, we added code that implements the conversion of RGB frames to 4:2:2 YUV frames. Next, we structured our M-JPEG* C-code as a set of routines (functions) that are called by the Matlab code shown in Figure 4.3.

This Matlab code is parameterized in the number of frames to be processed (`NumFrames`) and in the the vertical (`VNumBlocks`) and horizontal (`HNumBlocks`) size of a frame in number of 8×8-pixel blocks. For example, the code in line 1 specifies that the number of frames in the sequence can be any integer value between 8 and 100. The code in lines 5-8

```
1   %parameter NumFrames 8 100;
    %parameter VNumBlocks 16 100;
    %parameter HNumBlocks 8 100;

5   for k = 1:1:1,
      [ QTables, HuffTables,
        TablesInfo ] = DefaultTables();
    end

10  for k = 1:1:NumFrames,

      [ HeaderInfo, FrameType ] = VideoInInit();

      if FrameType = 1,
15       for j = 1:1:VNumBlocks,
           for i = 1:1:HNumBlocks,
              [ BlockRGB ] = VideoInRGB();
              [ Block(j,i) ] = RGB_to_YUV( BlockRGB );
           end
20       end
      else
         for j = 1:1:VNumBlocks,
           for i = 1:1:HNumBlocks,
              [ Block(j,i) ] = VideoInYUV();
25         end
         end
      end

      for j = 1:1:VNumBlocks,
30      for i = 1:1:HNumBlocks,
          [ Block(j,i) ] = DCT( Block(j,i) );
        end
      end

35    for j = 1:1:VNumBlocks,
        for i = 1:1:HNumBlocks,

          [ Block(j,i) ] = Q( Block(j,i), QTables );

40        [ Packets ] = VLE( Block(j,i), HuffTables );

          [ ] = VideoOut( HeaderInfo, TablesInfo,
                          Packets                 );

45      end
      end

48  end
```

Figure 4.3: Task-Level specification of the M-JPEG* application as a Weakly Dynamic Program in Matlab.

initializes the quantization (`QTables`) and Huffman (`HuffTables`) tables as well as other variables.

The Matlab code has a weakly dynamic (data-dependent) behavior because of the data-dependent "if"-condition at line 14. This condition checks whether an incoming frame is in YUV format or in RGB format. If an incoming frame is in RGB format then the frame is divided in 8×8-pixel blocks by the `VideoInRGB()` function and every block in the frame is converted to 4:2:2 YUV block by the `RGB_to_YUV()` function - see code lines 15–20. If an incoming frame is in YUV format then the frame is divided in 8×8-pixel blocks by the `VideoInYUV()` function where every block is a 4:2:2 YUV block - see code lines 22–26.

The code lines 29–46 execute the standard JPEG compression algorithm for one frame. A Discrete Cosine Transform (`DCT`) is applied on every 4:2:2 YUV block - line 31, followed by quantization (`Q`) and variable-length encoding (`VLE`) - line 38 and line 40, respectively. Function `VideoOut()` in line 42 adds header information to the compressed frame sequence.

The function calls communicate data via shared variables with different type and structure. For example, variable `FrameType` in line 12 is a simple boolean variable whereas variable

`QTables` in line 38 is a complex data structure that stores the quantization coefficients for the three components Y, U, and V, with which every 8×8-pixel block is quantized.

The weakly dynamic matlab program in Figure 4.3 is a convenient way to describe the M-JPEG* application. Nonetheless, this program does not reveal the inherent task-level parallelism available in the M-JPEG* due to the sequential nature of the program. Therefore, the first step in our system design flow is to convert this sequential program into an executable parallel specification, in our case a Kahn Process Network (KPN). We need a parallel specification of the M-JPEG* because we want to exploit efficiently the parallel resources provided by the platform described in Section 4.1.2 onto which we map the M-JPEG*.

In general, deriving an executable KPN specification by hand for an application described as a weakly dynamic program is difficult and time consuming. Instead, we relay on our prototype software that implements the COMPAANDYN approach to convert in a semi-automatic way the M-JPEG* Matlab program into the KPN specification shown in Figure 4.4-a). Our

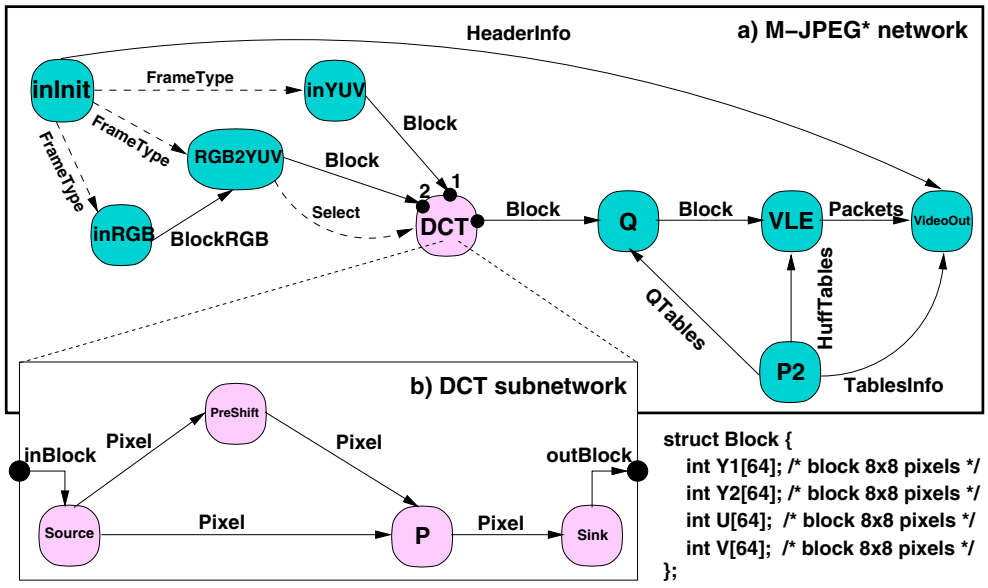

Figure 4.4: The hierarchical KPN for the M-JPEG* Application.

prototype COMPAANDYN software generates a Kahn Process Network as C++ code using the Y–chart Applications Programmers Interface (YAPI) [45]. In YAPI, each process is modeled as a light-weight thread that communicates data with other threads (processes) via unbounded FIFO channels. These channels are accessed using the primitives `read` and `write` to read/write data from/to FIFO channels. The `read` primitive blocks the execution of a process if the current channel from which a process reads data is empty. The `write` primitive is non-blocking. The blocking-read mechanism accomplishes the inter-process synchronization.

To obtain a KPN from a weakly dynamic matlab program, the general partitioning strategy

employed by our COMPAANDYN approach is to create a process for every function call in the program. Therefore, the Kahn Process Network shown in Figure 4.4-a) consists of nine processes. The DCT, Q (Quantizer), VLE (Variable Length Encoding), and VideoOut processes form the central data-flow processing of the M-JPEG encoding algorithm. The inInit process parses the header of every incoming video frame and sends information about the type of the video frame (RGB or YUV) to the inYUV, inRGB, and RGB2YUV processes. Based on this information if the type of the frame is RGB then processes inRGB and RGB2YUV are activated. Process inRGB splits the incoming frame in RGB blocks and sends them to the RGB2YUV process. This process converts the RGB blocks to 4:2:2 YUV blocks and sends them to process DCT. If the incoming frame is YUV then process inYUV is activated that splits the frame in 4:2:2 YUV blocks and sends them to DCT. Process P2 computes and sends quantization and Huffman tables to process Q and process VLE, respectively.

To obtain a specification that exploits distributed memory, all shared variables in the Matlab code shown in Figure 4.3 are replaced by FIFO channels in the KPN shown in Figure 4.4-a). For example, the shared variable `Block(j,i)` is distributed over four FIFO channels called Block in Figure 4.4-a). The type of data communicated over the FIFO channels is the same as the type of the shared variables from which the channels originate. For instance, the type of the variable `Block(j,i)` and the date communicated over the channels with the name Block is the structure shown in the lower right corner of Figure 4.4.

The KPN in Figure 4.4-a) is derived from the weakly dynamic program shown in Figure 4.3 where the data dependent "if"-condition at code line 14 makes the behavior of the program data dependent. The derived KPN has to handle this data dependent behavior. This is accomplished by introducing control FIFO channels over which control information is sent to some of the processes in order to control their internal behavior. These control FIFO channels are depicted as dashed arrows in Figure 4.4-a). The information communicated over control FIFOs denoted as `FrameType` is used to activate processes inYUV, inRGB, and RGB2YUV depending on the type of the incoming video frame. The control FIFO denoted as `Select` delivers information to process DCT that is used by DCT to select the port (port 1 or port 2) from where to read the blocks that have to be processed.

At the end of step 1 in our design flow, we have obtained an executable specification of the M-JPEG* application as a KPN in YAPI code. When the specification is executed, the YAPI code generates statistics on computational and communicational workload of the application. Based on this information, we perform a manual HW/SW partitioning of the application. We identify the most computational intensive process as a candidate we want to put on hardware to speedup the computation. This is done in step 3 of the system design flow. For the Kahn Process Network shown in Figure 4.4-a), the most computational intensive process is DCT that performs the Discrete Cosine Transform on every incoming block of pixels. The rest of the processes in the network will be implemented as software and mapped onto the microprocessor.

STEP 2

The processes selected to be put in software, need to execute on the microprocessor of our target platform. For this purpose, we use the YAPI multi-threading environment which is a

light-weight multi-threading environment. A standard C/C++ compiler is used to compile the YAPI code of the processes.

All the processes of the M-JPEG* KPN are mapped onto the microprocessor, except for the DCT process which is to be mapped on the FPGA. To integrate the execution of the DCT process on hardware with the software processes on the microprocessor, a small piece of interface code needs to execute on the microprocessor too. This code is given in Figure 4.5. The interface code is derived automatically by the COMPAANDYN prototype software.

```
 1  void DCT::main() {
      for (int k=1; k <= NumFrames; k++) {
        for (int j=1; j <= VNumBlocks; j++) {
          for (int i=1; i <= HNumBlocks; i++) {
 5          read(SelectPort, c);

            if ( c == k ) {
              read(inPort2, inBlock);
            } else {
10            read(inPort1, inBlock);
            }

            outBlock = DCT( inBlock );

15          write(outPort, outBlock);
          }
        }
      }
19  }
```

Figure 4.5: Interface code in YAPI format to connect the Software Processes with the Hardware implementation of DCT.

In line 5 of the code, the YAPI primitive `read()` is used to get control data from the control FIFO channel `Select` that is connected to the DCT process. The data read from this channel is stored in the control variable `c`. The value of this variable is used in code line 7 for selecting the data FIFO channel from which the data for the function call `DCT()` has to be read. This data can be read from port `inPort1` or port `inPort2` and the data is stored in variable `inBlock`. The type of this variable is the data structure shown in the lower right corner of Figure 4.4. This structure consists of four 8×8-pixel blocks. Two blocks for the luminance component (Y1 and Y2) and two for the chrominance component (U and V). In line 15, the YAPI primitive `write()` is used to put data that is stored in the variable `outBlock` to the output FIFO channel which is connected to the DCT process. The type of the variable `outBlock` is the same as the type of the variable `inBlock`.

In line 13, the function `DCT()` is called. This function executes a Discrete Cosine Transform (DCT) task implemented as hardware on the FPGA component shown in Figure 4.1-b). First, the data stored in the input argument `inBlock` of the `DCT()` is uploaded to the memory Bank0 of the *Memory* block shown in Figure 4.1-b). This is done via the *PCI bus* and the *Host Interface*. Next, the *HW Design* block executes the DCT task and stores the data in the memory Bank1. Finally, this data is downloaded from the memory Bank1 and returned in output argument `outBlock` of function call `DCT()` in line 13.

STEP 3

By performing a workload analysis, we identify a candidate process that is the most computational intensive process of a given KPN. Typically, the code of such process is a nested loop

program [70]. We want to implement this process as hardware running on the FPGA in the *HW Design* block of the target platform. For that purpose, we use the LAURA tool which generates synthesizable VHDL code from a KPN specification derived by the COMPAAN tool. This VHDL code is suitable for mapping onto FPGAs.

In our case, the candidate process is the DCT process of the KPN in Figure 4.4-a). The code executed inside this process is given in Figure 4.5 where the function call DCT() is to be implemented as hardware on the FPGA. Initially, the function call DCT() executes a static affine nested loop program. Using the COMPAAN tool we derive a KPN from this program. This KPN, shown in Figure 4.4-b), is a hierarchical subnetwork in the M-JPEG*. For this hierarchical subnetwork, LAURA generates synthesizable VHDL. By creating the subnetwork, we exploit more efficiently the parallelism available inside the DCT process. Moreover, we apply automatic type conversion as the hierarchical input/output to/from the DCT subnetwork is data of type Block. Inside the subnetwork, this is converted to streams of pixels (integers). By moving to streams of pixels, we get more fine-grained communication that can be mapped more efficiently onto the FPGA. The type conversion is automatically handled by COMPAAN.

The generation of hardware for the DCT process starts by converting the code in the function call DCT() in line 13 of Figure 4.5, into a Kahn Process Network specification. This is done by the COMPAAN compiler - step 3 in Figure 4.2. The code for the DCT() function call is described as a static affine nested loop program in Matlab as shown in Figure 4.6.

```
1   for k = 1:1:4,
      for j = 1:1:64,
        [ Pixel(k,j) ] = Source( inBlock );
      end
5   end

    for k = 1:1:4,

      if k <= 2,
10      for j = 1:1:64,
          [ Pixel(k,j) ] = PreShift( Pixel(k,j) );
        end
      end

15    for j = 1:1:64,
        [ Block ] = P_l_PixelsToBlock( Pixel(k,j) );
      end
      [ Block ] = P_l_2D_dct( Block );
      for j = 1:1:64,
20      [ Pixel(k,j) ] = P_l_BlockToPixels( Block );
      end

    end

25  for k = 1:1:4,
      for j = 1:1:64,
        [ outBlock ] = Sink( Pixel(k,j) );
      end
29  end
```

Figure 4.6: Matlab code of the DCT() function call.

Lines 1 to 5 transform the incoming data structure inBlock whose type is shown in the right-down corner of Figure 4.4 into pixels of type *int*. Then, the first 128 pixels (the luminance Y1 and Y2 components) are preprocessed so that their expected mean value is zero [65]. This is done by the code in lines 9 to 13. Next, the code in lines 15-21 applies 2D-DCT (Discrete Cosine Transform) [65] on the stream of pixels. Finally, the pixels are grouped again (see lines 25 to 29) in the data structure outBlock whose type is the same as

the type of the `inBlock` data structure. Applying the special preamble `P_l_` on the function names, we specify that the function calls `P_l_PixelsToBlock()`, `P_l_2D_dct()`, and `P_l_BlockToPixels()` have to be grouped in process P by COMPAAN.

The Kahn Process Network (KPN) generated by COMPAAN that corresponds to the Matlab code of the `DCT()` is depicted in Figure 4.4-b). It shows that this KPN is a subnetwork that implements the function call `DCT()` executed inside process DCT in the M-JPEG* network. The subnetwork consists of four processes. The Source and the Sink processes serve as hierarchical interfaces to the M-JPEG* network. The Source process transforms the incoming Block data structures to pixels and distributes the pixels corresponding to the luminance components to the PreShift process for preprocessing, while the pixels corresponding to the chrominance components go directly to the P process. The P process executes a 2D-DCT transformation. The Sink process groups the stream of pixels that comes out from the P process in Block data structures.

STEP4

In step 4 of the design flow shown in Figure 4.2, the LAURA tool transforms the KPN specification generated in step 3 together with predefined IP cores into synthesizable VHDL code. In our example, we provide to LAURA the KPN specification of the `DCT()` function call. The generation of the VHDL code for this KPN takes place in a number of steps.

First, LAURA creates a platform independent hardware model (HM) for the KPN of the `DCT()`. The obtained hardware model is depicted in Figure 4.7. It consists of four concurrent virtual processors VP1, VP2, VP3, and VP4 connected in a network that communicate data with each other asynchronously via FIFO buffers. The topology of the HM network is the same as the topology of the input KPN shown in Figure 4.4-b), as LAURA performs a one-to-one mapping. The virtual processors VP1, VP2, VP3, and VP4 implement the processes Source, PreShift, P, and Sink, respectively.

Every virtual processor is composed of four units: a *Read Unit*, a *Write Unit*, an *Execute Unit*, and a *Control Unit*. The *Control Unit* synchronizes the rest of the units. The *Execute Unit* implements the actual computational part of a virtual processor. It has input arguments that provide to the unit the necessary data for execution and output arguments that are the result of the computation. As an example, we show the internal structure of processor VP3 in Figure 4.7.

The *Execute unit* fires when all the input arguments have valid data and always produces data to all the output arguments. The *Read Unit* is responsible for assigning all the input arguments of the *Execute unit* with valid data. If there are more input ports than arguments, the *Read Unit* has to select from which port to read data. This information is stored in the *Control Table* of the *Read Unit*. *Input Ports* (IPs) are the input interfaces that connect the virtual processor to FIFO buffers. The *Write Unit* is responsible for distributing the results of the *Execute Unit* to the relevant processors in the network. A write operation can be executed only when all the output arguments of the *Execute Unit* are available to the *Write Unit*. *Output Ports* (OPs) are the output interfaces that connect the virtual processor to FIFO buffers. The *Read Unit* and the *Write Unit* can block the *Execution Unit* when a blocking-read or a blocking-write

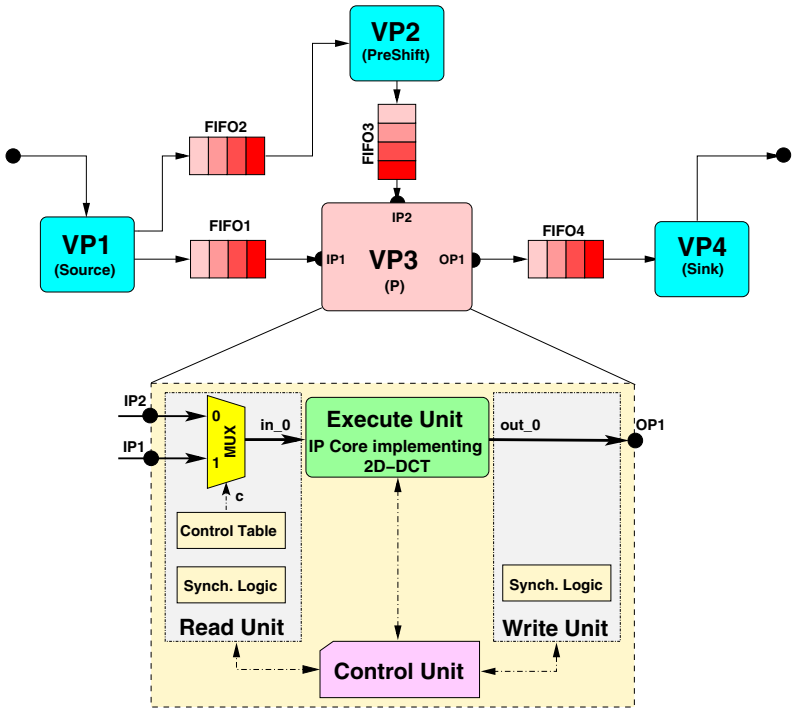

Figure 4.7: The LAURA hardware model of the DCT subnetwork.

situation occurs, thereby stalling the complete processor. A blocking-read situation occurs when data is not available at a given input port, i.e., the corresponding FIFO buffer is empty. A blocking-write situation occurs when data can not be written to a particular output port, i.e., the corresponding FIFO buffer is full. The blocking-read/write is the synchronization mechanism between the virtual processors, thereby implementing the KPN semantics directly into hardware.

From the information provided by the input KPN specification shown in Figure 4.4-b), LAURA synthesizes fully automatically the network of virtual processors shown in Figure 4.7 as well as the *Read Unit*, the *Write Unit*, and the *Control Unit* of every virtual processor. The *Execute Unit* of every virtual processor is implemented by instantiating an appropriate predefined IP Core. In order to select the appropriate IP core, LAURA searches through a library of predefined IP cores until it matches the required functionality. If an appropriate IP core can not be found, then the designer is responsible to add such a core to the library. LAURA provides an easy interface to do this. For the example in Figure 4.7, the IP cores for the *Execute Units* of the VP1 and VP4 processors were found in the library. The IP core (pipelined 2D-DCT) for the *Execute Unit* of the processor VP3 was downloaded from the Xilinx website [71] and added to the library. The IP core (pixel preprocessing) for the *Execute Unit* of the processor VP2 was written by us and added to the library as well. The hardware FIFO buffers FIFO1 to FIFO4 were implemented by instantiating predefined generic FIFO IP cores found in the

library.

In the second step in LAURA, the hardware model is annotated with additional information about the target platform. This is information about IP cores that are used in the virtual processors, the bit-width of the communicated data, and the type of this data. Also, the size of the hardware FIFO buffers is specified. Furthermore, the notion of a clock event is taken into consideration.

In the last step in LAURA, synthesizable VHDL code is generated that describes the annotated hardware model (HM). In LAURA a software procedure called *VHDL Visitor* is implemented that generates for each hardware component the correct VHDL syntax. By simply implementing other *Visitor* procedures, we can generate code in other formats, for example, Verilog or SystemC.

STEP5

In the last step of our design flow, we use commercial tools to synthesize and map the VHDL code, generated in step 4, onto the FPGA shown in Figure 4.1-b). For the synthesis, we used the SinplifyPro tool [72] of Synplicity, Inc. For the placement and routing and for the generation of the configuration file for the FPGA, we used the ISE Foundation package [73] provided by Xilinx.

4.1.4 Experiments and Results

In this section, we present some of the results we have obtained by mapping the M-JPEG* application onto the target platform using our system design flow presented in the previous sections above.

The main objective of this mapping experiment is to evaluate our design flow and to show that the COMPAANDYN approach presented in Chapter 2 can be used and integrated successfully in this flow which relies on the KPN model of computation to map efficiently real-life applications onto real hardware platforms in a systematic and automated way.

The input to our system design flow was an application described as a weakly dynamic program in Matlab. We started with publicly available sequential C code of a JPEG codec. This code was modified and structured by hand to meet the weakly dynamic subset of Matlab that our design flow accepts and to match the features of the M-JPEG* application. The only reason we used Matlab is because the prototype software that implements our COMPAANDYN approach uses a simple Matlab parser. Since the model of computation of Matlab and C is the same, you can read ”C” every time we speak about Matlab.

The writing of the Matlab code took four days together with the functional testing and debugging. After this preparation work, which is a *one-time* effort only, we started with the mapping of the M-JPEG* application using our system design flow.

Our first experiment was to measure how much time it takes to map the M-JPEG* application onto the target platform using our system design flow. Table 4.1 shows the processing times

for every step in the flow. The last column shows the time needed for every step to finish.

Table 4.1: Processing Times (hh:mm:ss).

	COMPAANDYN	COMPAAN	LAURA	*Other tools*	*Manually*	*Total*
STEP 1	00:00:24	–	–	–	00:40:00	00:40:24
STEP 2	–	–	–	00:00:35	–	00:00:35
STEP 3	–	00:00:08	–	–	–	00:00:08
STEP 4	–	–	00:00:07	–	03:00:00	03:00:07
STEP 5	–	–	–	00:13:10	–	00:13:10
Overall	00:00:24	00:00:08	00:00:07	00:13:45	03:30:00	03:54:24

The overall time of the whole design flow for the M-JPEG* experiment is around 3 hours and 55 minutes. The column *Manually* indicates that we had to do some manual manipulations. For example, the prototype software that implements our COMPAANDYN approach is not completed yet in the part where the fuzzy array data-flow analysis is done. So, we had to do some of the procedures in this analysis manually which took 10 minutes. Also, at the end of STEP 1, we had to do a manual HW/SW partitioning that took 30 minutes. In STEP 4, we had to download from Internet, modify and add some IP cores to our library of components, which took 3 hours. However, once the IP cores are in the library any manual manipulations in STEP 4 will disappear.

The results show that the mapping of the M-JPEG* application onto the target platform is done in a short amount of time - a few hours. The main reason is the great time performance of our prototype COMPAANDYN software and the tools COMPAAN and LAURA. For example, using our COMPAANDYN prototype software we were able to derive in a semi-automatic and systematic way a KPN for the M-JPEG* in about 11 minutes after the 4-day preparation work described above. For comparison, a KPN for the same M-JPEG encoder was derived by hand in [16] that took four weeks. The COMPAAN/LAURA tools convert fully automatically function call `DCT()` executed inside process `DCT` in synthesizable VHDL code in a few seconds. For comparison, a hand-made design of function call `DCT()` in VHDL will take several days.

In the second experiment, we use our design flow to evaluate the performance of the Kahn Process Network of the `DCT()` mapped onto the FPGA. We measured the time needed for the FPGA implementation to process a single datum of type `Block`. It took 35 micro seconds at clock frequency of 40MHz. In contrast, the execution of the function call `DCT()` as a program on the microprocessor (clock frequency 1.2GHz) took 98 micro seconds. We conclude, that we got a speedup of 2.8 on the FPGA. We can even improve the speedup by using in STEP 3 (Figure 4.2) the high-level transformations presented in Chapter 3. By performing the unfolding transformation, we can obtain a speedup of up to 10.

The mapping of the KPN of function call `DCT()` is efficient in terms of resource usage. Table 4.2 shows the FPGA resource utilization. The numbers in the table show that on average only 7% of the FPGA resources are used – 6% is taken by the IP cores and only 1% is taken by the FIFOs and the distributed control generated by LAURA to integrate these IP cores in the KPN of function call `DCT()`.

In the final experiment we measured the performance of the complete system: the M-JPEG* Kahn Process Network running on the target platform. We looked at the *throughput* of this

Table 4.2: DCT KPN on VirtexII 2V6000: Device utilization

FPGA resource	*Utilization*	*%*
Number of MULT18X18s	8 out of 144	5%
Number of RAMB16s	4 out of 144	2%
Number of SLICEs	2367 out of 33792	7%
Number of BUFGMUXs	2 out of 16	12%

system, measured in *frames per second*. For frames in CIF format of 128×128 pixels, the throughput of the system is 10.5 frames per second. This is below the standard minimum real-time throughput of 25 CIF frames per second. We found that the problem is not in the output of our design flow, but in the slow communication of data between the microprocessor and the FPGA. The bottleneck is the 32-bit width PCI bus operating at 33MHz. By switching the PCI bus to 64-bits at a frequency of 66MHz, we can increase the communication speed approximately 4 times. As a consequence, the system should be able to process 25 frames per second in CIF format of 128×128 pixels.

4.1.5 Conclusions and Discussion

We presented a system design flow in which an application written as a weakly dynamic program in Matlab is mapped onto a target platform composed of a CPU and an FPGA in a systematic and automated way. The novelty in this flow is that the CPU and the FPGA run concurrently, thereby exploiting efficiently task-level parallelism. Central to the flow is the use of the Kahn Process Network model of computation. This model inherently expresses applications in terms of distributed control and memory. This is required to get an efficient mapping onto the CPU and the FPGA. In realizing the flow, we integrated our COMPAAN-DYN approach presented in Chapter 2 with the COMPAAN and LAURA tools. This allow us to quickly go from a weakly dynamic application specification in Matlab to an implementation of the application running on the target platform. In conclusion, the use of our COMPAAN-DYN prototype software (that is still subject to further research and development) and the COMPAAN/LAURA tools together with other tools results in an efficient design flow for systems that execute high-performance real-time signal processing and multimedia applications.

We have demonstrated our system design flow by mapping the M-JPEG* application onto a platform that consists of a CPU and an FPGA. However, our flow is general enough to be used for a systematic mapping of applications onto multiple CPUs and FPGAs. The main reason for this is the Kahn Process Network (KPN) model of computation used in our flow. As the control and memory are distributed in a KPN, no global scheduler is needed. Hence, partitioning a KPN over a number of CPUs and FPGAs can easily be done.

Although we used in our system design flow a standard C++ compiler, a simple research multi-threading environment, and a simple target platform, the obtained results are already promising. Even better results should be achievable when employing, for example, more optimized and robust commercial solutions.

Discussion on Related Work

Mapping applications like MPEG and JPEG codecs onto a target architecture consisting of a CPU and a hardware co-processor (in our case an FPGA) has been the central question in Hardware/Software co-design in the last decade [74]. Researchers have already mapped successfully multi-media applications on such kind of platforms in a systematic way. The retargetable framework `Nimble` [75] and the work presented in [76] automatically compiles system-level applications specified in C onto a target architecture of a combined CPU and FPGA. However, the compiler only exploits instruction-level parallelism (ILP) in loops but not task-level parallelism. Loops are executed purely sequentially according to their original C specification even if mapped onto the FPGA for acceleration. In [70, 77], reconfigurable logic is used as a co-processor attached to a CPU. The co-processor is typically used to speeds-up certain instructions of the CPU.

All of the related work, mentioned above, exploits only ILP in loops mapped onto an FPGA that runs mutually exclusive with the CPU. In contrast, we show a systematic and automated system design flow to map an application onto a CPU and an FPGA in such a way that the CPU and the FPGA run *in parallel*, exploiting *task-level* parallelism.

Some recent efforts in mapping applications onto a CPU connected to reconfigurable logic (FPGAs) exploiting task-level parallelism has led to design flows that are somehow related to our design flow. Gokhale *et al.* [78] have developed a compiler that takes stream-based applications specified in Streams-C and generates synthesizable hardware for FPGAs and a multi-threaded software program for the control CPU. The Stream-C programming task-level model is the CSP [79] model of computation. The work in [80] also presents a design flow to map applications specified as CSPs onto a platform that consists of a CPU and FPGA. Conceptually, our design flow differs from these flows in the sense that we use the Kahn Process Network (KPN) model which specifies more naturally and efficiently (compared to CSP) the task-level parallelism in stream-based applications. The UC Berkeley's project SCORE [81] has developed a stream-based compute model which virtualizes reconfigurable computing resources (compute, storage, and communication) by dividing a computation up into fixed-size "pages" and time-multiplexing the virtual pages on available physical hardware. The specific language TDF is used to specify applications using the SCORE's model. This stream-based model is similar to the KPN model we use.

In the three design flows [78, 80, 81], mentioned above, the input application has to be analyzed and specified manually in terms of a concurrent task-level model of computation using very specific languages (Stream-C, Handle-C, TDF). This is very time consuming and error prone process because the system designer has to do manually the dependence analysis as well as to learn a specific description language. In contrast, our system design flow relies on our prototype COMPAANDYN software with which currently KPN specifications can be derived in a systematic and semi-automatic way from applications described as weakly dynamic programs in common languages like Matlab or C.

4.2 Exploring the Performance of Alternative Application Instances realized on a FPGA: a QR Case Study

In this section, we show how our algorithmic transformation techniques presented in Chapter 3 are used in practice for application exploration. Using the transformation techniques we derive systematically and fast alternative application instances, i.e., alternative Kahn Process Networks (KPNs) from a real-life signal processing algorithm, namely the QR-decomposition algorithm. These alternative KPNs express task-level concurrency hidden in the QR-algorithm in some degree of explicitness. We map the alternative KPNs onto a FPGA platform in order to demonstrate the effect of the transformation techniques on the performance of the QR algorithm.

4.2.1 Introduction

In system-level design of embedded signal-processing systems, a system designer sees the target system as the pair *Application specification - Architecture template*. An example of a typical signal-processing application specification is given in Figure 4.8 where the QR-decomposition algorithm [82] is described in Matlab. An example of an architecture template is the Virtex-II 2V6000 FPGA platform where a large amount of reconfigurable parallel computational resources and memories are available. The application specification provides the functional behavior of the system. The architecture template specifies the organization of the resources of the system onto which the functional behavior is to be mapped. Having both specifications, a designer has to make some design decisions, for example, how to partition the application into tasks, how to map the tasks onto the architecture template, etc. In order to evaluate different design decisions, a system designer uses a model of the target system and does performance analysis for alternative application instances, architecture instances and mappings, thereby exploring the design space of the *Application - Architecture* pair.

The transformations we have presented in Chapter 3 support efficient exploration of alternative application instances. An application instance is any feasible partitioning of an application into a composition of concurrent tasks. We use the Kahn Process Network (KPN) model of computation [10] to describe application instances. Each application instance differs from the others in the degree of exploited *task-level* parallelism. Therefore, the performance of the *Application - Architecture* pair significantly depends on the application instance. Many instances of a single application exist that are worth to be derived for exploration. In respect of this, our transformations support a system designer to derive systematically and fast a set of instances of an application in order to explore and evaluate the performance of the system (Application - Architecture pair). This gives a system designer an opportunity to select an application instance (partitioning) that satisfies performance/cost requirements the target system has to meet.

We reinforce the statements above with a case study. The main objective of this case study is to show that our algorithmic transformations presented in Chapter 3 can be used and integrated successfully in a design space exploration environment that facilitates systematic and automated exploration of alternative KPNs derived from a real-life application and mapped

onto a real-life architecture template. In the case study we demonstrate how for the QR-decomposition algorithm specified as a sequential program in Matlab, a set of KPN specifications in VHDL is derived in a systematic and automated way using our transformation techniques presented in Chapter 3 together with the COMPAAN/LAURA tools [18] [64]. These specifications are mapped onto a Virtex-II 2V6000 FPGA platform in order to demonstrate the effect of our algorithmic transformations on the performance of the QR-decomposition algorithm.

The choice of the QR-decomposition algorithm and the FPGA platform in our case study is not coincidental. In many modern signal processing applications like Digital Beamforming, Adaptive Digital Filtering etc., the QR-decomposition algorithm [82] is the main computational intensive kernel. QinetiQ Ltd.,UK is one of the leading companies that provides implementation solutions of the QR-decomposition algorithm for digital receivers and beamformers for military systems. Currently, QinetiQ explores the potential of the FPGA technology for implementation of the QR-decomposition algorithm. The designers at QinetiQ face the following problems:

- The QR algorithm they have to deal with is written as a sequential program in Matlab because this is a very convenient way to specify and test signal-processing algorithms. However, it is very hard to obtain straight from the Matlab a good implementation on a FPGA platform because the sequential nature of the Matlab code does not allow efficient exploitation of the parallel computational resources available in a FPGA platform. It took a single person almost a full year to manually partition, schedule, and map the QR algorithm on a FPGA platform.
- The QR algorithm can be partitioned in many different ways, i.e., many alternative application instances can be derived. They differ from each other in the degree of exploited task-level parallelism, thus by mapping alternative instances on a FPGA platform a different performance will be obtained. The QinetiQ designers understand very well that they have to derive alternative application instances and to explore their performance in order to select an instance that maps best on the target platform. In practice, however, they can not afford an application exploration because as said above it takes a designer a lot of time to derive and map a single instance of the QR algorithm, let alone alternatives. This is because the QinetiQ designers do not have a systematic and automated approach to derive and map alternative application instances.

QinetiQ finds our transformation techniques presented in Chapter 3 and implemented in the tool MATTRANSFORM together with the COMPAAN/LAURA tools [18] [64] a very promising and efficient solution to their problems described above. Therefore, they initiated the QR case study presented in the sections that follow. Our collaboration with QinetiQ on the QR case study gave us an opportunity to use our transformations and tools in solving real-life industrially relevant problems as well as to evaluate and show the usefulness of the transformations.

In the next section, we give a brief description of the QR-decomposition algorithm. Section 4.2.3 gives an overview of the experimentation set-up which we used to explore alternative application instances of the QR algorithm. In Section 4.2.4, we present some of the experiments that we have conducted and the corresponding results that we have obtained. In Section 4.2.5 we draw some conclusions.

4.2.2 The QR-decomposition Algorithm

In our case we derive and explore alternative application instances of the QR-decomposition algorithm [82]. This algorithm is the most computational intensive kernel in many signal processing applications like Digital Beamforming, Adaptive Digital Filtering etc. Matlab code for the QR-decomposition algorithm is shown in Figure 4.8.

```
for k = 1:1:T,
  for j = 1:1:N,
     [r(j,j), x(k,j), t] = Vec(r(j,j), x(k,j));
     for i = j+1:1:N,
        [r(j,i), x(k,i), t] = Rot(r(j,i), x(k,i), t);
     end
  end
end
```

Figure 4.8: QR-decomposition algorithm written in Matlab.

The upper bounds of the loops in the program are given as parameters `T` and `N`. The two inner loops with iterators `j` and `i` form a triangular-shaped iteration space of size `N`. The function calls `Vec` and `Rot` executed in the iterations that belong to the triangular-shaped space compute a single QR update. The outer loop with iterator `k` expresses the fact that a QR decomposition consists of computing several QR updates. In every QR update, i.e., in every iteration k the values of the triangular matrix `r(j,i)` computed in iteration $k-1$ are updated such that the final values of the input vector `x(k,j)` become zero.

The QR-decomposition algorithm shown in Figure 4.8 has a dependence graph representation as depicted in Figure 4.9. In this figure we show only one k-plane of the dependence graph (DG) that represents a single QR update. The complete dependence graph consists of several of these planes. The number of these planes is equal to the value of parameter `T`. Every k-

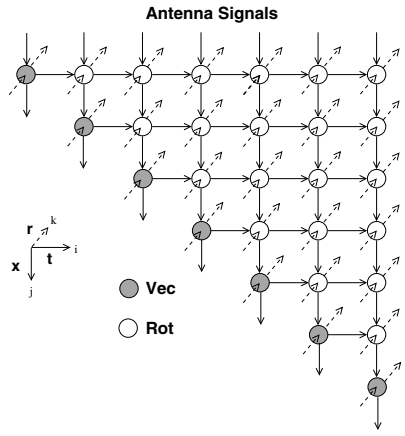

Figure 4.9: Dependence Graph Representation of one QR update, i.e., one k-plane of the QR-decomposition algorithm.

plane depends on its predecessor, i.e., plane $k-1$. Each node in the DG represents a function from the QR algorithm. A gray node represents function `Vec` and a white node represents function `Rot`. At the top of the triangle in Figure 4.9 the values of the input vector `x(k,j)` are taken for a QR update computation. These values arrive from external sources, say sensors of an `N`-antenna array (in our case `N = 7`), and they are propagated downwards through the plane such that the values become zero at the end. The values of matrix `r(j,i)` produced by the previous plane are updated using the functions `Rot` and `Vec`. Function `Rot` does rotation of its input arguments with an angle `t` using the Givens matrix [82]. For every row of `Rot` functions in the triangle, the angle `t` is computed first by the corresponding function `Vec` on the diagonal of the triangle. Next, the computed angle `t` for a row is propagated to all `Rot` functions in the row.

The dependence graph representation of the QR-decomposition algorithm reveals a lot of task-level parallelism that can be exploited by grouping nodes (functions) into concurrent tasks (processes). For this purpose the Kahn Process Network (KPN) model of computation is very suitable for parallel specification of the QR algorithm. Many alternative groupings of the functions into concurrent tasks are possible implying that many alternative KPN specifications can be derived for the QR algorithm. The alternative KPNs of the QR algorithm we call application instances of QR. Every application instance differs from the others in the degree of exploited task-level parallelism which determines the performance of the QR algorithm when mapped onto a parallel architecture. Therefore, it is worth to derive alternative application instances of the QR algorithm for performance exploration.

4.2.3 Using an Extended Y-chart Environment in the QR exploration

In Chapter 3 we presented a set of transformations that we have developed to derive systematically alternative application instances. Also, we presented how in general we can extend a Y-chart exploration environment with an *Application Transformation Layer* that allows exploration of alternative application instances - see Section 3.2. Here, we use such extended Y-chart environment for exploring our QR-decomposition algorithm. The main components and structure of the exploration environment are depicted in Figure 4.10.

The application transformation layer applies some of our transformations on the QR algorithm and generates alternative application instances - Kahn Process Networks (KPN) - as synthesizable VHDL. This is automated by integrating in the transformation layer the tools MATTRANSFORM, COMPAAN, and LAURA. First, the Matlab code in Figure 4.8 that describes the QR algorithm is given as an input to the MATTRANSFORM tool. MATTRANSFORM applies on the Matlab code our transformations `UNFOLD(U)` and `SKEW(M)` presented in Chapter 3. Transformation `UNFOLD(U)` is controlled by the unfolding vector `U` which specifies how the loops in the QR algorithm are to be unfolded. Transformation `SKEW(M)` is controlled by the skewing matrix `M` which specifies how the loops in the QR algorithm are to be skewed. Depending on the values of `U` and `M`, the original Matlab code of the QR algorithm is automatically transformed and structured in a particular way in order to express, in some degree of explicitness, the *task-level* parallelism inherently available in the algorithm.

Second, the transformed code is converted in a systematic and automated way to a KPN description in VHDL using the COMPAAN and LAURA tools. These tools were briefly in-

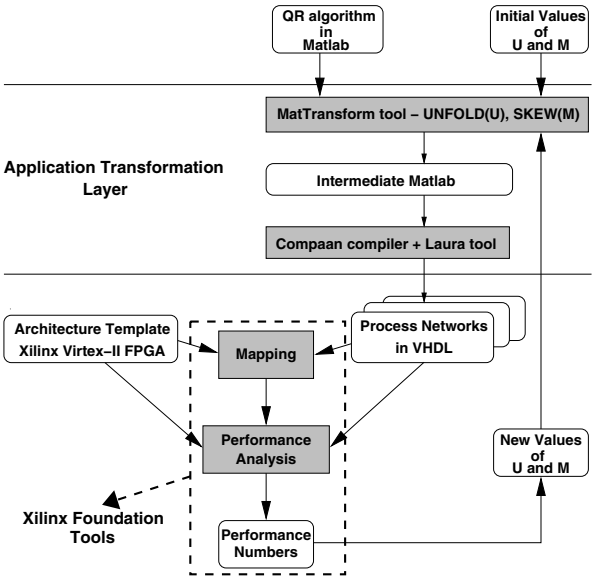

Figure 4.10: Exploring alternative Kahn Process Network specifications of the QR-decomposition algorithm using an extended Y-chart environment.

troduced in Section 4.1.1. Third, we use a Y-chart environment to map the KPN onto an architecture template and to do performance analysis. The architecture template for our experiments is the Virtex-II 2V6000 FPGA platform. The mapping is done automatically by a synthesizer and place-and-route tools provided by Xilinx. The performance analysis is done using the timing analysis and simulation tools from the Xilinx Foundation package. The result of the performance analysis gives us a hint about how to change the values of parameters `U` and `M` in order to improve the system performance.

Next, we change the parameters and repeat the steps described above resulting in a *design space exploration* of alternative KPN specifications of the QR algorithm. This is shown in Figure 4.10 as a feed-back arrow to the transformation layer. By changing the values of the parameters, the application transformation layer systematically derives a set of KPN specifications functionally equivalent to the sequential QR algorithm shown in Figure 4.8. The difference among the KPNs is the degree of the task-level parallelism that is exploited.

4.2.4 Experiments and Results

In this section, we present some of the experiments we have done in order to evaluate and show the usefulness of the algorithmic transformation techniques presented in Chapter 3. In our experiments we used the experimentation set-up described in the previous section to explore alternative application instances of the QR algorithm.

Experiment 1

In this experiment, we evaluate the effect of transformations `UNFOLD` and `SKEW` on the performance of the QR-decomposition algorithm implemented as hardware on an FPGA platform. We use the Matlab code in Figure 4.8. The parameter `T` is set to 21 which means that 21 QR updates are computed. The parameter `N` is set to 7 which means that every update consists of 7 `Vec` operations and 21 `Rot` operations that are dependent on each other as shown in Figure 4.9. From the Matlab code we generate automatically alternative Kahn Process Networks (KPN) in VHDL using the tools MATTRANSFORM, COMPAAN, and LAURA and map these KPNs on an FPGA platform in order to evaluate the performance. To realize the `Vec` operations on the platform we use a *3-stage pipelined* IP core called $BCELL$ that implements a vectorize operation on fixed point data. Similarly, to realize the `Rot` operations we use a *3-stage pipelined* IP core called $ICELL$ that implements a rotate operation on fixed point data.

First, we generate a KPN from the QR algorithm without applying any transformation. The KPN is depicted in Figure 4.11. It consists of two processes V and R. Process V executes

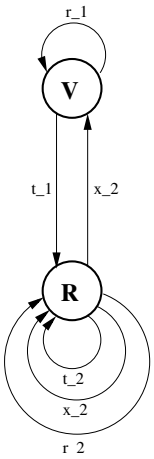

Figure 4.11: The Kahn Process Network derived from the Matlab code specifying the QR algorithm shown in Figure 4.8. No transformation is applied on the QR algorithm.

all vectorize operations `Vec` and process R executes all rotate operations `Rot`. The KPN is mapped on the FPGA platform where the `Vec` operations of process V are executed on one $BCELL$ core and the `Rot` operations of process R are executed on one $ICELL$ core. The performance numbers we obtain are given in the second row of Table 4.3. These numbers are our reference numbers in the evaluation of the effect of the transformations. The performance numbers obtained after applying a transformation are compared to these reference numbers. The second column of Table 4.3 gives the total execution time of the QR algorithm in clock cycles. The third column shows the number of hardware (HW) resources used to implement the functionality of the QR. The fourth column gives the utilization of the HW resources. The last column shows the achieved speedup compared to the performance of the non-transformed

Table 4.3: The effect of Unfolding on QR.

Transformation	*Total number of cycles*	*HW resources* BCELL	ICELL	*Utilization (%)* BCELL	ICELL	*Speedup*
NO transform	1735	1	1	8.64	25.59	1.00x
UNFOLD k by 2	913	2	2	8.21	24.31	1.90x
UNFOLD k by 3	600	3	3	8.33	24.66	2.89x
UNFOLD k by 4	516	4	4	7.27	21.51	3.36x
UNFOLD k by 5	445	5	5	6.74	19.96	3.89x

QR.

The KPN in Figure 4.11 is derived without applying any transformation, thus a low degree of task-level parallelism is exploited - there are only two parallel processes V and R. If the performance of this KPN mapped onto the FPGA is not satisfactory then we have to increase the degree of task-level parallelism. By applying transformation `UNFOLD` to one or more loops in the algorithm we distribute the computational workload of the algorithm on several parallel processes, thereby increasing the task-level parallelism, i.e., potentially decreasing the total execution time of the algorithm. Here, we demonstrate this by applying transformation `UNFOLD` with unfolding factors 2 to 5 on loop `k` of our QR algorithm. As a result four alternative KPNs are generated. Each of them has more than one V and R processes where different QR updates are distributed on different pairs of V and R processes. For example, the KPN depicted in Figure 4.12 is derived by applying the transformation `UNFOLD` with unfolding factor 3 on loop `k`. The KPN consists of three processes $V1$, $V2$, and $V3$ that execute

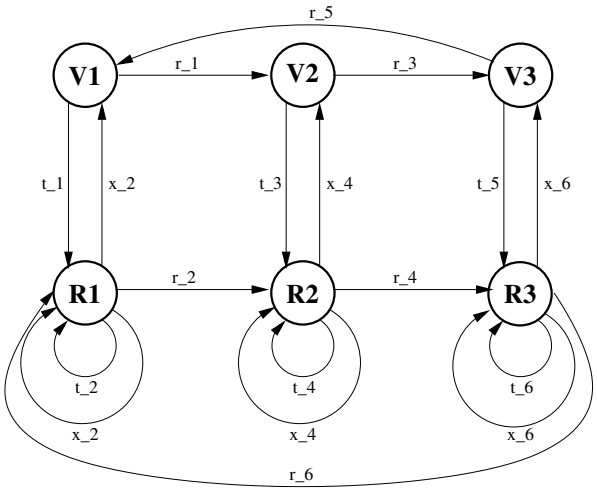

Figure 4.12: The Kahn Process Network derived after transformation `UNFOLD` is applied on loop `k` of the Matlab code describing the QR algorithm shown in Figure 4.8. The unfolding factor is three.

`Vec` operations and three processes $R1$, $R2$, and $R3$ that execute `Rot` operations. For every iteration `k` a QR update is computed that consists of 7 `Vec` operations and 21 `Rot` operations.

There are 21 QR updates because k takes values from 1 to 21. The Vec and Rot operations in QR updates corresponding to k = 1,4,7,10,13,16,19 are executed in processes $V1$ and $R1$, respectively. The Vec and Rot operations in QR updates corresponding to k = 2,5,8,11,14,17,20 are executed in processes $V2$ and $R3$, respectively. The Vec and Rot operations in QR updates corresponding to k = 3,6,9,12,15,18,21 are executed in processes $V3$ and $R3$, respectively.

The performance numbers obtained after mapping the four KPNs on the FPGA platform are summarized in Table 4.3 - see rows 3 to 6. The second column shows that by increasing the unfolding factor, the total execution time of the QR algorithm on the FPGA platform decreases. This means that by using transformation UNFOLD we gain performance improvement. The cost paid for this improvement is the increase of HW resources used to implement the KPNs - see column 3. The corresponding speedup compared to the non-transformed QR is given in the last column. With the increase of the unfolding factor, the number of parallel HW resources that perform calculations increases proportional to the unfolding factor as shown in column 3. Therefore, we should expect to see speedups approximately equal to the unfolding factor.

Let us look at row 4 where the unfolding factor is three, thus the HW resources $BCELL$ and $ICELL$ are three times more (see column 3) compared to the HW resources in the non-transformed QR. The speedup we obtain is 2.89x which is closer to 3x as we expect. It is not exactly 3x because the HW resources spend time for communication of data among each other. If we look at the last row we see that with unfolding factor of five the speedup is 3.89x which is not closer to the expected speedup of 5x. Here, again one of the reasons for this is the time spent for communication among the HW resources. However, another more important reason is the low utilization of the $BCELL$ and $ICELL$ resources - see column 4. As said earlier $BCELL$ and $ICELL$ are 3-stage pipelined IP cores. In order to exploit the pipelines efficiently we need to have in each QR update enough independent Vec and Rot operations that will allow to fill the pipelines. This can be achieved by applying our transformation SKEW on one or more loops in the algorithm.

We demonstrate the effect of transformation SKEW by skewing loop j in our QR algorithm with a factor of one. The topology of the generated KPN is the same as the topology of the KPN generated for the non-transformed QR algorithm - see Figure 4.11. However, the operations Vec and Rot executed in processes V and R, respectively, are re-ordered such that in every QR update, see Figure 4.9, the vertical data dependencies among the operations are broken. This leads to more independent operations in a QR update that allows to fill the pipelines more efficiently, thereby increasing the utilization of $BCELL$ and $ICELL$. We can see this by comparing column 4 in row 1 of Table 4.3 with column 4 in row 1 of Table 4.4.

Table 4.4: The effect of Skewing and Unfolding on QR.

Transformation	*Total number of cycles*	*HW resources* BCELL	ICELL	*Utilization (%)* BCELL	ICELL	*Speedup*
SKEW j by 1	1106	1	1	13.56	40.14	1.57x
SKEW j by 1 and UNFOLD k by 2	703	2	2	10.67	31.58	2.47x
SKEW j by 1 and UNFOLD k by 3	500	3	3	10.00	29.60	3.47x
SKEW j by 1 and UNFOLD k by 4	416	4	4	9.01	26.68	4.17x
SKEW j by 1 and UNFOLD k by 5	331	5	5	9.06	26.83	5.24x

By comparing row 1 of Table 4.3 with row 1 of Table 4.4 we see that with the same amount of HW resources we achieve speedup of 1.57x due to the improved utilization of the pipelined IP cores $BCELL$ and $ICELL$. The improved utilization achieved by transformation `SKEW` can be exploited by transformation `UNFOLD` to further speedup the execution of the QR algorithm. The performance numbers we obtain by applying transformation `SKEW` together with transformation `UNFOLD` are shown in rows 3 to 6 of Table 4.4. Comparing columns 3 and 5 of each row in this table with columns 3 and 5 of each row in Table 4.3 we see that better speedups are achieved with the same amount of HW resources when transformations `SKEW` and `UNFOLD` are used together.

The effect of our transformations `UNFOLD` and `SKEW` on the performance of the QR algorithm is visualized in Figure 4.13. We plot the total execution time of the QR measured in clock cycles that are given in the second column of Table 4.3 and Table 4.4.

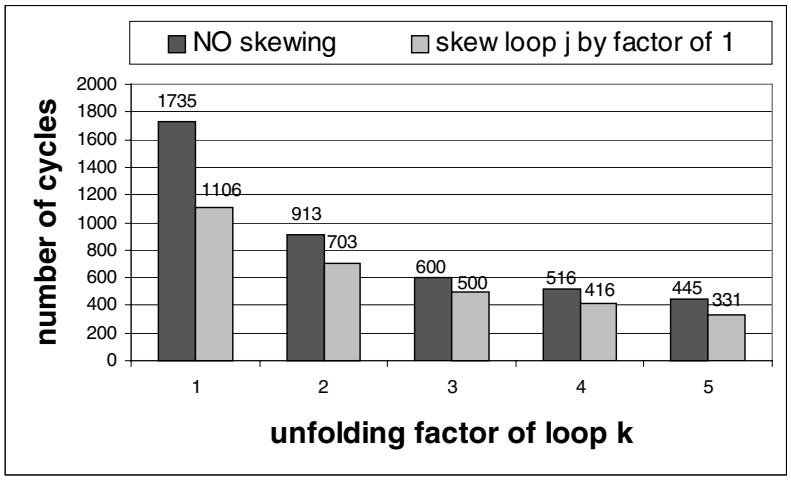

Figure 4.13: Performance of the QR algorithm transformed by using transformations `UNFOLD` and `SKEW`. The QR is mapped as KPNs on an FPGA platform. The `Vec` and `Rot` operations of the QR are implemented by 3-stage pipelined cores.

Our conclusion is that the performance of the QR algorithm mapped as a KPN onto an FPGA platform using the 3-stage pipelined IP cores $BCELL$ and $ICELL$ can be improved up to 3.89 times by unfolding loop `k` with unfolding factors up to five - see the dark bars in Figure 4.13. If we continue to increase the unfolding factor we will not get significant improvements any more because the degree of task-level parallelism that can be exploited in our QR algorithm is limited by the number of operations in a QR update and by the data dependencies among these operations. However, by skewing loop `j` by a factor of one we can break some data dependencies among operations in a QR update, thereby utilizing better the 3-stage pipelines in $BCELL$ and $ICELL$. Because of this the performance of the QR algorithm can be improved up to 5.24 times by skewing loop `j` with a factor of one and unfolding loop `k` with unfolding factors up to five - see the gray bars in Figure 4.13.

Experiment 2

This experiment is similar to Experiment 1 presented in the previous section. Again, we evaluate the effect of transformations `UNFOLD` and `SKEW` on the performance of the QR-decomposition algorithm mapped as KPNs on an FPGA platform. The difference between this experiment and Experiment 1 is that here the IP cores $BCELL$ and $ICELL$ are deeply pipelined. To realize the `Vec` operations on the platform we use a *55-stage pipelined* $BCELL$ core that implements a vectorize operation on floating point data. Similarly, to realize the `Rot` operations we use a *42-stage pipelined* $ICELL$ core that implements a rotate operation on floating point data.

First, we apply transformation `UNFOLD` on loop `k` of our QR algorithm with unfolding factors from 2 to 5. As a result, four alternative KPNs are generated that differ from each other in the degree of exploited task-level parallelism. By mapping these KPNs onto an FPGA platform we obtain the performance numbers given in Table 4.5.

Table 4.5: The effect of Unfolding on QR.

Transformation	*Total number of cycles*	*HW resources*		*Utilization (%)*		*Speedup*
		BCELL	*ICELL*	*BCELL*	*ICELL*	
NO transform	13190	1	1	1.43	3.76	1.00x
UNFOLD k by 2	7010	2	2	1.35	3.54	1.88x
UNFOLD k by 3	4452	3	3	1.42	3.71	2.96x
UNFOLD k by 4	3920	4	4	1.21	3.16	3.36x
UNFOLD k by 5	3302	5	5	1.14	3.00	3.99x

Again, the last column in the table shows that the performance of the QR algorithm can be improved and a speedup of up to 4x can be achieved. However, column 4 shows that the utilization of the hardware resources $BCELL$ and $ICELL$ is very low. This is because $BCELL$ and $ICELL$ are deeply pipelined IP cores and the data dependencies in a single QR update do not allow to fill the pipelines efficiently. To overcome this problem we apply transformation `SKEW` as we did in Experiment 1. The effect of this transformation on the performance is shown in the second row of Table 4.6.

Table 4.6: The effect of Skewing and Unfolding on QR.

Transformation	*Total number of cycles*	*HW resources*		*Utilization (%)*		*Speedup*
		BCELL	*ICELL*	*BCELL*	*ICELL*	
SKEW j by 1	2477	1	1	7.63	20.27	5.32x
SKEW j by 1 and UNFOLD k by 2	1823	2	2	5.18	13.60	7.24x
SKEW j by 1 and UNFOLD k by 3	1481	3	3	4.25	11.16	8.91x
SKEW j by 1 and UNFOLD k by 4	1215	4	4	3.89	10.21	10.86x
SKEW j by 1 and UNFOLD k by 5	1000	5	5	3.78	9.92	13.19x

Let us compare this second row with the second row of Table 4.5. We see that the utilization of $BCELL$ and $ICELL$ is improved approximately five times when transformation `SKEW` is applied. As a consequence we obtain a speedup of 5.32x with the same amount of HW resources - one $BCELL$ core and one $ICELL$ core. Moreover, by applying only transformation `SKEW` on the QR we get better performance than applying only transformation `UNFOLD`. This was not the case in Experiment 1 where the impact of `UNFOLD` on the

performance was higher compared to the impact of SKEW.

Here in our Experiment 2, the higher impact of transformation SKEW compared to transformation UNFOLD is because the IP cores $BCELL$ and $ICELL$ are deeply pipelined and transformation UNFOLD is not capable of exploiting this. However, applying transformation SKEW together with transformation UNFOLD can further improve the performance of our QR algorithm as shown in rows 3 to 6 of Table 4.6. We see that a speedup of up to 13.19x can be achieved by skewing loop j with a factor of 1 and unfolding loop k with factors up to 5.

The effect of our transformations UNFOLD and SKEW on the performance of the QR algorithm is visualized in Figure 4.14. We plot the total execution time of the QR measured in clock cycles that are given in the second column of Table 4.5 and Table 4.6.

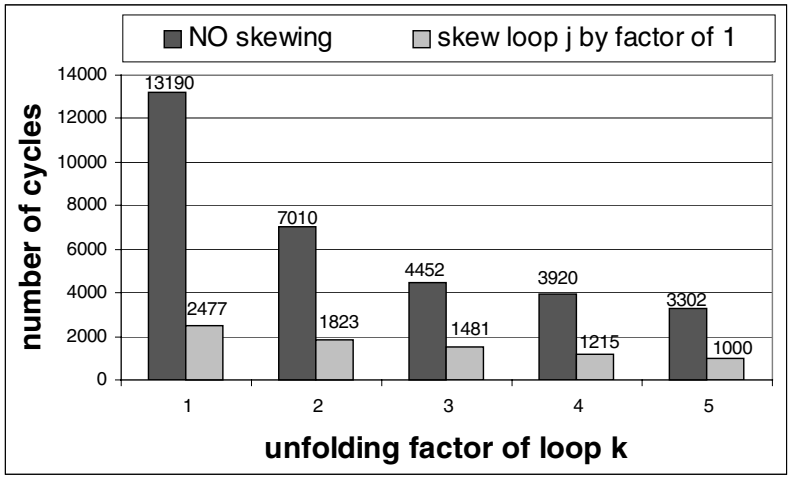

Figure 4.14: Performance of the QR algorithm transformed by using UNFOLD and SKEW transformations. The QR is mapped as KPNs on an FPGA platform. The Vec and Rot operations of the QR are implemented by using deeply pipelined IP cores: Vec is implemented by a 55-stage pipelined core and Rot is implemented by a 42-stage pipeline core.

We conclude that the performance of the QR algorithm mapped as a KPN onto an FPGA platform using deeply pipelined IP cores can be improved up to 3.99 times by unfolding loop k with unfolding factors up to five - see the dark bars in Figure 4.14. If we continue to increase the unfolding factor we will not get significant improvements any more because the utilization of the deeply pipelined IP cores $BCELL$ and $ICELL$ is very low. However, by skewing loop j by a factor of one we can break some data dependencies among operations in a QR update. This allows to fill the deep pipelines of $BCELL$ and $ICELL$ more efficiently, thereby increasing the utilization approximately five times. Because of this the performance of the QR is improved 5.32 times - see the first gray bar in Figure 4.14.

The impact of skewing on the performance of the QR is higher compared to the impact of unfolding when deeply pipelined cores are involved. This can be seen by comparing the first gray bar with the dark bars in Figure 4.14. Moreover, the high impact of transformation SKEW

can be exploited together with transformation UNFOLD to improve the performance of the QR algorithm up to 13.19 times. This can be achieved by skewing loop j with a factor of one and unfolding loop k with unfolding factors of up to five - see the gray bars in Figure 4.14.

4.2.5 Conclusions

We presented a case study in which we used our algorithmic transformation techniques described in Chapter 3 to derive systematically and fast a set of application instances (Kahn Process Networks) corresponding to a real-life application such as the QR-decomposition algorithm. We demonstrated how our techniques support a system designer in exploring the performance of alternative instances of the QR algorithm mapped onto an architecture template (in our case a FPGA platform).

The results we have obtained show that the effect of applying our transformations is that we can generate alternative application instances with different performance when mapping them onto an architecture template. It can be seen from Figure 4.13 and Figure 4.14 that our transformations UNFOLD and SKEW improve significantly the performance of the QR algorithm. Transformation UNFOLD is very useful in exploiting efficiently the task-level parallelism available in the algorithm when the algorithm is mapped onto many parallel hardware resources. Transformation SKEW improves the utilization of the hardware resources especially when the hardware resources are deeply pipelined IP cores. Our experiments show that transformations UNFOLD and SKEW have different impact on improving the performance of the QR algorithm. The impact of SKEW is higher compared to the impact of UNFOLD when deeply pipelined cores are involved. If the cores are not deeply pipelined then the impact of UNFOLD is higher compared to the impact of SKEW. Our transformation plane cutting was not mentioned in the QR case study because in the QR case this transformation does not have any significant impact on the performance.

We have implemented our transformations in the tool MATTRANSFORM. Using this tool together with the tools COMPAAN and LAURA we are able to fully automate the process of deriving alternative application instances. This helps a system designer to speedup significantly the process of exploring the performance of alternative application instances. In our experiments with the QR algorithm we explored the performance of 20 instances thereby obtaining the performance numbers given in Table 4.3, Table 4.4, Table 4.5, and Table 4.6. It took us two days to derive these 20 instances and to obtain the performance numbers. This indicates that an extensive design space exploration of alternative application instances can be done in a relatively short amount of time.

Chapter 5

Summary and Conclusions

In this dissertation we have presented a novel systematic and automated approach for converting Weakly Dynamic Programs (WDP) to equivalent Kahn Process Network (KPN) specifications. Our approach is essential for the systematic and automated design of the emerging embedded systems-on-chip platforms where a set of parallel specifications of an application has to be derived in order to allow systematic and efficient exploration and mapping of the application onto the platform. Many system-level design flows and application modeling and exploration approaches reported in the literature use the Kahn Process Network (KPN) model of computation for a parallel application specification. The derivation of a single KPN specification, let alone a set of KPNs, from an application is based on heuristic and time consuming manual approaches because there is not a sufficient amount of research work done in the area of systematic and automated derivation of KPNs.

The only research work known in this area is the work of Rijpkema et al. [19] [18] and the work of Turjan et al. [30]. They propose a systematic approach to derive a single KPN specification from an application specified as a static affine nested loop program. They put a restriction on the input program to be *static* in order to enable the automatic analysis and conversion of the input program to a KPN. Although, many applications in the domain of Scientific, Matrix Computation and Adaptive Digital Signal Processing can be specified as static programs the *static* restriction limits the applicability of their approach when media applications such as JPEG codecs, MPEG codecs, Smart Cameras, Software Radio, etc. have to be considered. This is because such applications have a dynamic (data-dependent) behavior which can not be expressed as a static affine nested loop program (SANLP). Our work presented in Chapter 2 of this dissertation demonstrated that the *static* restriction on the input program can be relaxed in a particular way and a more general class of programs called Weakly Dynamic Programs (WDP) can be automatically analyzed and converted to KPNs. This implies that our approach presented in Chapter 2 extends significantly the class of applications from which KPN specifications can be derived in a systematic and automated way. Our approach can handle not only Scientific, Matrix Computation and Adaptive Signal Processing applications but also media applications with dynamic (data-dependent) behav-

ior such as JPEG codecs, MPEG codecs, etc. because such applications can be specified as WDPs.

Converting a weakly dynamic program (WDP) to a KPN specification in a systematic and automated way is a challenging and complex problem because the exact behavior of a WDP is unknown at compile-time. This comes from the fact that in a WDP there are *if-then-else* constructs with dynamic (data-dependent) conditions, i.e., conditions that can be an arbitrary function of data variables which values may be unknown at compile-time. This means that the outcome of such conditions may be unknown at compile-time making the behavior of a WDP unknown. In Chapter 2 we demonstrated that although the exact behavior of a WDP is unknown at compile-time still such a program can be analyzed and converted to an executable KPN specification in a systematic and automated way. Our approach to do this consists of three main steps.

First, a WDP is converted to a functionally equivalent single-assignment program, i.e., a program in which every variable is written at most once. In this dissertation we have developed and used our special form of a single-assignment program called dynamic Single Assignment Code (dSAC) because the existing single-assignment forms can not capture efficiently the dynamic (data-dependent) behavior of a WDP. The procedure we have developed to convert a WDP to our dSAC is based on a very advanced dependence analysis technique called *fuzzy* array data flow analysis (FADA) because the state of the art exact data flow analysis can not handle the unpredictable behavior of a WDP. Using FADA in our procedure to obtain our dSAC leads to a code that expresses explicitly all possible data dependencies between the function calls in a WDP. Some of these data dependencies are not exactly defined, i.e., they depend on parameters introduced by FADA. To keep the functional behavior of our dSAC equivalent to the behavior of the initial WDP the values of these parameters have to be set dynamically at run-time. In Chapter 2 we presented our way of setting these parameters that is based on placing a very simple and efficient code called *control variables* in the dSAC.

Second, the dSAC obtained in the first step of our approach is transformed into more compact representation consisting of two models, namely *Approximated Dependence Graph* (ADG) and *Schedule Tree* (STree). We introduced the ADG model and the STree model in order to capture all the information that is present in the dSAC in a formal way. Transforming the dSAC into these models has several advantages: (1) the dSAC is very large and complex data structure to operate on whereas the ADG and the STree are very compact and easy models to operate on; (2) these models enable us to decompose the conversion problem into a number of well defined sub-problems; (3) the conversion to a KPN can be done in a structured and formal way because formal transformations can be easily defined and applied on the ADG and the STree instead of the dSAC. Because of the reasons above we decided to convert the dSAC into the ADG and STree models and derive the KPN from there.

The ADG model contains all the information that is related to the data dependencies between the functions in a dSAC as well as information about the iterations at which each function in the dSAC is executed. As said earlier some of the data dependencies in a dSAC are approximated, i.e., the exact data dependencies are not known at compile-time. We have developed the ADG model because the classical and widely used Dependence Graph (DG) or Polyhedral Reduced Dependence Graph (PRDG) models are not general enough to capture approximated data dependencies occurring in a dSAC. For every function in a dSAC there is

a node in the ADG. For every variable in a dSAC there is an edge in the ADG that indicates *possible* data dependency. Every node in the ADG is characterized by a set of integral points that describes the iterations at which the corresponding function in the dSAC is executed. For some functions the exact execution iterations are unknown at compile-time because of date-dependent "if"-constructs occurring in the dSAC. Therefore, we annotated the nodes in the ADG with *Linearly Bounded Sets* (LBS). We have developed the notion of a LBS in order to approximate the unknown information about the iterations at which functions in the dSAC are executed.

The STree model contains all the information about the execution order between the functions in a dSAC. The STree represents one valid schedule between all these functions that we call the global schedule. We have shown that from the STree a local schedule between any arbitrary set of the functions in the dSAC can be obtained by pruning operations on the STree.

Third, the ADG model and the STree model are converted into the KPN model. This is the last step in our approach. A KPN consists of concurrent processes that communicate data with each other over unbounded FIFO channels. Every process executes a sequential code. The synchronization between the processes is accomplished by a *blocking read* mechanism, i.e., a read operation from a FIFO channel blocks when no data is available in the channel. The generation of the KPN from the ADG and the STree is performed in two stages. In the first stage the topology of the process network is generated. The topology is determined by grouping nodes and edges of the ADG into processes and channels in the KPN. Our approach allows an arbitrary grouping of nodes into processes, thereby allowing the generation of KPNs with different topologies and degree of exploited parallelism. In the second stage the processes and their sequential code is generated. The code must be generated in such way that the functions which have to be executed inside a process are called in the proper order. This order is determined by using the information captured in the STree model and the order guarantees a deadlock free execution of the KPN.

An important characteristic of our approach for converting a weakly dynamic program (WDP) to a KPN is that we do this conversion without changing the semantics of the KPN model of computation. This is beneficial in the sense that all important properties of the KPN model are preserved. Introducing weakly dynamic behavior in the KPN model and still preserving its semantics implies the following as we showed in Chapter 2: (1) in a KPN derived from a WDP we distinguish two types of communication FIFO channels depending on the purpose of the communicated data. These two types are *data FIFO channels* and *control FIFO channels*. The control FIFO channels appear because the behavior of the WDP is not know completely at compile-time. The unknown behavior has to be resolved at run-time in the KPN and the control FIFO channels are used to communicate the necessary data to do this; (2) Because of the unknown behavior of a WDP at compile-time, some processes in the KPN may send more data into the data FIFO channels than actually needed at run-time. In order to discard the unnecessary data at run-time the data is tagged (colored) and a tag (color) matching is performed at run-time.

Control FIFO channels and tagging (coloring) of data are not necessary in case a KPN is derived from a static affine nested loop program. This means that the presence of control FIFO channels and tags (colors), i.e., extra communication workload is the "price" we have to pay when deriving a KPN from a weakly dynamic program.

Another important contribution of this dissertation is our work presented in Chapter 3 on deriving a set of alternative KPN specifications from an application specified as a weakly dynamic program (WDP). This work is important in system-level design, because it gives system designers an opportunity to perform design space exploration and to select a KPN specification that meets best the system requirements. In Chapter 3 we have presented several *task-level* algorithmic transformations which we have developed to facilitate a systematic derivation of alternative KPN specifications from a WDP. These KPN specifications are behaviorally equivalent to the input WDP but the degree of exploited task-level parallelism is different. We have demonstrated how our transformations can be encapsulated in an *application transformation layer* on the top of a Y-chart exploration environment in order to facilitate system designers in exploring the performance of alternative KPNs mapped onto an architecture template. To the best of our knowledge our application transformation layer provides for the first time a systematic and fast approach to derive alternative KPNs from an application specified as a WDP.

In Chapter 3 we have presented our set of four task-level transformations, namely unfolding, plane cutting, skewing, and merging. We have shown that our unfolding, plane cutting, and skewing transformations can be used to transform a WDP such that more task-level parallelism is revealed and exploited when converting the transformed WDP to a KPN specification. Moreover, the unfolding transformation or the plane cutting transformation can reveal the maximum task-level parallelism available in a WDP. This means that we can generate a KPN that exploits in full degree the task-level parallelism in a WDP. Our merging transformation has the opposite effect compared to the unfolding, plane cutting, and skewing transformations. The merging transformation decreases the degree of exploited task-level parallelism when converting the transformed WDP to a KPN. By merging we can generate a KPN that consists of only one process, i.e., a KPN where no parallelism is exploited.

Although, our set of algorithmic transformations is very small its transformation power is very large when the transformations are applied in combination on a WDP. By applying the unfolding or plane cutting transformation in combination with the merging transformation one can get whatever task-level distribution of the computational workload of a WDP over parallel processes. This means that our algorithmic transformations allow systematic derivation of a set of alternative KPN specifications ranging from a KPN where NO parallelism is exploited to a KPN where FULL task-level parallelism is exploited.

When a WDP is transformed using our transformations, extra control structures and operations are placed in the transformed WDP. The effect of these extra control structures and operations on the performance of the KPN derived from the transformed WDP gets lower when the granularity of the function calls executed inside the processes gets higher. This means that our transformations are very efficient when the function calls in the initial WDP represent relatively big and computational intensive tasks. By converting the transformed WDP into a KPN specification a lot of optimizations are done to minimize the effect of the additional control and operations on the performance of the KPN. Moreover, the extra control and operations are distributed over the parallel processes of the KPN and they are executed in parallel thereby minimizing the effect on the performance.

In order to validate and evaluate our approach to convert a WDP to a KPN (Chapter 2) and our transformations to derive a set of alternative KPNs (Chapter 3) we have performed the

following two activities.

First, we have prototyped our approach and transformations in software. The methods and techniques involved in the approach and the transformations are formulated in this dissertation in close resemblance to their corresponding software. The benefit of doing this is that this dissertation can be used to understand the internal structure and behavior of the prototype software we have developed. The following software has been developed: (1) most of the methods and techniques of our approach have been prototyped and tested as software procedures. There is work in progress for the complete implementation of the approach in software; (2) our algorithmic transformations are implemented in a software tool called MATTRANSFORM.

Second, we have conducted several experiments and two case studies using our prototype software. These case studies, the corresponding experiments, and the obtained results have been reported in Chapter 4 of this dissertation. The case studies have clearly shown that our research work presented in this dissertation can be applied successfully on real-life industrially relevant applications.

In the first case study we have shown how for a Motion-JPEG (MJPEG) encoder application written as a weakly dynamic program in Matlab, a Kahn Process Network specification was derived in a systematic and automated way using our approach presented in Chapter 2. This specification was systematically mapped onto a real hardware platform composed of a microprocessor and an FPGA using the COMPAAN/LAURA tools [18] [64]. Based on our experience with the case study and the results we have obtained we draw the following conclusions. First, we conclude that a large industrially relevant application such as a Motion JPEG encoder (MJPEG) can be easily specified as a weakly dynamic program (WDP) and converted to a KPN specification in a systematic and automated way using our approach presented in Chapter 2. Second, we conclude that our approach enables a system designer to derive a KPN from a large industrial relevant application in a relatively short amount of time - currently, a few days. Although, our approach was not fully automated, the MJPEG was converted to a KPN specification in a systematic and semi-automatic way in four days. When our approach is fully automated this time will be reduced to several minutes. For comparison, a KPN specification of the same MJPEG encoder was derived by hand in [16] that took four weeks. Third, we conclude that our approach can be used and integrated successfully in a system design flow which relies on the KPN model of computation to map efficiently real-life applications onto real hardware platforms in a systematic and automated way. In the case study we have shown such design flow where the KPN specification derived by our approach allowed the parallel hardware resources in the platform to run concurrently, thereby exploiting efficiently task-level parallelism. Finally, we conclude that our approach includes only basic techniques that we have developed in order to convert automatically WDPs to KPNs. The results, we have obtained for the MJPEG application, indicated that some optimization techniques have to be added to the approach that will help the improving of the quality of the generated KPNs in terms of optimal partitioning of the computation and communication workloads of a WDP over processes and channels in the KPN.

In the second case study we have used the algorithmic transformations presented in Chapter 3 to derive systematically and fast alternative Kahn Process Networks (KPNs) from a real-life signal processing algorithm, namely the QR-decomposition algorithm. We have mapped the

alternative KPNs onto a FPGA platform in order to demonstrate the effect of the transformations on the performance of the QR algorithm. Based on the results we have obtained and the experience we have gained from this case study we can conclude the following. First, we conclude that the effect of applying our transformations on a real-life algorithm is that we can derive systematically alternative KPN specifications from the algorithm with different degree of exploited task-level parallelism resulting in different performance and hardware utilization when mapping the KPNs onto the hardware platform. From the QR algorithm we derived 20 KPNs that were functionally equivalent to the algorithm but all of them showed different performance when they were mapped onto our FPGA platform. Second, we conclude that using our transformations implemented in the tool MATTRANSFORM together with the COMPAAN/LAURA tools [18] [64] allows a system designer to perform an extensive design space exploration of alternative KPNs in a relatively short amount of time. In our case study we explored the performance of 20 KPNs derived from the QR algorithm. It took us in total only two days to derive and map these 20 KPNs and to obtain performance numbers. For comparison, doing this by hand takes a single designer several months as reported in [83]. Finally, we conclude that the impact of each one of our transformations on the performance of the derived alternative KPNs is different and it depends on the data dependencies in the initial algorithm and on the hardware resources onto which the derived KPNs are mapped. Therefore, using and applying our transformations efficiently implies that a system designer is familiar with the algorithm and the target platform. Using and applying our transformations ”blindly” is also possible but it may result in an exhaustive and time consuming exploration process.

Bibliography

[1] Axel Jantsch and Hannu Tenhunen. *Networks on Chips*. Kluwer Academic Publishers, 2003.

[2] Alberto Sangiovanni-Vincentelli and Grant Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[3] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[4] F. Balarin, E. Sentovich, M Chiodo, P. Giusto, H. Hsieh, B Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, 1997.

[5] Daniel D. Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[6] Daniel D. Gajski. System Level Design Flow: What is needed and What is not. Technical report, CECS, University of California at Irvine, 2002. CECS-TR-02-33.

[7] Bart Kienhuis, Ed Deprettere, Pieter van der Wolf, and Kees Vissers. *A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach*. in Embedded Processor Design Challenges, LNCS 2268, Editors Ed F. Deprettere, Juergen Teich, and Stamatis Vassiliadis, Springer, 2002.

[8] Andrew Mihal and Kurt Keutzer. Mapping Concurrent Applications onto Architectural Platforms. In Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chips*, pages 39–59. Kluwer Academic Publishers, 2003.

[9] Edward Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[11] Andy Pimentel, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[12] E.A. Lee et al. PtolemyII: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[13] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[14] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.

[15] Pieter van der Wolf, Paul Lieverse, Mudit Goel, David La Hei, and Kees Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 3-5 1999.

[16] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System Level Design with SPADE: an M-JPEG Case Study. In *Proc. Int. Conference on Computer Aided Design (ICCAD'01)*, pages 31–38, San Jose CA, USA, November 4-8 2001.

[17] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. Int. Conference Design, Automation and Test in Europe (DATE'04)*, pages 340–345, Paris, France, February 16-20 2004.

[18] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[19] Edwin Rijpkema. Modeling Task Level Parallelism in Piece-wise Regular Programs, 2002. PhD thesis, Leiden University, The Netherlands.

[20] Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. A Compile-time based Approach for Solving Out-of-Order Communication in Kahn Process Networks. In *Proc. IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP'2002)*, San Jose, USA, July 17-19 2002.

[21] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. *Realizations of the Extended Linearization Model*. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc., 2003.

[22] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A Technique to Determine Interprocess Communication in the Polyhedral Model. In *Proc. Int. Workshop on Compilers for Parallel Computers (CPC'03)*, Amsterdam, The Netherlands, January 8-10 2003.

[23] Alexandru Turjan and Bart Kienhuis. Storage Management in Process Networks using the Lexicographically Maximal Preimage. In *Proc. of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, January 24-26 2003.

[24] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. Journal of Parallel Programming*, 20(1):23–53, 1991.

[25] William Pugh and David Wonnacott. An Exact Method for Analysis of Value-Based Array Data Dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, vol. 768*. Springer-Verlag, Berlin, 1993.

[26] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array Dataflow Analysis and its use in Array Privatization. In *Proc. Int. Conference on Principles of Programming Languages*, pages 2–15, January 1993.

[27] Paul Feautrier. Parametric Integer Programming. Operations Research, 22(3):243-268, 1988.

[28] Paul Feautrier and Jean-Francois Collard. Fuzzy Array Dataflow Analysis. Technical report, Ecole Normale Superieure de Lyon, 1994. ENS-Lyon/LIP N^o 94-21.

[29] Denis Barthou, Jean-Francois Collard, and Paul Feautrier. Fuzzy Array Dataflow Analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, 1997.

[30] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating Affine Nested-loop Programs to Process Networks. In *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'04)*, Washington D.C., USA, September 23-25 2004.

[31] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 107–120, San Diego, California, USA, January 1998.

[32] Paul Feautrier, Jean-Francois Collard, Michel Barreteau, Denis Barthou, Albert Cohen, and Vincent Lefebvre. The Interplay of Expansion and Scheduling in PAF. Technical report, PRiSM, University of Versailles, France, 1998. Report #1998/6.

[33] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal, and Francky Catthoor. A Step towards a Scalable Dynamic Single Assignment Conversion. Technical report, Katholieke Universiteit Leuven, 2003. Report CW360, April 2003.

[34] Patrice Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proc. of the the 11th Annual International Symposium on Computer Architecture*, pages 208–214, Ann Arbor, Michigan, USA, June 1984.

[35] Lothar Thiele. On the Design of Piecewise Regular Processor Arrays. In *Proc. IEEE Symposium on Circuits and Systems*, pages 2239–2242, 1989.

[36] Juergen Teich and Lother Thiele. Partitioning of Processor Arrays: a Piecewise Regular Approach. *INTEGRATION: The VLSI Journal*, 14(3):297 – 332, February 1993.

[37] Juergen Teich, Lother Thiele, and L. Zhang. Partitioning Processor Arrays under Resource Constraints. *Int. Journal on VLSI and Signal Processing Systems*, 15(1):5 – 21, 1997.

[38] Michael van Swaaij, Frank Franssen, Francky Catthoor, and Hugo De Man. Modeling Data Flow and Control Flow for DSP System Synthesis. VLSI Design Methodologies for DSP Systems, M. Bayoumi editor, Kluwer, 1993.

[39] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[40] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), December 1994.

[41] Sundararajan Sriram and Shuvra Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[42] Keshab Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, Inc., 1999.

[43] Keshab K. Parhi and David G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding. *IEEE Transaction on Computers*, 40(2):178–195, February 1991.

[44] Jurgen Teich and Lothar Thiele. Exact Partitioning of Affine Dependence Algorithms. *Lecture Notes in Computer Science (LNCS), Springer*, 2268:133–151, 2002.

[45] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.

[46] Joe Coffland and Andy Pimentel. A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems. In *Proc. of the 18th ACM Symposium on Applied Computing, Embedded Systems track*, pages 666–671, Melbourne, Florida, USA, March 2003.

[47] Todor Stefanov and Ed Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proc. IEEE-ACM-IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*, pages 90–96, Newport Beach, California, USA, October 1-3 2003.

[48] Paul Lieverse, Pieter van der Wolf, Kees Vissers, and Ed Deprettere. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Int. Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, 2001.

[49] Martijn J. Rutten, Jos T.J. van Eijndhoven, and Evert-Jan D. Pol. Design of Multi-Tasking Coprocessor Control for Eclipse. In *Proc. 10th Int. Symposium on Hardware/Software Codesign (CODES'02)*, pages 139–144, Estes Park, Colorado, USA, May 6-8 2002.

[50] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks. In *Proc. IEEE-ACM-IFIP Int. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*, pages 182–187, Newport Beach, California, USA, October 1-3 2003.

[51] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *Proc. 10th International Symposium on Hardware/Software Codesign (CODES'02)*, pages 7–12, Estes Park, Colorado, USA, May 6-8 2002.

[52] Bart Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[53] Peter Held. Functional Design of Data-Flow Networks, 1996. PhD thesis, Delft University of Technology, The Netherlands.

[54] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[56] Alexandru Turjan. A procedure to represent lattice as if-statements, 2002. Technical Memo N^o5, Leiden University, The Netherlands.

[57] Alexandru Turjan. The Compaan out-of-order detection step, 2002. Technical Memo N^o7, Leiden University, The Netherlands.

[58] Bart Kienhuis. Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools, January 1999. PhD thesis, Delft University of Technology, The Netherlands.

[59] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. The Construction of a Retargetable Simulator for an Architecture Template. In *Proc. 6-th Int. Workshop on Hardware/Software Codesign (CODES'98)*, Seattle, Washington, March 15-18 1998.

[60] Vladimir Zivkovic, Erwin de Kock, Ed Deprettere, and Pieter van der Wolf. Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs. In *Proc. Int. Conference Design, Automation and Test in Europe (DATE'03)*, Munich, Germany, March 3-7 2003.

[61] Vladimir Zivkovic, Pieter van der Wolf, Ed Deprettere, and Erwin de Kock. Design Space Exploration of Streaming Multiprocessor Architectures. In *Proc. Int. IEEE Workshop on Signal Processing Systems (SIPS'02)*, San Diego, California, USA, October 16-18 2002.

[62] Andy Pimentel, Simon Polstra, Frank Terpstra, Berry van Halderen, Joe Coffland, and Bob Hertzberger. *Towards Efficient Design Space Exploration of Heterogeneous Embedded Media Systems*. in E. Deprettere, J. Teich and S. Vassiliadis (eds), Embedded Processor Design Challenges: Systems, Architectures, MOdeling, and Simulation (SAMOS), Springer LNCS, number 2268, pp. 57-73, 2002.

[63] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. ACM SIGPLAN'91*, pages 39–50, June 1991.

[64] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September 1-3 2003.

[65] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards; Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

[66] W.B. Pennebacker and J.L. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[67] W.B. Pennebacker, J.L. Mitchel, C.E. Fogg, and D.J. LeGall. *MPEG Video Compression Standard*. Chapman and Hall, 1996.

[68] http://www.alpha-data.com/adm-xrc-ii.html. Alpha Data Parallel Systems, Ltd.

[69] PVRG-JPEG CODEC 1.1. Portable Video Research Group, Stanford University.

[70] J. Villarreal, G. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving Software Performance with Configurable Logic. *Kluwer Journal on Design Automation of Embedded Systems*, 7(4):325 – 339, November 2002.

[71] http://www.xilinx.com/. Xilinx, Inc.

[72] www.synplicity.com/products/synplifypro/index.html. Synplicity, Inc.

[73] http://www.xilinx.com/. Xilinx, Inc.

[74] Wayne Wolf. A Decade of Hardware/Software Codesign. *IEEE Computer*, 36(4):35 – 43, April 2003.

[75] Y. Li, T. Callahan, E. Dernell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proc. 37th Design Automation Conference (DAC'00)*, pages 507–512, Los Angeles, CA, June 5-9 2000.

[76] Timothy Callahan, John Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, pages 62–69, April 2000.

[77] Vinod Kathail, Shail Aditya, Robert Schreiber, and Bob Rau. PICO: Automatically Designing Custom Computers. *IEEE Computer*, 35(9), September 2002.

[78] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-Oriented FPGA Computing in the Stream-C High Level Language. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, April 2000.

[79] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[80] Eric Verhulst. Beyond the Von Neumann Machine: Communication as the driving design paradigm for MP-SoC from software to hardware. In Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chips*, pages 217–238. Kluwer Academic Publishers, 2003.

[81] Eylon Gaspi et al. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proc. 10th Int. Conference on Field Programmable Logic and Applications (FPL'00)*, August 28-30 2000.

[82] John Proakis, Charles Rader, Fuyun Ling, Chrysostomos Nikias, Mark Moonen, and Ian Proudler. *Algorithms for Statistical Signal Processing*. Prentice Hall, Inc., 2002.

[83] Tim Harriss, Richard Walke, Bart Kienhuis, and Ed F Deprettere. Compilation from Matlab to Process Networks Realized in FPGA. *Journal on Design Automation of Embedded Systems*, 7(4), 2002.

Index

Samenvatting

Dit proefschrift introduceert een nieuwe, systematische en automatische methode en procedure voor het vertalen van Zwak-Dynamische Sequentiële Programma's naar functioneel equivalente, parallelle specificaties in de vorm van Kahn-Proces-Netwerken. Het Kahn-Proces-Netwerken rekenmodel leent zich beter dan het Sequentiële Imperatieve rekenmodel voor de implementatie van, onder meer, Multimedia taken in multi-processor architecturen.

De methode en procedure die wordt voorgesteld in dit proefschrift verschaft een belangrijke en niet triviale uitbreiding van een eerder voorgestelde methode en procedure voor de automatische vertaling van Statische Sequentiële Programma's - in het bijzonder statische, affine geneste lus programma's zoals bekend uit de HPC wereld - naar functioneel equivalente Kahn-Proces-Netwerken specificaties. In dit geval is het Kahn-Proces-Network een aantrekkelijk compact alternatief voor het zogenoemde Cyclostatische Dataflow Netwerk rekenmodel. In de praktijk evenwel blijken vele toepassingen niet gespecificeerd te kunnen worden op deze manier omdat ze niet statisch, maar dynamisch zijn: Ze zijn data afhankelijk.

In hoofdstuk 2 van dit proefschrift wordt aangetoond dat de beperkingen die het gevolg zijn van de aanname dat data afhankelijkheid geen rol speelt ten dele kunnen worden opgegeven. Met andere woorden, de automatische vertaling van sequentiële programma's naar parallelle (Kahn-Proces-Netwerk) specificaties blijft mogelijk, ook als er data afhankelijke constructies in de programma's voorkomen - ook al moeten ze aan zekere zwakke voorwaarden voldoen. Vanwege die laatste beperking worden zulke programma's in dit proefschrift Zwak-Dynamische Programma's genoemd. Vele programma's die Multimedia taken, maar ook Adaptieve Signaalbewerkingstaken uit de Communicatie, Radar, en Radioastronomie, specificeren voldoen aan dit zwak dynamische criterium.

De uitbreiding van statische naar dynamische programma's wordt gehinderd door het feit dat een van de belangrijkste stappen in de vertaling naar parallelle Kahn-Proces-Netwerk specificaties faalt door de onzekerheden die optreden als gevolg van de data afhankelijke constructies in de dynamische programma's. In hoofdstuk 2 wordt onder andere aangetoond dat deze hindernis genomen kan worden wanneer het dynamisch gedrag 'zwak' is zoals gedefinieerd in dit proefschrift.

Hoofdstuk 3 bevat een andere bijdrage die van groot belang is wanneer een Kahn-Proces-Netwerk specificatie moet worden geimplementeerd in een multi-processor architectuur. De bijdrage betreft de mogelijkheid om een gegeven (zwak-dynamisch) sequentieel programma te vertalen naar meerdere functioneel equivalente parallelle Kahn-Proces-Netwerk specificaties. Deze keuzemogelijkheid is aantrekkelijk wanneer voorafgaand aan implementatie beslissingen nog optimalisatie exploraties gedaan moeten worden. In hoofdstuk 3 worden vier mogelijke, systematische en automatische transformaties van (zwak-dynamische) sequentiële programma's voorgesteld die samen vele mogelijke gradaties van parallellisme op een gebalanceerde manier over meerdere processoren in een architectuur kunnen distribueren. Drie van de vier transformaties worden automatisch toegepast op de gegeven sequentiële specificatie en bevatten een of meer parameters die, afhankelijk van hun waarden, een ander equivalent Kahn-Proces-Netwerk opleveren, typisch met een grotere mate van parallellisme. De vierde transformatie - waarmee parallellisme kan worden gereduceerd - wordt uitgevoerd op een Kahn-Proces-Netwerk specificatie, maar maakt gebruik van informatie die niet in die specificatie beschikbaar is maar wel in de gegeven sequentiële specificatie.

De methoden om Zwak-Dynamische Sequentiële programma's automatisch te vertalen naar parallelle Kahn-Proces-Netwerk specificaties zijn alle omgezet in software routines die samen een executeerbaar vertalingsprototype vormen. In hoofdstuk 4 zijn experimenten opgenomen die met dit prototype zijn uitgevoerd. Deze experimenten demonstreren en evalueren de theoretische onderbouwingen die zijn aangedragen in Hoofdstuk 2 en Hoofdstuk 3.

Curriculum Vitae

Todor Stefanov was born on July 11, 1974 in Samokov, Bulgaria. In 1993 he received his high school diploma at The High Technical School of Microprocessor Technology in Pravetz, Bulgaria. The same year he started his study in computer engineering at the Technical University of Sofia, Bulgaria. In 1998, Todor Stefanov received his Dipl.Ing. and M.Sc. degrees in Computer Engineering from the Technical University of Sofia after successfully defending his M.Sc. thesis titled "Design, Analysis, and Area Minimization of the Control Unit of Application Specific Microprocessor Core". During his M.Sc. study, he worked at Info MicroSystems, Ltd., Sofia, Bulgaria on designing application specific microprocessor IP cores. From 1998 till 2000, Todor Stefanov was a Research and Development Engineer at Innovative Micro Systems, Ltd., Sofia, Bulgaria where he worked on the development of a reconfigurable MicroSystems-on-Silicon In-Circuit Emulator based on FPGAs. In 2000, Todor Stefanov joined the Leiden Embedded Research Center (LERC) which is a part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University where he was appointed as a research assistant (Ph.D. student). He was involved in the ARTEMIS project which deals with ARchitectures and meThods for Embedded MedIa Systems. As a member of the ARTEMIS project he conducted research in the context of modeling of stream-oriented media applications and mapping them onto parallel architectures. In particular, he worked on methods and techniques for systematic and automated derivation of process networks from weakly dynamic applications. This research work culminated in the writing of this Ph.D. thesis.