Modeling Task Level Parallelism in Piece-wise Regular Programs

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 17 september 2002 klokke 14:15 uur

door

Edwin Rijpkema
geboren te Amstelveen
in 1970

samenstelling promotiecommissie:

promotor	Prof.dr.ir. E.F. Deprettere	
co-promotor	Dr.ir. A.C.J. Kienhuis	
referent	Dr. Z. Chamski	Philips Research, Eindhoven
overige leden:	Prof.dr. L. Thiele	KTH, Zürich, Zwitserland
	Prof.dr. P. Quinton	IRISA, Rennes, Frankrijk
	Prof.dr.-ing J. Teich	Paderborn Universiteit, Duitsland
	Prof.dr. H.A.G. Wijshoff	
	Dr. V. Loechner	ICPS, Straatsburg, Frankrijk
	Dr. T. Risset	ENS, Lyon, Frankrijk

Van dit proefschrift is ook
een handelseditie verschenen
onder ISBN 90-9016179-1

Printed in the Netherlands

Contents

Preface

This dissertation is the result of the work that has been conducted partly at the Delft University of Technology, partially at Leiden University, and partially at home. I am indebted to many people at these various places.

I want to mention Ralph Otten to give me the opportunity to start my Ph.D. at the Circuits and Systems (CaS) group at the department of Information Technology and Systems of the Delft University. This group covers a broad spectrum of disciplines from which I learned a lot. I am grateful to all people involved in the Jacobium project, Yijun Xu, Simone Fantini, Giuseppe Carcea, Daniel van Loenen, Alco Looye, Jun Ma, and, in particular Gerben Hekstra, for their contributions and their ideas on the initial work in this dissertation. I am also grateful to Peter Held and Paul Wielage for their assistance in understanding their tools, which helped me to built a basis for my own work, and Paul Lieverse for the discussions we had and experiments we did to proof the complementaryness of our research topics.

I want to mention Todor Stefanov, Vladimir Zivkovic, and Alexandru Turjan, at the Leiden Embedded Research Center (LERC) at the Leiden Institute of Advanced Computer Science of the Leiden University, for the technical discussions I had with them and valuable feedback I got from them at the later stage of my research project.

I am grateful to Vincent Loechner from ICPS, Straatsburg, France, for his time and the discussions we had on counting problems and various aspects of it, and for his assistance in using the PolyLib library.

I am grateful to Marion de Vlieger (CaS) and Eugenie Baken (LERC), for all kinds of non-technical support that helped me to realize this dissertation.

Specially, I want to mention my supervisor and promotor Ed Deprettere, for providing a stimulating and motivating research environment; first in the CaS group, later in the LERC. I am grateful for the many fruitful and pleasant discussions we had on all aspects within and many aspects outside the scope of this dissertation. I am also grateful for his continuous interest in this work and keeping me motivated all the time.

I also want to mention my co-promotor Bart Kienhuis, for the many fruitful discussions and cooperation on various aspects of this dissertation while he was a Ph.D. student in the CaS group. I am grateful to him for gaining my interest in software engineering while he had a Post Doctoral fellowship at UC Berkely and for his interest and feedback on this dissertation after he joined the LERC.

Finally and especially, I want to thank Marjolein for her love, her patience while I spent many “free” hours on this dissertation, and her encouragement to get me to do so.

Edwin Rijpkema

July 22, 2002
Eindhoven

Chapter 1

Introduction

Audio, video, radar, and sonar are examples of application domains that fall in the class of digital signal processing (DSP) applications. The increasing performance demands of these applications goes hand in hand with the increasing potential performance capacity of chips. However, exploiting this potential capacity is a main challenge.

Digital signals are sequences or *streams* of data samples or *tokens*. The processing of signals consists of procedures that convert tokens to tokens and streams to streams by passing signals through operators. Operators take data tokens from input streams as arguments and return values as data tokens in output streams.

A compact way to specify signal processing procedures is by means of *nested loop programs*. Nested loop programs are compact because the operations are scheduled in an easily conditionally guarded lexicographical order. However, by ordering the operations in this manner, concurrency that may be present in the procedure is completely undone or *hidden*. In fact, nested loop program specifications of signal processing procedures are only appealing when the target implementation architecture is a shared memory instruction set architecture (ISA). However, this architecture, though, is not the only one that could be envisaged. The shared memory instruction set architecture, also known as the general purpose processor (GPP) is the most flexible among all possible architectures. This is because the instructions are so chosen that almost all operations can be executed in the architecture, be it that only one or at most a few instructions can be executed at a time, and instructions are encoding a given set of rather low level operations. Now, applications built on stream-based signal processing procedures are typically requiring that a high throughput must be sustained. That is, tokens in the streams must be processed at a high rate. The procedures themselves are usually such that this requirement can in principle be satisfied. However, their common specifications together with the matching architectures, the GPPs, including the digital signal processors, even when they are very long instruction word (VLIW) architectures, cannot sustain that high throughput except maybe at the cost of a very high power consumption. There are several ways to overcome this problem.

A first option is to add to the instruction set of the GPP a few powerful instructions, and to modify the GPPs data path and micro instructions in such a way that the new instructions can be executed in one clock cycle, see [1]. Without such modifications, the subset of instructions in the original ISA having the same overall functionality would take several clock cycles.

A second option consists of adding to the GPP a dedicated co-processor, e.g. one or more systolic arrays, and corresponding instructions, in such a way that the instructions can be executed in the co-processors, again in one clock cycle, see [2, 3]. Instead of adding several co-processors, one can also include a single

configurable co-processor, which can be configured before the corresponding instruction is to be executed. This option is clearly an extension to the first option and can yield faster execution of (part of) the procedure. This goes at the expense of more hardware, although the reduction of power dissipation that goes with it may pay off handsomely.

The instructions and hardware added to a GPP are usually chosen to accelerate certain routines that are pertinent to a particular application domain, such as multimedia, image processing, and array signal processing. In other words, the GPP is customized to that application domain.

A third option is to go (almost) fully dedicated. In contrast to the (modified) GPP and GPP plus co-processor approach, the dedicated architecture solution is not flexible anymore and can execute at most a few (similar) signal processing procedures. This solution is very costly, especially when a set of signal processing procedures constitute the application, since it becomes necessary to provide an implementation for each and every procedure in the application. As a result there is (almost) no reuse of hardware and therefore this solution is far from efficient.

A fourth solution is to go away from a sequential execution of instructions or even from instruction level parallelism (ILP), however powerful (some of) the instructions may be – as in the first two options – and opt for task level parallelism (TLP) instead. As the name suggests, TLP is aiming at exploiting concurrency at the level of tasks, that is, of large portions of (sub) procedures. This is in contrast to ILP, in which only the independence between instructions in a small instruction window is exploited. Exploiting TLP also needs an architecture that is different from the instruction set architecture (ISA) or VLIW architectures. The TLP architecture is composed of a micro processor, some memory, and a number of processing units (PUs) that are linked together via some kind of interconnection network. Examples of these new architectures are the *Prophid* architecture [4], The *Jacobium* architecture [5], and the *Pleiades* architecture [6], intended for video consumer appliances, adaptive array signal processing, and wireless mobile communications, respectively. These architectures have in common that they exploit TLP over the PUs and ILP within the micro processor and possibly the PUs. The problem is then how to map signal processing procedures of an application from an application domain onto a TLP architecture instance.

As already mentioned before, the signal processing procedures are commonly specified in terms of nested loop programs (NLP), because they form a compact description, and match the shared memory GPP architectures on which algorithm developers usually develop the specifications. However, such specifications do not match the TLP architectures: mapping NLPs onto TLP architectures is (almost) impossible. A procedure specification that better matches the TLP architecture is a 'process network' specification, that is, a network of communicating processes or tasks. However, converting ISA programs to process network programs is also a complex task.

In this dissertation I show that for a subclass of NLPs – the so-called *piece-wise affine nested loop programs* – the conversion to process network programs can be automated. The resulting concurrent tasks specification can then be more easily mapped onto a TLP architecture, e.g., as elaborated upon in [7–9].

This chapter is further organized as follows. First, Section 1.1 provides the motivation of this work by stating what the actual problem is that we want to solve. Then, knowing what the problems is, Section 1.2 briefly sketches the approach I have taken to solve the problem. Section 1.3 gives a brief overview of related work and Section 1.4 summarizes my contributions laid down in this dissertation. Finally, Section 1.5 describes the organization of this dissertation.

1.1 Problem statement

As I have noticed above, it is difficult – if at all possible – to map an application specified in an imperative language into an architecture structure that does not resemble the classical shared-memory single-processor-unit structure of the instruction set architecture. In an ISA, whether single threaded, multi threaded, or even instruction level parallel, instructions are executed one at a time, at best in a pipelined fashion, and generally no faster that one instruction per clock cycle. For example, the application specification given in Figure 1.1(a) may at best execute in $(\frac{1}{2}(N-1)Nn_R + Nn_V)K$ clock cycles on an ISA, if n_R and n_V

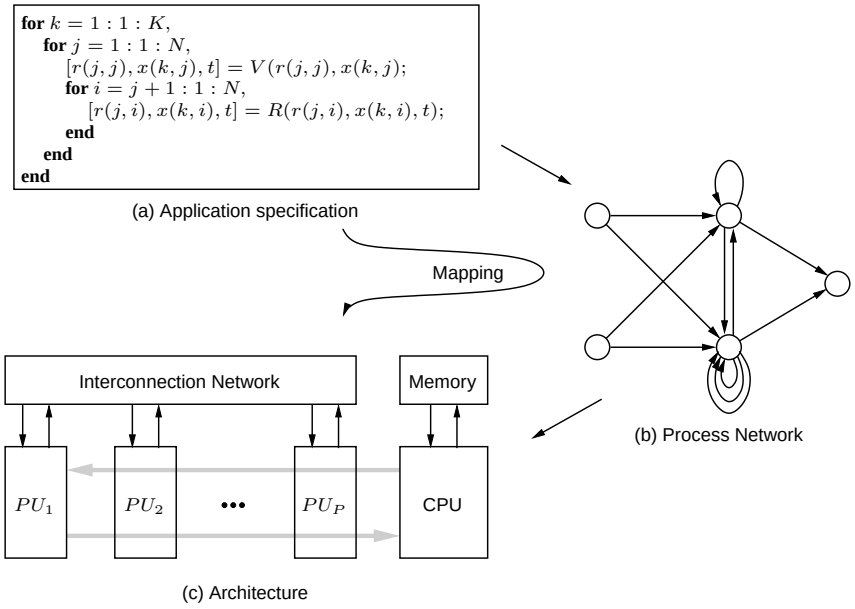


Figure 1.1: Mapping the given application specification onto a TLP architecture is difficult because the specification does not match the organization of the architecture.

are the number of clock cycles needed to execute the – not specified – functions $R()$ and $V()$, respectively. Thus, assuming that $n_R = n_V = 12$ clock cycles, and $N = 8$, the ISA can sustain a throughput of approximately one x-vector (one k iteration) per 432 clock cycles. Now, consider a task level parallel architecture organization as shown in figure 1.1(c).

This architecture consists of a classical shared memory (ILP) ISA and a number of – not necessary identical – processor units embedded in a communication network and having low-bandwidth interaction means with the (ILP) ISA.

The PUs typically execute tasks, such as, for example (sets of) functions like the functions $V()$ and $R()$. Assuming, again, that $n_R = n_V = 12$ clock cycles latency and one clock cycle throughput, and $N = 8$, this architecture could sustain a throughput of three x-vectors per 36 clock cycles, with only three PUs, $P = 3$, in the network [10]. Here I assumed that communication and (distributed) memory access is overlapped with computation. Thus the processing of one x-vector on the TLP architecture shown in Figure 1.1(c) takes only 3% of the time the ISA would take.

Given this enormous difference in throughput, it is natural that applications that require high throughput be mapped on TLP architectures, not on ISAs. The problem, however, is that an application specification as given in Figure 1.1(a) can be easily mapped on an ISA, not on a TLP architecture. This is so, because the imperative model of computation (MoC) underlying the application specification does not match the TLP architecture organization.

The mapping onto the TLP architecture would be much easier when the underlying MoC would be a Process Network (PN) model, a model where autonomous tasks communicate with each other via some kind of channel. A (symbolic) Process Network model specification for the imperative program in Figure 1.1(a) is shown in Figure 1.1 (b). It consists of two source processes, one sink process, and two processes that execute functions $V()$ and $R()$, respectively. The arrows are communication channels, which in case the Process Network is a Kahn Process Network [11] are unbounded FIFO channels with a blocking read synchronization, see Chapter 4. Although the mapping of the Process Network onto the TLP architecture is no a trivial task, it will be clear that it must be easier than mapping the imperative program onto that architecture because there is now a clear separation between computation (processes) and communication (channels) as is the case with the TLP architecture: PUs and communication network. The question of how the processes themselves are specified will depend on the type of PU on which they are mapped. The obvious conclusion of the above reasoning is this one: rely on the PN MoC to specify the application and not on the imperative MoC. Correct this conclusion may be, there are two new obstacles that arise here. Firstly, application developers do invariantly specify their applications in an imperative language. The have been doing so in the past and they will continue to do so. Secondly, even if an application developer would be willing to specify his or her application in a PN MoC, that specification would just be one of the many possible Process Networks that have the same behavior, and would most likely not be constructed with a mapping on a TLP architecture in mid as the application developer is only interested in behavior, not in performance.

So it seems that we are stuck: we have to accept that applications are specified as imperative programs that we cannot map onto TLP architectures. The only remaining route is to translate imperative programs to process networks. Now, if mapping of imperative programs to TLP architectures is difficult, and if mapping of Process Networks onto TLP architectures is easier, why would translating imperative programs to Process Networks be doable to begin with? The answer is that it is not, in general (except in an ad hoc way that may provide useless networks), but that it is possible in case the imperative program a so-called piece-wise affine nested loop program [12] (see Chapter 3), as is the one given in Figure 1.1(a). The reason why a translation is possible for these programs is that Process Networks are appropriately defined, then it turns out that systolic arrays are special cases of Process Networks, and because the relation between piece-wise affine nested loop programs and systolic arrays is well understood, it must be possible to translate these NLPs directly to Process Networks.

Thus, the problem dealt with in this dissertation is the translation of application kernels specified as a piece-wise affine nested loop programs into process networks, in particular Kahn Process Networks. This section introduces the three step approach that is further worked out in the dissertation.

Section 1.2 sketches the approach that I have taken. Before doing so, I shall first give in more precise terms the differences between the imperative model and the Process Network model. This will allow the reader to appreciate the translation problem.

On the one hand there is the imperative specification of the application, and thus also of the kernel. On the other hand there is the Kahn process network model to which the imperative specification of the kernel must be translated. The problem of translating a specification in the imperative model to a specification in the KPN model is rooted in two fundamental differences between these models, viz., the differences in terms of *assignments* and *communication*. In imperative languages, *assignment* refers to the assignment of a value to a variable, while in KPNs it refers to the assignment of a value to a token. In imperative languages, *communication* refers to the assignment of values to variables and the referencing to them, whereas in KPNs it refers to the writing and reading of tokens to and from channels.

Imperative languages have so called *destructive assignments*. This means that the value assigned to a variable may change during the execution of a program. Because of this, changing the execution order of

the statements might change the functional behavior of the program, and therefore, the programmer must specify the execution order. For NLPs, this means that the **for** loops and the nesting of statements within these loops specify the intended execution order. A program written in an imperative language specifies *what* is to be done by this program and *how* this is to be done.

This is in contrast with declarative languages which have so called *non-destructive assignments*. This means that during the execution of a program, to each variable a value is assigned only once. Because of this the execution order of the statements doesn't matter as long as the variables that are referenced to have a value assigned to them. A program written in a declarative language specifies *what* is to be done by this program, not *how* this is to be done.

In the terminology just defined, communication in both imperative and declarative languages is done via shared memory: there is a shared space that contains the values (most recently) assigned to the variables. The order in which one wants to refer to these values is independent of the order in which they were assigned.

In Kahn process networks values are assigned to tokens. This assignment happens when a value that exists inside a process is written onto a channel. As part of the write operation the token to which the value is assigned is created and consequently, token assignments are non-destructive . There is no way to assign a value to a token more than once. The token is written on the channel from which it can be read some other time. Since channels are unbounded FIFO queues, the order in which tokens are written on the channel is the same as the order in which the tokens are read from that channel.

The differences between the shared memory and KPN models is given in Table 1.1.

	assignments	**communication**
imperative language	destructive	via shared memory
declarative language	non-destructive	via shared memory
Kahn process network	non-destructive	via unbounded queues

Table 1.1: Differences and similarities between three programming models with respect to assignments and communication.

1.2 Approach

The problem dealt with in this dissertation is the translation of piece-wise affine nested loop programs to Kahn process networks. In order to do so this translation is carried out in three steps which are introduced in this section and which are described in detail in the chapters that follow.

The arrow from the nested loop program in Figure 1.1(a) to the Kahn process network in Figure 1.1(b) hides an intermediate model, called the *polyhedral reduced dependence graph* (PRDG) model and shown in Figure 1.2(b). The reason that the PRDG model has been introduced is twofold: 1) it partitions the conversion from the NLP to the KPN into two tractable problems, as shall be seen soon, and 2) it is based on models for which a rich set of transformations and methods have been developed in the past, notably in the areas of array synthesis and automatic parallelization. However, this dissertation focuses on the first point.

The conversion of a piece-wise affine NLP to a KPN is carried out in three steps represented by the tools MATPARSER, DGPARSER, and PANDA, as shown in Figure 1.3. In this Figure, a box represents a result and an ellipsoid represents a method or tool. The set of tools is called the *Compaan tool set* or *Compaan* for short.

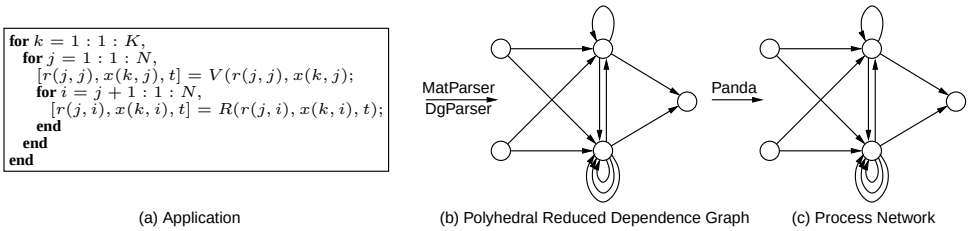


Figure 1.2: Translation of NLPs to polyhedral reduced dependence graphs to KPNs.

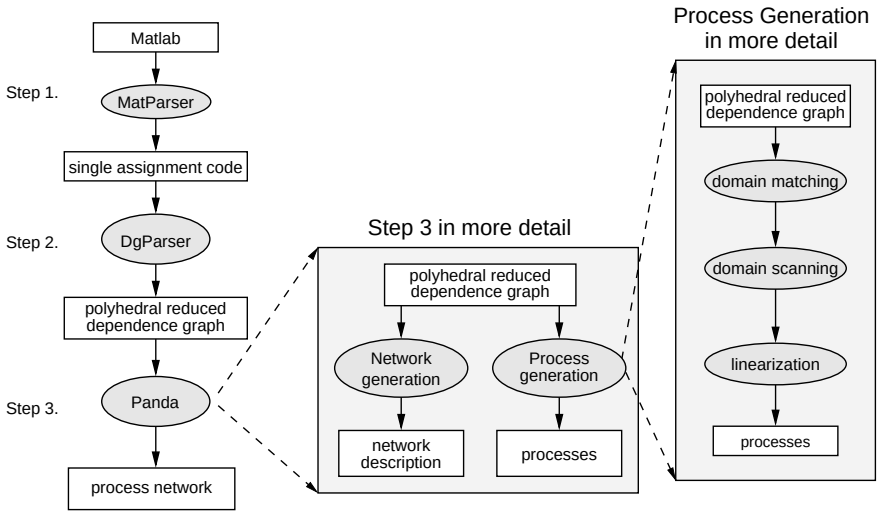


Figure 1.3: Compaan consists of three tools that translate a Matlab specification into a process network specification.

The first step is the translation of an imperative nested loop program specification into a *single-assignment program* (SAP) specification. As specification language we selected Matlab and this translation is done by the tool MATPARSER [13]. In a SAP, to every variable a value is assigned only once. This means that this step translates a NLP program with destructive assignments in a program with non-destructive assignments. Consequently this step results in a declarative version of the kernel, see Table 1.1. Although the program is declarative, it uses exactly the same nested loop structure as the original program. This allows the declarative program to be interpreted as an imperative program such that it can be executed, and thus validated, in the same programming environment as the original specification, in my case Matlab.

The second step is to derive the *polyhedral reduced dependence graph* (PRDG) specification from the SAP. This step is performed by the tool DGPARSER. In the SAP some loops may depend on other loops. As the name suggests, a PRDG is a graph. The nodes and edges a PRDG represent the computations and communications (data dependencies) from the SAP the PRDG derived from. Rather that introducing a node for each iteration in the SAP, a node is introduced for a regular set of iterations in the SAP. This regular set of iterations is defined in terms of polyhedra. A *polyhedron* is a convex set defined by a finite set of inequalities and is a useful mathematical representation that is used in the rest of this dissertation. Similarly, an edge in the PRDG represent a regular set of data dependencies. A PRDG thus represent an application in terms of topology (it is a graph), behavior (it models the computations), and geometry (regular sets of iterations are modeled in terms of polytopes. The PRDG model and constructing instances of the model from SAPs is described in detail in Section 3. The explicit separation between communication and computation and the geometrical characterization simplifies the translation from communication via shared memory to communication via unbounded FIFO queues. In addition, the geometrical characterization makes this model useful for transformations such as index (dependency) transformation, loop reordering,

and partitioning of computational domains.

The third step is to convert the PRDG into a Kahn network description and the individual Kahn processes. This step is performed by the tool PANDA. The topological characterization of the PRDG is used to derive the network description of the KPN. The generation of the individual processes is further decomposed in three sub-steps: *domain matching*, *domain scanning*, and *linearization*, shown in Figure 1.3 and described in detail in Section 4. Two computational domains that communicate with each other not necessary communicate for all points in those domains. Domain matching takes care that data producers only write the tokens onto the channels that are eventually consumed by a consumer. When a PRDG is translated to a KPN, the domains in the PRDG are translated to sets of nested loops. This translation is called domain scanning. When such a sequential execution of a domain is not desired, the domain should be first partitioned into sets of domains. However, the sequential execution might be actually be a parallel execution in case the processing unit the on which the operations are executed are are pipelined. Linearization of data references is the main challenge in the translation, and addresses the FIFO behavior of the channel. This means that the higher-dimensional data structures that appear in the original specification must be scheduled over the one-dimensional communication channel such that the FIFO behavior of those channels is respected.

1.3 Related work

This dissertation directly relates to two main research areas. The first one is the synthesis of regular processor arrays. The second one is the automatic parallelization of regular nested loop programs. The latter area, however, relies heavily on methods that were developed in the former area.

Array synthesis
A systolic array is a set of simple processors that are interconnected in a regular manner. This concept was first introduced by H. Kung [14] where he showed how some typical matrix computation algorithms can be executed efficiently onto such an architecture. In the context of systems of difference equations, Karp et al. [15] studied uniform recurrence equations (URE) and systems of UREs (SURE). They also introduced the reduced dependence graph (RDG), a formal graph based description of the (S)URE whose size only depends on the structure of the computations, not the actual number of computations. Based on the RDG their focus was on the computability of (S)UREs. It was recognized by Moldovan [16] and Quinton [17] that specifying algorithms by means of (S)UREs opens the doors to the automatic mapping of applications onto systolic arrays (or the automatic generation of systolic arrays from applications). In order to improve the mapping/generation process, transformations on SURE models have been studied extensively by many authors [18–21].

Since then, a lot of research has been conducted on this topic, leading to a number of generalizations to the (S)URE model and the methods applied to it: The indexing of variables (multi-dimensional arrays) has been generalized to (systems of) affine recurrence equations, and parameterized models have been introduced. Recent work that relates to the work in this dissertation, was done in the context of the CATHEDRAL IV [22], ALPHA [23], PHIDEO [24], and HIFI [25] projects.

Within the context of the CATHEDRAL IV project, Swaaij [22] has studied the synthesis of systolic arrays from the applicative language *Silage*. In [22], Silage programs are first translated into a model called *polyhedral dependence graph* (PDG), i.e., a graph in which nodes are associated with *conditional affine recurrence equations* defined on a polyhedral domain. The PDG model is more general that our PRDG model in that there are less restrictions on specifying the computational domains which makes this model difficult to deal with. However, the PRDG model is capable to capture more regularity of the application. The main difference is that the work in [22] focuses on transformations to optimize the control flow of the application, whereas this dissertation focuses on the transformation from the PRDG model to the KPN

model.

In the ALPHA environment [23], a program is described as a system of affine recurrence equations (SARE). Starting from such a specification both the synthesis of regular architectures and the compilation to sequential or parallel machines is considered. The rationale behind writing programs in ALPHA rather than in some imperative language is that a functional/mathematical specification matches the way people think of an algorithm and that all parallelism in the algorithm is naturally preserved. The domains associated with the SARE are $\mathbb{Z}$-polyhedra [26], see also Section 2.5. The PRDG model that this dissertation presents is more general that the SARE in that the recursions in the PRDG model allows for multiple left hand side variables in assignment statements and that the index sets are more general than $\mathbb{Z}$-polyhedra, see also Section 2.5.

In [27] Thiele and in [28] Teich and Thiele describe the systematic design of processor arrays. In order to do so they propose to map piece-wise regular (linear) algorithms onto processor arrays. Such an algorithm is described by a set of recurrence equations in which the indexing functions are simple translations (affine functions) of the points in a linearly bounded lattice (LBL) [29], see also Section 2.5. In [30] Thiele introduces *piece-wise regular dependence graphs* and *reduced piece-wise regular dependence graphs* to represent (what was later called) *piece-wise regular programs*. The sets of iterations of these programs are defined by LBLs. These LBLs are associated with the nodes of the reduced piece-wise regular dependence graphs [29]. Also in [29] the class of piece-wise regular programs is extended to *piece-wise linear programs* that allows for affine indexing functions rather that constant. In this dissertation has I propose a model in which the index sets are defined by a stricter means than LBLs.

The high-level synthesis methodology PHIDEO [24] starts with a specification in the single assignment form, and converts this description into an instance of an target architecture template. An important part of PHIDEO is the address generation for memories that are introduced by the synthesis tools. The address generation method used in PHIDEO can be viewed as a special case of the address generation method presented in this dissertation, see Section 4.5. This is due to somewhat more restricted geometry of the iteration domains used in PHIDEO.

The HIFI project [25] focused on a methodological approach of designing regular processor arrays. In particular, the work of Held [12] focuses on the functional design of dataflow networks. Starting with a MATLAB program (piece-wise affine NLP), an exact dependence analysis, based on work by Feautrier [31], is performed to yield a single-assignment program. The *piece-wise regular dependence graph* of this program is modeled by a hierarchical graph. Every node represents a regular piece of the application and is converted back into single-assignment code (MATLAB again). These programs are each converted into the *applicative state transition* [32, 33] model. The main difference between the work in [12] and the work in this dissertation is that in [12], the problem of communication via one dimensional communication channels is not considered.

Although the above work relates more or less to the work presented in this dissertation, the main difference is that the above work relates to (array) synthesis whereas the work in this dissertation relates to compilation and does not have processor arrays as target architectures.

Automatic parallelization
Automatic parallelization mainly focuses on the generation of code for multi-computer and multi processor systems.

Two programming paradigms are found in programming parallel machines: *message passing* and *data parallelism*. In the message passing type of programming, the program explicitly contains the communication between the processors. The two most popular message passing libraries are the message passing interface (MPI) [34] and the parallel virtual machine (PVM). For a comparison between the two see [35].

Message-passing programs are very well suited for multiple-instruction, multiple-data architectures. Often, the single-program multiple-data (SPMD) model is used (as in MPI and PVM), in which each and every processor in the architecture runs the same program. In the data-parallel type of programming, low-level communication details are left to the compiler, as, for example, in compilers for high performance Fortran (HPF), see for example [36–38], and therefore this programming model eases the development and maintenance of parallel programs considerably compared to message passing programs. Data parallel programs are usually compiled for single-instruction, multiple-data (SIMD) architectures.

Most work in automatic parallelization focuses on the derivation of programs in the data-parallel programming model. Some approaches translate (mainly HPF) data parallel code into SPMD code. Within the SUIF project [39], the focus is on parallelization and optimization of sequential programs for share- memory multi-processor architectures by means of loop transformation methods [40]. In the LOOPO project [41] they develop a prototype implementation of parallelization methods, and produce data parallel code in HPF. As a back-end to this generated HPF code the Adaptor compiler, developed in the ADAPTOR project [42], can be used to produce a SPMD program, mainly for the use with MPI and PVM. In the PRISM/SCPDP project [43] the focus is also on generating data parallel programs, but current research goes also in the direction of code generation for heterogeneous systems in the SPMD model. In the PIPS Workbench project [44], data parallel code is generated by automatic parallelization. On top of PIPS the HPF compiler, HPFC, is implemented to generate SPMD code for use with PVM. In PIPS there is also a tool WP65, that generates code for distributed memory architectures. In the BOUCLETTES project [45] perfectly nested loops are transformed into data parallel HPF programs.

Wilde [46] describes an approach to compile Alpha programs [23] into imperative programs and data-parallel imperative programs. The construction of efficient loop nests is, for example, dealt with in [47,48], however, generation of the communication code between tasks is not dealt with. This thesis proposes to use similar techniques, but my focus is the generation of communication code between concurrently operating tasks.

In the PCO (program compilation & optimization) project [49], work is done is to improve locality in regular nested loop programs to exploit computer systems' memory hierarchy. Two methods to improve locality, one for temporal locality and one for spatial locality are presented in [50] and [51], respectively. The relation with the work in this dissertation is the use of counting methods in parameterized convex sets.

The work in this dissertation relates to the data-parallel and message-passing paradigms in the following sense. The original program is specified as a nested loop program that is translated into a single-assignment program. The SAP can be interpreted as a parallel program in which the functions communicate with each other via shared memory. Then the SAP is converted into our PRDG model which is then converted into a Kahn process network. The processes in this network communicate with each other by writing tokens to and reading tokens from channels. The tokens may be seen as messages. However, the derived Kahn process network makes no assumptions on how the communication is eventually implemented on a target architecture. One may map the channel onto shared memory, by using C-HEAP [52] for example, or onto dedicated buffers that are connected via a network-on-chip [53].

1.4 Contributions

The work presented in this dissertation focuses on the derivation of Kahn process networks from imperative code; both methods as well as the implementation of these methods are presented. The main contributions are

- **derivation of methods to translate parameterized piece-wise affine NLPs to KPNs**
  Literature reports many design flows that start from a KPN (or some other process or data flow model) as the initial application specification for the purpose of high level synthesis or design space exploration [9, 54–58]. This dissertation presents an approach to derive such an initial specification from a parameterized piece-wise affine NLP. The case that such an NLP contains broadcasts is mentioned but not solved.

- **introduction of the intermediate PRDG model**
  Several models are presented in the context of regular array design and automatic parallelization. These models have expressiveness that matches the restrictions imposed on the specification of the nested loop program. For example, the URE model (defined on a polyhedral index set) is well suited for modeling perfectly nested loop programs, whereas the SARE model is better suited for modeling sets of non-perfectly nested loop programs. In this dissertation, the class of nested loop programs dealt with is the class of piece-wise affine NLPs with pseudo-affine control statements, see Section 3.3. These statements allow the specification of iteration domains that are defined by periodic patterns of integer points, bounded by polytopes. Two well known models are the $\mathbb{Z}$-polyhedron model [26] and the linearly bounded lattice (LBL) model [28]. The nature of the pattern of integer points that I consider is such that the $\mathbb{Z}$-polyhedron model is not generic enough. However, the LBL model is more general than required. Therefore, the PRDG model that is introduced in this dissertation is based on a restricted form of the linearly bounded lattice (LBL) model. Moreover, as opposed to the systems of recurrence equations that are found in literature, the PRDG model allows to model vector functions, i.e., functions that have multiple return values. In addition, in order to capture as much regularity as possible, the PRDG model is communication oriented rather that computation oriented; the iteration domains of the input arguments and return values of a set of indexed functions are specified independently from each other.

- **partitioning of the translation from PRDG to KPN in three well-defined problems and deriving solutions to these problems**
  The problem of converting the PRDG to a KPN is defined and is partitioned in three well defined sub-problems shown in Figure 1.3; domain matching, domain scanning, and linearization. This dissertation presents a solution to each of these sub-problems in sections 4.3–4.5. Many of the problems are formulated in terms of a counting problem, which is addressed using Ehrhart polynomials.

- **implementation of the presented methods in software versions**
  Many methods presented in the literature are not verified by an implementation of the method. All methods presented in this dissertation have a corresponding software version. Moreover, to enable and stimulate ongoing research, the algebra and methods in this dissertation are carefully formulated. This formulation is such that the structure of theoretical concepts matches software implementation. As a result there is a clear correspondence between the software versions on one hand and the algebra and these methods on the other hand. Also modern software engineering techniques are applied to have the software well organized.

1.5 Outline

This dissertation is organized as follows. Chapter 2 introduces the mathematical background that is used in the rest of this dissertation. The chapter deals with combinatorial geometry and deals with parameterized polytopes and how they can be used to define domains, special sets of integer vectors. Also, great attention is paid to the counting of the number of integral points inside these polytopes by means of Ehrhart polynomials. Chapter 3 deals with the class of NLPs and defines the polyhedral reduced dependence graph

(PRDG). Then the conversion from NLPs to their single-assignment program (SAP) and the conversion from SAPs to their PRDG is described. Chapter 4 deals with the conversion of the PRDG to Kahn process networks. The decomposition of this problem into the three sub-problems and the solution to these problems is presented. The methods in this chapter heavily rely on the counting methods that are presented in Chapter 2. Chapter 5 deals with the organization of the software versions of these methods and data structures in chapters 2–4. The chapter also serves as a "putting it all together" and gives a global overview of the chapters 3 and 4.

Chapter 2

Basics

As far as required for the next chapters, this chapter contains basic material from the theory of integer linear algebra. Aside an introduction of notations and a limited number of elementary notions this chapter deals with polytopes and operations on them, especially counting the number of points in parameterized polytopes. Also attention is paid to sets of regular spaced integral points.

This chapter is further organized as follows. Section 2.1 gives some preliminary definitions used in the other sections of this chapter. Section 2.2 introduces polyhedra and polytopes in both implicit and dual form. Since my goal is to deal with parameterized applications these definitions are then extended to include parameterized versions. Parameterized polyhedra, polytopes, and their vertices are dealt with in Section 2.3. Later I will be interested in counting the number of integer points contained by a polytope. These numbers are expressed by Ehrhart polynomials which is the topic of Section 2.4. Finally special sets of integral points are introduced in Section 2.5. Much of the material used in this section is from [59–61].

2.1 Preliminaries

This section gives some notations and definitions that are used throughout this chapter and the rest of this dissertation. All vectors in this dissertation are column vectors. A *vector* $\mathbf{x}$ of dimension d is denoted as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

or alternatively as $\mathbf{x} = (x_1, x_2, \cdots, x_d)$, where $x_i, i = 1, 2, \cdots, d$ are elements of some set S. A row vector is indicated by putting a superscript T to a column vector, that is, if $\mathbf{x}$ is a column vector, then $\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_d \end{bmatrix}$ is a row vector. A *point* is a column vector that has scalar entries from the sets $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$.

Definition 2.1 (combination)
Let $\mathbf{x} = (x_1, x_2, \cdots, x_d)$ be a vector and let $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \cdots, \lambda_d)$ be a vector of coefficients. Four different combinations of the form $\sum_{i=1}^{d} \lambda_i x_i$ can be identified:

- $\sum_{i=1}^{d} \lambda_i x_i$ is called a linear combination

- $\sum_{i=1}^{d} \lambda_i x_i$ is called a positive combination when all $\lambda_i \geq 0$
- $\sum_{i=1}^{d} \lambda_i x_i$ is called an affine combination when $\sum_{i=1}^{d} \lambda_i = 1$
- $\sum_{i=1}^{d} \lambda_i x_i$ is called a convex combination when $\lambda_i \geq 0$ and $\sum_{i=1}^{d} \lambda_i = 1$ ■

Definition 2.2 (hyper-plane and closed half-space)
Let a non-zero vector $\mathbf{a} = (a_1, a_2, \cdots, a_d)$ and a scalar b be given. Two types of sets are defined in terms of $\mathbf{a}$ and b.

- $\mathcal{H} = \{\mathbf{x} \in \mathbb{Q}^d \,|\, \mathbf{a}^T\mathbf{x} = b\}$ is called a hyper-plane
- $\mathcal{H} = \{\mathbf{x} \in \mathbb{Q}^d \,|\, \mathbf{a}^T\mathbf{x} \geq b\}$ is called a closed half-space ■

Note that a hyper plane itself is the intersection of two closed half-spaces, i.e., $\{\mathbf{x} \,|\, \mathbf{a}^T\mathbf{x} = b\} = \{\mathbf{x} \,|\, \mathbf{a}^T\mathbf{x} \geq b \ \wedge \ \mathbf{a}^T\mathbf{x} \leq b\}$.

Definition 2.3 (vertex)
Let a set $\mathcal{S}$ be given. A *vertex* of $\mathcal{S}$ is a point in $\mathcal{S}$ that cannot be expressed as a convex combination of any other points in $\mathcal{S}$. The set of vertices of a set is called its *vertex set*. ■

Definition 2.4 (line)
Let $\mathcal{S}$ be a set in $\mathbb{Q}^d$. A *line* of $\mathcal{S}$ is a vector $\boldsymbol{\ell}$ that has the property that for any $\mathbf{s} \in \mathcal{S}$ for all $\mu \in \mathbb{Q}$ it is true that $\mathbf{s} + \mu\boldsymbol{\ell} \in \mathcal{S}$. ■

Definition 2.5 (linear (affine) independence)
Let $\mathcal{S}$ be a set of points. The points in S are said to be linearly (affinely) independent if and only if no point in $\mathcal{S}$ is a linear (affine) combination of other points in S. When a set of points are not linearly (affinely) independent these points are said to be linearly (affinely) *dependent*. ■

Definition 2.6 (linear (affine) basis)
Let $\mathcal{S}$ be a set of points. A basis of $\mathcal{S}$ is a subset of $\mathcal{S}$ that has the following two properties: 1) the points in the subset are linearly (affinely) independent and 2) every point in $\mathcal{S}$ is a linear (affine) combination of the points in the subset. The *rank* of a set is the cardinality of its linear basis. ■

Definition 2.7 (affine subspace)
Let $\{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_n\}$ be any set of affinely independent vectors in $\mathbb{Q}^d$. An *affine subspace* of $\mathbb{Q}^d$ is the set of all affine combinations of the vectors $\mathbf{b}_i$, $i = 1, 2, \cdots n$. The dimension of an affine subspace is defined as the rank of set of lines that spans the affine subspace. Thus with the set of n vectors above, the dimension of the affine subspace is $n - 1$. ■

Definition 2.8 (affine span)
Let $\mathcal{S}$ be a set in $\mathbb{Q}^d$. The *affine span*, also called *affine hull*, of $\mathcal{S}$ is the smallest dimensional affine subspace that entirely contains $\mathcal{S}$. ■

2.2 Polyhedra and polytopes

This section introduces polyhedra and polytopes. A polyhedron is a convex subset of some space in some dimension, say n. In the literature, this space is either $\mathbb{R}^n$ or $\mathbb{Q}^n$. To be compatible with the theory of Ehrhart polynomials (see Section 2.4) I define polyhedra in the space $\mathbb{Q}^n$.

Definition 2.9 (polyhedron)
A polyhedron $\mathcal{P}$ is the intersection of a set of finitely many closed half-spaces, i.e.,

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} = \mathbf{b} \wedge C\mathbf{x} \geq \mathbf{d}\} \tag{2.1}$$

where A is an integral $k \times n$ matrix, $\mathbf{b}$ is an integral vector of size k, C is an integral $\ell \times n$ matrix, $\mathbf{d}$ is an integral vector of size ℓ, and k and ℓ are the number of equalities and inequalities, respectively. The dimension of a polyhedron is the dimension of its affine span. A k-dimensional polyhedron is called a k-polyhedron. The *interior* of a polyhedron $\mathcal{P}$ is the set of points in $\mathcal{P}$ for which no of the constraints are satisfied with equality. The *boundary* of a polyhedron $\mathcal{P}$ is the set of points in $\mathcal{P}$ that are not in the interior of $\mathcal{P}$. ■

Note that when a polyhedron lies in an affine subspace with a dimension lower than the dimension of the space that contains the polyhedron then this polyhedron has an empty interior.

Definition 2.10 (supporting hyper-plane)
Let $\mathcal{P}$ be a polyhedron and $\mathcal{H}$ be a hyper-plane. $\mathcal{H}$ is called a supporting hyper-plane of $\mathcal{P}$ when it intersects the polyhedron of $\mathcal{P}$ but not its interior. ■

Definition 2.11 (face)
Let $\mathcal{P}$ be an n-polyhedron and $\mathcal{H}$ be a supporting hyper-plane of $\mathcal{P}$. The intersection $\mathcal{P} \cap \mathcal{H}$ defines a *face* of $\mathcal{P}$. Note that the faces of a polyhedron are polyhedra themselves. The dimension of a face is the dimension of its affine span. A face of dimension d is called a d-face. Faces of dimension 0, 1, and $n-1$ are a called a vertices, edges, and *facets* respectively. ■

Definition 2.12 (polytope)
A bounded polyhedron is called a *polytope*. A k-dimensional polytope is called a k-polytope. ■

Since polytopes are polyhedra, the definitions for supporting hyper-planes and faces also hold for polytopes.

In the above definition, a polytope is defined as the intersection of finitely many closed half spaces. There is however an alternative representation in terms of a convex combination of vertices.

Definition 2.13 (geometric representation)
A polytope $\mathcal{P}$ has a dual geometric representation as a convex combination of vertices:

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{x} = V\boldsymbol{\lambda} \wedge \lambda_i \geq 0 \wedge \sum_{i=1}^{\ell} \lambda_i = 1\} \tag{2.2}$$

where the columns of matrix V are the elements of the vertex set of $\mathcal{P}$ and $\boldsymbol{\lambda}$ is a vector with elements $\lambda_i, 1 \leq i \leq \ell$.

There exist methods that convert polytopes from their implicit representation to their dual geometric representation and vice versa [61–63].

Figure 2.1 shows the two representations of a polytope. On the left hand side is a polytope in its implicit representation, on the right hand side is its dual geometric representation. ■

The *denominator* of a vertex is the least common multiple of the denominators of its coordinates. The *denominator* of a polytope is the least common multiple of the denominators of its vertices.

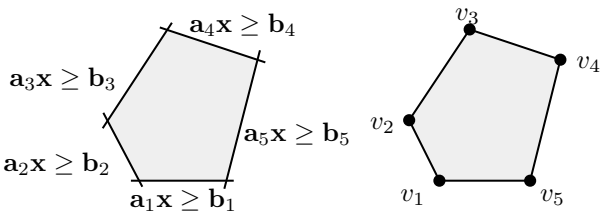


Figure 2.1: Implicit and dual geometric representation of a polytope.

2.3 Parameterized polyhedra and polytopes

This section introduces parameterized polyhedra and parameterized polytopes. The vertices of a parameterized polytope are parameterized themselves and are described next.

2.3.1 Parameterized polyhedra

Often polyhedra depend on a vector of parameters. I shall use the symbol $\mathbf{p}$ for this parameter vector.

Definition 2.14 (parameterized polyhedron)
A parameterized polyhedron $\mathcal{P}(\mathbf{p})$ is a polyhedron whose defining closed half-spaces are affinely dependent on a vector $\mathbf{p} \in \mathbb{Q}^m$ of parameters, as follows:

$$\mathcal{P}(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} = B\mathbf{p} + \mathbf{b} \wedge C\mathbf{x} \geq D\mathbf{p} + \mathbf{d}\} \tag{2.3}$$

where B and D are an integral $k \times m$ and a $\ell \times m$ matrices, respectively, and where A, C, $\mathbf{b}$, $\mathbf{d}$, k, and ℓ are as in (2.1). ■

Alternatively a parameterized polyhedron $\mathcal{P}(\mathbf{p})$ can be interpreted as a function from the parameter space $\mathbb{Q}^m$ to the space of all subsets of $\mathbb{Q}^n$. The domain for which this function is defined is called the *context* of the parameterized polyhedron $\mathcal{P}(\mathbf{p})$.

A parameterized polyhedron can be represented as a non-parameterized polyhedron by writing it in the combined data-parameter space as follows

$$\mathcal{P}' = \{\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} \in \mathbb{Q}^{n+m} \mid A' \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} = \mathbf{b} \wedge C' \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} \geq \mathbf{d}\} \tag{2.4}$$

where $A' = \begin{bmatrix}A & -B\end{bmatrix}$, and $C' = \begin{bmatrix}C & -D\end{bmatrix}$, with A, B, C, and D as in (2.3). Note that the parameter vector $\mathbf{p}$ is replaced by the free variable vector $\mathbf{y}$.

The parameterized polyhedron $\mathcal{P}(\mathbf{p})$ can be obtained from its non-parameterized representation by intersecting $\mathcal{P}'$ with the hyper-plane $\mathbf{y} = \mathbf{p}$ and projecting the intersection onto the space $\mathbb{Q}^n$, the space that contains $\mathcal{P}(\mathbf{p})$. Let $\mathcal{S}(\mathbf{p}) = \{\begin{bmatrix}\mathbf{x}\\ \mathbf{p}\end{bmatrix} \in \mathbb{Q}^{n+m}\}$

$$\begin{aligned}\mathcal{P}(\mathbf{p}) &= \mathrm{proj}_{\mathbb{Q}^n}(\mathcal{P}' \cap \mathcal{S}(\mathbf{p}))\\ &= \{\mathbf{x} \mid \begin{bmatrix}\mathbf{x}\\ \mathbf{p}\end{bmatrix} \in \mathcal{P}'\}\end{aligned} \tag{2.5}$$

where $\mathbf{p}$ is the parameter vector.

2.3.2 Parameterized vertices of polytopes

This section deals with the vertices of parameterized polytopes. The parameters in the parameter vector $\mathbf{p} = (p_1, p_2, \cdots, p_m)$ of a parameterized polytope also parameterize the vertices of this polytope. Hence these vertices are referred to as *parameterized vertices* and are denoted by $v_i(\mathbf{p})$ where the index i identifies the vertex. In line with the literature a parameterized vertex is confined to be an affine function of the parameters.

In general, a parameterized vertex $v_i(\mathbf{p})$ of a parameterized polytope is not defined for all possible values of the parameters but rather for a subset of all possible parameter values. The set of parameters for which a parameterized vertex is defined is called the *context of the vertex*.

Let me illustrate the notion of context of a vertex with the following example. Figure 2.2 shows the parameterized polytope $\mathcal{P}(p) = \{(x_1, x_2) \in \mathbb{Q}^2 \,|\, 0 \leq x_2 \leq 4 \wedge x_2 \leq x_1 \leq x_2 + 9 \wedge x_1 \leq p \wedge p \leq 40\}$, where the last constraint is the context of the parameterized polytope. The figure shows the eight parameterized

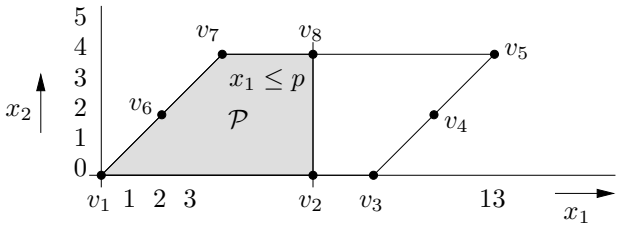


Figure 2.2: A parameterized polytope with its parameterized vertices.

vertices $v_1(p), v_2(p), \cdots, v_8(p)$ of $\mathcal{P}(p)$. Clearly, for the particular value of p shown in the figure, only v_1, v_2, v_7, and v_8 are the actual valid vertices of $\mathcal{P}(p)$. Thus, depending on the value of p, eight possible vertices may show-up.

The contexts of all the parameterized vertices of $\mathcal{P}(p)$ in Figure 2.2 are given by (2.6).

$$\begin{array}{lll}
v_1(p) = (0,0) & \text{when} & 0 \leq p \leq 40 \\
v_2(p) = (p,0) & \text{when} & 0 \leq p \leq 9 \\
v_3(p) = (9,0) & \text{when} & 9 \leq p \leq 40 \\
v_4(p) = (p,p-9) & \text{when} & 9 \leq p \leq 13 \\
v_5(p) = (13,4) & \text{when} & 13 \leq p \leq 40 \\
v_6(p) = (p,p) & \text{when} & 0 \leq p \leq 4 \\
v_7(p) = (4,4) & \text{when} & 4 \leq p \leq 40 \\
v_8(p) = (p,4) & \text{when} & 4 \leq p \leq 13
\end{array} \tag{2.6}$$

A method to find all parameterized vertices of a parameterized polytope and their corresponding contexts is presented in [64]. The method is based on the correspondence between the parameterized vertices of a parameterized polytope $\mathcal{P}(\mathbf{p})$ in m parameters and m-faces of $\mathcal{P}'$. It is shown in [64] that 1) for every parameterized vertex $v_i(\mathbf{p})$ in $\mathcal{P}(\mathbf{p})$ there is a corresponding m-face $\mathcal{F}_i^m$ in $\mathcal{P}'$ such that $v_i(\mathbf{p}) = \text{proj}_{\mathbb{Q}^n}(\mathcal{F}_i^m \cap \mathcal{S}(\mathbf{p}))$, where $\mathcal{S}(\mathbf{p})$ is defined as in (2.5) and 2) every vertex $v_i(\mathbf{p})$ is defined only for the projection of its corresponding m-face onto the parameter space via $\text{proj}_{\mathbb{Q}^m}(\mathcal{F}_i^m)$ which is, thus, the context of the parameterized vertex. Methods to compute the parameterized vertices $v_i(\mathbf{p})$ and their contexts is presented in [64] and implemented in the POLYLIB [65].

Section 2.4 deals with counting the number of integral points in parameterized polytopes. Since the theorem presented there uses the notion of affine-vertex polytope, this notion is first defined here.

Definition 2.15 (affine-vertex polytope)
Let $\mathcal{P}(\mathbf{p}) \subset \mathbb{Q}^n$ be a parameterized polytope with $\mathbf{p} = (p_1, p_2, \cdots, p_m)$ and with vertex set $\{v_i(\mathbf{p})\}_{i=1,2,\cdots,k}$. $\mathcal{P}(\mathbf{p})$ is said to be an affine-vertex polytope when every vertex in the vertex set has the form

$$v_i(\mathbf{p}) = M_i\mathbf{p} + \mathbf{m}_i\ ,\ \ i = 1, 2, \cdots k \tag{2.7}$$

where M_i is a rational n-by-m matrix and $\mathbf{m}_i$ is a rational n-vector and where all parameterized vertices are valid for the whole parameter range. Thus each coordinate of each vertex is an affine function of the parameters. ■

The common context of the vertices of an affine-vertex polytope is the context of this polytope itself.

A parameterized polytope $\mathcal{P}(\mathbf{p})$ can be described in terms of affine-vertex polytopes by partitioning[1] the context such that the restriction of $\mathcal{P}(\mathbf{p})$ to any partition is an affine-vertex polytope. For example take the polytope from Figure 2.2 with vertices given in (2.6). Four sets of the parameterized vertices are found, each one with its own context, the context of the affine-vertex polytope. They are given in (2.8).

$$\begin{array}{lll}
0 \le p \le 4 & : V_1(p) = \{v_1, v_2, v_6\} & = \{(0,0), (p,0), (p,p)\} \\
4 \le p \le 9 & : V_2(p) = \{v_1, v_2, v_7, v_8\} & = \{(0,0), (p,0), (4,4), (p,4)\} \\
9 \le p \le 13 & : V_3(p) = \{v_1, v_3, v_4, v_7, v_8\} & = \{(0,0), (9,0), (p,p-9), (4,4), (p,4)\} \\
13 \le p \le 40 & : V_4(p) = \{v_1, v_3, v_5, v_7\} & = \{(0,0), (9,0), (13,4), (4,4)\}
\end{array} \tag{2.8}$$

A method to partition the context is presented in [66]. Let $\mathcal{P}(p_1, p_2, \cdots, p_m)$ be a parameterized polytope. All m-faces of $\mathcal{P}'$ that corresponds to the parameterized vertices of $\mathcal{P}(\mathbf{p})$ are projected onto the m-dimensional parameter space $\mathbb{Q}^m$. These projections partition the parameter space, and consequently the context of $\mathcal{P}(\mathbf{p})$, and every such a partition is a context for which $\mathcal{P}(\mathbf{p})$ is an affine-vertex polytope.

Figure 2.3 shows the polytope of 2.2 in the combined data-parameter space. Since the dimension of the

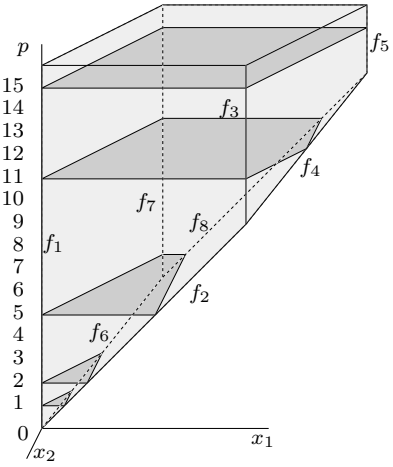


Figure 2.3: The parameterized polytope of Figure 2.2 in the combined data-parameter space together with its 1-faces that corresponds to the parameterized vertices of $\mathcal{P}(p)$. The intersections of the projection of these faces result in the contexts for which $\mathcal{P}(\mathbf{p})$ is an affine-vertex polytope.

parameter space is one, the 1-faces (which are the edges) of $\mathcal{P}'$ are projected onto $\mathbb{Q}^1$. Since projection of the four 1-faces at the top of the figure onto $\mathbb{Q}^1$ are single points, and thus of dimension 0, they do not correspond to parameterized vertices of $\mathcal{P}(p)$. The eight remaining 1-facets labeled $f_1, f_2, \cdots, f_8$ are

[1] The partitioning of the context is not a strict partitioning. The intersection of the partitions, however, are the intersections of the faces of the partitions.

projected onto the p-axis. Indeed the parameterized vertices and their contexts are shown in (2.6) and the intersections of the contexts of the vertices that form the contexts of the affine-vertex polytopes of $\mathcal{P}(\mathbf{p})$ are shown in (2.8). Note that the contexts of the affine-vertex polytopes partition the context of $\mathcal{P}(\mathbf{p})$ but not $\mathcal{P}(\mathbf{p})$ itself.

2.4 Ehrhart polynomials

In this section I consider a particular counting problem. It is the problem to count the number of integral points contained in a parameterized polytope. Thus if $\mathcal{P}(\mathbf{p})$ is a parameterized polytope in $\mathbb{Q}^n$, then we are interested in the number of points in the set $\mathcal{P}(\mathbf{p}) \cap \mathbb{Z}^n$. This number is referred to as the *enumerator* of $\mathcal{P}(\mathbf{p})$[2].

Clearly the enumerator of a non-parameterized polytope is a constant. Ehrhart [68], and Clauss and Loechner [69] have proved that the enumerator of a parameterized affine-vertex polytope $\mathcal{P}(\mathbf{p})$ can be expressed as a certain *pseudo-polynomial*.

I introduce pseudo-polynomials in Section 2.4.1. A pseudo-polynomial that is the enumerator of a parameterized affine-vertex polytope is called an *Ehrhart polynomial*. In Section 2.4.2 I elaborate on Ehrhart polynomials. Most of the material used in Section 2.4.1 and Section 2.4.2 is from [61, 66, 69, 70].

2.4.1 Pseudo-polynomials

A *pseudo-polynomial* extends the notion of a polynomial in that its coefficients are periodic numbers. Hence I first define periodic numbers and then I define pseudo-polynomials. The definitions I give here are in line with the definitions in [71].

Definition 2.16 (one-dimensional periodic coefficient)
Let be given an integer ℓ, and ℓ rational numbers $c_i, i = 0, 1, \cdots, \ell - 1$. The function $c() : \mathbb{Z} \rightarrow \mathbb{Q}, c(p) = c_{p \bmod \ell}$ is called a *periodic coefficient* with *period* ℓ. ■

The ℓ possible values of $c(p)$ are usually made explicit by representing $c(p)$ as an indexed array,

$$c(p) = [c_0, c_1, \cdots, c_{\ell-1}]_p \tag{2.9}$$

Definition 2.17 (multi-dimensional periodic coefficient)
Let be given an m-dimensional integral vector $\boldsymbol{\ell} = (\ell_1, \ell_2, \cdots, \ell_m)$, and $\prod_{i=1}^{m} \ell_i$ rational numbers $c_{\mathbf{i}}$, $\mathbf{i} = (i_1, i_2, \cdots, i_m)$, $i_j \in \{0, 1, \cdots, \ell_j - 1\}$, $j = 1, 2, \cdots, m$. The function $c() : \mathbb{Z}^m \rightarrow \mathbb{Q}, c(\mathbf{p}) = c_{\mathbf{p} \bmod \boldsymbol{\ell}}$, with $\mathbf{p} = (p_1, p_2, \cdots, p_m)$ and $\mathbf{p} \bmod \boldsymbol{\ell} = (p_1 \bmod \ell_1, p_2 \bmod \ell_2, \cdots, p_m \bmod \ell_m)$, is called an *$m$-dimensional periodic coefficient* with *multi-period* $\boldsymbol{\ell}$ and *period* $lcm(\ell_1, \ell_2, \cdots, \ell_m)$. Again, $c(\mathbf{p})$ is represented as an m-dimensional array, shown here for $m = 2$ (for notational convenience I use short-hands ℓ'_1 and ℓ'_2 for $\ell_1 - 1$ and $\ell_2 - 1$, respectively):

$$c(\mathbf{p}) = [[c_{00}, c_{10}, c_{\ell'_1 0}]_{p_1}, [c_{01}, c_{11}, c_{\ell'_1 1}]_{p_1}, \cdots, [c_{0\ell'_2}, c_{1\ell'_2}, c_{\ell'_1 \ell'_2}]_{p_1}]_{p_2} \tag{2.10}$$

The interpretation of (2.10) is that if $\mathbf{p} \bmod \boldsymbol{\ell} = \mathbf{i}$, then $c(\mathbf{p}) = c_{\mathbf{i}}$. ■

Example 2.1 gives three example of periodic numbers.

[2] In general the *enumerator* of any set $\mathcal{S}$ with respect to a *lattice* $\mathcal{L}^n$ is $|\mathcal{S} \cap \mathcal{L}^n|$, where $\mathcal{L}^n = G(\mathbb{Z}^n)$, G being an non-singular rational matrix. Unless otherwise stated, I will assume – without loss of generality [67] – that G is the identity matrix.

Example 2.1 (periodic number)

$$\sin(p\tfrac{\pi}{2}) = [0,1,0,-1]_p\ ,\ p \in \mathbb{Z}$$

$$\lceil \tfrac{p}{3} \rceil - \lfloor \tfrac{p}{3} \rfloor = [0,1,1]_p\ ,\ p \in \mathbb{Z}$$

$$\begin{bmatrix} -14 & -27 & -18 \\ 1 & 2 & 1 \\ 9 & 17 & 12 \end{bmatrix}^p = \begin{bmatrix} [1,-14,7]_p & [0,-27,18]_p & [0,-18,9]_p \\ [0,1,-3]_p & [1,2,-6]_p & [0,1,-4]_p \\ [0,9,-1]_p & [0,17,-5]_p & [1,12,-1]_p \end{bmatrix}\ ,\ p \in \mathbb{Z}$$

■

The last example says that the entries of the p-th power of the given matrix are periodic.

A generalization of a polynomial is a *pseudo-polynomial*.

Definition 2.18 (pseudo-polynomial)
Let $c_i(p), i = 0, 1, \cdots, k$ be scalar periodic coefficients. The function

$$f(p) = c_k(p)p^k + c_{k-1}(p)p^{k-1} + \cdots + c_0(p) \tag{2.11}$$

is a (one-dimensional) *pseudo-polynomial* of degree k. The generalization to higher dimensions is as follows.

Let $\mathbf{p} = (p_1, p_2, \cdots, p_m)$ be an m-dimensional integral vector and let $c_{i_1,i_2,\cdots,i_m}(\mathbf{p})$ be periodic numbers whose representing multi-dimensional arrays have dimension of at most m. The function

$$f(\mathbf{p}) = \sum_{i_1=0}^{k} \sum_{i_2=0}^{k-i_1} \cdots \sum_{i_m=0}^{k-i_1-i_2-\cdots-i_{m-1}} c_{i_1,i_2,\cdots,i_m}(\mathbf{p}) p_1^{i_1} p_2^{i_2} \cdots p_m^{i_m} \tag{2.12}$$

is an (m-dimensional) pseudo-polynomial of degree k. The *pseudo-period* of $f(\mathbf{p})$ is defined as the least common multiple of the periods of its periodic coefficients $c_{\mathbf{i}}(\mathbf{p})$. ■

2.4.2 The Ehrhart theorem

In [68], Ehrhart conjectured that the enumerator $E(p)$ of an affine-vertex polytope $\mathcal{P}(p)$ parameterized in a single positive integer parameter p is a certain pseudo-polynomial. The extension of this result to higher dimensional integer parameter vectors $\mathbf{p}$ has been established in [70] by Clauss and later in [69] by Clauss and Loechner.

Theorem 2.1 below gives the combined results. The proof is based on the theory of enumerative combinatorics [72] and is beyond the scope of this dissertation.

Theorem 2.1 (Ehrhart theorem)
The enumerator $E(\mathbf{p})$ of any affine-vertex k-polytope $\mathcal{P}(\mathbf{p}), \mathbf{p} = (p_1, p_2, \cdots, p_m)$ is expressed by a multivariate pseudo-polynomial in the parameter vector $\mathbf{p}$ of degree k whose pseudo-period ℓ is the denominator of $\mathcal{P}(\mathbf{p})$

$$E(\mathbf{p}) = \sum_{i_1=0}^{k} \sum_{i_2=0}^{k-i_1} \cdots \sum_{i_m=0}^{k-i_1-i_2-\cdots-i_{m-1}} c_{i_1,i_2,\cdots,i_m}(\mathbf{p}) p_1^{i_1} p_2^{i_2} \cdots p_m^{i_m} \tag{2.13}$$

The dimension m of the periodic coefficients equals the dimension of the parameter vector $\mathbf{p}$. The periods of the periodic coefficients $c_{\mathbf{i}}$ are ℓ in each dimension.

Proof
The proof of this theorem can be found in [68], [72], and [69]. ■

$E(\mathbf{p})$ is called the Ehrhart polynomial associated with $\mathcal{P}(\mathbf{p})$.

Since parameterized polytopes can be described in terms of affine-vertex polytopes, Theorem 2.1 is used to express the enumerator of any parameterized polytope $\mathcal{P}(\mathbf{p})$ by a set of Ehrhart polynomials, each one defined on the context of the affine-vertex polytopes of $\mathcal{P}(\mathbf{p})$.

Equation (2.13) defines a template for the Ehrhart polynomial of an affine-vertex polytope. To find the actual Ehrhart polynomial, the elements in all periodic coefficients of (2.13) must be determined. There are $\binom{m+k}{k}$ such coefficients each counting ℓ^m elements. To find these elements, $\ell^m\binom{m+k}{k}$ initial countings are performed to construct a system of $\ell^m\binom{m+k}{k}$ equations with $\ell^m\binom{m+k}{k}$ unknowns to be solved.

Example 2.2 (Ehrhart theorem I) Let $\mathcal{P}(p) = \{(x_1, x_2) \in \mathbb{Q}^2 \mid 0 \leq x_2 \leq 4 \wedge x_2 \leq x_1 \leq x_2+9 \wedge x_1 \leq p \wedge p \leq 40\}$ be the parameterized polytope given in Figure 2.2. $\mathcal{P}(p)$ is expressed as a set of affine-vertex

polytopes each defined on its own context which are given in (2.8).

$$
\begin{array}{l}
0 \le p \le 4 \\
\quad V_1(p) = \{(0,0),(p,0),(p,p)\} \\
\quad \text{all } v \in V_1(p) \in \mathbb{Z} \Rightarrow \text{all pseudo-periods are 1} \\
\quad \mathcal{P}_1(p) \text{ is a 2-polytope, hence } E_1(p) = c_2p^2 + c_1p + c_0 \\
\quad \text{initial countings: } E_1(0) = 1, E_1(1) = 3, E_1(2) = 6 \\
\quad \text{system to solve: } \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix} \\
\quad \text{result: } (c_0, c_1, c_2) = (1, 1\frac{1}{2}, \frac{1}{2}) \Rightarrow E_1(p) = \frac{1}{2}p^2 + 1\frac{1}{2}p + 1 \\
\\
4 \le p \le 9 \\
\quad V_2(p) = \{(0,0),(p,0),(4,4),(p,4)\} \\
\quad \text{all } v \in V_2(p) \in \mathbb{Z} \Rightarrow \text{all pseudo-periods are 1} \\
\quad \mathcal{P}_2(p) \text{ is a 2-polytope, hence } E_2(p) = c_2p^2 + c_1p + c_0 \\
\quad \text{initial countings: } E_2(4) = 15, E_2(5) = 20, E_2(6) = 25 \\
\quad \text{system to solve: } \begin{bmatrix} 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 6 & 36 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 15 \\ 20 \\ 25 \end{bmatrix} \\
\quad \text{result: } (c_0, c_1, c_2) = (-5, 5, 0) \Rightarrow E_2(p) = 5p - 5 \\
\\
9 \le p \le 13 \\
\quad V_3(p) = \{(0,0),(9,0),(p,p-9),(4,4),(p,4)\} \\
\quad \text{all } v \in V_3(p) \in \mathbb{Z} \Rightarrow \text{all pseudo-periods are 1} \\
\quad \mathcal{P}_3(p) \text{ is a 2-polytope, hence } E_3(p) = c_2p^2 + c_1p + c_0 \\
\quad \text{initial countings: } E_3(9) = 40, E_3(10) = 44, E_3(11) = 47 \\
\quad \text{system to solve: } \begin{bmatrix} 1 & 9 & 81 \\ 1 & 10 & 100 \\ 1 & 11 & 121 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 40 \\ 44 \\ 47 \end{bmatrix} \\
\quad \text{result: } (c_0, c_1, c_2) = (-41, 13\frac{1}{2}, -\frac{1}{2}) \Rightarrow E_3(p) = -\frac{1}{2}p^2 + 13\frac{1}{2}p - 41 \\
\\
13 \le p \le 40 \\
\quad V_4(p) = \{(0,0),(9,0),(13,4),(4,4)\} \\
\quad \text{all } v \in V_4(p) \in \mathbb{Z} \Rightarrow \text{all pseudo-periods are 1} \\
\quad \mathcal{P}_4(p) \text{ is a 2-polytope, hence } E_4(p) = c_2p^2 + c_1p + c_0 \\
\quad \text{initial countings: } E_4(13) = 50, E_4(14) = 50, E_4(15) = 50 \\
\quad \text{system to solve: } \begin{bmatrix} 1 & 13 & 169 \\ 1 & 14 & 196 \\ 1 & 15 & 225 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 50 \\ 50 \\ 50 \end{bmatrix} \\
\quad \text{result: } (c_0, c_1, c_2) = (50, 0, 0) \Rightarrow E_4(p) = 50 \qquad \blacksquare
\end{array}
\tag{2.14}
$$

In Example 2.2 all vertices are integral and, therefore, all periodic coefficients of the pseudo-polynomials are ordinary scalars. The next example deals with the enumerator of a single affine-vertex polytope with rational vertices.

Example 2.3 (Ehrhart theorem II) Let $\mathcal{P}(p,q)$ be given: $\mathcal{P}(p,q) = \{(x_1,x_2) \in \mathbb{Q}^2 \,|\, 0 \le x_2 \le \frac{1}{2}q \ \wedge\ 2x_2 \le x_1 \le 2x_2 + \frac{1}{2}p\}$ with context $p,q \ge 0$. The parameterized vertices of $\mathcal{P}(p,q)$ are $V(p,q) = \{(0,0),(\frac{1}{2}p,0),(q,\frac{1}{2}q),(q+\frac{1}{2}p,\frac{1}{2}q)\}$. Since the vertices are defined on whole $\mathcal{P}(p,q)$'s context, $\mathcal{P}(p,q)$

is an affine-vertex polytope. Observe that the dimension of both $\mathcal{P}(p,q)$ and its denominator are 2. As a consequence, the Ehrhart polynomial is a multi variate pseudo-polynomial in two variables of degree 2 and pseudo-period 2:

$$E(p,q) = c_{20}p^2 + c_{11}pq + c_{02}q^2 + c_{10}p + c_{01}q + c_{00} \tag{2.15}$$

where every $c_{ij}(p,q) = [[c_{ij}^{00}, c_{ij}^{01}]_p, [c_{ij}^{10}, c_{ij}^{11}]_p]_q$. There are 24 unknowns and, as a consequence, a system of 24 equalities in 24 unknowns is formulated.

Let me denote the elements of the periodic coefficients c_{ij} by $c_{ij}^{\Delta p \Delta q}$, where $\Delta p, \Delta q \in \{0,1\}$. Rather than building one system of 24 equation with 24 unknowns, I construct four independent systems of 6 equations with 6 unknowns each, and each system containing only the $c_{ij}^{\Delta p \Delta q}$ for particular values of Δp and Δq. Since $\Delta p = p \mod 2$ and $\Delta q = q \mod 2$, one system corresponds to the elements with both p and q even, two systems with either p or q odd, and one system with both p and q odd.

Four sets of initial countings[3], one for each setting of Δp, Δq:

$E(0+\Delta p, 0+\Delta q) = 1$
$E(2+\Delta p, 0+\Delta q) = 2$
$E(4+\Delta p, 0+\Delta q) = 3$
$E(0+\Delta p, 2+\Delta q) = 2$
$E(0+\Delta p, 4+\Delta q) = 3$
$E(2+\Delta p, 2+\Delta q) = 4$

Four systems to solve, one for each setting of Δp, Δq:

$$\begin{bmatrix} 1 & 0+\Delta q & (0+\Delta q)^2 & 0+\Delta p & (0+\Delta p)(0+\Delta q) & (0+\Delta p)^2 \\ 1 & 0+\Delta q & (0+\Delta q)^2 & 2+\Delta p & (2+\Delta p)(0+\Delta q) & (2+\Delta p)^2 \\ 1 & 0+\Delta q & (0+\Delta q)^2 & 4+\Delta p & (4+\Delta p)(0+\Delta q) & (4+\Delta p)^2 \\ 1 & 2+\Delta q & (2+\Delta q)^2 & 0+\Delta p & (0+\Delta p)(2+\Delta q) & (0+\Delta p)^2 \\ 1 & 4+\Delta q & (4+\Delta q)^2 & 0+\Delta p & (0+\Delta p)(4+\Delta q) & (0+\Delta p)^2 \\ 1 & 2+\Delta q & (2+\Delta q)^2 & 2+\Delta p & (2+\Delta p)(2+\Delta q) & (2+\Delta p)^2 \end{bmatrix} \begin{bmatrix} c_{00}^{\Delta p \Delta q} \\ c_{01}^{\Delta p \Delta q} \\ c_{02}^{\Delta p \Delta q} \\ c_{10}^{\Delta p \Delta q} \\ c_{11}^{\Delta p \Delta q} \\ c_{20}^{\Delta p \Delta q} \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Solving these systems for the four possible settings of $(\Delta p, \Delta q)$ gives

$c_{00}(p,q) = [\ [1, \frac{1}{2}]_p,\ [\frac{1}{2}, \frac{1}{4}]_p\ ]_q = [[1, \frac{1}{2}]_p, [\frac{1}{2}, \frac{1}{4}]_p]_q$
$c_{01}(p,q) = [\ [\frac{1}{2}, \frac{1}{4}]_p,\ [\frac{1}{2}, \frac{1}{4}]_p\ ]_q = [\frac{1}{2}, \frac{1}{4}]_p$
$c_{02}(p,q) = [\ [0,0]_p,\ [0,0]_p\ ]_q = 0$
$c_{10}(p,q) = [\ [\frac{1}{2}, \frac{1}{2}]_p,\ [\frac{1}{4}, \frac{1}{4}]_p\ ]_q = [\frac{1}{2}, \frac{1}{4}]_q$
$c_{11}(p,q) = [\ [\frac{1}{4}, \frac{1}{4}]_p,\ [\frac{1}{4}, \frac{1}{4}]_p\ ]_q = \frac{1}{4}$
$c_{20}(p,q) = [\ [0,0]_p,\ [0,0]_p\ ]_q = 0$

As a result, the Ehrhart polynomial is $E(p,q) = \frac{1}{4}pq + [\frac{1}{2}, \frac{1}{4}]_q p + [\frac{1}{2}, \frac{1}{4}]_p q + [[1, \frac{1}{2}]_p, [\frac{1}{2}, \frac{1}{4}]_p]_q$.

■

2.5 Images of integer polyhedra

This section deals with particular sets of regular spaced integral points. These sets are obtained by the application of an affine function to all integral points contained in a polyhedron. Such set of points are extensively used in modeling the set of iterations that are specified by a specific class of nested loop programs, as further discussed in Section 3.3 and Section 3.4.

[3] It just turned out that the results of the initial countings of the four sets are the same. This allowed me to write this example in this compact way.

2.5.1 Integer polyhedral images

Previous sections dealt with polyhedra which are subset of a rational space $\mathbb{Q}^n$. This section deals with three types of images of integer polyhedra under affine mapping. In this dissertation I define an integer polyhedron[4] as the set of integer points contained in a polyhedron. That is, let $\mathcal{P} \in \mathbb{Q}^n$ be a polyhedron, the integer polyhedron defined by $\mathcal{P}$ is $\mathcal{P} \cap \mathbb{Z}^n$. The tree types of integer polyhedral images discussed in this section are *linearly bounded lattices*, *$\mathbb{Z}$-polyhedra*, and *periodic lattice polyhedra*, and are discussed in this and the following subsection.

Let be given two sets $\mathcal{A} \in \mathbb{Z}^n$ and $\mathcal{B} \in \mathbb{Z}^d$. An integral affine function $M() : \mathcal{A} \rightarrow \mathcal{B}$ is defined by an integral matrix $d \times n$ matrix M and an integral d-vector $\mathbf{m}$ by $M(\mathbf{k}) = M\mathbf{k} + \mathbf{m}$. When $\mathbf{m}$ is the zero vector, $M()$ is an integral linear function. The *rank* of $M()$ is defined as the rank of the matrix M, $\mathrm{rank}(M)$. The function $M()$ is called non-singular when its matrix has an inverse. Let be given a set $\mathcal{S} \subseteq \mathcal{A}$, the set $\mathcal{I} = M(\mathcal{S}) = \{\mathbf{i} \mid \mathbf{i} = M(\mathbf{s}) \ \wedge \ \mathbf{s} \in \mathcal{S}\}$ is called the *image* of $\mathcal{S}$ under $M()$.

All three kinds of integer polyhedral images are defined by the image of an integral polyhedron $\mathcal{P} \cap \mathbb{Z}^n$ under integral affine function $M()$.

Definition 2.19 (integer polyhedral image)
Let be given a polyhedron $\mathcal{P} \in \mathbb{Q}^n$ and an integral affine function $M()$ with $d \times n$ matrix M and d-vector $\mathbf{m}$. An integer polyhedral image $\mathcal{I} \subset \mathbb{Z}^d$ is defined by

$$\mathcal{I} = M(\mathcal{P} \cap \mathbb{Z}^n) = \{\mathbf{i} \mid \mathbf{i} = M\mathbf{k} + \mathbf{m} \wedge \ \mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n\} \tag{2.16}$$

■

Without any constraints on the rank of $M()$ and the geometry of $\mathcal{P}$, (2.16) defines the first kind of integer polyhedral image which is called *linearly bounded lattice* (LBL) [28].

Because in the LBL there are no additional constraints on $M()$ and $\mathcal{P}$, multiple points in $\mathcal{P} \cap \mathbb{Z}^n$ are, in general, mapped onto the same point $\mathbf{i} \in \mathbb{Z}^d$, and hence, $M()$ has no inverse. The absence of an inverse mapping in the LBL makes this model unsuitable in analyses that require such inverse. The other two kinds of integer polyhedral images are the *$\mathbb{Z}$-polyhedron* [73] and *periodic lattice polyhedron* which are special special cases the LBL model; they impose additional constraints on $M()$ and $\mathcal{P}$ such that they have an inverse of their mappings.

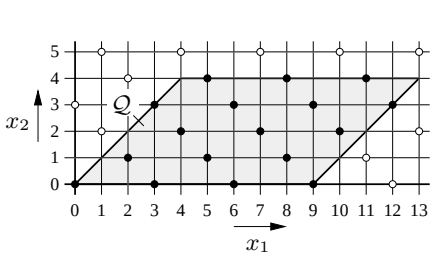
Definition 2.20 ($\mathbb{Z}$-polyhedron)
Let be given an integral non-singular $d \times d$ matrix G and a polyhedron $\mathcal{Q} \in \mathbb{Q}^d$. A (full-dimensional) $\mathbb{Z}$-polyhedron is defined by $\mathcal{I} = \mathcal{Q} \cap G(\mathbb{Z}^d)$ where $G()$ is the linear function with matrix G. The set $\mathcal{L} = G(\mathbb{Z}^d)$ defines a so called full-dimensional integer lattice; the set of all integral linear combinations of the columns of G. ■

Interestingly, $\mathbb{Z}$-polyhedra have a canonical representation called the $\mathbb{Z}$-polyhedron normal form [26]. Every $\mathbb{Z}$-polyhedron $\mathcal{I} = \mathcal{Q} \cap G(\mathbb{Z}^d)$ is represented by an integral polyhedral image $\mathcal{I} = M(P \cap \mathbb{Z}^d)$ with $M = BH$, where B is an linear unimodular basis of $\mathbb{Z}^d$ such that the first k columns of B are a lowest cardinality linear basis of $\mathcal{I}$, H is an integral matrix in Hermite normal form, and $\mathcal{P}$ is a rational polyhedron that is derived from $\mathcal{Q}$ and $G()$ by a number of transformations [26].

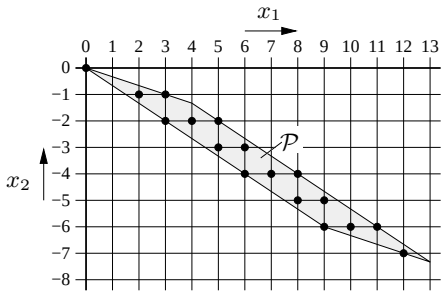
From this it follows that a $\mathbb{Z}$-polyhedron is an integer linear image with $n = d$ and $M()$ being defined by a non-singular matrix M. So one can view $M()$ as a bijective function $M() : \mathbb{Z}^d \rightarrow \mathbb{Z}^d$. The following example illustrates a $\mathbb{Z}$-polyhedron.

[4] Sometimes integer polyhedra are defined as rational polyhedra whose vertices are all integer. This definition is not used here.

Example 2.4 ($\mathbb{Z}$-polyhedron) Let $\mathcal{I}$ be the $\mathbb{Z}$-polyhedron given by $\mathcal{Q} \cap G(\mathbb{Z}^2)$ where $\mathcal{Q} = \{(x_1, x_2) \in \mathbb{Q}^2 \,|\, 0 \leq x_2 \leq 4 \ \wedge \ x_2 \leq x_1 \leq x_2 + 9\}$, and $G = \left[\begin{smallmatrix} 3 & 2 \\ 0 & 1 \end{smallmatrix}\right]$. $\mathcal{I}$ is shown in Figure 2.4 (a). Since the basis that contains $\mathcal{I}$ is the identity matrix, matrix B is not shown.



(a) $\mathcal{I} = \mathcal{Q} \cap G(\mathbb{Z}^2), G = \left[\begin{smallmatrix} 3 & 2 \\ 0 & 1 \end{smallmatrix}\right]$, dots show $G(\mathbb{Z}^2)$, and black dots show $\mathcal{I}$



(b) $\mathcal{I} = H(\mathcal{P} \cap \mathbb{Z}^2), H = \left[\begin{smallmatrix} 1 & 0 \\ 2 & 3 \end{smallmatrix}\right]$, and black dots show $\mathcal{P} \cap \mathbb{Z}^2$

Figure 2.4: A domain represented by (a) a $\mathbb{Z}$-polyhedron and (b) its canonical representation. ■

Now consider the set of points in Figure 2.5. This set is constructed from subsets of three consecutive points that is repeated along the x axis with a *pitch* of four, and that is bounded by the polytope $2 \leq x \leq 10$, indicated by the black colored dots. Note that there exists no full-dimensional integral lattice $G(\mathbb{Z})$ that intersected with a polytope results in the set of points in Figure 2.5, hence this set cannot be represented by a (single) $\mathbb{Z}$-polyhedron.

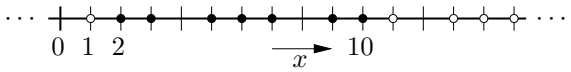


Figure 2.5: A set of points that cannot be presented by the intersection of a polytope and a full-dimensional lattice.

2.5.2 Periodic lattice polyhedra

This section describes the third kind of integer polyhedral images. To my best knowledge this is a novel model, which I call *periodic lattice polyhedra*. They are less general than linearly bounded lattices but more general than $\mathbb{Z}$-polyhedra.

Let me introduce periodic lattice polyhedra by the following example. The set of points in Figure 2.5 can be represented by the image of the integer points contained by polytope $\mathcal{P} = \{(x, c) \in \mathbb{Q}^2 \,|\, 2 \leq x \leq 10 \wedge 4c+1 \leq x \leq 4c+3\}$ and the affine mapping $L()$ represented by the matrix $L = \begin{bmatrix} 1 & 0 \end{bmatrix}$. $\mathcal{P}$ and $L(\mathcal{P})$ are illustrated in Figure 2.6.

Definition 2.21 (lattice defining polyhedron)
Let be given two integers d and k, an integral $k \times d$ matrix A, an integral positive non-singular diagonal $k \times k$ matrix Λ and two integral k-vectors $\mathbf{b} = (b_1, b_2, \cdots, b_k)$ and $\mathbf{c} = (c_1, c_2, \cdots, c_k)$ with $\mathbf{b} \leq \mathbf{c}$. A *$d$-lattice defining polyhedron* $\mathcal{P}_L$ is defined by

$$\mathcal{P}_L = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{b} \leq \begin{bmatrix} A & \Lambda \end{bmatrix} \mathbf{x} \leq \mathbf{c}\} \tag{2.17}$$

where $n = d + k$ and $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \cdots, \lambda_k)$ with $\lambda_j > c_j - b_j$. ■

Note that the region in between (and including) the two diagonal parallel lines in Figure 2.6 is a 1-lattice defining polyhedron with $A = -1$, $\Lambda = 4$, $\mathbf{b} = -3$, and $\mathbf{c} = -1$.

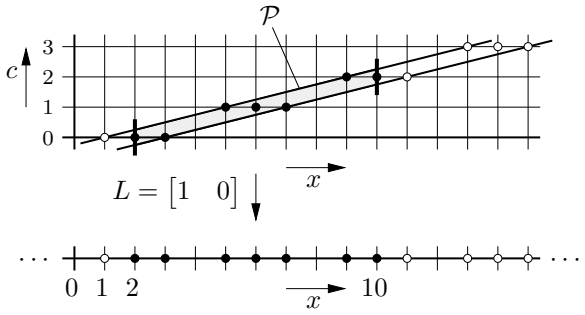


Figure 2.6: The set of points in Figure 2.5 defined by the projection $L(\mathcal{P} \cap \mathbb{Z}^2)$ onto $\mathbb{Z}$.

Theorem 2.2
The projection of any d-lattice defining polyhedron onto $\mathbb{Q}^d$ is $\mathbb{Q}^d$.

Proof
The proof is to show that for every $\mathbf{y} \in \mathbb{Q}^d$ there exists a $\mathbf{x} \in \mathcal{P}_L$ such that $\mathbf{y} = \text{proj}_{\mathbb{Q}^d}(\mathbf{x})$. Since the projection of $\mathbf{x}$ onto $\mathbb{Q}^d$ is the vector composed of the first d elements of $\mathbf{x}$, $\mathbf{x}$ must have the form $(\mathbf{y}, \mathbf{z})$. Thus, it must be proved that for any vector $\mathbf{y}$ there exists a vector $\mathbf{z}$ such that $(\mathbf{y}, \mathbf{z}) \in \mathcal{P}_L$. Existence is proved by solving (2.17) for equality with $\mathbf{b}$, that is, $\mathbf{b} = A\mathbf{y} + \Lambda\mathbf{z}$. By definition Λ is non-singular and hence $\mathbf{z} = \Lambda^{-1}(\mathbf{b} - A\mathbf{y})$. ■

Lemma 2.1
For any integral vector $\mathbf{m}$, there is at most one integral vector $\boldsymbol{\ell}$ such that $\mathbf{0} \leq \mathbf{m} + \boldsymbol{\ell} < \mathbf{1}$.

Proof
The proof is by contradiction. Assume that there exist two different integral vectors $\mathbf{u}$ and $\mathbf{v}$ that meet the constraints in Lemma 2.1. Because these vectors are different, there must be at least one coordinate i such that $v_i = u_i + k, k \neq 0$. Clearly with u_i such that $0 \leq m_i + u_i \leq 1$ the only integral value of k such that $0 \leq m_i + u_i + k < 1$ is $k = 0$. This is in contradiction with the assumption that $\mathbf{u}$ and $\mathbf{v}$ are different and hence the solution in Lemma 2.1, if exists, is unique. ■

Theorem 2.3
Let a d-lattice defining polyhedron $\mathcal{P}_L \subset \mathbb{Q}^n$ be given. For any point $\mathbf{i} \in \mathbb{Z}^d$, there exists at most one point $\mathbf{k} \in \mathcal{P}_L \cap \mathbb{Z}^n$ such that $\mathbf{i} = \text{proj}_{\mathbb{Q}^d}(\mathbf{k})$.

Proof
Let $\mathbf{i}$ be given. Like in the proof of Theorem 2.2, the solutions to $\mathbf{i} = \text{proj}_{\mathbb{Q}^d}(\mathbf{k})$ must have the form $\mathbf{k} = (\mathbf{i}, \boldsymbol{\ell})$. So, the proof is to show that if $\boldsymbol{\ell}$ exists it is unique. Substituting $\mathbf{k} = (\mathbf{i}, \boldsymbol{\ell})$ in (2.17) results in $\mathbf{b} \leq A\mathbf{i} + \Lambda\boldsymbol{\ell} \leq \mathbf{c}$. Subtracting $\mathbf{b}$ from each term results in $\mathbf{0} \leq A\mathbf{i} - \mathbf{b} + \Lambda\boldsymbol{\ell} \leq \mathbf{c} - \mathbf{b}$. By definition Λ is non-singular and positive and hence Λ^{-1} can be applied to each term without changing the "$\leq$"s, this results in $\mathbf{0} \leq \Lambda^{-1}(A\mathbf{i} - \mathbf{b}) + \boldsymbol{\ell} \leq \Lambda(\mathbf{c} - \mathbf{b})$. By using $\lambda_j > c_j - b_j$ we get $\Lambda^{-1}(\mathbf{c} - \mathbf{b}) < \mathbf{1}$ and hence $\mathbf{0} \leq \Lambda^{-1}(A\mathbf{i} - \mathbf{b}) + \boldsymbol{\ell} < \mathbf{1}$. Calling $\mathbf{m} = \Lambda^{-1}(A\mathbf{i} - \mathbf{b})$ results in $\mathbf{0} \leq \mathbf{m} + \boldsymbol{\ell} < \mathbf{1}$ and by Lemma 2.1, if $\boldsymbol{\ell}$ exists it is unique and hence $\mathbf{k}$ is unique. ■

Definition 2.22 (periodic lattice polyhedron I)
Let be given a d-lattice defining polyhedron $\mathcal{P}_L \subset \mathbb{Q}^n$, the $d \times n$ projection matrix $L = \begin{bmatrix} I & 0 \end{bmatrix}$, and a polyhedron $\mathcal{Q} \subset \mathbb{Q}^d$. A *periodic lattice polyhedron* $\mathcal{I}$ is defined by $\mathcal{I} = \mathcal{Q} \cap L(\mathcal{P}_L \cap \mathbb{Z}^n)$, where $L()$ is the linear function that has L as its matrix. ■

The periodic lattice polyhedron has an alternative representation. This representation is more useful in next sections and is derived in the following.

Definition 2.23 (embedded polyhedron)
Let a polyhedron $\mathcal{Q} \subset \mathbb{Q}^d$ be given. The n-dimensional *embedded polyhedron* $\mathcal{Q}_E$ of $\mathcal{Q}$ is defined by

$$\mathcal{Q}_E = \{(x_1, x_2, \cdots, x_n) \in \mathbb{Q}^n \,|\, (x_1, x_2, \cdots, x_d) \in \mathcal{Q}, d \leq n\} \tag{2.18}$$

where n is the dimension of the space $\mathcal{Q}$ is embedded in. ■

Note that for $d = n$, $\mathcal{Q}_E = \mathcal{Q}$. Moreover, the embedding of $\mathcal{Q}$ into higher dimensional space corresponds to adding rays for any additional dimension in the corresponding direction.

Theorem 2.4
Let $\mathcal{I} = \mathcal{Q} \cap L(\mathcal{P}_L \cap \mathbb{Z}^n)$ be a periodic lattice polyhedron in $\mathbb{Z}^d$. Let $\mathcal{J}$ be the set $\mathcal{J} = L(\mathcal{P} \cap \mathbb{Z}^n)$ where $\mathcal{P} = \mathcal{Q}_E \cap \mathcal{P}_L$ with $\mathcal{Q}_E$ the n-dimensional embedded polyhedron of $\mathcal{Q}$. The two sets $\mathcal{I}$ and $\mathcal{J}$ are equivalent.

Proof
The sets $\mathcal{I}$ and $\mathcal{J}$ are equivalent when $\mathbf{i} \in \mathcal{I}$ if and only if $\mathbf{i} \in \mathcal{J}$. I first prove that a) $\mathbf{i} \in \mathcal{I} \Rightarrow \mathbf{i} \in \mathcal{J}$ and then that b) $\mathbf{i} \in \mathcal{J} \Rightarrow \mathbf{i} \in \mathcal{I}$.
a) $\mathbf{i} \in \mathcal{I}$ implies that $\mathbf{i} \in \mathcal{Q} \wedge \mathbf{i} \in L(\mathcal{P}_L \cap \mathbb{Z}^n)$. Since $\mathbf{i} \in L(\mathcal{P}_L \cap \mathbb{Z}^n)$ there exists, according to Theorem 2.3, a unique integral vector $\mathbf{k} = (\mathbf{i}, \boldsymbol{\ell}) \in \mathcal{P}_L \cap \mathbb{Z}^n$ such that $\mathbf{i} = L(\mathbf{k})$. Since $\mathbf{i} \in \mathcal{Q}$ the vector $(\mathbf{i}, \mathbf{z}) \in \mathcal{Q}_E$ for all $\mathbf{z} \in \mathbb{Q}^{n-d}$ and thus $\mathbf{k} = (\mathbf{i}, \boldsymbol{\ell}) \in \mathcal{Q}_E$. Since $\mathbf{k} \in \mathcal{Q}_E$ and $\mathbf{k} \in \mathcal{P}_L \cap \mathbb{Z}^n$ with $\mathbf{i} = L(\mathbf{k})$, $\mathbf{k} \in \mathcal{Q}_E \cap \mathcal{P}_L \cap \mathbb{Z}^n$ and thus $\mathbf{i} \in L(\mathcal{Q}_E \cap \mathcal{P}_L \cap \mathbb{Z}^n) = \mathcal{J}$.
b) $\mathbf{i} \in \mathcal{J}$ implies that there exists a $\mathbf{k} \in \mathcal{Q}_E \cap \mathcal{P}_L \cap \mathbb{Z}^n$ such that $\mathbf{i} = L(\mathbf{k})$, moreover, this $\mathbf{k}$ is unique. Clearly for this $\mathbf{k}$ it holds that $\mathbf{k} \in \mathcal{Q}_E \wedge \mathbf{k} \in \mathcal{P}_L \cap \mathbb{Z}^n$. Applying $L()$ two both sides of the "$\wedge$" results in $L(\mathbf{k}) \in \mathcal{Q} \wedge L(\mathbf{k}) \in L(\mathcal{P}_L \cap \mathbb{Z}^n)$ and hence $\mathbf{i} = L(\mathbf{k}) \in \mathcal{Q} \cap L(\mathcal{P}_L \cap \mathbb{Z}^n) = \mathcal{I}$. ■

Theorem 2.4 allows me to define periodic lattice polyhedra alternatively as follows.

Definition 2.24 (periodic lattice polyhedron II)
Let be given a polyhedron $\mathcal{P} = \mathcal{Q}_E \cap \mathcal{P}_L \subset \mathbb{Q}^n$ and the $d \times n$ matrix $L = \begin{bmatrix} I & 0 \end{bmatrix}$. A periodic lattice polyhedron $\mathcal{I}$ is defined by

$$\mathcal{I} = L(\mathcal{P} \cap \mathbb{Z}^n) \tag{2.19}$$

where $L()$ is the linear function whose matrix is L, that is, $L() = \mathrm{proj}_{\mathbb{Q}^d}()$. ■

From Theorem 2.3 it follows that $L()$ is injective from $\mathcal{P}_L$ to $\mathbb{Z}^d$ and hence from $\mathcal{P}$ to $\mathbb{Z}^d$. This means that $L() : \mathcal{P} \rightarrow \mathcal{I}$ is bijective and hence invertible.

Theorem 2.5
Let polyhedron $\mathcal{Q} \in \mathbb{Q}^d$ and d-lattice defining polyhedron $\mathcal{P}_L$ be given. Further let $\mathcal{Q}_E$ be the n-dimensional embedded polyhedron of $\mathcal{Q}$. $\mathcal{Q}$ is a polytope if and only if $\mathcal{P} = \mathcal{Q}_E \cap \mathcal{P}_L$ is a polytope.

Proof
This proof first shows that a) $\mathcal{Q}$ being a polytope implies that $\mathcal{P}$ is a polytope and then b) $\mathcal{P}$ being a polytope implies that $\mathcal{Q}$ is a polytope.
a) When $\mathcal{Q}$ is a polytope, the embedding in $\mathbb{Q}^n$ results in a polyhedron that has all its lines perpendicular to $\mathbb{Q}^d$ and has no rays (lines in one direction). Moreover, $\mathcal{P}_L$ has no rays or lines that are perpendicular to $\mathbb{Q}^d$ because it would violate Lemma 2.1 (similar to the proof in Theorem 2.3). Therefore, since $\mathcal{P} = \mathcal{Q}_E \cap \mathcal{P}_L$, it contains no lines and no rays and is therefore bounded. Since the projection of a polyhedron is a polyhedron, $\mathcal{P}$ is bounded polyhedron and therefore a polytope.
b) It is a well known that the projection of a polytope is a polytope and hence $\mathcal{Q} = \mathrm{proj}_{\mathbb{Q}^d}(\mathcal{P})$ is a polytope.
■

Figure 2.7 illustrates the relations between the polyhedra in the two representations of periodic lattice polyhedra. In all cases the polyhedron $\mathcal{Q} \subset \mathbb{Q}^d$ is the projection of its embedded polyhedron $\mathcal{Q}_E \subset \mathbb{Q}^n$ onto $\mathbb{Q}^d$. In figures (a) and (b), $d = 1$ and $n = 2$; in figures (c) and (d), $d = 1$ and $n = 3$; and in figures (e) and (f), $d = 2$ and $n = 3$. In the figure, I used the notation c_1 and c_2 for x_{d+1} and x_{d+2}, respectively. The first column of the figure shows the special cases where the vectors $\mathbf{b}$ and $\mathbf{c}$ of $\mathcal{P}_L$ in 2.17 are equal, i.e., the rows of matrix A describe hyper-planes. The second column shows the general cases. Solid lines represent bounds of the polyhedra. For example, $\mathcal{Q}_E$ in figure (d) is unbounded in the directions c_1 and c_2 but is bounded in the x_1 direction. Note, in all cases $\mathcal{Q}$ is a polytope and hence $\mathcal{P}$ is a polytope.

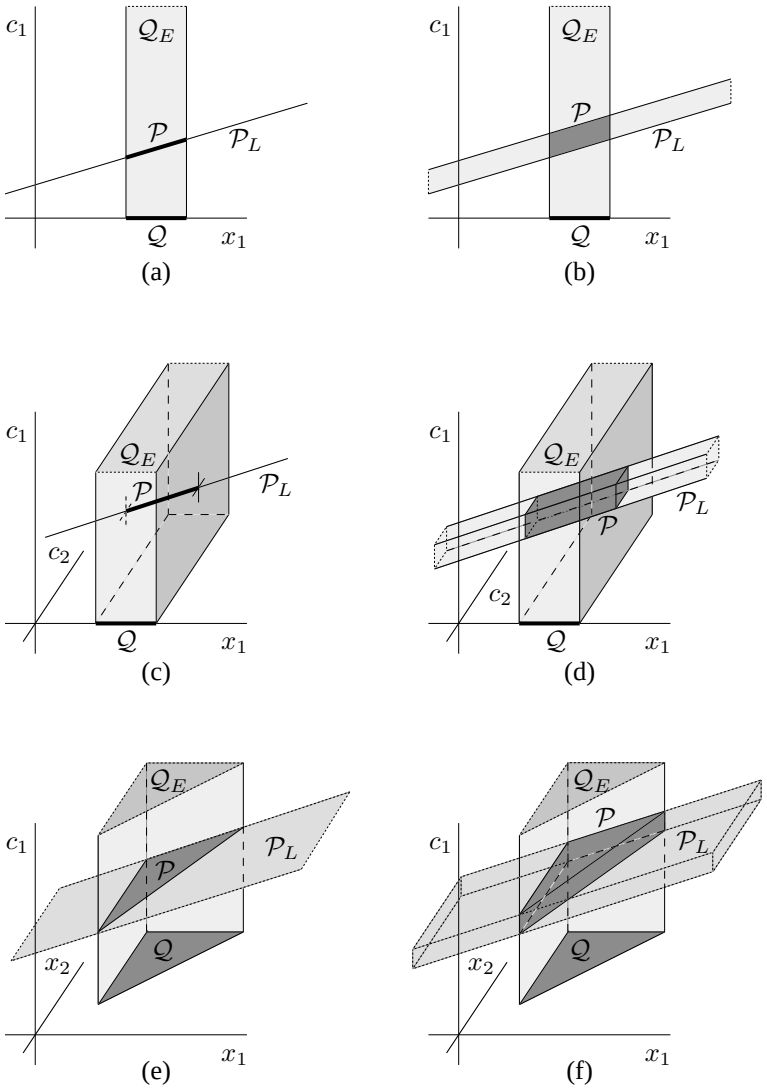


Figure 2.7: Illustration of periodic lattice polyhedra $\mathcal{I} = \mathcal{Q} \cap \mathrm{proj}_{\mathbb{Q}^d}(\mathcal{P}_L \cap \mathbb{Z}^n) = proj_{\mathbb{Q}^d}(\mathcal{P} \cap \mathbb{Z}^n)$. $\mathcal{I}$ itself is not depicted. Here $\mathcal{P} = \mathcal{Q}_E \cap \mathcal{P}_L$ and $\mathcal{Q} = \mathrm{proj}_{\mathbb{Q}^d}(\mathcal{P})$. In figures (a,b), $n = 2, d = 1$, in figures (c,d), $n = 3, d = 1$, and in figures (e,f), $n = 3, d = 2$. In the first and second column $\mathcal{P}_L$ is defined by only hyper-planes and only pairs of parallel hyper-planes, respectively.

2.6 Ehrhart test

This section describes a novel method which I call the *Ehrhart test*. The Ehrhart test is used to test whether or not a point does belong to a certain integer polyhedral image. In addition, the Ehrhart test computes how many integral points in the polytope that defines the integer polyhedral image are mapped onto points in the space that contains the integer polyhedral image.

Let be given a parameterized polytope $\mathcal{P}(\mathbf{p}) \in \mathbb{Q}^n$ and an integral affine function $M()$ from $\mathbb{Z}^n$ to $\mathbb{Z}^d$ defining the integer polyhedral image $M(\mathcal{P}(\mathbf{p}) \cap \mathbb{Z}^n)$. Often we are interested how many points in $\mathcal{P}(\mathbf{p}) \cap \mathbb{Z}^n$ map onto a certain point $\mathbf{j} \in \mathbb{Z}^d$. This number is called the multiplicity of $\mathbf{j}$ with respect to $M(\mathcal{P}(\mathbf{p}) \cap \mathbb{Z}^n)$.

In other words, let be given

$$\mathcal{K}(\mathbf{p}, \mathbf{j}) = \{\mathbf{x} \in \mathcal{P}(\mathbf{p}) \,|\, \mathbf{j} = M(\mathbf{x})\} \tag{2.20}$$

the problem is to find $\mathcal{M}(\mathbf{p}, \mathbf{j}) = |\, \mathcal{K}(\mathbf{p}, \mathbf{j}) \cap \mathbb{Z}^n \,|$, the number of integral points in $\mathcal{K}(\mathbf{p}, \mathbf{j})$.

Theorem 2.6
Let be given the integer polyhedral image $M(\mathcal{P}(\mathbf{p}) \cap \mathbb{Z}^n)$, $\mathcal{P}(\mathbf{p}) \in \mathbb{Q}^n$ and $M() : \mathbb{Z}^n \rightarrow \mathbb{Z}^d$. Further let $\mathcal{P}(\mathbf{j})$ be the parameterized polyhedron $\{\mathbf{x} \in \mathbb{Q}^n \,|\, M(\mathbf{x}) = \mathbf{j}\}$. The multiplicity $\mathcal{M}(\mathbf{p}, \mathbf{j})$ of $\mathbf{j} \in \mathbb{Z}^d$ is a set of Ehrhart polynomials associated with the affine-vertex polytopes of $\mathcal{P}(\mathbf{p}) \cap \mathcal{P}(\mathbf{j})$.

Proof
Since the intersection of a polytope with a polyhedron is a polytope, $\mathcal{P}(\mathbf{p}) \cap \mathcal{P}(\mathbf{j})$ is a polytope. Clearly $\mathcal{K}(\mathbf{p}, \mathbf{j})$ in 2.20 is $\mathcal{P}(\mathbf{p}) \cap \mathcal{P}(\mathbf{j})$ and, thus, is a polytope. By Section 2.3.2 the parameter space of $\mathcal{K}(\mathbf{p}, \mathbf{j})$ can be partitioned such that $\mathcal{K}(\mathbf{p}, \mathbf{j})$ is an affine-vertex polytope on each of the partitions and consequently by theorem 2.1 the enumerator of $\mathcal{K}(\mathbf{p}, \mathbf{j})$ is an Ehrhart polynomial. ■

Example 2.5 (Ehrhart test) Let be given the integer polyhedral image $\mathcal{I} = M(\mathcal{P}(N) \cap \mathbb{Z}^2)$, where $\mathcal{P}(N) = \{(x_1, x_2) \in \mathbb{Q}^2 \,|\, 0 \leq x_1 \leq N \ \wedge \ x_1 \leq x_2 \leq N\}$ and $M(\mathbf{k}) = 2k_1 + k_2 + 3, (k_1, k_2) \in \mathbb{Z}^2$. Figure 2.8 shows $\mathcal{P}(N) \cap \mathbb{Z}^2$, and $\mathcal{I} = M(\mathcal{P}(N) \cap \mathbb{Z}^2)$. To find the multiplicity of points j the parameterized

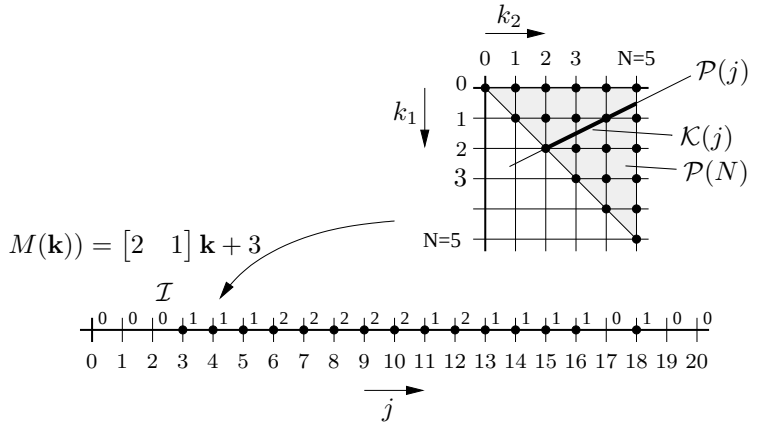


Figure 2.8: Multiplicity of points j in the domain $\mathcal{I} = (M, \mathcal{P}(N))$.

polyhedron $\mathcal{P}(j)$ is constructed, $\mathcal{P}(j) = \{(x_1, x_2) \in \mathbb{Q}^2 \mid j = 2x_1 + x_2 + 3\}$. As a result $\mathcal{K}(N, j) = \mathcal{P}(N) \cap \mathcal{P}(j) = \{(x_1, x_2) \,|\, 0 \leq x_1 \leq N \ \wedge \ x_1 \leq x_2 \leq N \wedge j = 2x_1 + x_2 + 3\}$. By using the Ehrhart

theory from a previous subsection the multiplicity $\mathcal{M}(N,j) = |\mathcal{K}(N,j) \cap \mathbb{Z}^2|$ is

$$\mathcal{M}(N,j) = \begin{cases} \frac{1}{3}j + [0, -\frac{1}{3}, -\frac{2}{3}]_j & \text{if } 3 \leq j \leq N+3 \\ \frac{1}{2}N + [-\frac{1}{6}j + [1, \frac{7}{6}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{5}{6}]_j, & \\ \qquad -\frac{1}{6}j + [\frac{3}{2}, \frac{2}{3}, \frac{5}{6}, 1, \frac{7}{6}, \frac{1}{3}]_j]_N & \text{if } N+3 \leq j \leq 3N+3 \\ 0 & otherwise \end{cases} \quad \blacksquare \tag{2.21}$$

Note that in the example the domain $\mathcal{I}$ is not a polytope (intersected with $\mathbb{Z}$) since there is a "hole" at $j = 17$.

Since we sometimes are not interested to know how many integral points in a polytope map onto a point **j** but rather whether there is such a point, the Ehrhart test can also map the integer-valued multiplicity onto a Boolean value.

2.A Homogeneous representation of polyhedra

The non-parameterized polyhedra seen so far describe a system of *inhomogeneous* equations and inequalities. They can be, however, represented as a system of *homogeneous* equations and inequalities. This is done by transforming the polytope

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n | A\mathbf{x} = \mathbf{b} \wedge C\mathbf{x} \geq \mathbf{d}\} \tag{2.22}$$

into the cone

$$\mathcal{C} = \{\begin{bmatrix}\mathbf{x}\\ t\end{bmatrix} \in \mathbb{Q}^{n+1} | A\mathbf{x} - \mathbf{b}t = \mathbf{0} \wedge C\mathbf{x} - \mathbf{d}t \geq \mathbf{0} \wedge t \geq 0\} \tag{2.23}$$

In [74] it is shown that this transformation is bijective (one-to-one correspondence) and inclusion-preserving [5].

The inhomogeneous polyhedron $\mathcal{P}$ can be obtained from its homogeneous representation via the inverse transformation. This is done by intersecting $\mathcal{C}$ with the hyper-plane $t = 1$ and projecting this intersection onto the data space

$$\mathcal{P} = \{\mathbf{x} \mid \begin{bmatrix}\mathbf{x}\\ 1\end{bmatrix} \in \mathcal{C}\} \tag{2.24}$$

Let $\mathcal{P}$ be as in (2.3). By combining (2.4) and (2.22), $\mathcal{P}$ is represented in the homogeneous non-parameterized form by

$$\begin{aligned} \mathcal{C} &= \{\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ t\end{bmatrix} \in \mathbb{Q}^{n+m+1} \mid \begin{bmatrix}A & -B & -\mathbf{b}\end{bmatrix} \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ t\end{bmatrix} = \mathbf{0} \wedge \begin{bmatrix}C & -D & -\mathbf{d}\\ \mathbf{0}^T & \mathbf{0}^T & 1\end{bmatrix} \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ t\end{bmatrix} \geq \mathbf{0}\} \\ &= \{\tilde{\mathbf{x}} \in \mathbb{Q}^{n+m+1} \mid \tilde{A}\tilde{\mathbf{x}} = \mathbf{0} \wedge \tilde{C}\tilde{\mathbf{x}} \geq \mathbf{0}\} \end{aligned} \tag{2.25}$$

where $\tilde{\mathbf{x}} = \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ t\end{bmatrix}$, $\tilde{A} = \begin{bmatrix}A & -B & -\mathbf{b}\end{bmatrix}$, and $\tilde{C} = \begin{bmatrix}C & -D & -\mathbf{d}\\ \mathbf{0}^T & \mathbf{0}^T & 1\end{bmatrix}$.

[5] This means that $\mathcal{P}_1 \subset \mathcal{P}_2$ implies $\mathcal{C}_1 \subset \mathcal{C}_2$ and vice versa. From this property it follows that operations on polytopes, such as intersection, can be performed on their cones.

Theorem 2.7
Let $\mathcal{P}$ be a parameterized polyhedron as defined in (2.3), and let $\mathcal{C}$ be the homogeneous system that corresponds with the non-parameterized representation $\mathcal{P}'$ of $\mathcal{P}$, then $\mathcal{P}$ can be obtained from $\mathcal{C}$ as

$$\mathcal{P} = \{\mathbf{x} | \begin{bmatrix}\mathbf{x}\\ \mathbf{p}\\ 1\end{bmatrix} \in \mathcal{C}\} \tag{2.26}$$

Proof
Let $\mathcal{P}'$ be the non-parameterized polyhedron that represents $\mathcal{P}$ and let $\mathcal{C}$ be the homogeneous representation of $\mathcal{P}'$. From (2.24) we find

$$\mathcal{P}' = \{\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} \mid \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ 1\end{bmatrix} \in \mathcal{C}\} \tag{2.27}$$

Substituting (2.27) into (2.5) gives

$$\begin{aligned}\mathcal{P} &= \{\mathbf{x} \mid \begin{bmatrix}\mathbf{x}\\ \mathbf{p}\end{bmatrix} \in \{\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} \mid \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ 1\end{bmatrix} \in \mathcal{C}\}\}\\ &= \{\mathbf{x} \mid \begin{bmatrix}\mathbf{x}\\ \mathbf{p}\\ 1\end{bmatrix} \in \mathcal{C}\}\end{aligned} \tag{2.28}$$

■

A parameterized polyhedron $\mathcal{P}$ is completely characterized by the triple $(A, C, \mathbf{p})$ where A and C are the matrices $\tilde{A}$ and $\tilde{B}$ in (2.25). We use the notation $\mathcal{P} = (A, C, \mathbf{p})$ as a shorthand.

Note that the part $(\mathbf{x}, \mathbf{p}, 1) \in \mathcal{C}$ in (2.26) is the intersection of $\mathcal{C}$ with the hyper-planes $t = 1$ and $\mathbf{y} = \mathbf{p}$. Call this intersection $\mathcal{P}^+$

$$\mathcal{P}^+ = \{\begin{bmatrix}\mathbf{x}\\ \mathbf{p}\\ 1\end{bmatrix} \in \mathcal{C}\} \tag{2.29}$$

Now (2.26) can be rewritten in

$$\mathcal{P} = \{\mathbf{x} = L\mathbf{z} \,|\, \mathbf{z} \in \mathcal{P}^+\}. \tag{2.30}$$

where L is a matrix of the form $\begin{bmatrix}L_1 & L_2\end{bmatrix}$, with L_1 an $n \times n$ identity matrix and L_2 an $n \times (m+1)$ zero matrix. n is the dimension of the space containing $\mathcal{P}$ and m is the number of parameters.

Example 2.6 (parameterized polytope represented by cone) Consider the parameterized polytope $\mathcal{P} = \{i \in \mathbb{Q} \,|\, 1 \leq i \leq N + 1\}$, depending on the parameter N. The homogeneous system, as in (2.25), becomes $\mathcal{C} = \{(i, y, t) \in \mathbb{Q}^3 \,|\, t \leq i \leq t + y, t \geq 0\}$ where y is the free variable version of parameter N. Figure 2.9 shows the cone together with the intersections with the hyper-planes $t = 1$ and $y = N$. The result of the intersections gives the polytope

$$\{\begin{bmatrix}i\\ N\\ 1\end{bmatrix} \in \mathbb{Q}^3 \,|\, 1 \leq i \leq 1 + N\}.$$

Projecting this polytope onto $\mathbb{Q}^1$ gives back the original polytope. ■

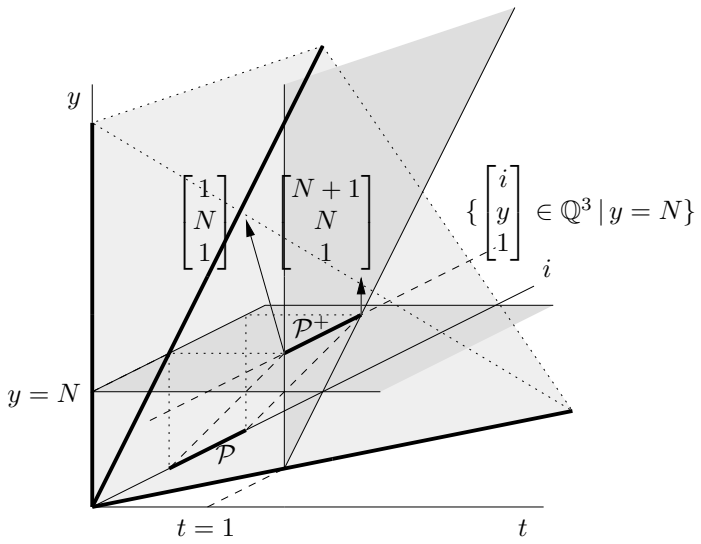


Figure 2.9: Parameterized polytope represented as non-parameterized polyhedron in the homogeneous form. The light shaded region is the cone $\mathcal{C}$. The dark shaded planes are the hyper-planes $t = 1$ and $y = N$; their intersection is the set $\{[i \;\; y \;\; 1]^T \in \mathbb{Q}^3 \,|\, y = N\}$. The short bold lines represent $\mathcal{P}^+$ and its projection $\mathcal{P}$ onto the i-axis.

Chapter 3

Modeling and analysis of piece-wise regular programs

In Section 1 I have argued that it is beneficial to have an intermediate application model. This application model must enable us to derive Kahn process networks in order to perform design space exploration or mapping onto a target architecture. This section defines the intermediate model that is used in this dissertation, the Polyhedral Reduced Dependence Graph (PRDG), and how a special class of nested loop programs is converted into the PRDG model.

As indicated in Figure 1.3 our flow starts with programs written in Matlab. The first tool in the tool chain is MATPARSER. The approach in this dissertation is that all programs that MATPARSER accepts are considered valid programs for the other tools. The set of iterations traversed by these programs can always be described by periodic lattice polyhedra, but not always by $\mathbb{Z}$-polyhedra, see Section 2.5. In this dissertation we take the general approach by defining these iteration sets as periodic lattice polyhedra, although it might be more efficient from a computational point of view to use $\mathbb{Z}$-polyhedra, whenever possible, instead. The use $\mathbb{Z}$-polyhedra is seen as an optimization which might be considered in the future.

Section 3.1 extends the usual definition of a graph to a graph that also contains ports. This extended graph definition defines the topology of the PRDG model that is defined in Section 3.2. Section 3.3 defines the class of piece-wise affine nested loop programs in terms of their statements and how they are represented as parse trees. Section 3.4 describes the conversion from parse trees to PRDGs.

3.1 Graphs

In this section I summarize some definitions for directed graphs from [75]. The terms introduced in this section may have some different meaning for undirected graphs, like trees, and I shall mention such differences when needed.

A *directed graph* (or *digraph*) G is a pair $G = (V, A)$, where V is a finite set and A is a binary relation on V. The set V is called the *vertex set* of G. The elements of V are called *vertices*. The set A is called the *arc set* of G. The elements of A are called the *arcs*. When we view a graph pictorially, vertices are represented by circles and arcs are represented by arrows. Note that *self-loop*s – arcs from a vertex to itself – are possible.

If (u, v) is an arc in a directed graph $G = (V, A)$, we say that vertex v is adjacent to vertex u and that arc

(u, v) is *incident from* vertex u and *incident to* vertex v.

A *path* of *length* k from a vertex u to a vertex u' in a directed graph $G = (V, A)$ is an ordered set $(v_0, v_1, \cdots, v_k)$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in A$ for $i = 1, 2, \cdots, k$. The length of the path is the number of arcs in the path. A path is *simple* if all vertices in the path are distinct. The path is said to *contain* the vertices $v_0, v_1, \cdots, v_k$ and the arcs (v_0, v_1), (v_1, v_2), $\cdots$, (v_{k-1}, v_k). A path $(v_0, v_1, \cdots, v_k)$ forms a *cycle* if $v_0 = v_k$ and the path contains at least one arc. A self-loop is a cycle of length one. A graph with no cycles is called *acyclic*.

The standard definition of a graph does not allow the graph $G = (V, A)$ to have a pair (u, v) to be present in the arc set more than once. Moreover, even if this was allowed (as in *multi-graphs*), the model provides no handles to associate behavior with the vertices unambiguously. Therefore I extend the notion of (multi-)graph and introduce the *extended graph* structure. In an extended graph nodes have ports and edges connect pairs of ports. In this way the internal behavior of each node can refer unambiguously to its ports.

Let S be a set. A collection S^* of nonempty subsets of S is said to be a *partition* of the set S if every element in S is contained in exactly one member of S^*. An *ordered partition* is a partition whose members are ordered sets. Let S^* be a set of pairwise disjoint (ordered) sets, we write S to denote the set of which S^* is a (ordered) partition.

Definition 3.1 (node)
A *node* is a pair (I, O) consisting of two disjoint ordered sets of ports, input ports (I) and output ports (O), respectively, with $I \cap O = \emptyset$. Let n be a node, then we write $n = (I_n, O_n)$ and call I_n and O_n the *input port set* and *output port set* of n, respectively. The union of I_n and O_n is called the *port set*, denoted by P_n, of node n. ■

Definition 3.2 (edge)
Let I and O be sets of input ports and output ports, respectively, such that $I \cap O = \emptyset$. A directed *edge* is a pair (o, i) where $o \in O$ and $i \in I$. ■

Definition 3.3 (extended graph)
An extended graph is a pair $G = (N, E)$ where N is a set of nodes and E is a binary relation from $O = \cup_{n \in N} O_n$ to $I = \cup_{n \in N} I_n$ represented by a set of edges such that $(O \cup I, E)$ is a graph with the property that for every port $i \in I$ there is at most one port $o \in O$ for which $(o, i) \in E$. N is called the *node set* of G and E is called the *edge set* of G. ■

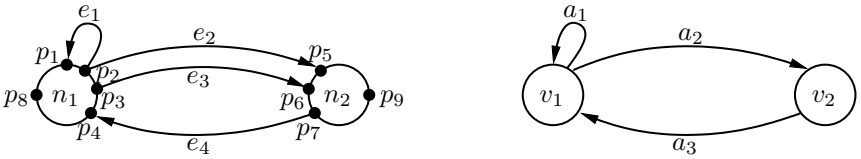
Pictorially we view an extended graph with circles for the nodes, arrows for the edges, and dots for the ports which are placed on the circumferences of the nodes to which they belong.

The graph G' of an extended graph $G = (N, E)$ is the graph $G' = (V, A)$ where V is the set of vertices obtained by associating every node $n \in N$ with a unique vertex v. We write $node(v)$ to denote the node v is associated with. The arc set of G' is the set $A = \{(u, v) \in V^2 \mid \exists (o, i) \in E$, such that $i \in I_{node(u)} \wedge o \in O_{node(v)}\}$.

Let G be an extended graph. G is said to be *acyclic* when its G' is acyclic. An ordered set of k nodes $(n_1, n_2, \cdots, n_k)$ of G form a *k-cycle* in G if the ordered set of vertices $(v_1, v_2, \cdots, v_k)$ associated with the nodes form a k-cycle in G'. A 1-cycle in G is called a *self-loop*.

Example 3.1 Let G be the extended graph (N, E) , $N = \{n_1, n_2\} = \{(\{p_1, p_4, p_8\}, \{p_2, p_3\}), (\{p_5, p_6\}, \{p_7, p_9\})\}$, and $E = \{e_1, e_2, e_3, e_4\} = \{(p_2, p_1), (p_2, p_5), (p_3, p_6), (p_7, p4)\}$. Figure 3.1 gives a pictorial view of G and its graph G'.

In next section a computational model, known as the dependence graph, is defined. In order to do so, *nodes* and *edges* are redefined to take behavior into account. Moreover, *ports* are specialized to *input ports* and *output ports* which also have behavior.



(a) extended graph G (b) graph G'

Figure 3.1: Example of an extended graph (a) and its graph (b). ■

3.2 Polyhedral Reduced Dependence Graphs

Previous section introduced the notion of extended graph. By adding behavior to the nodes of the extended graph and by adding semantics to the nodes, ports and edges, we obtain a *computational network*. This section describes two types of computational networks, viz., *dependence graph*s (DGs) and *polyhedral reduced dependence graph*s (PRDGs).

3.2.1 Dependence graphs

In a dependence graph (DG), ports, nodes, and edges have attributes added to their topological counterparts in extended graphs; ports are labeled, nodes encapsulate functionality and edges are dependence relations.

A DG is an acyclic extended graph in which nodes are associated with operations onto variables and edges are associated with communications of variables. An operation is a function that is encapsulated in a node. A function receives its input arguments via *input ports* of the node, and releases its return values via *output ports* of the node. From now on I shall use the term *output arguments* for the return values of a function. The input arguments together with the output arguments of a function are the arguments of the function. Output ports and input ports of nodes are connected via edges. These edges specify the communication of variables between the ports of the nodes.

Definition 3.4 (input port)
An *input port* is a triple (a, v, t) consisting of a name a, a variable v and a *type* t. The type represents the set of values which can be assigned to the variable v. The name is used to bind the port to an input argument of a function. ■

Definition 3.5 (output port)
An *output port* is a triple (a, v, t) consisting of a name a, a unique variable v and a *type* t. The type represents the set of values which can be assigned to the variable v. The name is used to bind the port to an output argument of a function. ■

The *domain* $P^\times$ of an ordered set of ports $P = (p_0, p_1, \cdots, p_{|P|-1})$ is the *product set* of the types of the ports in P. A port is either active or passive; it is active when to its variable v a value of type t is assigned, and it is passive otherwise. Initially a port is always passive.

Definition 3.6 (node)
A *node* is a triple $n = (I_n, O_n, f_n)$, where I_n is an ordered set of input ports, O_n is an ordered set of output ports, and $f_n : I_n^\times \rightarrow O_n^\times$ is a function encapsulated in the node. The input arguments of the input ports in I_n are bound to the input arguments of f_n by name. Similarly, the output arguments of the output ports in O_n are bound to the output arguments of f_n by name. A node is either enabled or disabled (blocking); it is enabled when all its input ports are active and all its output ports are passive and it is blocking otherwise. When a node is enabled it fires, i.e., all values assigned to the variables of the node's

input ports are taken, making them passive, and are passed to the input arguments of the node's function. The function is evaluated and its return values are assigned to the variables of the node's output ports, one for every port, making them active. ■

Definition 3.7 (edge)
An edge is a pair (o, i), where o and i are an output port and input port, respectively. All edges incident from some active output port o copy the value assigned to o's variable to the ports the edges are incident to. ■

The above definitions of node, edge, input port, and output port define a computational model whose underlying structure is defined by an extended graph. A node fires when all its input ports are active. The activation of input ports of some node n may depend on some other node m, n is adjacent to. So, the dependency between nodes is specified by the edges of the graph.

Definition 3.8 (dependence graph)
A dependence graph is an acyclic extended graph $G = (N, E)$, specialized in the sense that N is a set of nodes $\{(I_n, O_n, f_n)\}_{n \in N}$ where I_n and O_n are sets of input and output ports, respectively, and E is a set of edges defining a binary relation from $O = \cup_{n \in N} O_n$ to $I = \cup_{n \in N} I_n$. ■

Note that a dependence graph is being acyclic by definition. If we would have allowed a cycle to be present in such a graph, all nodes in the cycle would be be blocked initially and forever – because all ports are passive initially, and would be blocked forever, since the ports in the cycle could only be activated when a node in the cycle would fire.

Moreover, because a dependence graph is acyclic, every port is activated at most once, and consequently a value is assigned to every variable at most once. This property is referred to as the *single assignment* property. Computer programs having the single assignment property are called *single-assignment program*s (SAPs). For single-assignment programs that have static control, it is well known that they may be represented by a dependence graph [76]. This property will be further discussed in Chapter 4.

3.2.2 Polyhedral Reduced Dependence Graphs

A dependence graph describes the topology of the dependencies between the functions of its nodes, and hides any additional regularity in the structure that may be present. However, often dependence graphs of nested loop programs are of interest. The loops in nested loop programs represent repetitions of the statements in the program in a compact manner. If loops are specified by a set of affine inequalities, then there is an underlying regularity that can be exploited to reduce the polyhedral geometry in the DG which is the result of the affine loop property.

Note: in previous chapter I used the word *domain* to denote the set a function is defined upon, exclusively. For example, the periodic lattice polytope $\mathcal{I} = L(\mathcal{P} \cap \mathbb{Z}^d)$ describes a set of points defined by the projection $L()$ on the domain $\mathcal{P} \cap \mathbb{Z}^d$. Confusingly, literature often denotes the set $\mathcal{I}$ as a domain. To avoid this confusion, I use the term *index set* to describe the sets of integral points. These sets may be defined by $\mathbb{Z}$-polyhedra, LBLs, or periodic lattice polyhedra.

Definition 3.9 (input port domain)
Let $\mathcal{I}_p$ be an index set defined by a periodic lattice polyhedron and let p be an input port. An input port domain is a set of indexed ports $p_{\mathbf{i}}$ given by

$$\{p_{\mathbf{i}} \,|\, \mathbf{i} \in \mathcal{I}_p \wedge p_{\mathbf{i}} = (p, \mathbf{i})\} \tag{3.1}$$

where $(p, \mathbf{i})$ represents an instance of port p at the index point $\mathbf{i}$. As a shorthand for (3.1) we write $P = (p, \mathcal{I}_p)$. ■

Definition 3.10 (output port domain)
Let $\mathcal{I}_q$ be an index set defined by a periodic lattice polyhedron and let q be an output port. An output port domain is a set of indexed ports $q_{\mathbf{j}}$ given by

$$\{q_{\mathbf{j}} \,|\, \mathbf{j} \in \mathcal{I}_q \wedge q_{\mathbf{j}} = (p, \mathbf{j})\} \tag{3.2}$$

where $(q, \mathbf{j})$ represents an instance of port q at the index point $\mathbf{j}$. Let d be the dimension of the space that contains $\mathcal{I}_q$, then the set of variables associated with the ports in an output port domain are all contained in a d-dimensional variable array v. The variable associated with the port $q_{\mathbf{j}}$ is $v(\mathbf{j})$. As a shorthand for (3.2) we write $Q = (q, \mathcal{I}_q)$. ■

Definition 3.11 (node domain)
Let $\mathcal{I}_n$ be an index set defined by a periodic lattice polyhedron and let n be a node. A *node domain* is a set of indexed nodes $n_{\mathbf{i}}$ given by

$$\{n_{\mathbf{i}} | \mathbf{i} \in \mathcal{I}_n \wedge n_{\mathbf{i}} = (n, \mathbf{i})\}. \tag{3.3}$$

where $(n, \mathbf{i})$ represents an instance of node n at the iteration point $\mathbf{i}$. ■

How attractive Definition 3.11 seems, it makes sense to come up with an alternative representation of a node domain. This is so because the definition above may be interpreted as computation centric, that is, at every point $\mathbf{i} \in \mathcal{I}$ the node $n = (I, O, f)$ is present. In this dissertation the main concern is communication. Therefore, the alternative representation that is given below is communication centric, that is, the node domain in (3.3) is represented in terms of a set of input port domains I_N, a set of output port domains O_N, a function f_n, and an index set $\mathcal{I}_n$.

Let the node in (3.3) have K input ports and L output ports with input port set $I_n = (p_1, p_2, \cdots, p_K)$ and output port set $O_n = (q_1, q_2, \cdots, q_K)$, respectively. Substitution of the port sets into $n = (I_n, O_n, f_n)$ into (3.3) results in

$$N = \{((p_1, p_2, \cdots, p_K), (q_1, q_2, \cdots, q_L), f_n)_{\mathbf{i}} \,|\, \mathbf{i} \in \mathcal{I}\} \tag{3.4}$$

Rather then indexing the nested tuples in (3.4), the indexing can be moved within the nesting. This means that $p_k, k = 1, 2, \cdots, K$ is replaced by $\{p_k, \mathbf{i} \,|\, \mathbf{i} \in \mathcal{I}\}$ and that $q_\ell, \ell = 1, 2, \cdots, L$ is replaced by $\{q_\ell, \mathbf{i} \,|\, \mathbf{i} \in \mathcal{I}\}$. By using the shorthands $(p_k, \mathcal{I})$ and $(q_\ell, \mathcal{I})$ for these replacements we get

$$\begin{aligned} N =& (((p_1, \mathcal{I}), (p_2, \mathcal{I}), \cdots, (p_K, \mathcal{I})), ((q_1, \mathcal{I}), (q_2, \mathcal{I}), \cdots, (q_L, \mathcal{I})), \{f_{n,\mathbf{i}} | \mathbf{i} \in \mathcal{I}\}) \\ =& ((P_1, P_2, \cdots, P_K), (Q_1, Q_2, \cdots, Q_L), f_n, \mathcal{I}_n) \end{aligned} \tag{3.5}$$

where $P_k = (p_k, \mathcal{I}), k = 1, 2, \cdots, K$ are input port domains and $Q_\ell = (q_\ell, \mathcal{I}), \ell = 1, 2, \cdots, L$ are output port domains of the node domain.

Because (3.4) is a set of nested tuples and (3.5) is a nested tuple of sets, (3.4) and (3.5) are not equivalent. However, one can derive the (3.5) from the (3.4), as is done above, and vice versa.

Now, any of the P_k and Q_ℓ may be partitioned by partitioning their defining index set $\mathcal{I}$. For every $k = 1, 2, \cdots, K$ there is a partitioning $\mathcal{I}_k^*$ of $\mathcal{I}$, $\mathcal{I}_k^* = \{\mathcal{I}_{k,1}, \mathcal{I}_{k,2}, \cdots, \mathcal{I}_{k,u_k}\}$, where u_k is the number of partitions in $\mathcal{I}_k^*$. For every $\ell = 1, 2, \cdots, L$ there is a collection (not necessary a partitioning) $\mathcal{J}_\ell^*$ of $\mathcal{I}$, $\mathcal{J}_\ell^* = \{\mathcal{J}_{\ell,1}, \mathcal{J}_{\ell,2}, \cdots, \mathcal{J}_{\ell,v_\ell}\}$, where v_ℓ is the number of sets in $\mathcal{J}_\ell^*$. The input port p_k and the partitions $\mathcal{I}_{k,i}, i = 1, 2, \cdots u_k$ define a partitioning of the input port domains P_k, $P_{k,i} = (p_k, \mathcal{I}_{k,i})$. The output port

q_ℓ and the index sets $\mathcal{J}_{\ell,j}, j = 1, 2, \cdots, v_\ell$ define a set of (possibly overlapping) output port domains Q_ℓ, $Q_{\ell,j} = (q_\ell, \mathcal{J}_{\ell,j})$. The resulting node domain representation is

$$\begin{aligned} N &= ((P_{1,1}, P_{1,2}, \cdots P_{1,u_1}, \cdots, P_{K,u_K}), (Q_{1,1}, Q_{1,2}, \cdots, Q_{1,v_1}, \cdots, Q_{L,v_L}), f_n, \mathcal{I}_n) \\ &= (I_N, O_N, f_n, \mathcal{I}_n) \end{aligned} \tag{3.6}$$

where I_N is the ordered set of input port domains $(P_{1,1}, P_{1,2}, \cdots P_{1,u_1}, \cdots, P_{K,u_K})$ and O_N is the ordered set of output port domains $(Q_{1,1}, Q_{1,2}, \cdots, Q_{1,v_1}, \cdots, Q_{L,v_L})$. From now on a node domain is represented by the tuple $(I_N, O_N, f_n, \mathcal{I}_n)$. Note that this is an extension of the node in the extended graph with the pair $(f_n, \mathcal{I}_n)$.

Input port domains and output port domains are related by edge domains. A compact way of representing a set of indexed edges is to specify a function from the index set of the input ports to the index set of the output ports.

Definition 3.12 (edge domain)
Let $P = (p, \mathcal{I}_p)$, with $\mathcal{I}_p = (L_p, \mathcal{P}_p)$, and $Q = (q, \mathcal{I}_q)$ be an input port domain and an output port domain, respectively. An edge domain is an indexed set of edges $e_{\mathbf{ij}}$ given by

$$\{(e_{\mathbf{ij}} = (q_{\mathbf{j}}, p_{\mathbf{i}}) \,|\, \mathbf{i} = L_p\mathbf{k} \wedge \mathbf{j} = D\mathbf{k} \wedge \mathbf{k} \in \mathcal{P}_p\} \tag{3.7}$$

where $p_{\mathbf{i}} \in P$ and $q_{\mathbf{j}} \in Q$. As a shorthand for (3.7) we write $E = (e, \mathcal{I}_e)$, where $e = (Q, P)$ and $\mathcal{I}_e = (D, \mathcal{P}_p)$. ■

The definitions of input port domain, output port domain, node domain, and edge domain are used to reduce the representation of a dependence graph. Rather than enumerating all nodes, edges, and ports in the DG, the reduced representation contains a collection of node domains, edge domains and port domains.

I have shown that an input port domain, output port domain, node domain, and edge domain is obtained by associating an index set with an input port, output port, node, and edge. Because these objects are attributed versions of the objects in extended graphs, a set of input port domains, output port domains, node domains, and edge domains construct an attributed version of an extended graph as well.

Definition 3.13 (polyhedral reduced dependence graph)
A polyhedral reduced dependence graph is a reduced representation of a dependence graph. It is an extended graph $G = (\mathcal{N}, \mathcal{E})$, specialized in the following way

- every node $N \in \mathcal{N}$ has an extra pair $(f_n, \mathcal{I}_N)$ associated with it, that is, it is a node domain $N = (I_N, O_N, f_n, \mathcal{I}_n)$, where I_N and O_N are sets of input port domains and output port domains, respectively, and
- every edge $E \in \mathcal{E}$ has an extra index set $\mathcal{I}_e$ associated with it, that is, it is an edge domain $E = (e, \mathcal{I}_e)$, where $e = (Q, P), Q \in O_N \wedge P \in I_M$, where O_N and I_M are the output port domain set and input port domain set of node domains N and M, respectively.

In addition, the collection of sets of nodes in N, the collection of sets of edges in E, and the collection of sets of input ports in I_N are pairwise disjoint over all N. ■

3.3 Parse tree representation of single-assignment programs

This section describes the relation between a nested loop program, its single-assignment program, and its parse tree representation. The next section describes how single-assignment programs are converted into polyhedral reduced dependence graphs.

A single-assignment program is a computer program in which to every variable, a value is assigned only once [77]. Such programs may be written by hand or may be the result of an *array dataflow analysis* of an imperative program. An array dataflow analysis is the analysis that finds all flow dependencies in a program [78].

One such array dataflow analysis compiler is MATPARSER [13]. The input of MATPARSER is an imperative program expressed in a subset of the MATLAB language which is described in sections 3.3.1 – 3.3.3. The output of MATPARSER is either a single-assignment program, or a parse tree representing a single-assignment program. In this dissertation I assume that a single-assignment program is generated with MATPARSER and that it is represented by a parse tree with annotated nodes.

Nested loop programs are well represented by parse trees. In this dissertation I focus on a specific class of nested loop programs, the so-called *parameterized piece-wise affine nested loop programs* [12, 79] which are precisely the programs accepted by the MATPARSER tool. An example of a parameterized piece-wise affine nested loop program and its single-assignment program is given in Program 3.1 and Program 3.2, respectively. The initialization of the array a has been omitted for brevity purposes.

Parameterized piece-wise affine NLPs can be expressed as parameterized piece-wise affine SAPs[1], using statements specified in subsections 3.3.1–3.3.3. Single-assignment programs contain two types of statements: Assignment statements and control statements.

Program 3.1: EXAMPLE NLP

```
%parameter N 10 20;
%parameter M 10 20;
for i = 1 : 1 : N,
    for j = 1 : 1 : M,
        [a(i + j)] = f(a(i + j));
    end
end
```

Program 3.2: EXAMPLE SAP

```
%parameter N 10 20;
%parameter M 10 20;
for i = 1 : 1 : N,
    for j = 1 : 1 : M,
        if i − 2 ≥ 0,
            if j ≤ M − 1,
                [in0] = ipd (a1(i − 1, j + 1));
            else
                [in0] = ipd (a(i + j));
            end
        else
            [in0] = ipd (a(i + j));
        end
        [out0] = f(in0);
        [a1(i, j)] = opd (out0);
    end
end
```

3.3.1 Assignment statements

Assignment statements occur in of the functional part of the program (as opposed to the control part of the program); they deal with the assignment of values to variables, the binding of variables to arguments of functions, and function calls. There are three types of assignment statements, viz., **ipd** statements, **opd** statements, and **node** statements. They have the following semantics and syntax:

- **ipd** statement. This is a variable-to-argument binding. An example of an **ipd** statement is "$[in_0] =$

[1] That is, the property "piece-wise affine" is preserved. A SAP is piece-wise affine if a PRDG for it exists.

ipd $(a(i+j))$;" in Program 3.2. The syntax of the **ipd** statement is:

$$[\textit{arg}] = \textbf{ipd}\ (\textit{var}(\textit{index}));$$

where *arg* is a string representing the input argument of a function, *var* is the variable array name, and *index* is an integral affine function of index variables, control variables, and parameters (all three will be defined later). This function is called the *indexing function* or *dependence function*. For example, the *index* in $[in_0] =$ **ipd** $(a_1(i-1, j+1))$ is the integer affine function $\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}i\\ j\end{smallmatrix}\right] + \left[\begin{smallmatrix}-1\\ 1\end{smallmatrix}\right]$.

- **opd** statement. This is an argument-to-variable binding. An example of an **opd** statement is "$[a_1(i,j)] =$ **opd** (out_0);" in Program 3.2. The syntax of the **opd** statement is:

$$[\textit{var}(\textit{index})] = \textbf{opd}\ (\textit{arg});$$

where *arg* is a string representing the output argument of a function, *var* is the variable array name, and *index* is an integral affine function, the indexing or dependence function.

- **node** statement. This is an assignment statement containing a function call. An example of a **node** statement is "$[out_0] = f(in_0)$;" in Program 3.2. The syntax of the **node** statement is:

$$[\textit{arg}, \textit{arg}, \cdots, \textit{arg}] = \textit{function}(\textit{arg}, \textit{arg}, \cdots, \textit{arg});$$

where $[\textit{arg}, \textit{arg}, \cdots, \textit{arg}]$ is the list of output arguments of the function with name *function* and where $(\textit{arg}, \textit{arg}, \cdots, \textit{arg})$ is the list of input arguments of the function.

3.3.2 Control statements

Control statements describe the structure of a SAP. There are four types of control statements: the **for** statement, the **conditional** statement, the index **transformation** statement, and the **parameter** declaration statement. The four control statements have the following semantics and syntax:

- **for** statement. This statement is used to specify repetitive execution of a set of statements, called *body*. The syntax of the **for** statement is:

  for *var* = *expression* : *integer* : *expression*,
  body
  end

  where *var* is a variable called the *loop variable* or *index variable*. The left hand side *expression* and the right hand side *expression* are the *lower bound* and upper bound of the loop, respectively. The $integer$ is an integral value, called the *stride* or *step size* of the loop. The first line of the **for** statement is called the *header* of the loop. Let the header of the loop be **for** $i = \ell : s : u$, with s being an integer. The loop index i specifies a set of index points $\mathcal{I} = \{\ell + ks\}_{k=0}^{p}$, where p is the largest integer for which $\ell + ps \leq u$. For every $i \in \mathcal{I}$, *body* is executed once.

- **conditional** statement. This statement is used to specify the conditional execution of one of two sets of statements. The syntax of the **conditional** statement is:

  if *expression* ≥ 0,
  if-body
  else
  else-body
  end

where the expression *expression* ≥ 0 is called the *condition*. Expressions of the form $e_1 \leq e_2$ or $e_1 \geq e_2$ can be written in the default form as $e_2 - e_1 \geq 0$ and $e_1 - e_2 \geq 0$, respectively. If the condition is true, then the *if-body* is executed, else the *else-body* is executed. The **else** keyword together with the *else-body* are optional.

- **index transformation** statement. This statement is used to define *control variable*s that can be used in the expressions of **conditional** and **for** statements. The syntax of the **index transformation** statement is:

  var $=$ *expression*;

  where *var* is the control variable defined and where *expression* is a pseudo-affine expression, see Subsection 3.3.3.

- **parameter** declaration statement. This statement is used to declare parameters. The syntax of the **parameter** declaration statement is:

  %**parameter** *var* $integer$ $integer$;

  where *var* is the parameter and the left and right hand side $integer$ specify the *range* of values the parameter can be given. This range is the *context* of the parameter.

3.3.3 Type of expressions in statements

All statements except the **node** statement contain expressions. There are two types of expressions: *affine expressions* and *pseudo-affine expressions*. The type of expression allowed in the statements differs per statement. An affine expression is a multivariate polynomial of degree one

$$a_1 k_1 + a_2 k_2 + \cdots + a_{n-1} k_{n-1} + a_n \tag{3.8}$$

where the a_is are rational coefficients and the k_is are integer variables. When an affine expression evaluates to an integer value for all integer k_is we say that the expression is *integral*.

A pseudo-affine expression extends the affine expression in the sense that at least one of its terms is pseudo-linear:

$$a_1 c_1 + a_2 c_2 + \cdots + a_{n-1} c_{n-1} + a_n \tag{3.9}$$

where the a_is are rational coefficients and at least one c_i an integer division function, the other c_is being integer variables. In case c_i is an integer division function it takes the form of one of the equations 3.10-3.13.

$$c_i = \mathrm{mod}(e_i, b_i) \tag{3.10}$$
$$c_i = \mathrm{div}(e_i, b_i) \tag{3.11}$$
$$c_i = \mathrm{floor}(e_i) \tag{3.12}$$
$$c_i = \mathrm{ceil}(e_i) \tag{3.13}$$

where e_i is an integral affine expression and b_i is a strictly positive integer. Moreover, in (3.10) and (3.11) e_i must have integral coefficients. Pseudo-affine expressions are found in

- lower and upper bounds of **for** statements,
- conditions in **conditional** statements, and
- expressions in **index transformation** statements.

All other expressions are affine expressions.

One more remark about the expressions in statements of a SAP is in order. The statement must be in the scope of all variables used in its expressions. For example, it is not allowed that the index variable of a **for** statement is used in expressions outside the loop body.

3.3.4 Properties of single-assignment programs

The code produced by MATPARSER exhibits two important properties:

Property 1: The code has new variable arrays introduced for each variable array in the original program to which values are potentially assigned during the execution of the program. Potentially, because it can be the case that assignment statements appear in dead code or code that is dead depending on parameter settings. The names of the newly introduced variable arrays have in addition to the original variable array name a unique subscript.

Property 2: The code is in so called *output normal form*, this means that the variables in the left hand side of each **opd** statement are fully indexed, that is, if $(i_1, i_2, \cdots, i_n)$ is the tuple of loop indices of the loops enclosing the **ipd** statement, in outer loop to inner loop order, then the left-hand side variable of the **opd** statement is indexed by the same tuple. For example, the statement $[a_1(i,j)] =$ **opd** (out_0); – in Program 3.2 – is fully indexed.

Lemma 3.1
Programs produced by MATPARSER have the single assignment property.

Proof
Because all variable names in the statements of the program are unique (property 1), multiple assignments can only take place in a single statement, and because the program is in output normal form (property 2), the repetition of the execution of that statement goes with a unique index set, such that no multiple assignment can occur. ■

As we will see, these two properties are important when interpreting the SAP.

3.3.5 Parse trees

In the parse tree representation of a SAP, the statements are associated with the nodes in the parse tree. These statements must be seen as parse subtrees themselves, embedded in the parse tree and described by the syntax of the statements. In this way the parse tree as described here corresponds to the parse tree in [80]. For the complete grammar of the parse trees see [13]. The topology of the parse tree represents the control structure of the program. The parse tree representation of Program 3.2 is given in Figure 3.2.

The node labeled with the "@" symbol is the *root* of the tree. The parse tree is an ordered tree and must be interpreted depth first, left to right. In an ordered tree the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, $\cdots$, and a k-th child [75]. When we draw a parse tree, the children of each node are drawn in a left-to-right order.

The parse tree is constructed is such way that the scope of each control variable, loop variable, and parameter in the SAP corresponds to the set of subtrees rooted by the children of the statement defining the variable. For this reason, the **parameter** declaration statements are the ancestors of all other statements. As an example, consider the program in Figure 3.3 (a). The program contains a **parameter** declaration statement and two **index transformation** statements. Figure 3.3 (b) shows the parse tree of the program and illustrates that the scope of parameter N is the complete program. The parse tree also illustrates that

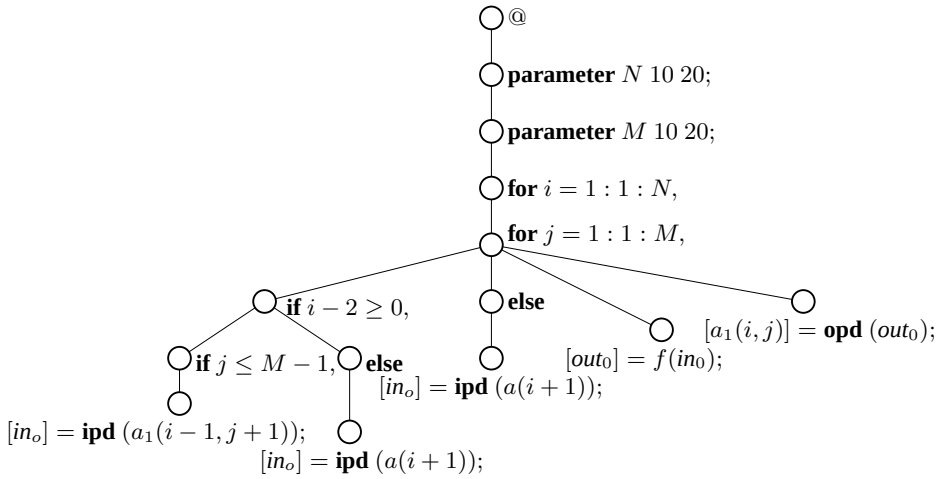
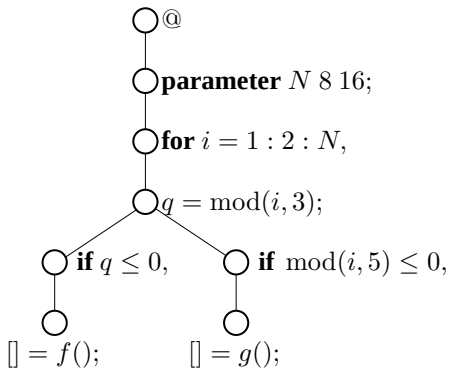


Figure 3.2: Parse tree of Program 3.2.

the **index transformation** statement "$q = \mathrm{mod}(i, 3)$;" is the parent of the two **if** statements that follow it in the program, rather than their left sibling.

```
%parameter N 8 16;
for i = 1 : 2 : N,
    q = mod(i, 3);
    if q ≤ 0,
        [] = f();
    end
    if mod(i, 5) ≤ 0,
        [] = g();
    end
end
```

(a) SAP



(b) parse tree

Figure 3.3: A SAP with two index transformation statements (a) and its parse tree (b).

From now, on the term NLP is used to denote parameterized piece-wise affine nested loop programs, and the term "SAP" is used to denote the single-assignment program of an NLP generated with MATPARSER as code or parse tree. The next step is to convert a SAP into a polyhedral reduced dependence graph. For more information on the class of nested loop programs discussed in this dissertation, their single-assignment programs, and their parse tree representations, see [12, 13, 79, 81].

3.4 From parse tree to polyhedral reduced dependence graph

This section describes the procedure used to convert single assignment programs into the polyhedral reduced dependence graph (PRDG) model. I assume the single-assignment program is generated with the tool MATPARSER.

Section 3.2 defined the elements of a PRDG by attributing the elements of an extended graph with both behavior and geometry. Because the extended graph describes topological properties only, the elements of a PRDG are described by a triple (*topology*, *behavior*, *geometry*). Table 3.1 summarizes the definitions of

the elements of the PRDG and explicitly shows how these elements are constituted from their behavioral, topological, and geometrical part.

type	notation	behavior	topology	geometry
input port domain	P	p		$\mathcal{I}_p$
output port domain	Q	q		$\mathcal{I}_q$
node domain	N	f_n	(I_N, O_N)	$\mathcal{I}_n$
edge domain	E		(Q, P)	$\mathcal{I}_e$

Table 3.1: Elements of the PRDG and their behavioral, topological, and geometrical constituents.

The conversion of a SAP to a PRDG thus comprises the extraction of behavior, topology and geometry from the SAP.

- The elements in the column *geometry* in Table 3.1 are encoded in the SAP in its control statements and the indexing functions of its **ipd** and **opd** statements.
- The elements in the column *behavior* in Table 3.1 are encoded in the leaf nodes of the parse tree. Depending on the type of the statement, a primitive node, primitive input port, or primitive output port has to be constructed.
- The elements in the column *topology* in Table 3.1 are also encoded in the leaf nodes of the parse tree. Their relative position and the variable array name, in case of **ipd** or **opd** statement, are of importance.

Section 3.4.1 deals with the derivation of geometry. Section 3.4.2 deals with the derivation of behavior and topology.

3.4.1 Domain construction

The *index vector* or *iteration vector* of a statement is defined as the vector composed of the index variables of the **for** statements in the path from the root of the parse tree to the parent of the statement. An *iteration* of a statement is a specific value of its index. The set of all iterations for which a statement is executed is called the *index set* of that statement.

Besides the index vector of a statement, two more vectors are significant. The *control vector* of a statement is defined as the vector composed of all control variables of the statements in the path from the root of the parse tree to the parent of that statement. The *parameter vector* of a statement is defined as the vector composed of all parameters of the statements in the path from the root of the parse tree to the parent of that statement.

When specifying the index set of a statement as a periodic lattice polyhedron as defined in (2.19), a set of affine constraints is associated with every control statement. This is done in three steps: First a set of (possibly pseudo-affine) constraints are associated with every statement, second all pseudo-affine constraints are rewritten into sets of affine constraints, third the set of affine constraints from the root of the parse tree to the statement are collected in a single system of constraints.

Extracting constraints from the parse tree

In the first step of domain construction, the syntax of the control statements leads straightforwardly to the constraints:

- Let **for** $i = \ell : s : u$ be the header of a **for** statement. Three constraints are associated with this statement, viz., $\ell \leq i \leq u$ for the loop bounds and $i = \ell + qs$, q being a free variable. When $s = 1$ the constraint $i = \ell + qs$ is redundant and therefore dropped.
- Let **if** $e \geq 0$ be the header of an **if** statement. The constraint associated with this statement is its condition $e \geq 0$. The constraint associated with the **else** statement is the integral complement of $e \geq 0$, that is, $-e - 1 \geq 0$.
- Let $q = e;$ – be an **index transformation** statement. The constraint associated with this statement is the statement $q = e$ itself (without the semicolon).
- Let $\%$**parameter** $N\ \ell\ u;$ – be a **parameter** declaration statement. Two constraints are associated with this statement, viz., $\ell \leq N \leq u$.

Conversion of pseudo-affine expressions into affine expressions

In the second step of domain construction, the pseudo-affine expressions are converted into affine expressions. To do this, I use a technique similar to that used in [12,82]. For every pseudo-linear term that appears in an expression, two additional constraints are introduced.

Recall that in $\mathrm{mod}(e_i, b_i)$ and $\mathrm{div}(e_i, b_i)$, e_i has integral coefficients and b_i is strictly positive. Moreover, e_i in $\mathrm{floor}(e_i)$ and $\mathrm{ceil}(e_i)$ can be alternatively rewritten into $e_i = \frac{e'_i}{b_i}$ where b_i is the least common multiplier $lcm()$ of the denominators of the coefficients. It is assumed that the coefficients of e_i are co-prime. If not, then the coefficients must be put in their *co-prime form* first by dividing numerator and denominator by their $lcm()$. Now e'_i has integer coefficients such that it can converted into an affine expression with integer coefficients easily.

Let e_i be an affine expression and let b_i be a positive integer. Further, let $\{\cdots + a_i c_i + \cdots \geq 0\}$ be a constraint with $a_i c_i$ being a pseudo-linear term. Equations 3.14-3.17 show the conversion of the pseudo-linear term for any of the four integer division functions.

$$\cdots + a_i\ \mathrm{mod}(e_i, b_i) + \cdots \geq 0 \Leftrightarrow \begin{cases} \cdots + a_i(e_i - b_i d) + \cdots \geq 0 \\ 0 \leq e_i - b_i d \leq b_i - 1 \end{cases} \tag{3.14}$$

$$\cdots + a_i\ \mathrm{div}(e_i, b_i) + \cdots \geq 0 \Leftrightarrow \begin{cases} \cdots + a_i d + \cdots \geq 0 \\ 0 \leq e_i - b_i d \leq b_i - 1 \end{cases} \tag{3.15}$$

$$\cdots + a_i\ \mathrm{floor}(e_i) + \cdots \geq 0 \Leftrightarrow \begin{cases} \cdots + a_i d + \cdots \geq 0 \\ 0 \leq e'_i - b_i d \leq b_i - 1 \end{cases} \tag{3.16}$$

$$\cdots + a_i\ \mathrm{ceil}(e_i) + \cdots \geq 0 \Leftrightarrow \begin{cases} \cdots + a_i d + \cdots \geq 0 \\ 1 - b_i \leq e'_i - b_i d \leq 0 \end{cases} \tag{3.17}$$

where d is the newly introduced control variable.

Collecting the constraints

In the third step of domain construction, the individual sets of constraints per statement are combined into one system of constraints. For a given statement, the vector composed of its index vector $\mathbf{i}$, its control vector $\mathbf{c}$, the parameter vector $\mathbf{p}$, and the homogeneous constant $t = 1$ is called the *data-parameter vector*

k, defined as

$$\mathbf{k} = \begin{bmatrix} \mathbf{i} \\ \mathbf{c} \\ \mathbf{p} \\ t \end{bmatrix}. \tag{3.18}$$

Let s be a statement. Let $(@, s_1, s_2, \cdots, s_n)$ be the path from the root of the parse tree to the parent of s. Let $\mathbf{k}$ be the data-parameter vector of s, and let $\{A_i\mathbf{k} = \mathbf{0} \wedge C_i\mathbf{k} \geq \mathbf{0}\}$ be the sets of constraints associated with s_i, $i = 1, 2, \cdots n$. By construction, the index set of s is a periodic lattice polyhedron and is given by $\mathcal{I} = (L, \mathcal{P})$, where the number of rows of L equals the dimension of the index vector of the statement s and where $\mathcal{P}$ is the polytope constructed as

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^d \mid \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \mathbf{x} = \mathbf{0} \wedge \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{bmatrix} \mathbf{x} \geq \mathbf{0}\} \tag{3.19}$$

where d is the dimension of the data-parameter vector.

Figure 3.4 illustrates the sets of affine constraints associated with the statements in the parse tree in the program in Figure 3.3.

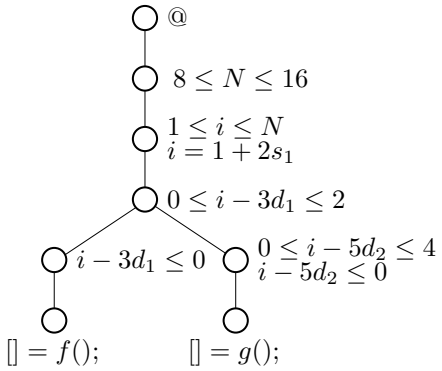


Figure 3.4: New parse tree of program in Figure 3.3.

In this figure the additional control variables d_1 and d_2 are introduced. The method of this section is implemented by Algorithm GET-DOMAIN().

```
GET-DOMAIN(l)
1  s ← parent[l]
2  D ← NIL
3  while s ≠ NIL
4  do C ← GET-CONSTRAINTS(s)
5     D ← ADD-CONSTRAINTS(D, C)
6     s ← parent[s]
7  return D
```

where GET-CONSTRAINTS() converts the statement into a set of constraints and ADD-CONSTRAINTS() constrains D with the just derived set of constraints C.

The construction of the SAP is such that the **opd** statements are direct siblings at the right-hand side of the **node** statements, and consequently, they have the same index set.

3.4.2 Creating the PRDG

This section deals with the extraction of behavior and topology from the parse tree. The PRDG is constructed by using the derived behavior and topology together with the domain construction presented in previous section.

The conversion of the parse tree to a PRDG $G = (\mathcal{N}, \mathcal{E})$ encompasses the conversion of the **node**, **ipd**, and **opd** statements to node domains, input port domains, output port domains, and edge domains. Let s be a leaf node of the parse tree and let its index set $\mathcal{I}$ be given by $(L, \mathcal{P})$. Depending on the type of statement s one of the following must be done for each leaf node s.

- if s is an **opd** statement
  Let Q be the corresponding output port domain $Q = (q, \mathcal{I}_q)$. Then $\mathcal{I}_q = \mathcal{I}$ and q is the primitive output port $q = (\textit{arg}, \textit{var}, \texttt{double}^2)$ with *arg* and *var* parsed from s.

- if s is an **ipd** statement
  Let P be the corresponding input port domain $P = (p, \mathcal{I}_p)$. Then $\mathcal{I}_p = \mathcal{I}$ and p is the primitive input port $p = (\textit{arg}, \textit{var}, \texttt{double})$ with *arg* and *var* parsed from s.

- if s is an **ipd** statement
  Let E be the corresponding edge domain $E = ((Q, P), \mathcal{I}_e)$. Then P is the input port domain associated with s and Q is the output port domain that has the same variable in its primitive port as P has. Further, $\mathcal{I}_e$ is the index set $(D, \mathcal{P}_e)$ where $\mathcal{P}_e = \mathcal{P}$ and D is the mapping matrix, defined at the end of Section 3.3.1, of s.

- if s is an **node** statement
  Let N be the corresponding node domain $N = (I_N, O_N, f_n, \mathcal{I}_n)$. Then $\mathcal{I}_n = \mathcal{I}$ and f_n is the *function* parsed from s. Further, I_N and O_N are the set of input port domains and output port domains constructed from the **ipd** and **opd** statements directly preceding and following the **node** statement in the parse tree, respectively. The parsing of the function f_n from the statement s is implemented by the method PARSE-FUNCTION(s).

This conversion is implemented in the tool DGPARSER and is described in Section 5.2. Its input is a SAP converted into a parse tree using the parser from the MATPARSER tool. The parse tree is then transformed into a PRDG.

[2] I use the type `double` here to represent the 54 bit floating point precision standard in MATLAB.

Chapter 4

Synthesis of KPNs from dependence graphs

Chapter 3 dealt with the polyhedral reduced dependence graph (PRDG) model and with the techniques to convert a single assignment program into this model. This section focuses on the conversion of a PRDG into a Kahn process network (KPN).

Again, parse trees are used as an intermediate representation. First the PRDG is converted into a *set of* parse trees, then from the parse trees the final code of the processes are generated. However, the network that connects the process is directly generated from the PRDG. Note that I use parse trees as output representation, but these trees will now represent a significantly different computation model; KPN instead of nested loop programs (NLP).

Roughly stated, the conversion works as follows: For every node domain in the PRDG, a Kahn process in parse tree representation is constructed. Apart from the statements found in NLPs, these parse trees contain communication primitives that allow to the processes to communicate with each other over channels. These channels connects the processes and are constructed from the edge domains in the PRDG.

The remainder of this chapter is organized as follows. Section 4.1 motivates the use of Kahn process networks and relates this model to other process network models and data flow networks found in literature. Section 4.2 describes a structuring of the parse tree representation of the Kahn processes. This structuring is such that it naturally fits the model of the node domain. The conversion of a node domain into a parse tree is divided in three steps, shown in Figure 1.3: *domain scanning*, *domain matching*, and *linearization*. Section 2.6 describes a method that is used in any of the three steps that is dealt with in sections 4.3 – 4.5. Section 4.6 finally deals with the construction of the process network itself.

4.1 Kahn process networks

Chapter 1 motivated that in order to map an application specified by an NLP onto a task level parallel (TLP) architecture, that this application is first translated into a process network. This section motivates the use of a specific model; the *Kahn process network* model.

To map applications onto such a TLP architecture it is beneficial to specify the applications in terms of a model that maps naturally onto the architecture. It is desired that for such a model the following requirements are met:

- The model should enable the designer to understand the specification and enable him to reason about it.

- The model should be general enough to model a reasonable variety of applications in a specific application domain.
- The model should be implementation-independent, as far as possible.
- The model should naturally expose the parallelism inherent in the application.
- The model should enable the use of both coarse-grain as well as fine-grain parallelism.
- The model should make the communication and synchronization explicit.

Basically there are two types of models that roughly fulfill these requirements, viz., *dataflow networks* and *process networks*. Both dataflow networks and process networks are graphs where the nodes represent computation and the edges represent communication and synchronization. I distinguish the two types of networks as follows.

In dataflow networks, a node represents an atomic unit of computation (a mathematical function) and is called an actor. An edge in a dataflow graph represents a channel that can carry a (possibly infinite) number of tokens. A token is a container of information. Actors communicate with each other by producing tokens on their output channels and consuming tokens from their input channels.

An actor is said to be enabled when all tokens on its input channels are available. An enabled actor can fire (start its execution). A firing goes as follows: first, the actor consumes all required tokens from its input channels and passes them as arguments to the function. Second, the function is executed. Third, the results of the execution are produced as tokens on the actor's output channels. These three steps form semantically one atomic action.

Dataflow actors have firing rules. These rules specify what tokens must be available at the inputs for the actor to fire. When an actor fires, it consumes some finite number of input tokens and produces some finite number of output tokens. A process may be formed by repeated firings of the same dataflow actor so that infinite streams of data may be operated upon.

There is a large variety of dataflow actors and associated firing rules. Some well known examples are *homogeneous dataflow* (HDF) [83], *synchronous dataflow* (SDF) [54], *cyclo-static dataflow* (CSDF) [56], *boolean dataflow* (BDF) [84], and *dynamic dataflow* (DDF) [85–87].

HDF has a single token passing firing rule and is a special case of SDF in which the number of tokens read or written may be more than one, but must be invariant. In CSDF, the number of tokens consumed and produced may be different for different firings but must have a cyclic behavior. BDF introduces deterministic conditional behavior (*if-then-else*; *while* loops) with the actors SWITCH and SELECT. Finally, DDF introduces non-determinism (non-deterministic MERGE). Restricting the type of dataflow actors to those that have predictable token production and consumption patterns makes it possible to perform static scheduling and to bound the memory required to implement the communication channel buffers [88].

In a process network, the nodes represents processes. Unlike a dataflow actor, a process is not an atomic unit of computation. A process represents a set of operations that are executed sequentially. In contrast with dataflow actors, processes in process networks can have state. Two well known process network models are the *Kahn process networks* [11] and the *communicating sequential processes* (CSP) [89].

In CSP, the channels are not buffered and synchronization is by means of rendez-vous. In KPN, processes communicate over unbounded FIFO channels as dataflow actors in dataflow networks do. However, the KPN model does not rely on firing rules, and uses a blocking-read mechanism for synchronization instead.

The CSP model is non-deterministic and not well suited for modeling streaming applications. As the applications we are interested in are both stream-based and deterministic, we exclude CSP as candidate model.

The KPN model, on the other hand, is a natural model for stream-based applications. Moreover, a KPN is deterministic, that is, its functionality is independent of the execution order of the processes. This is so, because processes block when attempting to get data from an empty channel. A process is either executing, or it is blocked waiting for data on one of its input channels [88]. Although dataflow network models are very powerful, the KPN model is more general and is close to the PRDG, as I shall show.

4.2 Derivation of Kahn process networks from PRDGs

Although several KPNs could be derived from a given PRDG, one can convert that PRDG into a unique initial KPN by defining a process for each node domain and a communication channel for each interconnecting output port domain and input port domain. Since a process in a KPN is fully sequential, the node domains which are mapped into Kahn processes have to be sequentialized. This may seem counter-productive as the PRDG was constructed to expose all parallelism in the first place. However, by sequentializing the node domains, the inherent parallelism is not lost because the information is still available in the PRDG and parallel implementations of Kahn processes can be easily created by inspecting the underlying node domains in the PRDG. Since a node domain in a PRDG is defined by a single polytope, its conversion to a Kahn process can be easily done by imposing a lexicographic ordering on the operations in the domain and generating a corresponding nested-loop program for the Kahn process.

In Section 4.2.1 I illustrate the basic concept by means of an example. In Section 4.2.2, the process is formalized by introducing the parse tree representation of the Kahn process that is derived from the PRDG node domain. This section further identifies the three sub-problems in the generation of the Kahn processes from PRDG node domains which are dealt with in detail in next sections.

4.2.1 Introducing the structure of the Kahn processes

Figure 4.1 shows a part of a PRDG, consisting of node domains N_P and N_C, the output port domain (OPD) of N_p (shaded triangle), the input port domain (IPD) of N_c (shaded triangle), and the affine mapping M from IPD to OPD. Also shown is the process network that is to be derived from the PRDG. It consists of two processes, called the producer ($P()$) and the consumer ($C()$) and an unbounded FIFO channel (ch_1) for the communication between the two.

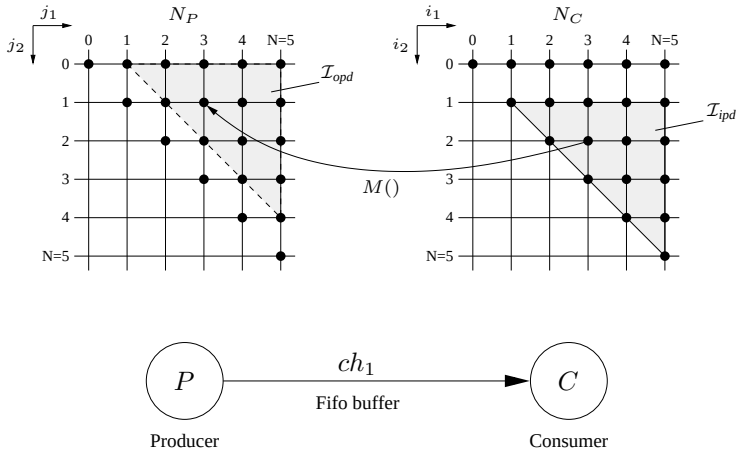


Figure 4.1: Mapping of a PRDG onto a Process Network, running example.

The goal is to map the the PRDG shown in the top-part of the figure onto the KPN shown in the bottom-part.

Figure 4.2 shows the internal structure of the two Kahn processes.

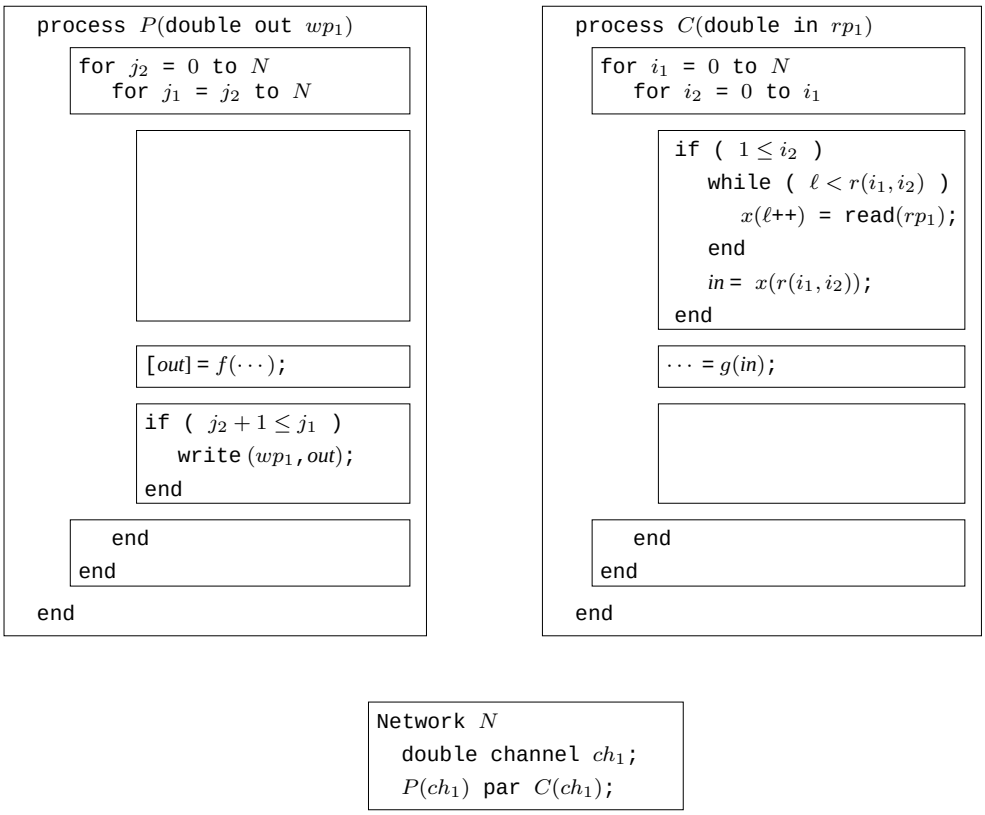


Figure 4.2: Internal structure of the two Kahn processes from Figure 4.1.

The processes $P()$ and $C()$ correspond to the node domains N_P and N_C, respectively. It is assumed that the primitive nodes in N_P and N_C, represented by the black dots, contain the functions $f()$ and $g()$, respectively. Process $P()$ starts with a declaration of the process[1] immediately followed by a set of nested loops. These loops iterate over all the points in N_P. Similarly, the loops in process $C()$ iterate over all the points in N_C. Since all primitive nodes in N_P invoke the same function $f()$, $f()$ appears exactly once in the body of the loop nest in $P()$. Similarly, there is exactly one invocation of $g()$ in the body of $C()$. The shaded regions in Figure 4.1 are the port domains that communicate with each other. Therefore, only the results of the functions at the iteration points in the shaded index set of N_P are to be written onto the channel. In process $P()$ this is specified by the **if** statement surrounding the **write** statement. The block of code that contains the **write** statement in Figure 4.2 is surrounded by a rectangle. There is one such block for every output port domain of the node domain from which the process is derived. Notice that process $P()$ writes tokens to the channel in the order specified by the loop nest.

Similarly, reading from the channel should only take place at the iteration points in the shaded index set of N_C. The **read** statement in $C()$ is not only surrounded by an **if** statement but also by a **while** statement. Since Kahn channels are FIFO queues, tokens are read from channel in the the same order as they were written to it. Process $C()$ will in general process tokens in a different order. To reorder these tokens an array x is introduced for channel wp_1. The array x is written to as if it were a queue, this is shown by the counter ℓ that is the logical address written to, and is read from as if it were a random access memory. The random access allows the reordering of incoming tokens and is specified by the read function $r(i_1, i_2)$. This

[1]The original Kahn paper [11] uses the keywords `in` and `out` to specify whether there will be read from or written to the channel, respectively. Since our tools currently generate Kahn Process Networks in the target language C++ we textually distinguish between input ports and output ports by using port names rp and wp, standing for *read port* and *write port*, respectively.

function is derived from the order in which the iteration points in N_P are visited and the mapping function $M()$, as will be explained in Section 4.5. For every iteration point (i_1, i_2) in the input port domain, process $C()$ binds an element from x to input argument *in* of function $g()$. The **while** loop models the blocking-read semantics of the KPN model. The **while** loop iterates as long as the data that is to be read from x, that is $x(r(i_1, i_2))$, is not yet in x. In the **while** loop, tokens are read from the channel and are stored in array x. The index in x at which the data is written is determined by the local counter ℓ that is incremented every time a token has been read from the channel. The while loop terminates in case $\ell \geq r(i_1, i_2)$, in this case it is guaranteed that $x(r(i_1, i_2))$ contains valid data that can be passed to function $g()$. When $\ell < r(i_1, i_2)$ and the channel is empty, the process blocks. In this way, the **while** loop implements an out-of-order blocking read without sacrificing the KPN semantics. The block of code that contains the **read** statement in Figure 4.2 is surrounded by a rectangle delimiting the actions associated with the corresponding input port. There is one such block for every input port domain of the node domain from which the process is derived.

The next section formalizes the structure of the Kahn process by means of a parse tree.

4.2.2 Parse tree representation of the Kahn processes

The previous section introduced the structure of a Kahn process by means of the code example in Figure 4.2. This figure is neither formal nor complete. This section formally describes the complete structure of a Kahn process by means of a parse tree and introduces the problems that arise when deriving the Kahn process from the PRDG. Parse trees have already been described in Section 3.3. The structure of the parse tree of a Kahn process in shown in Figure 4.3. All nodes in the tree, except for the root and the leave nodes

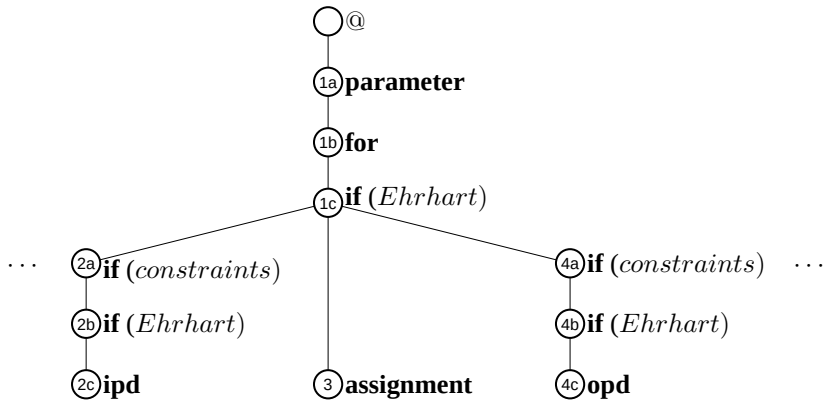


Figure 4.3: The structure of the parse tree generated from the PRDG.

represent a nesting of nodes of the same type. For example, the **if** $(Ehrhart)$ node represents a set of nested **if** $(Ehrhart)$ statements. The dots "$\cdots$" on the left and right hand side of the figure represent repetitions of the leftmost branch and rightmost branch, respectively.

Among the statements in the parse tree, there are two types of **if** statements, i.e., **if** $(constraints))$ and **if** $(Ehrhart)$. An **if** $(constraints)$ statement uses a constraint (equality or inequality) as its condition. This statement narrows a polytope to a smaller one. An **if** $(Ehrhart)$ statement has a condition of the form $E \geq 1$ where E is an Ehrhart polynomial. This statement filters out points to create holes in an index set.

To explain the structure of the parse tree, I distinguish four partitions in it. The partition a node belongs to is indicated by the number depicted inside the node. Partition 1 has the three nodes 1a, 1b, and 1c, and define to the loop structure of the program. Partition 2 has the tree nodes 2a, 2b, and 2c, and defines the reading of tokens from the channels and their assignment to variables. Partition 3 has the single node 3 and defines the function call statement. Partition 4 has the three nodes 4a, 4b, and 4c, and defines the code for

writing the return values of the function to the channels.

partition 1
The nodes in the first partition define the parameters and the index set of the node domain.

a) The set of **parameter** statements defines the parameters that are used in the rest of the program. Every **parameter** statement defines one parameter together with its lower and upper bound.

b) The set of **for** statements defines a set of index variables and an index set. The index set is a dense integral set that contains the index set of the node domain.

c) The set of **if** $(Ehrhart)$ statements filters out all points that do not belong to the node domain. Thus, whereas the set of **for** statements defines a dense index set, only points that do actually belong to the node domain pass the **if** statement. Since the domains of both node domains in the example in Figure 4.1 are dense themselves, no **if** $(Ehrhart)$ statements are found in the code in the example in Figure 4.2.

partition 2
The nodes in the second partition define the points of the index set at which a **read** operation from a specific channel must be executed. The nodes in this partition correspond to the blocks of code containing the **read** statement in process $C()$ in Figure 4.2.

a) The set of **if** $(constraint)$ statements bounds the polytope that encloses the index set of the node domain to a polytope that encloses the index set of an input port domain. The resulting region corresponds to the shaded part in N_C in Figure 4.1. The constraint corresponds to the constraint $1 \leq i_2$ in process $C()$ in Figure 4.2.

b) The set of **if** $(Ehrhart)$ statements filters out points from the index set for which no input is to be read from a specific channel. This must be done when, for example, the lattice of the input port domain is coarser than the lattice of the node domain.

c) The **ipd** statement represents the block of code, excluding the **if** statement, that contains the **read** operation like in the consumer process in Figure 4.2.

partition 3
The **assignment** statement represents the function call in the body of the Kahn process. Its input arguments get values assigned in partition 2. The results are written to the channels in partition 4.

partition 4
The nodes in the fourth partition define the points of the index set at which a **write** operation to a specific channel must be executed. The nodes in this partition correspond to the block of code containing the **write** statement in process $P()$ in Figure 4.2.

a) The set of **if** $(constraint)$ statements bounds the polytope that encloses the index set of the node domain to a polytope that encloses the index set of an output port domain. The resulting region corresponds to the shaded part in N_P in Figure 4.1. The constraint corresponds to the constraint $j_2 + 1 \leq j_1$ in process $P()$ in Figure 4.2.

b) The set of **if** $(Ehrhart)$ statements filters out points from the index set for which no output is to be written to a specific channel.

c) The **opd** statement represents the **write** operation like in the producer process in Figure 4.2.

Now that the structure of the parse tree representing a Kahn process is defined, the problem is how to convert a node domain of an PRDG into such a parse tree. This can be split into three sub-tasks. Each sub-task will generate a set of nodes of the parse tree in Figure 4.3, except the creation of the assignment statement (node 3). The creation of node 3 is just the conversion of a function in the PRDG model to an assignment statement and is given in Algorithm NODE2ASSIGNMENT below.

NODE2ASSIGNMENT(N)
1 **A** $\leftarrow$ NIL $\triangleright$ the vector of statements to be returned
2 $f \leftarrow function[N]$
3 $A \leftarrow$ new `AssignStatement`
4 $function[A] = functionName[f]$
5 **for all** arguments a in $inputArguments[f]$
6 **do** $var \leftarrow$ new `VariableStatement`$(name[a])$
7 ADD-CHILD$(rhs[A], var)$
8 **for all** arguments a in $outputArguments[f]$
9 **do** $var \leftarrow$ new `VariableStatement`$(name[a])$
10 ADD-CHILD$(lhs[A], var)$
11 INSERT-ELEMENT$(\mathbf{A}, A)$
12 **return A**

The first sub-task is referred to as *domain scanning* and covers nodes 1a (shorthand for the first node in partition 1), 1b, and 1c of the parse tree. Domain scanning deals with finding and specifying a schedule for the **assignment** statement, but not with the derivation of the **assignment** statement itself. Given a scan order, domain scanning is in charge of finding the lower and upper bound expressions of the **for** statements in node 1b, and of excluding the points that are not iterations in the index set with the **if** $(Ehrhart)$ statements in node 1c. The bounds of the loops of the **for** statements may be parameterized. The bounds of these parameters are derived in a similar way as the bounds of the **for** statements.

The second sub-task is referred to as *domain matching* and covers nodes 2a, 2b, 4a, and 4b of the parse tree. Edge domains of the PRDG are converted into channels of the KPN. An input port domain of a node domain specifies for which points in the node domain data is to be read by the corresponding **read** operation in the corresponding process. An edge domain maps points from an input port domain onto a corresponding output port domain and, thus, specifies for what points in the node domain data is to be written by the corresponding **write** operation of the corresponding process. An output port domain (OPD) is said to *match* an input port domain (IPD) when for every point in the OPD there is a corresponding non-empty set of points in the IPD and when for every point in the IPD there is a corresponding single point in the OPD. The matching of an output port domain to an input port domain ensures that output values of producing node domain functions are only written to channels if they will be consumed by the consumer at the end of the channel. This domain matching corresponds to the relating of nodes 4a and 4b of a parse tree to nodes 2a and 2b of another (possibly the same) parse tree.

The third sub-task is referred to as *linearization* and covers node 2c of the parse tree. Recall that a consuming process may process data that arrive at an incoming channel in a different order. This reordering is performed in a queue-like data structure into which incoming data is written in order. Linearization is about the address generation to access this data structure such that the proper reordering is performed.

The remainder of this chapter is organized as follows. Section 4.3, Section 4.4, and Section 4.5 deal with the sub problems domain scanning, domain matching, and linearization, respectively.

4.3 Domain scanning

This section deals with the conversion of the index set of a node domain to a nested loop structure. The generation of the nested loop structure is called *domain scanning* and the resulting nested loop structure defines a lexicographic ordering of the nodes in the node domain.

The index set of a node domain is a periodic lattice polytope (cf. Definition 3.11 and Definition 2.24). The index set $\mathcal{I} \subset \mathbb{Z}^d$ is defined by the projection of the integral points contained in a parameterized polytope $\mathcal{P}(\mathbf{p}) \subset \mathbb{Z}^n$ onto $\mathbb{Z}^d$. In this section I consider three cases of (2.19) of increasing complexity.

1. $d = n$, hence the projection matrix $L = I$ and hence the index set is dense because $\mathcal{P}(\mathbf{p}) \in \mathbb{Z}^n$ is.
2. $d < n$, hence the projection matrix $L = \begin{bmatrix} I & 0 \end{bmatrix}$, and the index set is dense
3. $d < n$, hence the projection matrix $L = \begin{bmatrix} I & 0 \end{bmatrix}$, and the index set is non-dense.

Before moving on, I first introduce the three cases to be considered informally using the examples shown in Figure 4.4

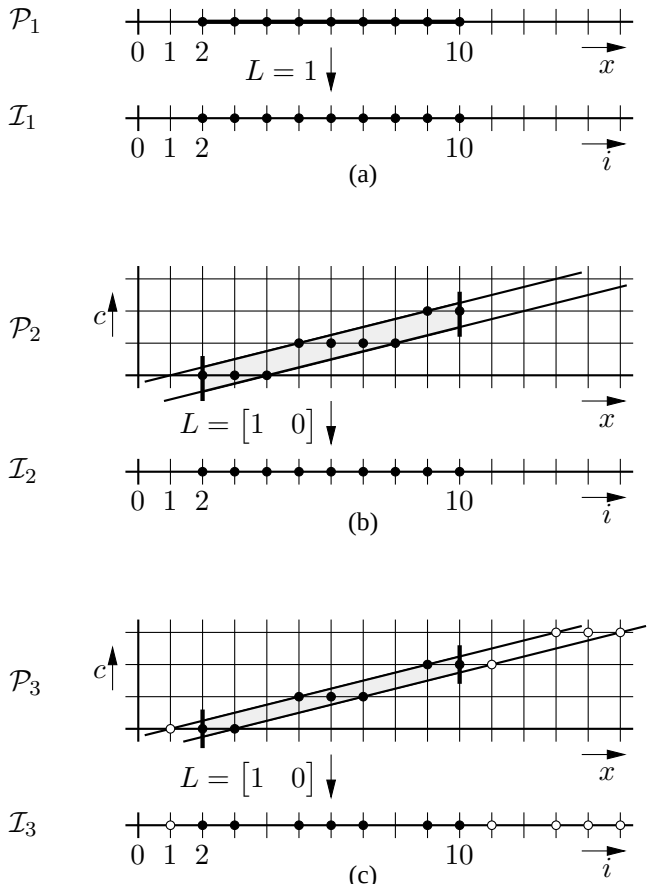


Figure 4.4: Three index sets for which the scanning problem has increasing complexity.

The first case is illustrated by Figure 4.4 (a). Here, polytope $\mathcal{P}_1 \subset \mathbb{Q}$ and index set $\mathcal{I}_1$ is defined by the mapping $i = Lk$, $k \in \mathcal{P}_1 \cap \mathbb{Z}$ onto $\mathbb{Z}$. In this case, the index set consists of just the integral points contained by the polytope. So, by scanning the integral points in $\mathcal{P}_1$, we have a scanning of the points in $\mathcal{I}_1$. The scanning of polytopes is dealt with in Section 4.3.1.

The second case is illustrated by Figure 4.4 (b). Here polytope $\mathcal{P}_2 \subset \mathbb{Q}^2$ and index set $\mathcal{I}_2$ is defined by the mapping $i = L(x, c), (x, c) \in \mathcal{P}_2 \cap \mathbb{Z}^2$ onto $\mathbb{Z}$. In this case the index set is still dense and contains the same points as $\mathcal{I}_1$. The main difference with the first case is that its defining polytope is in $\mathbb{Q}^2$ rather than in $\mathbb{Q}$. The scanning of the points in $\mathcal{I}_2$ is performed by scanning the integral points in $\mathcal{P}_2$ with x in the outer loop and c in the inner loop. The loop for x alone specifies the required scanning. So, the scanning of index sets of the second case is based on the scanning of polytopes and is dealt with in Section 4.3.2.

The third case is illustrated by Figure 4.4 (c). Here polytope $\mathcal{P}_3 \subset \mathbb{Q}^2$ and index set $\mathcal{I}_3$ is defined by the mapping $i = L(x, c), (x, c) \in \mathcal{P}_3 \cap \mathbb{Z}^2$ onto $\mathbb{Z}$. Note that this is the same index set as in Figure 2.6. In this case the index set differs from the index sets $\mathcal{I}_1$ and $\mathcal{I}_2$ in that there are additional holes in it.

Like in the second case, polytope $\mathcal{P}_3$ is scanned, but unlike the second case the inner loop for the variable c is not ignored, and is used to identify the holes. The scanning of non-dense index sets is dealt with in Section 4.3.3.

4.3.1 Scanning polyhedra

Given a polytope $\mathcal{P}(\mathbf{p})$, the polyhedral scanning problem is to construct a set of nested **for** loops that orders the integral points contained by the polytope lexicographical. The lexicographic order is specified by an iteration vector $(i_1, i_2, \cdots, i_d)$, where d is the dimension of the space that contains the polytope, i_1 is the iterator of the outer most loop, and i_d is the iterator of the inner most loop. The structure of the resulting loop nest is given in Program 4.1.

Program 4.1: STRUCTURE-OF-LOOP-NEST

for $i_1 = \ell_1(\mathbf{p}) : 1 : u_1(\mathbf{p})$,
  for $i_2 = \ell_2(i_1, \mathbf{p}) : 1 : u_2(i_1, \mathbf{p})$,
    $\ddots$
      for $i_d = \ell_d(i_1, i_2, \cdots, i_{d-1}, \mathbf{p}) : 1 : u_d(i_1, i_2, \cdots, i_{d-1}, \mathbf{p})$,

In this program, $\ell_j()$ and $u_j()$ are affine functions that specify the lower bound and upper bound of iterator i_j, respectively. The *level* of an iterator is defined as its position in the iteration vector. The iterator bounds $\ell_j()$ and $u_j()$ are piece-wise affine functions of the parameter vector $\mathbf{p}$ and the lower level iterators $i_1, i_2, \cdots, i_{j-1}$. Piece-wise affine means that the $\max()$ and the $\min()$ functions might be applied to a set of affine functions in the lower and upper bound functions respectively.

For the polyhedral scanning, I use the function POLYHEDRAL-SCAN($\mathcal{P}, \mathcal{C}$) that is implemented in the PolyLib[2] and described in [90]. The strategy that is used in this algorithm is the same as in [73] but differs in that it does not use the Fourier-Motzkin pairwise projection algorithm [59].

Basically the algorithm recursively projects the polytope in the direction that corresponds to highest level loop iterator not yet projected along. Let $\mathcal{P} \in \mathbb{Q}^d$ be the polytope to be scanned in the lexicographic order $(x_1, x_2, \cdots, x_d)$. Initialize the recursion with $\mathcal{P}_d = \mathcal{P}$. Then, at every level $\ell = d, d-1, \cdots, 1$ of the recursion the projection $\mathcal{P}_{\ell-1} = \mathrm{proj}_{x_\ell}(\mathcal{P}_\ell)$ is obtained. $\mathcal{P}_0$ is the context $\mathcal{C}$ of $\mathcal{P}$. Note that the projection $\mathrm{proj}_{x_\ell}(\mathcal{P}_\ell)$ makes $\mathcal{P}_{\ell-1}$ independent of x_ℓ, and hence, due to the recursion, independent of all $x_i, i \geq \ell$. All $\mathcal{P}_\ell$ are polyhedra in $\mathbb{Q}^d$ [3]. Constraints that occur the polyhedron $\mathcal{P}_\ell$ may be redundant with the constraints in $\mathcal{P}_{\ell-1}$. At every level ℓ in the recursion, a new polyhedron $\mathcal{Q}_\ell$ is derived that has such redundant constraints. The removal of redundant constraints is done with the function DOMAINSIMPLIFY() from the PolyLib;

[2]The PolyLib I am referring to is the one from the Université Louis Pasteur, Strasbourg, `http://icps.u-strasbg.fr/ loechner/polylib/`.

[3]To be precise, actually the algorithm works on the double description of polyhedra that is, it uses the representation in (2.3) and the dual representation. In the dual representation a polyhedron is specified in terms of vertices, rays, and lines [59].

$\mathcal{Q}_\ell = \text{DOMAINSIMPLIFY}(\mathcal{P}_\ell, \mathcal{P}_{\ell-1})$. As a result we have $\mathcal{P}_\ell = \mathcal{Q}_\ell \cap \mathcal{P}_{\ell-1}$. The result of the polyhedral scanning algorithm is the set of polyhedra $\{\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_d\}$.

As a result of the projections onto lower-dimensional spaces every $\mathcal{Q}_\ell$ has the form $\mathcal{Q}_\ell = \{\mathbf{x} \in \mathbb{Q}^d \mid A_\ell \mathbf{x} = B_\ell \mathbf{p} + \mathbf{b}_\ell \wedge C_\ell \mathbf{x} \geq D_\ell \mathbf{p} + \mathbf{d}_\ell\}$, where $A_\ell = \begin{bmatrix} A'_\ell & 0 \end{bmatrix}$ and $C_\ell = \begin{bmatrix} C'_\ell & 0 \end{bmatrix}$ with 0 being the zero matrix with $d - \ell$ columns.

Every constraint in $\mathcal{Q}_\ell$ is of the form $a_1x_1 + a_2x_2 + \cdots + a_\ell x_\ell \geq b_1p_1 + b_2p_2 + \cdots b_mp_m + c$ (here I assumed that constraints with equality are rewritten into two constraints with inequality of different signs). When $a_\ell \geq 1$ the constraint is rewritten to $x_\ell \geq (-a_1x_1 - a_2x_2 - \cdots - a_{\ell-1}x_{\ell-1} + b_1p_1 + b_2p_2 + \cdots b_mp_m + c)/a_\ell$ where the right hand side is a lower bound of the loop that is constructed for $\mathcal{Q}_\ell$. Similarly, when $a_\ell \leq -1$ the constraint is rewritten to $x_\ell \leq (-a_1x_1 - a_2x_2 - \cdots - a_{\ell-1}x_{\ell-1} + b_1p_1 + b_2p_2 + \cdots b_mp_m + c)/a_\ell$ where the right hand side is an upper bound of the loop that is constructed for $\mathcal{Q}_\ell$. When a lower bound and/or upper bound is non-integral then the ceiling and/or floor of the bounds must be taken, respectively. When there are multiple lower bounds and/or upper bounds, the maximum and/or minimum among these multiple bounds must be taken, respectively. To indicate that the loop nests scan over the integral points in the polytope the variables $x_1, x_2, \cdots, x_d$ are replaced by $i_1, i_2, \cdots, i_d$.

Example 4.1 (Polyhedral Scanning) The example is to scan $\mathcal{P}(p) = \{(x_1, x_2) \in \mathbb{Q}^2 \mid 0 \leq x_2 \leq 4 \wedge x_2 \leq x_1 \leq x_2 + 9 \wedge x_1 \leq p \wedge p \leq 40\}$ from Figure 2.2. Suppose we want to scan the polytope in the (x_1, x_2) order. The polyhedral scanning algorithm returns the following two polyhedra $\mathcal{Q}_1 = \{(x_1, x_2) \in \mathbb{Q}^2 \mid 0 \leq x_1 \wedge x_1 \leq 13 \wedge x_1 \leq p\}$ and $\mathcal{Q}_2 = \{(x_1, x_2) \in \mathbb{Q}^2 \mid 0 \leq x_2 \wedge x_1 - 9 \leq x_2 \wedge x_2 \leq 4 \wedge x_2 \leq x_1\}$.

$\mathcal{Q}_1$ is converted into the outer loop with index variable i_1. The constraint $0 \leq x_1$ represents the lower bound 0. The two constraints $x_1 \leq 13$ and $x_1 \leq p$ represent the upper bounds 13 and p. Since there are multiple upper bounds, the minimum of 13 and p is taken.

$\mathcal{Q}_2$ in converted into the inner loop with index variable i_2. The two constraints $0 \leq x_2$ and $x_1 - 9 \leq x_2$ represent the lower bounds 0 and $i_1 - 9$. Since there are multiple lower bounds, the maximum of 0 and $i_1 - 9$ is taken. The two constraints $x_2 \leq 4$ and $x_2 \leq x_1$ represent the upper bounds 4 and i_1. Since there are multiple upper bounds, the minimum of 4 and i_1 is taken.

Program 4.2 shows the loop nest constructed from the loop bounds derived above.

Program 4.2: SCANNING THE INDEX SET IN FIGURE 2.2

for $i_1 = 0 : 1 : \min(13, p)$,
  for $i_2 = \max(0, i_1 - 9) : 1 : \min(4, i_1)$, ■

4.3.2 Scanning dense index sets

The method described in the previous subsection is applicable when the index set of a node domain coincides with the set of integral points contained in its defining polytope, as in Figure 4.4(a). Recall that the index sets to be scanned are periodic lattice polyhedra, see Definition 2.24 and Definition 3.11. The case when the index set is not contained in its defining polytope, yet is dense, as in Figure 4.4(b), is treated in this section.

According to Definition 3.11, the index set of a node domain is a periodic lattice polyhedron, that is, it is of the form $(L, \mathcal{P})$ where the mapping function $L()$ specifies the orthogonal projection of the integral points in $\mathcal{P}$ onto the space that contains the index set. One of the properties of a periodic lattice polyhedron is that the points in the index set are in one-to-one correspondence with the points in $\mathcal{P} \cap \mathbb{Z}^n$ where n is the dimension of the space that contains the polytope.

Let be given an index set $\mathcal{I} = L(\mathcal{P}), \mathcal{I} \subset \mathbb{Z}^d, \mathcal{P} \subset \mathbb{Q}^n$. When the index set is dense, the projection of the

integral points in $\mathcal{P}$ onto $\mathbb{Z}^d$ result in the same set of points as the set of integral points in the projection of $\mathcal{P}$ onto $\mathbb{Q}^d$. Thus for dense index set we have that $L(\mathcal{P} \cap \mathbb{Z}^n) = L(\mathcal{P}) \cap \mathbb{Z}^d$. Denote the points in $\mathbb{Q}^n$ by $\mathbf{x} = (x_1, x_2, \cdots, x_n)$. Note that $L(\mathcal{P})$ projects $\mathcal{P}$ in the directions of the variables $x_{d+1}, x_{d+2}, \cdots, x_n$. Thus, by using the polyhedral scanning of the previous section, the first $n-d$ recursions project $\mathcal{P}$ onto $\mathbb{Q}^d$ and the other d recursions construct the polyhedra required for construction of the loops.

The polyhedral scanning applied to $\mathcal{P}$ returns the set of polyhedra $\{\mathcal{Q}_\ell\}_{\ell=1}^n$ and must be interpreted as follows.

1. All polyhedra $\mathcal{Q}_\ell, \ell > d$ should be ignored.
2. All polyhedra $\mathcal{Q}_\ell, \ell \leq d$ are to be interpreted as polyhedra in $\mathbb{Q}^d$ rather than $\mathbb{Q}^n$.

Example 4.2 (scanning a dense index set) The index set $\mathcal{I}_2$ in Figure 4.4(b) is defined by the polytope $\mathcal{P}_2 = \{(x, c) \in \mathbb{Q}^2 \,|\, 2 \leq x \leq 10 \wedge 4c+1 \leq x \leq 4c+4\}$. Since the i-direction of the index set corresponds with the x-direction of $\mathcal{P}_2$, $\mathcal{P}_2$ is scanned in the lexicographic order (x, c). The scanning algorithm returns two polyhedra $\mathcal{Q}_1$ and $\mathcal{Q}_2$; only $\mathcal{Q}_1$ is used. The polyhedron found is $\mathcal{Q}_1 = \{x \in \mathbb{Q} \,|\, 2 \leq x \leq 10\}$, and the resulting loop is given in Program 4.3.

Program 4.3: SCANNING $\mathcal{I}_2$ IN FIGURE 4.4(B)
for $i = 2 : 1 : 10$, ■

For every iterator $i_\ell, \ell = 1, 2, \cdots, d$, the corresponding polyhedron $\mathcal{Q}_\ell$ is converted in a **for** statement to form the set of **for** statements in node 1b of the parse tree in Figure 4.3.

The domain scanning procedure is given in Algorithm DOMAIN2FORSTATEMENTS(). It takes as the input an index set specified by a single system of constraints M and returns a vector of **for** statements. Matrix M combines all constraints in (2.3) and has the form

$$M = \begin{bmatrix} \mathbf{1} & A & -B & \mathbf{b} \\ \mathbf{0} & C & -D & \mathbf{d} \end{bmatrix} \tag{4.1}$$

where a 1 in the first column indicates that the constraint is an equality, and a 0 indicates that the constraint is an inequality.

```
DOMAIN2FORSTATEMENTS(M)
 1  I ← CONSTRAINTS2POLYHEDRON(constraints[M])
 2  U ← UNIVERSE-POLYHEDRON(dim[parameterVector[M]])
 3  S ← POLYHEDRAL-SCAN(I,U) ▷ S is linked list of polyhedra
 4  x ← dataParameterVector[M]
 5  d ← dim[M] ▷ d is dimension of space that contains the index set
 6  for ℓ = 1 to d
 7  do F ← new ForStatement
 8     (iterator[F], lowerBound[F], upperBound[F]) ← POLYHEDRON2BOUNDS(S_ℓ, ℓ, x)
 9     stride[F] ← 1
10     INSERT-ELEMENT(F, F)
11  return F
```

Here the Algorithm POLYHEDRON2BOUNDS() is used to convert a polyhedron into two sets of constraints. One set of constraints for constraints of the form $\mathbf{x}_\ell \leq ...$ and one set of the form $\mathbf{x}_\ell \geq ....$

```
POLYHEDRON2BOUNDS(Q, ℓ, x)
1  M ← POLYHEDRON2CONSTRAINTS(Q) ▷ again, M is as in (4.1)
2  a ← m_ℓ ▷ a is the (ℓ+1)^th column of M
3  m_ℓ = 0 ▷ set the (ℓ+1)^th column of M to 0
4  v = M · [0; x; 1] ▷ v is vector of polynomials
```

```
5  for i = 0 to number of rows in M minus one
6  do if a_i > 0 or M_{i,0}
7        then INSERT-ELEMENT(lb, -v_i/a_i)
8     if a_i < 0 or M_{i,0}
9        then INSERT-ELEMENT(ub, -v_i/a_i)
10 return (x_{ℓ-1}, lb, ub)
```

4.3.3 Scanning non-dense index sets

The way I deal with a non-dense index set is to first handle it as if it were dense, that is, as if it were the set of integral points in the projection of $\mathcal{P}$. So, the scanning of dense index sets is performed and the related **for** statements are constructed. However, since the **for** loops now iterate also over points that do not belong to the index set, additional statements to filter them out have to be introduced in the nested loop program.

Let $\mathcal{Q}_\ell, \ell = 1, 2, \cdots, n$ be the set of all polyhedra found by the polyhedral scanning algorithm of an index set $\mathcal{I}$. Note that $\mathcal{P} = \mathcal{P}_0 \cap \mathcal{Q}_1 \cap \mathcal{Q}_2 \cap \cdots \cap \mathcal{Q}_n$. Since points in $\mathcal{I}$ are in one-to-one correspondence with integral points in $\mathcal{P}$, an integral point that is in $\mathcal{P}_d = \mathcal{P}_0 \cap \mathcal{Q}_1 \cap \mathcal{Q}_2 \cap \cdots \cap \mathcal{Q}_d$ but that is not in $\mathcal{I}$ is not in $\mathcal{Q}_{d+1} \cap \mathcal{Q}_{d+2} \cap \cdots \cap \mathcal{Q}_n$. Conversely, an integral point that is not in $\mathcal{Q}_{d+1} \cap \mathcal{Q}_{d+2} \cap \cdots \cap \mathcal{Q}_n$ is not in $\mathcal{P}$ and, thus, is not in $\mathcal{I}$. Consequently, for all points $\mathbf{i}$ iterated over by the **for** loops it must be tested if $\mathbf{i}$ is in $\mathcal{Q}_{d+1} \cap \mathcal{Q}_{d+2} \cap \cdots \cap \mathcal{Q}_n$.

Now call $\mathcal{Q}_{lat} = \mathcal{Q}_{d+1} \cap \mathcal{Q}_{d+2} \cap \cdots \cap \mathcal{Q}_n$. Clearly $\mathcal{P} = \mathcal{P}_d \cap \mathcal{Q}_{lat}$. Since the constraints in $\mathcal{Q}_{lat}$ are irredundant in the constraints in $\mathcal{P}_d$ is must be the case that $\mathcal{Q}_{lat}$ is a d-lattice defining polyhedron.

Section 2.6 shows that whether a point $\mathbf{i}$ is in $\mathcal{Q}_{lat}$ is indicated by the multiplicity $\mathcal{M}(\mathbf{i})$ of point $\mathbf{i}$ with respect to $\mathcal{Q}_{lat}$ and mapping $L()$. Since $\mathcal{Q}_{lat}$ is a d-lattice defining polyhedron by Theorem 2.3, $\mathcal{M}(\mathbf{i}) \in \{0, 1\}$. Moreover, there is only one Ehrhart polynomial associated with $\mathcal{M}(\mathbf{i})$ which is valid for all $\mathbf{i} \in \mathbb{Z}^d$; this is seen as follows. Let me denote the points in $\mathbb{Q}^n$ space by $(\mathbf{x}, \mathbf{c})$ where $\mathbf{x}$ and $\mathbf{c}$ are a rational d-vector and $(n-d)$-vector, respectively. Since $\mathcal{Q}_{lat}$ is a d-lattice defining polyhedron, see (2.17), it has the form

$$\mathcal{Q}_{lat} = \{(\mathbf{x}, \mathbf{c}) \in \mathbb{Q}^n \,|\, \mathbf{s} \leq \begin{bmatrix} A & \Lambda \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{c} \end{bmatrix} \leq \mathbf{t}\} \tag{4.2}$$

The multiplicity $\mathcal{M}(\mathbf{i})$ is the number of integral points in the intersection of $\mathcal{Q}_{lat}$ with the line $\mathbf{i} = L(\mathbf{x}, \mathbf{c}) = \mathbf{x}$, $\mathcal{Q}_{lat}(\mathbf{i}) = \mathcal{Q}_{lat} \,|_{\mathbf{x}=\mathbf{i}}$. Thus $\mathcal{M}(\mathbf{i})$ is the number of points in

$$\mathcal{Q}_{lat}(\mathbf{i}) = \{(\mathbf{i}, \mathbf{c}) \in \mathbb{Q}^n \,|\, \mathbf{s} \leq \begin{bmatrix} A & \Lambda \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \mathbf{c} \end{bmatrix} \leq \mathbf{t}\} \tag{4.3}$$

Now call $\mathbf{s}(\mathbf{i}) = \mathbf{s} - A\mathbf{i}$ and $\mathbf{t}(\mathbf{i}) = \mathbf{t} - A\mathbf{i}$ and substitute this in (4.3).

$$\mathcal{Q}_{lat}(\mathbf{i}) = \{(\mathbf{i}, \mathbf{c}) \in \mathbb{Q}^n \,|\, \mathbf{s}(\mathbf{i}) \leq \Lambda\mathbf{c} \leq \mathbf{t}(\mathbf{i})\} \tag{4.4}$$

Since Λ is a diagonal matrix the constraints of $\mathcal{Q}_{lat}(\mathbf{i})$ are $s_j(\mathbf{i}) \leq \lambda_j c_j \leq t_j(\mathbf{i})$, $j = 1, 2, \cdots, n-d$, where $s_j(\mathbf{i})$ and $t_j(\mathbf{i})$ is the j^{th} element from $\mathbf{s}(\mathbf{i})$ and $\mathbf{t}(\mathbf{i})$, respectively. Clearly all c_j are independent of each other, and the parameterized vertices of $\mathcal{Q}_{lat}(\mathbf{i})$ can be derived easily. The extreme values of c_j are $\frac{s_j(\mathbf{i})}{\lambda_j}$ and $\frac{t_j(\mathbf{i})}{\lambda_j}$. In this way there are 2^{n-d} possible extreme values of $\mathbf{c}$ which define the parameterized vertices of $\mathcal{Q}_{lat}(\mathbf{i})$. Let $\alpha = \alpha_1, \alpha_2, \cdots, \alpha_{n-d}$ be a binary word, then for every α there is a parameterized vertex $\mathbf{v}_\alpha(\mathbf{i}) = (v_{\alpha_1}(\mathbf{i}), v_{\alpha_1}(\mathbf{i}), \cdots, v_{\alpha_{n-d}}(\mathbf{i}))$ where $v_{\alpha_j}(\mathbf{i}) = s_j(\mathbf{i})$ if $\alpha_j = 0$ and $v_{\alpha_j}(\mathbf{i}) = t_j(\mathbf{i})$ if $\alpha_j = 1$. Clearly, all parameterized vertices are an affine function of $\mathbf{i}$ on the complete space $\mathbb{Z}^d$.

Example 4.3 (scanning a non-dense index set) The index set $\mathcal{I}_3$ in Figure 4.4(c) is defined by the polytope $\mathcal{P}_3 = \{(x, c) \in \mathbb{Q}^2 \,|\, 2 \leq x \leq 10 \ \wedge 4c + 1 \leq x \leq 4c + 3\}$. Since the i direction of the index

set corresponds with the x direction of $\mathcal{P}_3$, $\mathcal{P}_3$ is scanned in the lexicographic order (x, c). The scanning algorithm returns two polyhedra $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Polyhedron $\mathcal{Q}_1 = \{x \in \mathbb{Q} \,|\, 2 \leq x \leq 10\}$ and defines the outer loop bounds. Polyhedron $\mathcal{Q}_2 = \{(x, c) \in \mathbb{Q}^2 \,|\, 4c + 1 \leq x \leq 4c + 3\}$. The multiplicity of points $\mathcal{M}(i), i \in \mathbb{Z}$ with respect to $\mathcal{Q}_2$ and mapping $L()$ is found by the Ehrhart test; $\mathcal{M}(i) = [0, 1, 1, 1]_i$. The result of the scanning of Figure 4.4(c) is given in Program 4.4.

Program 4.4: SCANNING $\mathcal{I}_3$ IN FIGURE 4.4(C)
for $i = 2 : 1 : 10$,
if $[0, 1, 1, 1]_i = 1$, ■

The result of the domain scanning of a non-dense index set is an **if** $(Ehrhart)$ statement that has the pseudo polynomial $\mathcal{M}(\mathbf{i})$ as its condition. A node 1c in the parse tree in Figure 4.3 is constructed and is annotated with this statement. Moreover, the polyhedron $\mathcal{P}_d$ on its own defines a dense index set and is scanned as described in previous section to construct parse nodes 1b.

The filtering out of points that do not belong to the index set of the node domain is implemented by Algorithm NODEDOMAIN2GENERICIFSTATEMENT().

```
NODEDOMAIN2GENERICIFSTATEMENT(D)
 1  if dim[controlVector[D]] > 0
 2    then E ← MULTIPLICITY(D)
 3         e ← ENUMERATIONTOPOLYNOMIAL(E, indexVector[D])
 4         SIMPLIFY(e)
 5         lcm ← MAKEINTEGRAL(e)
 6         G ← EHRHARTPOLYNOMIAL2SUBTREE(ehrhartPol, "n")
 7         G ← new GenericIfStatement
 8         condition[G] ← "("TOINTEGRALSTRING(e, "n")")/"lcm" >= 0"
 9         INSERT-ELEMENT(G, G)
10  return G
```

Here the Algorithm EHRHARTPOLYNOMIAL2SUBTREE() is used to convert the Ehrhart polynomial into a vector of which each element defines a periodic coefficient of the polynomial.

```
EHRHARTPOLYNOMIAL2SUBTREE(e, name)
 1  j ← 0
 2  for all terms t of e
 3  do if denom[coeff[t]] = 0
 4       then decl ← TOCDECLARATION(coeff[t])
 5            decl ← name + "_" + j + decl
 6            S ← new SimpleStatement
 7            declaration[simple] ← decl
 8            INSERT-ELEMENT(G, S)
 9            j ← j + 1
10  return G
```

The creation of the **parameter** -statements in node 1a is quite similar to the domain scanning problem. Only the scanning procedure for dense index sets is used since the original Matlab specification does not allows to specify non-dense parameter index sets. Algorithm DOMAIN2PARSTATEMENT() shows the implementation of method to create the **parameter** statements.

```
DOMAIN2PARSTATEMENT(D)
1  D ← CONSTRAINTS2POLYHEDRON(context[D])
2  U ← UNIVERSE-POLYHEDRON(0)
3  S ← POLYHEDRAL-SCAN(D, U) ▷ S is linked list of polyhedra
4  p ← parameterVector[D]
5  L ← dim[p]
6  for ℓ = 1 to L
7  do P ← new ParameterStatement
```

```
 8    (iterator[P], lowerBound[P], upperBound[P]) ← POLYHEDRON2BOUNDS(S_ℓ, ℓ, p)
 9    INSERT-ELEMENT(P, P)
10  return P
```

The main difference between DOMAIN2FORSTATEMENTS and DOMAIN2PARSTATEMENTS is that for the latter the context of the index set is used instead of the constraints.

4.4 Domain matching

This section deals with the conversion of the input and output port domains of the PRDG to parse tree nodes 2a, 2b, 4a, and 4b in Figure 4.3.

In a Kahn process network (KPN) every channel connects a unique output port to a unique input port. Since the topology of the PRDG is described by an extended graph, an output port domain may be connected to multiple input port domains. Since the KPN under construction has the same topology as the PRDG, the PRDG is first brought into a form where each output port domain connects to a single input port domain. This transformation changes the topology of the PRDG by introducing new output port domains and updating the edge domains. I call this transformation POINT-TO-POINT().

After the topology has been adapted to the point-to-point communication the geometry of the output port domains is refined by restricting each domain to the region for which there is actual communication. I call this transformation RECONSTRUCT-OUTPUT-PORT-DOMAINS().

When the transformations POINT-TO-POINT() and RECONSTRUCT-OUTPUT-PORT-DOMAINS() are applied to the PRDG, every output port domain is converted into the parse nodes 4a and 4b, and every input port domain is converted into the parse nodes 2a and 2b of the parse tree in Figure 4.3.

4.4.1 Transforming the PRDG

When the PRDG is created from the SAP, there is exactly one output port domain for every output argument of a function. Moreover, the index set of each output port domain is the index set of the node domain it belong to. Note that this modeling corresponds to the output normal form of the SAP.

To explain the transformations POINT-TO-POINT() and RECONSTRUCT-OUTPUT-PORT-DOMAINS() consider Figure 4.5. The figure shows the extended graph of a PRDG composed of its nodes, ports, and edges. The output port of node P is connected to two input ports, one of C_1 and one of C_2. The figure also shows a number of boxes and diamond-shaped arrows which have their semantics defined in the Unified Modeling Language (UML). A diamond-shaped head is known as an *aggregation* and specifies that the object that has the arrowhead connected to it, has the other object associated with it.

an index set is associated with every node, input port, output port, and edge. Thus node P has the index set $\mathcal{I}_{ND}$ associated with it. When the PRDG is constructed, every output port domain has the index set of the node domain associated with it.

The function POINT-TO-POINT() transforms the PRDG such that every output port domain is associated with at most one edge domain. The function RECONSTRUCT-OUTPUT-PORT-DOMAIN() changes the index set of the output port domain from the index set of the node domain to the index set of the edge domain. The result of applying these transformations to the PRDG of Figure 4.5 is shown in Figure 4.6. This figure differs from Figure 4.5 in that the output port domain is duplicated for both edge domains, and that the index sets associated with the duplicates are the index sets of their associated edge domains (only showed for the top output port domain and top edge domain).

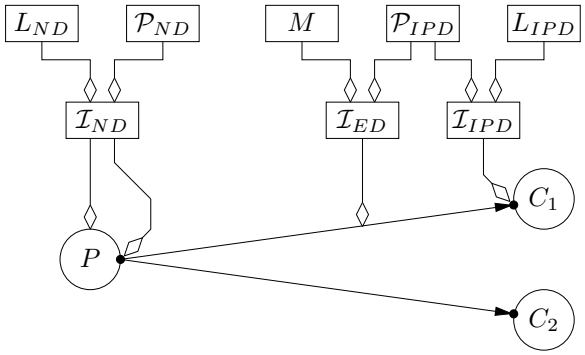


Figure 4.5: An PRDG viewed as an annotated extended graph before POINT-TO-POINT() is applied to it.

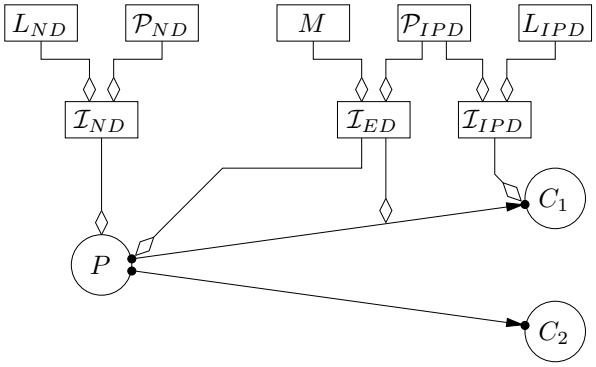


Figure 4.6: The transformed PRDG of Figure 4.5 viewed as an annotated extended graph after POINT-TO-POINT() is applied to it.

The POINT-TO-POINT() procedure is as follows. Let be given the PRDG $G = (\mathcal{N}, \mathcal{E})$. For each edge domain $E_i = ((Q, P_i), \mathcal{I}_e) \in \mathcal{E}$ duplicate Q, call the duplicated version of Q, Q_i, and replace the Q in E_i by Q_i. This results in $E_i = ((Q_i, P_i), \mathcal{I}_e) \in \mathcal{E}$. In addition, determine the node domain N for which $Q \in O_N$ and insert Q_i in the set O_N. Finally, clean up all node domains $N \in \mathcal{N}$ by removing the original Qs from O_N. The pseudo-code for this procedure is given in Algorithm POINT-TO-POINT.

POINT-TO-POINT(G)
1 $\triangleright$ $G = (\mathcal{N}, \mathcal{E})$
2 **for all** $E_i \in \mathcal{E}, E_i = ((Q, P_i), \mathcal{I}_e)$
3 **do** $Q_i \leftarrow Q$ $\triangleright$ duplicate the output port domain
4 $E_i \leftarrow ((Q_i, P_i), \mathcal{I}_e)$
5 ADD-DOMAIN(O_N, Q_i) $\triangleright$ where O_N is the node domain that contains Q
6 CLEAN-UP() $\triangleright$ remove all original output port domains from all O_N of all $N \in \mathcal{N}$

The RECONSTRUCT-OUTPUT-PORT-DOMAIN() procedure is as follows. Let be given the PRDG $G = (\mathcal{N}, \mathcal{E})$. For each edge domain $E = ((Q, P), \mathcal{I}_e) \in \mathcal{E}$ set the index set of Q to $\mathcal{I}_e$. The pseudo code for this procedure is given in Algorithm RECONSTRUCT-OUTPUT-PORT-DOMAINS.

RECONSTRUCT-OUTPUT-PORT-DOMAINS(G)
1 $\triangleright$ $G = (\mathcal{N}, \mathcal{E})$
2 **for all** $E_i \in \mathcal{E}, E_i = ((Q_i, P_i), \mathcal{I}_e)$
3 **do** SET-DOMAIN($Q_i, \mathcal{I}_e$)

After these two transformations are applied, the communicating input/output port domain pair is said to *match*. This means that for every $\mathbf{j} \in \mathcal{I}_{OPD}$ there is at least one $\mathbf{i} \in \mathcal{I}_{IPD}$ at which the data produced at $\mathbf{j}$ is to be consumed, and, for every $\mathbf{i} \in \mathcal{I}_{IPD}$ there is exactly one $\mathbf{j} \in \mathcal{I}_{OPD}$ that produces that data to be consumed at iteration $\mathbf{i}$. This is expressed by the the following two series of implications.

$$\mathbf{j} \in \mathcal{I}_{OPD} \Rightarrow \mathbf{j} \in \mathcal{I}_{ED} \Rightarrow \exists\, \mathbf{k} \in \mathcal{P}_{IPD}\ s.t.\ \mathbf{j} = M(\mathbf{k}) \Rightarrow \mathbf{i} = L_{IPD}(\mathbf{k}) \in \mathcal{I}_{IPD}$$
$$\mathbf{i} \in \mathcal{I}_{IPD} \Rightarrow \exists\, \mathbf{k} \in \mathcal{P}_{IPD}\ s.t.\ \mathbf{i} = L_{IPD}(\mathbf{k}) \Rightarrow \mathbf{j} = M(\mathbf{k}) \in \mathcal{I}_{ED} = \mathcal{I}_{OPD}$$

4.4.2 Creating the parse nodes

This section deals with the actual generation of the nodes 2a, 2b, 4a, and 4b of Figure 4.3. These nodes represent a set of nested **if** statements that filter out iterations for which there is no communication between an output and input port domain.

When the transformations that are described in the previous section have been performed, output port domains match with input port domains. Let be given node domain $N = (I_N, O_N, f_n, \mathcal{I}_n)$, and let $IPD \in I_N$ and $OPD \in O_N$ be an input port domain and an output port domain, with index sets $\mathcal{I}_{IPD}$ and $\mathcal{I}_{OPD}$, respectively. Every input port domain in I_N is converted to an input port of the Kahn process and, similarly, every output port domain in O_N is converted to an output port of the Kahn process. The domain scanning procedure described in the previous section scans index set $\mathcal{I}_n$. So, for every iteration $\mathbf{i} \in \mathcal{I}_n$ data must be read from the port that is derived from IPD when $\mathbf{i} \in \mathcal{I}_{IPD}$. Similarly, data must be written to the port derived from OPD when $\mathbf{i} \in \mathcal{I}_{OPD}$. Because of the domain scanning, it is already known that $\mathbf{i} \in \mathcal{I}_n$. So, it remains to test whether $\mathbf{i} \in \mathcal{I}_{IPD}$ and/or $\mathbf{i} \in \mathcal{I}_{OPD}$. (4.5) shows what the test is for every $IPD \in I_N$.

$$\mathbf{i} \in \mathcal{I}_{IPD} \quad \text{when } \mathcal{M}_{\mathcal{I}_{IPD}}(\mathbf{i}) = 1 \tag{4.5}$$

where $\mathcal{M}_{\mathcal{I}_{IPD}}(\mathbf{i})$ is the multiplicity of $\mathbf{i}$ with respect to $\mathcal{P}_{IPD}$ and $L_{IPD}()$, where $\mathcal{P}_{IPD}$ and $L_{IPD}()$ are the polytope and projection that define $\mathcal{I}_{IPD}$. (4.6) shows what the test is for every $OPD \in O_N$.

$$\mathbf{i} \in \mathcal{I}_{OPD} \quad \text{when } \mathcal{M}_{\mathcal{I}_{OPD}}(\mathbf{i}) \geq 1 \tag{4.6}$$

where $\mathcal{M}_{\mathcal{I}_{OPD}}(\mathbf{i})$ is the multiplicity of $\mathbf{i}$ with respect to $\mathcal{P}_{OPD}$ and $M_{OPD}()$, where $\mathcal{P}_{OPD}$ and $M_{OPD}()$ are the polytope and mapping that define $\mathcal{I}_{OPD}$.

Constructing the parse nodes for input port domains

Rather than directly deriving $\mathcal{M}_{\mathcal{I}_{IPD}}(\mathbf{i})$ in (4.5), it makes sense to use the fact that $\mathcal{I}_{IPD}$ is a periodic lattice polyhedron and that $\mathcal{I}_{IPD} \subseteq \mathcal{I}_{ND}$, where $\mathcal{I}_{ND}$ is the index set of the node domain to which IPD belongs. Let, according to Definition 2.24, $\mathcal{I}_{IPD} = L(\mathcal{P}_{IPD} \cap \mathbb{Z}^n), \mathcal{I}_{IPD} \in \mathbb{Z}^d$. Like in the polyhedral scanning of non-dense index sets, the function POLYHEDRAL-SCAN is used to decompose $\mathcal{P}_{IPD}$ into $\mathcal{P}_{E_{IPD}} \cap \mathcal{P}_{L_{IPD}}$, where $\mathcal{P}_{E_{IPD}}$ is an embedded polyhedron and $\mathcal{P}_{L_{IPD}}$ is a d-lattice defining polyhedron. By Theorem 2.4, $\mathcal{I}_{IPD} = \mathcal{Q}_{IPD} \cap L(\mathcal{P}_{L_{IPD}} \cap \mathbb{Z}^n)$, where $\mathcal{Q}_{IPD}$ is the projection of $\mathcal{P}_{E_{IPD}}$ onto $\mathbb{Q}^d$. Call $\mathcal{I}_L = L(\mathcal{P}_{L_{IPD}})$. Since $\mathbf{i} \in \mathcal{I}_{IPD}$ when $\mathbf{i} \in \mathcal{Q}_{IPD} \wedge i \in \mathcal{I}_L$, the multiplicity $\mathcal{M}_{\mathcal{I}_{IPD}}(\mathbf{i})$ can be rewritten as follows:

$$\mathcal{M}_{\mathcal{I}_{IPD}}(\mathbf{i}) = \begin{cases} \mathcal{M}_{\mathcal{I}_L}(\mathbf{i}) & \text{if } \mathbf{i} \in \mathcal{Q}_{IPD} \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Note that, as with the domain scanning, the validity domain of $\mathcal{M}_{\mathcal{I}_L}(\mathbf{i}) = \mathbb{Z}^d$.

The test $\mathbf{i} \in \mathcal{Q}_{IPD}$ is simplified by using the fact that $\mathbf{i} \in \mathcal{I}_{ND}$. Similar to the alternative representation of $\mathcal{I}_{IPD}$, POLYHEDRAL-SCAN is used to derive the alternative representation of $\mathcal{I}_{ND}$, $\mathcal{I}_{ND} = \mathcal{Q}_{ND} \cap L(\mathcal{P}_{L_{ND}} \cap \mathbb{Z}^m)$. The fact that $\mathbf{i} \in \mathcal{I}_{ND}$ implies $\mathbf{i} \in \mathcal{Q}_{ND}$. Let $\mathcal{P}_{\mathbf{if}}$ be the polytope $\mathcal{P}_{IPD}$ with all

redundant constraints in $\mathcal{Q}_{ND}$ removed from it; $\mathcal{P}_{\textbf{if}} = \text{DOMAINSIMPLIFY}(\mathcal{P}_{IPD}, \mathcal{P}_{ND})$. Therefore, the test if $\mathbf{i} \in \mathcal{Q}_{IPD}$ is simplified to the test if $\mathbf{i} \in \mathcal{P}_{\textbf{if}}$.

With both simplifications (4.5) becomes (4.8).

$$\mathbf{i} \in \mathcal{I}_{IPD} \quad \text{when } \mathbf{i} \in \mathcal{P}_{\textbf{if}} \wedge \mathcal{M}_{\mathcal{I}_L}(\mathbf{i}) = 1 \tag{4.8}$$

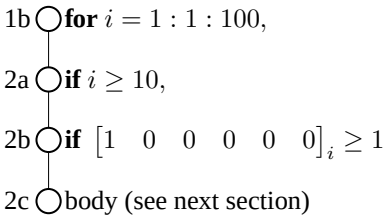
The nodes 2a are constructed from the constraints in $\mathcal{P}_{\textbf{if}}$. Every constraint in $\mathcal{P}_{\textbf{if}}$ is the condition of an **if** -statement. Like in DOMAIN SCANNING, the Ehrhart polynomial $\mathcal{M}_{\mathcal{I}_L}(\mathbf{i})$ is converted to a single **if** -statement.

Example 4.4 (input port domain code generation) Let be given a node domain with index set $\mathcal{I}_{ND} = L_{ND}(\mathcal{P}_{ND} \cap \mathbb{Z}^2)$. Let $\mathcal{P}_{ND} = \{(x_1, x_2) \in \mathbb{Q}^2 \,|\, 0 \leq x_1 \leq 100 \wedge x_1 = 2x_2\}$, and let $L_n = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

Further, let be given an input port domain with index set $\mathcal{I}_{IPD} = L_{IPD}(\mathcal{P}_{IPD} \cap \mathbb{Z}^3)$ with $\mathcal{P}_{IPD} = \{(x_1, x_2, x_3) \in \mathbb{Q}^3 \,|\, 10 \leq x_1 \leq 100 \wedge x_1 = 2x_2 \wedge x_1 = 3x_3\}$, and $L_{IPD} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$.

The index sets $\mathcal{I}_{ND}$ and $\mathcal{I}_{IPD}$ are rewritten by using the polyhedral scanning method into $\mathcal{I}_{ND} = \mathcal{Q}_{ND} \cap L(\mathcal{P}_{L_{ND}} \cap \mathbb{Z}^2)$ and $\mathcal{I}_{IPD} = \mathcal{Q}_{IPD} \cap L(\mathcal{P}_{L_{IPD}} \cap \mathbb{Z}^3)$. Here $\mathcal{Q}_{ND} = \{x_1 \in \mathbb{Q} \,|\, 0 \leq x_1 \leq 100\}$, $\mathcal{Q}_{IPD} = \{x_1 \in \mathbb{Q} \,|\, 10 \leq x_1 \leq 100\}$. The non redundant constraints in $\mathcal{Q}_{ND}$ with respect to $\mathcal{Q}_{IPD}$ are $\mathcal{P}_{\textbf{if}} = \{x_1 \in \mathbb{Q} \,|\, x_1 \geq 10\}$.

The 1-lattice defining polyhedron for the input port domain is $\mathcal{P}_{L_{IPD}} = \{(x_1, x_2, x_3) \in \mathbb{Q}^3 \,|\, x_1 = 2x_2 \wedge x_1 = 2x_3\}$. Let i be the index variable of the index set. By using the Ehrhart test for this index set we get $\mathcal{M}_{\mathcal{I}_{L_{IPD}}}(i) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_i, i \in \mathbb{Q}$. Figure 4.7 shows the parse tree that is constructed and the corresponding piece of code; where the outer loop is the result of domain scanning as described in the previous section. The annotations at the left hand side of the nodes correspond to the annotations in Figure 4.3.



(a) parse tree

Program 4.5: PROGRAM OF EXAMPLE 4.4

```
for i = 1 : 1 : 100,
    if i ≥ 10,
        if [1 0 0 0 0 0]_i ≥ 1,
            body (see next section)
        end
    end
end
```

(b) corresponding piece of code

Figure 4.7: The parse tree that is generated (a) and the code it represents (b). ■

The creation of the parse nodes 2a and 2b is implemented by Algorithms INPUTPORT2IFSTATEMENT() and INPUTPORT2GENERICIFSTATEMENT(), respectively. The former algorithm creates **if** -statements that tightens the index set of the node domain. The latter algorithm filters out point that do belong to the index set of the node domain but not to the index set of the input port domain.

INPUTPORT2IFSTATEMENT(P)
1 $\mathcal{I}_{nd} \leftarrow domain[node[P]]$
2 $\mathcal{I}_{ipd} \leftarrow domain[P]$
3 $A_{nd} \leftarrow indexConstraints[\mathcal{I}_{nd}]$
4 $A_{ipd} \leftarrow indexConstraints[\mathcal{I}_{ipd}]$
5 $A = \text{CONSTRAINTSSIMPLIFY}(A_{ipd}, A_{nd})$
6 **return** DOMAIN2IFSTATEMENT($A, indexParameterVector[\mathcal{I}_{ipd}]$)

Here DOMAIN2IFSTATEMENT is used to convert a constraint matrix into an vector of **if** -statements.

```
DOMAIN2IFSTATEMENT(A, x)
 1  v ← TOLINEAREXPRESSION(A, x)
 2  for j = 0 to dim[v] − 1
 3  do if v_j ≠ "1 ≥ 0"
 4        then I ← new IfStatement
 5             expression[I] ← v_j
 6             INSERT-ELEMENT(I, I)
 7  return I
```

```
INPUTPORT2GENERICIFSTATEMENT(P)
 1  N ← node[P]
 2  I_nd ← domain[N]
 3  I_ipd ← domain[P]
 4  if (nbRows[mapping[I_ipd]] < dim[dpVec[I_ipd]] − dim[pVec[I_ipd]] ||
 5    dim[dpVec[I_ipd]] > dim[dpVec[I_nd]])
 6    then E ← MULTIPLICITY(I_ipd) ▷ E is an enumeration
 7         if E ≠ NIL
 8           then e ← ENUMERATIONTOPOLYNOMIAL(E, ipVecI_nd)
 9                SIMPLIFY(e)
10                lcm ← MAKEINTEGRAL(e)
11                G ← EHRHARTPOLYNOMIAL2SUBTREE(e, "m" + count)
12                G ← new GenericIfStatement
13                condition[G] ← "("TOINTEGRALSTRING(e, name)")/"lcm" >= 0"
14                INSERT-ELEMENT(G, G)
15                count ← count + 1
16  return G
```

Constructing the parse nodes for output port domains

Equation (4.6) states that $\mathbf{i} \in \mathcal{I}_{OPD}$ when $\mathcal{M}_{\mathcal{I}_{OPD}}(\mathbf{i}) \geq 1$. $\mathcal{I}_{OPD}$ is a linearly bounded lattice defined by an integral affine function $M()$ and a polytope $\mathcal{P}_{IPD}$. Since $\mathcal{P}_{IPD}$ is not defined in the space of the node domain to which $\mathcal{I}_{OPD}$ belongs, we cannot simplify as with input port domains. So, the Ehrhart test is directly applied to the index set $\mathcal{I}_{OPD} = (M, \mathcal{P}_{IPD})$. The result is a list of Ehrhart polynomials, each having its own parameter validity domain.

Let the Ehrhart test result in the Ehrhart polynomials $p_0(\mathbf{i})$, $p_1(\mathbf{i})$, $\cdots$, $p_{n-1}(\mathbf{i})$ with corresponding parameter validity domains $\mathcal{P}_0$, $\mathcal{P}_1$, $\cdots$, $\mathcal{P}_n$, respectively. Since the iteration vector $\mathbf{i} \in \mathbb{Z}^d$ is used to specify the counting problem, the result is parameterized parameterized in $\mathbf{i}$ and the parameter validity domains are in $\mathbb{Z}^d$ as well. So, a point $\mathbf{i}$ is in $\mathcal{I}_{OPD}$ when it is any of the $\mathcal{P}_j, j = 0, 1, \cdots, n-1$ and when for the corresponding polynomial $p_j(\mathbf{i})$ it is true that $p_j(\mathbf{i}) \geq 0$. Moreover, let $\mathcal{I}_{ND} = L(\mathcal{P}_{ND} \cap \mathbb{Z}^n)$ be the index set of the node domain that contains $\mathcal{I}_{OPD}$. Since by the domain scanning $\mathbf{i} \in \mathcal{I}_{ND}$, it is also true that $\mathbf{i} \in \mathcal{P}_{\mathcal{I}_{ND}}$, where $\mathcal{P}_{\mathcal{I}_{ND}} = \text{proj}_{QQ^d}(\mathcal{P}_{ND})$. This fact is used to simplify the test whether $\mathbf{i}$ is in any of the polytopes $\mathcal{P}_j, j = 0, 1, \cdots, n-1$ by removing the redundant constraints in $\mathcal{P}_{ND}$ from each of them; $\mathcal{P}'_j = \text{DOMAINSIMPLIFY}(\mathcal{P}_j, \mathcal{P}_{ND}), j = 0, 1, \cdots, n-1$. Figure 4.8 shows the structure of the part of the parse tree that corresponds to the filtering process for output port domains. A boolean variable w is used to indicate whether point $\mathbf{i}$ is in some $\mathcal{P}'_j, j = 0, 1, \cdots, n-1$ and whether the multiplicity for this point is greater than zero. The nodes in the figure of the form **if** $\mathbf{i} \in \mathcal{P}'_j, j = 0, 1, \cdots, n-1$ are converted to a set of **if** -statements, one for each constraint in $\mathcal{P}'_j$.

Example 4.5 (output port domain code generation) Let be given a node domain with index set $\mathcal{I}_n = (I, \mathcal{P}_n)$, where I is the identity matrix and $\mathcal{P}_n = \{x_1 \in \mathbb{Q} \,|\, 0 \leq x_1 \leq 3N + 3\}$.

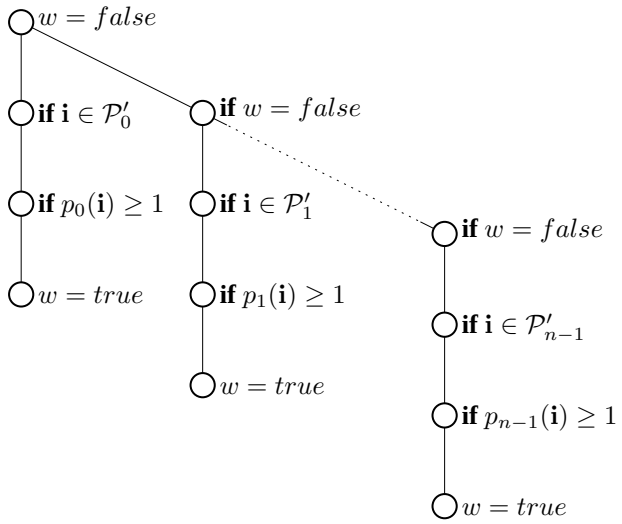


Figure 4.8: Structure of the sub tree for the output port domains.

Further, let be given the output port domain of Example 2.5, that is, $\mathcal{P}(N) = \{(x_1, x_2) \in \mathbb{Q}^2 \,|\, 0 \leq x_1 \leq N \ \wedge \ x_1 \leq x_2 \leq N\}$ and $M(\mathbf{k}) = \begin{bmatrix} 2 & 1 \end{bmatrix} \mathbf{k} + 3, \mathbf{k} \in \mathcal{P}(N) \cap \mathbb{Z}^2$.

The multiplicity is expressed by the Ehrhart polynomials with their parameter validity domains as in (2.21). The parameter validity domains are simplified by removing their redundant constraints in $\mathcal{I}_n$. In this example the only redundant constraint is $j \leq 3N + 3$.

The parse tree that is generated has the form as depicted in Figure 4.8, and the corresponding piece of code is given in Program 4.6.

Program 4.6: PROGRAM OF EXAMPLE 4.5

```
 1   parameter N 8 16;
 2   for j = 1 : 1 : 3N + 3,
 3     w = false;
 4     if −j + N + 3 ≥ 0,
 5       if j − 3 ≥ 0,
 6         if 1/3 j + [0, −1/3, −2/3]_j ≥ 1,
 7           w = true;
 8         end
 9       end
10     end
11     if w == false,
12       if j − N − 3 ≥ 0,
13         if 1/2 N + [−1/6 j + [1, 7/6, 1/3, 3/2, 2/3, 5/6]_j, −1/6 j + [3/2, 2/3, 5/6, 1, 7/6, 1/3]_j]_N ≥ 1,
14           w = true;
15         end
16       end
17     end
18     if w == true,
19       body (see next section)
20     end
21   end
```

Lines 1 and 2 are the result of the domain scanning procedure and is shown for the context here. Lines 3 – 17 show the actual code that is produced by the method in this subsection. Lines 18–20 ensure that the proper value is written to the output port of the process. This is the topic of the next section. ■

The creation of the parse nodes 4a and 4b are implemented by Algorithm OUTPUTPORT2- GENERICIFSTATEMENTS. The algorithm is called with the output port $\mathcal{Q}$ for which the code is to be generated.

OUTPUTPORT2GENERICIFSTATEMENTS(Q)

```
 1  G ← NIL
 2  I ← NIL
 3  I_nd ← domain[node[Q]]
 4  A_nd ← indexConstraints[I_nd]
 5  I_opd ← domain[Q]
 6  S ← new SimpleAssignStatement
 7  lhs[S] ← "doWrite"
 8  rhs[S] ← "false"
 9  INSERT-ELEMENT(G, S)
10  s ← S ▷ s is a stitch node
11  E ← MULTIPLICITY(I_opd, I_nd)
12  for all enumerations e in E
13  do ENUMERATIONDOMAINSIMPLIFY(e, I_nd)
14     A_opd ← POLYHEDRON2CONSTRAINTS(validityDomain[e])
15     I ← DOMAIN2IFSTATEMENT(A_opd)
16     s ← STITCH(I, s)
17     p ← ENUMERATIONTOPOLYNOMIAL(e, ipVec[I_nd])
18     SIMPLIFY(p)
19     lcm ← MAKEINTEGRAL(e)
20     G ← EHRHARTPOLYNOMIAL2SUBTREE(p, "d" + count)
21     G ← new GenericIfStatement
```

```
22      condition[G] ← "("TOINTEGRALSTRING(e, name)")/"lcm" >= 0"
23      INSERT-ELEMENT(G, G)
24      count ← count + 1
25      s ← STITCH(s, G)
26      S ← new SimpleAssignStatement
27      lhs[S] ← "doWrite"
28      rhs[S] ← "true"
29      INSERT-ELEMENT(s, S)
30      if e is not the last enumeration in E
31        then G ← new GenericIfStatement
32             condition[G] ← "doWrite == false")
33             INSERT-ELEMENT(G, G)
34             s ← G
35  return G
```

The pseudo code for the STITCH() function and the role of the stitch node is dealt with in Chapter 5.3.

4.5 Linearization

This section deals with the conversion of the indexing functions that access the higher dimensional arrays to parse tree nodes 2c and 4c in Figure 4.3.

The linearization procedure has already been introduced at the beginning of this Chapter. Let $E = (Q, P, \mathcal{J})$ be an edge domain that is mapped onto the Kahn channel that connects the two processes onto which Q and P are mapped. The scanning of the node domain that contains Q also imposes a scanning of Q itself. The rank is determined for every point in the index set of Q. Let $\mathcal{J}$ be the index set $\mathcal{J} = M(\mathcal{P} \cap \mathbb{Z}^n)$. The *rank* of point $\mathbf{j} \in \mathcal{J}$ is the number of points in $\mathcal{J}$ that are lexicographic smaller than $\mathbf{j}$. The expression that ranks all points $\mathbf{j} \in \mathcal{J}$ is called the *ranking function* and is denoted by $\mathrm{rank}(\mathbf{j})$. The procedure to derive the ranking function is called *ranking*.

I follow the approach that tokens to be sent are written on the channel in the same order as they are produced. The consuming process reads the tokens from the channel and stores the tokens in private memory in the same order as they are read from the channel. The function that specifies the write addresses of the tokens is called the *write function*. Because the complete process from production to storage is in order, the write function equals the ranking function.

4.5.1 Counting in index sets

Chapter 2 describes a method to count the number of integral points contained in a polytope. In this section I give a method to count the number of integral points contained by an index set.

Let an index set $\mathcal{I} = M(\mathcal{P} \cap ZZ^n)$ be given. When M is a bijection from $\mathcal{P}$ to $\mathcal{I}$, the points in $\mathcal{I}$ are in one-to-one correspondence with the integral points in $\mathcal{P}$. So, for this case the number of points contained in $\mathcal{I}$ is equal to the number of points in $\mathcal{P} \cap \mathbb{Z}^n$ and, thus, $|\mathcal{I}| = EP(\mathcal{P})$.

Now consider the case that M is not a bijection from $\mathcal{P}$ to $\mathcal{I}$. An example of such an index set is already given in Example 2.5. Since the multiplicity of some of the points $\mathbf{i} \in \mathcal{I}$ is greater than one, counting the number of integral points in $\mathcal{P}$ would count some iterations $\mathbf{i}$ more than once. In the following I describe a method to count the number of points in an index set in which all points in the polytope that map onto the same point lie on the same line in the polytope. This case corresponds to a integral affine function $M()$ whose matrix M has row rank $n - 1$, n being the dimension of the space that contains the polytope.

Lemma 4.1
Let $\mathcal{I} \in \mathbb{Z}^d$ be the index set $M(\mathcal{P} \cap \mathbb{Z}^n)$. The number of point in $\mathcal{I}$ is given by:

$$|\mathcal{I}| = |\mathcal{P} - \mathcal{P}_{br}| \tag{4.9}$$

where "-" is the polyhedral difference and $\mathcal{P}_{br}$ is any polyhedral subset of $\mathcal{P}$ that meets the following two requirements:

1. $\mathcal{I} = M((\mathcal{P} - \mathcal{P}_{br}) \cap \mathbb{Z}^n)$
2. for every $\mathbf{i} \in \mathcal{I}$, $\mathcal{M}_{\mathcal{P}_u}(\mathbf{i}) \in \{0, 1\}$, where $\mathcal{P}_u = \mathcal{P} - \mathcal{P}_{br}$ and $\mathcal{M}_{\mathcal{P}_u}(\mathbf{i})$ is the multiplicity with respect to the polytope $\mathcal{P}_u$.

Proof
Requirement 1 states that every point in $\mathcal{I}$ is mapped onto from $\mathcal{P}_u$ by M and requirement 2 states that at most one point from $\mathcal{P}_u$ maps onto the points in $\mathcal{I}$. These two requirements thus state that M is a bijection from $\mathcal{P}_u$ to $\mathcal{I}$ and thus (4.9) holds. ■

I will call $\mathcal{P}_{br}$ and $\mathcal{P}_u$ the broadcast and unicast polytopes of $\mathcal{P}$ respectively.

Let n be the dimension of the space that contains $\mathcal{P}$. To find $\mathcal{P}_u$ the problem then is to derive a $\mathcal{P}_{br}$ that meets the two requirements in Lemma 4.1. When the matrix M_1, the matrix composed of the first n columns of M, has a row rank equal to $n - 1$, there is a procedure to find $\mathcal{P}_{br}$ [91].

4.5.2 Ranking

Definition 4.1 (pre-image of a polyhedron)
Let be given a polyhedron $\mathcal{Q} = \{\mathbf{y} \in \mathbb{Q}^d \mid A\mathbf{y} = B\mathbf{p} + \mathbf{b} \wedge C\mathbf{y} \geq D\mathbf{p} + \mathbf{d}\}$, and an integral affine function $M() : \mathbb{Q}^n \rightarrow \mathbb{Q}^d, \mathbf{y} = M(\mathbf{x}) = M\mathbf{x} + \mathbf{m}$. The pre-image $\mathcal{Q}'$ of Q under $M()$ is defined by

$$\mathcal{Q}' = \{\mathbf{x} \in \mathbb{Q}^n \mid AM(\mathbf{x}) = B\mathbf{p} + \mathbf{b} \wedge CM(\mathbf{x}) \geq D\mathbf{p} + \mathbf{d}\} \tag{4.10}$$

Note that there are no constraints on $M()$, that is, M is allowed to be any integral matrix of proper dimensions. Also note that in the case when there is no $\mathbf{x}$ that satisfies (4.10) then $\mathcal{Q}' = \emptyset$. As a short hand for pre-image under integral affine function $M()$, I write $\mathcal{Q}' = M^{-1}(\mathcal{Q})$. ■

It is clear that for every point $\mathbf{x} \in \mathcal{Q}'$ in (4.10) it is true that $M(\mathbf{x}) \in \mathcal{Q}$. This is seen by substituting $\mathbf{y} = M(\mathbf{x})$ into (4.10). This results in $A\mathbf{y} = B\mathbf{p} + \mathbf{b} \wedge C\mathbf{y} \geq D\mathbf{p} + \mathbf{d}$ and, thus, $\mathbf{y} \in \mathcal{Q}$.

Lemma 4.2
Let the pre-image $\mathcal{Q}' = M^{-1}(\mathcal{Q})$, $\mathcal{Q}' \subset \mathbb{Q}^n$ as in (4.10). $\mathcal{Q}'$ is the largest subset of $\mathbb{Q}^n$ for which $M(\mathcal{Q}') \subseteq \mathcal{Q}$. In other words, there are no points $\mathbf{x}$ outside $\mathcal{Q}'$ for which $M(\mathbf{x}) \in \mathcal{Q}$.

Proof
The proof is by contradiction. Assume there is an $\mathbf{x}$ outside $\mathcal{Q}'$ for which $M(\mathbf{x}) \in \mathcal{P}$. Call $\mathbf{y}$ this point in $\mathcal{P}$, $\mathbf{y} = M(\mathbf{x})$. Substitution $M(\mathbf{x})$ of $\mathbf{y}$ into $A\mathbf{y} = B\mathbf{p} + \mathbf{b} \wedge C\mathbf{y} \geq D\mathbf{p} + \mathbf{d}$ results in $AM(\mathbf{x}) = B\mathbf{p} + \mathbf{b} \wedge CM(\mathbf{x}) \geq D\mathbf{p} + \mathbf{d}$. Therefore $\mathbf{x} \in \mathcal{Q}'$ with contradicts the assumption that $\mathbf{x}$ is outside $\mathcal{Q}$. ■

Theorem 4.1
Let be given an index set $\mathcal{I} \subset \mathbb{Z}^d, \mathcal{I} = M(\mathcal{P} \cap \mathbb{Z}^n)$, and a polyhedron $\mathcal{Q} \subset \mathbb{Q}^d$. The intersection of $\mathcal{I}$ with $\mathcal{Q}$ is given by

$$\mathcal{I} \cap \mathcal{Q} = M(\mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n) \tag{4.11}$$

where $\mathcal{Q}' = M^{-1}(\mathcal{Q})$ is the pre-image of $\mathcal{Q}$ under $M()$.

Proof
The theorem states that $\mathcal{I} \cap \mathcal{Q} = \{\mathbf{i} \,|\, \mathbf{i} = M(\mathbf{k}), \mathbf{k} \in \mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n\}$. The proof is given in two parts. Part a) proves that $\mathbf{i} \in \mathcal{I} \cap \mathcal{Q}$ implies that $\exists \mathbf{k} \in \mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n$ such that $\mathbf{i} = M(\mathbf{k})$. Part b) proves that $\mathbf{k} \in \mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n$ implies that $\mathbf{i} = M(\mathbf{k}) \in \mathcal{I} \cap \mathcal{Q}$. Note that for the case that $\mathcal{I} \cap \mathcal{Q} = \emptyset$ that part b) also proves that $\mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n = \emptyset$ since else $\mathcal{I} \cap \mathcal{Q}$ would not be empty.

a) Let $\mathbf{i} \in \mathcal{I} \cap \mathcal{Q}$. Because $\mathbf{i} \in \mathcal{I}$, there exists at least one $\mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n$ such that $\mathbf{i} = M(\mathbf{k})$; let $\mathbf{k}$ be such a point. Because $\mathbf{i} \in \mathcal{Q}$ and $\mathbf{k}$ is such that $\mathbf{i} = M(\mathbf{k})$ it follows that $M(\mathbf{k}) \in \mathcal{Q}$ and by Lemma 4.2 $\mathbf{k} \in \mathcal{Q}'$. Because $\mathbf{k} \in \mathcal{Q}'$ and $\mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n$, $\mathbf{k} \in \mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n$.

b) Let $\mathbf{k} \in \mathcal{P} \cap \mathcal{Q}' \cap \mathbb{Z}^n$. Because $\mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n$ it follows that $\mathbf{i} = M(\mathbf{k}) \in \mathcal{I}$, by definition. Because $\mathbf{k} \in \mathcal{Q}'$ it follows that $M(\mathbf{k}) \in \mathcal{Q}$. Because $\mathbf{i} = M(\mathbf{k})$, $\mathbf{i} \in \mathcal{Q}$ as well. Because $\mathbf{i}$ is in both $\mathcal{I}$ and $\mathcal{Q}$ it follows that $\mathbf{i} \in \mathcal{I} \cap \mathcal{Q}$. ■

Example 4.6 (intersection of LBL with polytope) Let be given $\mathcal{I} = M(\mathcal{P} \cap \mathbb{Z}^2)$, where $\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^2 \,|\, 11 \leq x_1 \leq 15 \wedge 16 - x_1 \leq x_2 \leq 5)\}$ and $M = [1 \; -2]$, and polytope $\mathcal{Q} = \{y \in \mathbb{Q} \,|\, 9 \leq y \leq 15\}$. The derivation of $\mathcal{I} \cap \mathcal{Q}$ is illustrated in Figure 4.9. The bottom of Figure 4.9 shows the index set $\mathcal{I}$ indicated by the set of all dots. The intersection of $\mathcal{I}$ with $\mathcal{Q}$ is indicated by the black dots. The set $\mathcal{I} \cap \mathcal{Q}$ is obtained as follows. First $\mathcal{Q}' = M^{-1}(\mathcal{Q})$ is derived. Second the polytope $\mathcal{P} \cap \mathcal{Q}$ is derived, indicated by the dark gray shaded region. By applying $M()$ to the integral points in this region $\mathcal{I} \cap \mathcal{Q}$ is obtained.

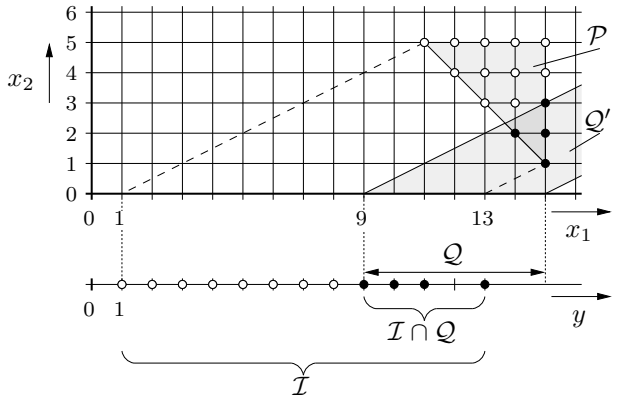


Figure 4.9: Intersection of index set $\mathcal{I} = M(\mathcal{P} \cap \mathbb{Z}^2)$, $M = [1 \; -2]$ with $\mathcal{Q}$. $\mathcal{I}$ is indicated by all the dots on the j-axis. $\mathcal{I} \cap \mathcal{Q}$ is indicated by the black dots. ■

Definition 4.2 (lexicographic order)
Let $\mathcal{Q}_\ell(\mathbf{j})$ define a family of parameterized polyhedra

$$\mathcal{Q}_\ell(\mathbf{j}) = \{\mathbf{y} \in \mathbb{Q}^d \,|\, y_\ell \leq j_\ell - 1 \wedge y_i = j_i, i = 1, 2, \cdots, \ell - 1\} \tag{4.12}$$

for $\ell = 1, 2, \cdots, d$ and $\mathbf{j} \in \mathbb{Z}^d$. Further let R be the relation defined on $\mathbb{Z}^d$ given by

$$R = \{(\mathbf{i}, \mathbf{j}) \in \mathbb{Z}^{d \times d} \,|\, \exists\, \ell \in \{1, 2, \cdots, d\} \text{ s.t. } \mathbf{i} \in \mathcal{Q}_\ell(\mathbf{j})\} \tag{4.13}$$

R is called the *lexicographic order*. When a pair $(\mathbf{i}, \mathbf{j}) \in R$ we write $\mathbf{i} \prec \mathbf{j}$ and say that $\mathbf{i}$ lexicographical precedes $\mathbf{j}$. ■

Definition 4.3 (lexicographic expansion)
Let be given a set $S \subset \mathbb{Z}^d$. The lexicographic expansion $\mathcal{S}_{lex}(\mathbf{j})$ of $\mathcal{S}$ with respect to a point $\mathbf{j} \in \mathbb{Z}^d$ is the following collection of d mutually disjoint subsets of $\mathcal{S}$:

$$S_{lex}(\mathbf{j}) = \{\mathcal{S}_1(\mathbf{j}), \mathcal{S}_2(\mathbf{j}), \cdots, \mathcal{S}_d(\mathbf{j})\} \tag{4.14}$$

where

$$\mathcal{S}_\ell(\mathbf{j}) = \{\mathbf{i} \in \mathcal{S} \,|\, \mathbf{i} \in \mathcal{Q}_\ell(\mathbf{j})\} = \mathcal{S} \cap \mathcal{Q}_\ell(\mathbf{j}) \tag{4.15}$$

and $\mathcal{Q}_\ell(\mathbf{j})$ as defined in (4.12). ■

Theorem 4.2
Let index set $\mathcal{I} \subset \mathbb{Z}^d$ be given. Further let $\mathcal{I}_{lex}(\mathbf{j}) = \{\mathcal{I}_1(\mathbf{j}), \mathcal{I}_2(\mathbf{j}), \cdots, \mathcal{I}_d(\mathbf{j})\}$ be the lexicographic expansion of $\mathcal{I}$ with respect to $\mathbf{j}$. For any $\mathbf{i} \in \mathcal{I}$ and $\mathbf{j} \in \mathbb{Z}^d$, $\mathbf{i}$ lexicographical precedes $\mathbf{j}$ if and only if $\mathbf{i} \in \mathcal{I}_\ell(\mathbf{j})$ for some $\ell \in \{1, 2, \cdots, d\}$.

Proof
The proof is given in two parts, part a) and part b). Part a) proves that $\mathbf{i} \in \mathcal{I}_\ell(\mathbf{j})$ implies $\mathbf{i} \prec \mathbf{j}$. Part b) proves that for $\mathbf{i} \in \mathcal{I}$, $\mathbf{i} \prec \mathbf{j}$ implies that $\mathbf{i} \in \mathcal{I}_\ell(\mathbf{j})$ for some ℓ.

a) Let $\mathbf{i} \in \mathcal{I}_\ell(\mathbf{j}) = \mathcal{I} \cap \mathcal{Q}_\ell(\mathbf{j})$. Since $\mathbf{i} \in \mathcal{Q}_\ell(\mathbf{j})$, it follows that $\mathbf{i} \prec \mathbf{j}$, by Definition 4.2.

b) Let $\mathbf{i} \prec \mathbf{j}, \mathbf{i} \in \mathcal{I}$. By Definition 4.2, $\mathbf{i} \in \mathcal{Q}_\ell(\mathbf{j})$ for some $\ell \in 1, 2, \cdots, d$. Because $\mathbf{i}$ is both in $\mathcal{I}$ and some $\mathcal{Q}_\ell(\mathbf{j})$, it follows that $\mathbf{i} \in \mathcal{I} \cap \mathcal{Q}_\ell(\mathbf{j}) = \mathcal{I}_\ell(\mathbf{j})$. ■

Definition 4.4 (rank)
Let be given the index set $\mathcal{I} \subset \mathbb{Z}^d$. The *rank* $\mathcal{R}(\mathbf{j})$ of any $\mathbf{j} \in \mathbb{Z}^d$ with respect to $\mathcal{I}$ is the number of points in $\mathcal{I}$ that lexicographical precedes $\mathbf{j}$.

$$\mathcal{R}(\mathbf{j}) = |\,\{\mathbf{i} \in \mathcal{I} \,|\, \mathbf{i} \prec \mathbf{j}\}\,| \tag{4.16}$$

where $|\,\mathcal{S}\,|$ denotes the cardinality of the set $\mathcal{S}$. ■

Theorem 4.3
Let index set $\mathcal{I} \subset \mathbb{Z}^d$ be given. Further let $\{\mathcal{I}_1(\mathbf{j}), \mathcal{I}_2(\mathbf{j}), \cdots, \mathcal{I}_d(\mathbf{j})\}$ be the lexicographic expansion of $\mathcal{I}$. The rank of $\mathbf{j} \in \mathbb{Z}^d$ with respect to $\mathcal{I}$ is given by

$$\mathcal{R}(\mathbf{j}) = \sum_{\ell=1}^{d} |\,\mathcal{I}_\ell(\mathbf{j})\,| \tag{4.17}$$

Proof
By Theorem 4.2 we have that $\mathbf{i} \prec \mathbf{j}$ is contained in some $\mathcal{I}_\ell(\mathbf{j}), \ell \in 1, 2, \cdots, d$, and that every $\mathcal{I}_\ell(\mathbf{j}), \ell = 1, 2, \cdots, d$ contains only points that lexicographical precedes $\mathbf{j}$. By construction every $\mathbf{i}$ is in at most one $\mathcal{I}_\ell(\mathbf{j})$. Therefore counting the number of points $\mathbf{i} \prec \mathbf{j}$ is equivalent with counting the number of points in the $\mathcal{I}_\ell(\mathbf{j})$. ■

In order to use the ranking function in (4.17) the number of points in any of the $\mathcal{I}_\ell(\mathbf{j})$ must be counted. For a given index set $\mathcal{I}$ these sets are defined in (4.15); $\mathcal{I}_\ell(\mathbf{j}) = \mathcal{I} \cap \mathcal{Q}_\ell(\mathbf{j})$. Let $\mathcal{I} = M(\mathcal{P} \cap \mathbb{Z}^n)$, by Theorem 4.1, $\mathcal{I}_\ell(\mathbf{j})$ is written as follows:

$$\mathcal{I}_\ell(\mathbf{j}) = M(\mathcal{P} \cap \mathcal{Q}'_\ell(\mathbf{j}) \cap \mathbb{Z}^n) \tag{4.18}$$

where $\mathcal{Q}'_\ell(\mathbf{j})$ is the pre-image of $\mathcal{Q}_\ell(\mathbf{j})$ under mapping $M()$. By substituting (4.18) into (4.17) we get

$$\mathcal{R}(\mathbf{j}) = \sum_{\ell=1}^{d} |\,M(\mathcal{P} \cap \mathcal{Q}'_\ell(\mathbf{j}) \cap \mathbb{Z}^n)\,| \tag{4.19}$$

This method is implemented in PANDA by the function RANK() for the index sets with unicasts only. The function RANK() can be extended to one-dimensional broadcasts as described in Section 4.5.1. At the time of writing, no solution is known to the author for multi-dimensional broadcasts.

4.5.3 Linearization of the communication

The preceding section describes a method to perform ranking in an index set. In this section, the ranking procedure is used to derive the parse nodes 2c and 4c in Figure 4.3.

Logical memory model

The unidirectional unbounded queues in Kahn Process networks are responsible for the fact that the tokens are written onto a channel and read from that channel in the same order. In order to allow a token-consuming process to process the data out of order, additional memory is required. In order to allow efficient hardware implementation all tokens that arrive at an input port are stored in a so called *random access queue* (RAQ). Like a queue, in an RAQ data is added to the back and is removed from the front. Unlike a (FIFO) queue, an RAQ allows random access for reading data. Every element in an RAQ has an associated semaphore with it. When data is written into the RAQ the value of the semaphore is set to the number of times the data is to be read. Every read access to a specific address inside the RAQ first tries to decrease the value of the semaphore by one and then reads the data from that address. When the value of the semaphore is zero, the read operation is blocking. All consecutive addresses that have a zero valued semaphore at the front of the RAQ are removed. An efficient implementation of an RAQ in shared memory that has a binary valued semaphore is the *logical storage structure* [92]. Since the elements in an RAQ are addressable, two (internal) values are associated with it, a write address and a front address. The write address is initially set to zero and is incremented by one at the end of every write into the RAQ. The front address is initially set so zero and is incremented by the number elements removed from the RAQ.

Example 4.7 (random access queue) Let a, b, c, and d be values to be written into and read from an RAQ. Now consider the following evolution of an RAQ that is reset initially. At time 1, a is written; at time 2, b is written; at time 3, c is written and the value from address 1 is read; and at time 4, d is written and the value from address 0 is read. It is assumed that the semaphore of every value written to the RAQ is set to one. The filling of the RAQ for time $T = 0, 1, \cdots, 4$ is shown in Figure 4.10

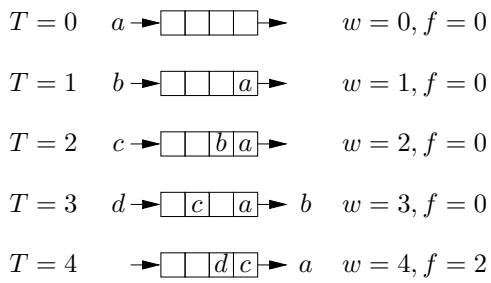


Figure 4.10: The evolution of an RAQ for reads and writes into and from it.

Clearly, when this RAQ is implemented, its size should be at least $\max(w - f) = 3$. ■

4.5.4 Address generation

Now, given that the RAQ memory model of previous subsection is used, this subsection deals with deriving the addresses to read from.

Let $\mathcal{I}_{OPD}$ and $\mathcal{I}_{IPD}$ be a matching input-output port domain pair as described in Section 4.4, that is, the data produced at points $\mathbf{j} \in \mathcal{I}_{OPD}$ is consumed at points $\mathbf{i} \in \mathcal{I}_{IPD}$. Let $\mathcal{I}_{ND}$ be the index set of the node domain of which $\mathcal{I}_{OPD}$ is a subset. Now, because $\mathcal{I}_{OPD} \subset \mathcal{I}_{ND}$, the scanning of $\mathcal{I}_{ND}$ also determines the

scanning of points in $\mathcal{I}_{OPD}$. Since consecutive tokens are written in the RAQ at consecutive addresses, the rank $\mathcal{R}(\mathbf{j})$ of $\mathbf{j} \in \mathcal{I}_{OPD}$ is the address at which the data produced at point $\mathbf{j}$ is stored in the RAQ. In this context I refer to $\mathcal{R}(\mathbf{j})$ as the *write polynomial* and denote it by $w(\mathbf{j})$.

Let $\mathbf{i} \in \mathcal{I}_{IPD}$ be a point the input port domain. The question is from what address in the RAQ the data for this point must be read. By Definition 3.9, the index set of an input port domain is a periodic lattice polyhedron. From Definition 2.24, this means that for every $\mathbf{i} \in \mathcal{I}_{IPD}$ there is a unique $\mathbf{k} \in \mathcal{P}_{IPD}$ such that $\mathbf{i} = L\mathbf{k}$, where $L = \begin{bmatrix} I_d & 0 \end{bmatrix}$ with I_d the $d \times d$ identity matrix where d is the dimension of the space that contains $\mathcal{I}_{IPD}$. This means that the first d elements of $\mathbf{k}$ for which $\mathbf{i} = L\mathbf{k}$ is precisely the vector $\mathbf{i}$.

Now it is straightforward to derive the addresses to be read from for $\mathbf{i} \in \mathcal{I}_{IPD}$. The data that must be read at point $\mathbf{i}$ is the data produced at point $\mathbf{j} = M\mathbf{k}$ where $\mathbf{k}$ is such that $\mathbf{i} = L\mathbf{k}$. Since the data produced at point $\mathbf{j}$ is stored at point $w(\mathbf{j})$, the address to be read from is $w(M(\mathbf{k}))$.

Since $\mathbf{j} = M(\mathbf{k}) = M_1(\mathbf{k}) + M_2(\mathbf{p}) + \mathbf{m}$, and $M_1 = \begin{bmatrix} M_1' & 0 \end{bmatrix}$ where M_1' is a matrix having d columns, it follows that

$$\mathbf{j} = M(\mathbf{k}) = M_1'\mathbf{i} + M_2(\mathbf{p}) + \mathbf{m} \tag{4.20}$$

noting that the first d elements of $\mathbf{k}$ is the vector $\mathbf{i}$. So, the procedure is first to derive $w(\mathbf{j})$ and then to substitute $\mathbf{j} = M(\mathbf{k}) = M_1'\mathbf{i} + M_2(\mathbf{p}) + \mathbf{m}$. The resulting polynomial is called the *read polynomial* and is denoted by $r(\mathbf{i})$. In general, $w(\mathbf{j})$ is not a single polynomial, but a set of polynomials $w_i(\mathbf{j})$, $i = 0, 1, \cdots n-1$, each defined on a corresponding parameter validity domain $w_i(\mathbf{j})$, $i = 0, 1, \cdots n-1$. Just like the $r_i(\mathbf{i})$ are derived by substitution of $\mathbf{j} = M(\mathbf{k})$ into the $w_i(\mathbf{j})$, the parameter validity domains $\mathcal{P}_i(\mathbf{i})$ are derived by substitution of $\mathbf{j} = M(\mathbf{k})$ into the $\mathcal{P}_i(\mathbf{j})$. The definition of the read polynomial $r(\mathbf{i})$ is given in (4.21).

$$r(\mathbf{i}) = \begin{cases} r_0(\mathbf{i}), & \mathbf{i} \in \mathcal{P}_0(\mathbf{i}) \\ r_1(\mathbf{i}), & \mathbf{i} \in \mathcal{P}_1(\mathbf{i}) \\ \vdots & \vdots \\ r_{n-1}(\mathbf{i}), & \mathbf{i} \in \mathcal{P}_{n-1}(\mathbf{i}) \end{cases} \tag{4.21}$$

Example 4.8 Let two node domains ND_p and ND_c be given, as in Figure 4.11. Further let $\mathcal{I}_{OPD}$ and $\mathcal{I}_{IPD}$ be an output port domain and input port domain of ND_p and ND_c, respectively. These port domains corresponds with the shaded regions in the figure. Now assume that the producing node domain ND_p is scanned with j_2 in the outer loop and with j_1 in the inner loop. By using the ranking procedure from a previous subsection the write polynomial $w(j_1, j_2)$ is derived. The ranking is showed in the figure by the numbers placed at the points in $\mathcal{I}_{OPD}$.

$$w(j_1, j_2) = -\frac{1}{2}j_2^2 + (N - \frac{1}{2})j_2 + j_1 - 1$$

The mapping function $M()$ in the figure is given by

$$\mathbf{j} = \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} = M(\mathbf{i}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_i \\ i_2 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The corresponding read polynomial $r(i_i, i_2)$ is found after substitution of $j_1 = i_1$ and $j_2 = i_2 - 1$ into $w(j_1, j_2)$.

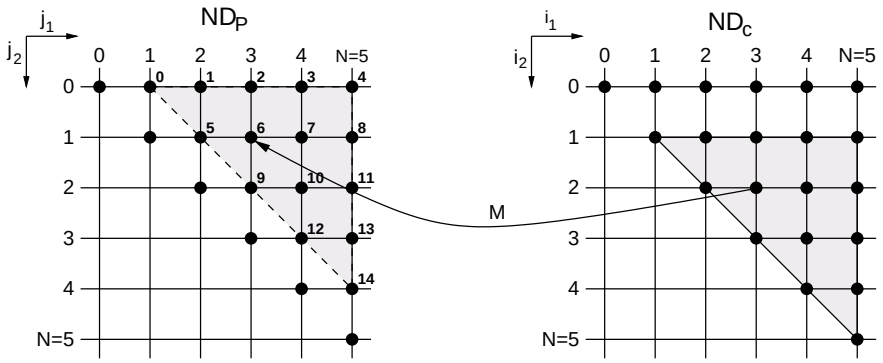


Figure 4.11: Ranking of output port domain domain used to derive write polynomial for the input port domain.

$$r(i_1, i_2) = -\frac{1}{2}i_2^2 + (N + \frac{1}{2})i_2 + i_1 - (N + 1)$$

Take for example the point $(i_1, i_2) = (3, 2)$ in $\mathcal{I}_{IPD}$. The ranking polynomial evaluates for this point to $r(3, 2) = -\frac{1}{2} \cdot 2^2 + (5 + \frac{1}{2}) \cdot 2 + 3 - (5 + 1) = 6$. This is precisely the rank of the point onto which $\mathbf{i} = (3, 2)$ is mapped. ■

4.5.5 Generating the OPD and IPD blocks

Each node domain of the PRDG is converted into a process of the Kahn process network. Every input port domain IPD_j of a node domain specifies when to read from channel ch_j, the channel that is written onto in OPD_j. Since by convention data is written onto the channels in order, the *body* statement in line 19 in Program 4.6 (page 68) is just the write operation in Program 4.7.

Program 4.7: PROTOTYPE OPD BLOCK

```
1   write(wp_j, out);
```

where *out* is a return value of the function that is bound to the output port domain. That is all to be done to generate the *body* statement in line 19 in Program 4.6.

The generation of this line is implemented by Algorithm OUTPUTPORT2ASSIGNMENTSTATEMENT().

```
OUTPUTPORT2ASSIGNMENTSTATEMENT(Q)
1  V ← new VariableStatement(variable[Q], domain[Q])
2  G ← new GenericIfStatement("doWrite == true")
3  INSERT-ELEMENT(A, G)
4  O ← new OpdStatement(name[Q], argument[Q], name[edge[Q]], V)
5  INSERT-ELEMENT(A, O)
6  return A
```

In order to generate the code for the *body* statement in Figure 4.4, Program 4.8 shows a template of the code that is to be generated for an input port domain IPD_j that corresponds to a variable array x. Line 1 in the program is only shown for the context; its actual derivation is described in Section 4.4.

Program 4.8: PROTOTYPE IPD BLOCK

```
1   if i ∈ I_IPD_j,
2       r_j = r_j(i);
3       while w_j ≤ r_j,
4           x(w_j) = read(rp_j);
5           w_j = w_j + 1;
6       end
7       in = x(r_j);
8   end
```

In line 2, r_j specifies from which address in array x a value must be read. r_j is set to be equal to the value of the *read polynomial* $r_j(\mathbf{i})$, evaluated for the current iteration $\mathbf{i}$.

Similarly w_j is the value of the *write polynomial* but by construction its value is its previous value plus one. w_j indicates the write address for array x, i.e. the position in the array x where the next token read from the channel is to be written. The write address w_j is initialized with $w_j = 0$. In line 3 there are two possibilities: $w_j \leq r_j$, or $w_j > r_j$.

In the former case, the value to be read from the array has not yet been written into the array. Lines 4 and 5 are executed and new values are read from channel ch_j and are stored at incremental addresses in array x until the value of the write address equals the value of the read polynomial plus one. Then in line 7 this value is assigned to in which is an argument of the function that is executed for every point in the node domain (not shown in the program).

In the latter case, the read polynomial is smaller than the current value of the write address. This means that the value has previously been written to the array and control jumps to line 7 in the program.

The blocking semantics that was described in the previous section is implemented by the $read()$ operation from channel ch_j in line 4. The parse tree representation of Program 4.8 is given in Figure4.12.

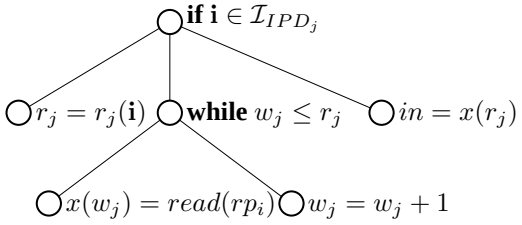


Figure 4.12: Structure of the sub-tree for a single input port domain.

The variable x is the variable of the primitive input port of the input port domain, cf. Definition 3.4 and Definition 3.9. The argument of the function to which the value read from memory must be passed is the argument of the primitive port of the input port domain.

The node labeled $r_j = r_j(\mathbf{i})$ in Figure 4.12 is a sub-tree itself, reflecting the multiple read polynomials with their parameter validity domains, see (4.21). The structure of the sub-tree to be derived is quite similar to the structure in Figure 4.8, and is given in Figure 4.13.

In Figure 4.13, $\mathcal{P}_{j,k}$ is the k^{th} parameter validity domain of r_j and $r_{j,k}()$ is its corresponding pseudo polynomial. Since the parameter validity domains the result of a counting problem in the context of $\mathcal{I}_{IPD_i}$, there will be an $\mathcal{P}_{j,k}$ such that $\mathbf{i} \in \mathcal{P}_{j,k}$. The Boolean flag b ensures an assignment to r_j is made only once to avoid redundant computations.

Example 4.9 Let IPD_1 be the input port domain of ND_c in Example 4.8. The generated code is given in Program 4.9.

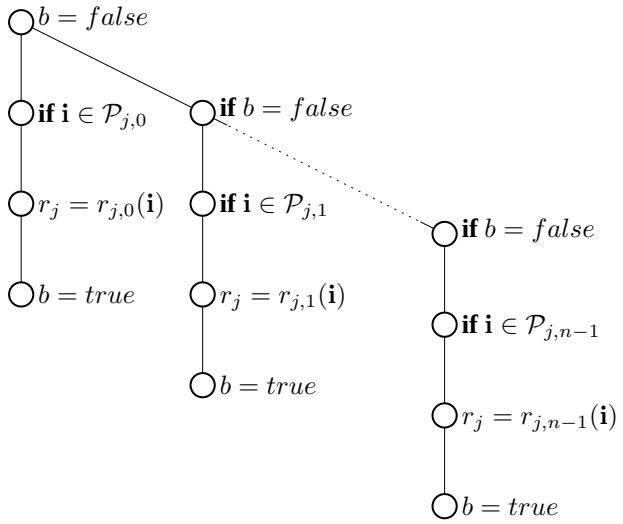


Figure 4.13: Structure of the sub tree for deriving the read polynomial.

Program 4.9: EXAMPLE IPD BLOCK

```
1  if i2 ≥ 1,  ▷ check if (i1, i2) ∈ I_ND
2      r1 = −½i2² + (N + ½)i2 + i1 − (N + 1);
3      while w1 ≤ r1,
4          x(w1) = read(rp1);  ▷ use array x to store data read
5          w1 = w1 + 1;
6      end
7      in = x(r1);
8  end
```

■

The method to convert an input port domain into the parse tree node 2c that is presented in this chapter is implemented by the Algorithm INPUTPORT2ASSIGNMENTSTATEMENT.

```
INPUTPORT2ASSIGNMENTSTATEMENT(P)
 1  R ← new RootStatement
 2  Iopd ← domain[fromPort[edge[P]]]
 3  p ← RANK(I_opd);  ▷ Actually only first element of rank vector is used.
 4  var ← variable[P]
 5  arg ← argument[P]
 6  name ← "c"count
 7  count ← count + 1
 8  lcm ← MAKEINTEGRAL(p)
 9  R ← EhrhartPolynomial2SubTree(p, name)
10  S1 ← new SimpleAssignStatement
11  lhs[S1] ← "readAddress"var
12  rhs[S1] ← "("TOINTEGRALSTRING(p, name)")/"lcm
13  ADD-CHILD(R, S1)
14  W ← new WhileStatement(var".getWriteAddress() <= readAddress"var)
15  ADD-CHILD(R, W)
16  S2 ← new SimpleAssignStatement(arg, var".getElement(readAddress"var")")
17  ADD-CHILD(R, S2)
18  S3 ← new SimpleStatement("read(RP"name[edge[P]].substring(3)", value)")
19  ADD-CHILD(R, S3)
20  S4 ← new SimpleStatement(var".put(value)")
```

21 ADD-CHILD(W, S_4)
22 **return** R

4.6 Network generation

The generation of the Kahn process network basically consists of two steps: the generation of the processes and the generation of the network that allows these processes to communicate. The previous sections dealt with the generation of the processes. This section deals with the generation of network.

The channels in the Kahn process network are derived from the edge domains in the PRDG. Let ED_i be an edge domain. After the POINT-TO-POINT() procedure has been applied to the PRDG, the index i also identifies the output port domain OPD_i and input port domain IPD_i for which a communication channel must be set up.

Let ND_k be a node domain and let P_k be the process derived from ND_k. For every input port domain IPD_i in ND_k, a single read operation $read(rp_i)$ in P_k is generated during the IPD block generation. Similarly, for every output port domain OPD_j in ND_k, a single write operation $write(wp_j)$ is generated during the OPD block generation. Since the index i in rp_i originates from the edge domain ED_i and the index j in wp_j originates from the edge domain ED_j, there is only one rp_i and only one rw_j in the complete Kahn process network for a particular value of i and j. So the following procedure is followed to construct the network:

- Every process declares the list of channels from which it reads data and to which is writes data, thereby using the local names rp_i and wp_j to avoid name clashes. The order in which the ports appear in the list is not important at this point.
- For every edge domain ED_i a channel ch_i is declared and is bound to the local channel names rp_i and wp_i of the processes. This binding is done by taking the channel lists of the process declaration and replacing all rp_i by ch_i and replacing all wp_j by ch_j.

An example of the network generation is the network at the bottom of Figure 4.2. A more complete example will be given in next chapter.

Chapter 5

Software organization

This chapter puts together the methods that are described in previous chapters. This is done by focusing on the design flow that is implemented by the Compaan tool set.

One of the objectives of this dissertation is to work out the idea of *concept matches software*. This means that the theoretical concepts presented in chapters 2, 3, and 4 are organized and formulated in such way that it is straightforward to implement them in software. The reverse is also true; to understand each functionality of the software one can read the corresponding section in the previous chapters. The "concept matches software" approach reveals itself in the strong relationship between the algebraic constructs and their structural counterparts in software.

The software versions of the methods described in this dissertation are written in the object oriented language Java. Modern software engineering techniques, including *design patterns* and *UML*, have been used to have this software well organized.

This chapter is organized as follows. Section 5.1 deals with the design flow that is implemented by the Compaan tool set, and describes the top-level organization of these tools. Sections 5.2 and 5.3 deal with the internal operation of the tools and relates them to the methods and data structures that are described in previous sections. The operation of the tools are illustrated by an application that transposes a triangular matrix.

5.1 Design Flow

For the methods presented in this dissertation, corresponding software versions have been implemented as part of Compaan tool set to allow the user to convert a nested loop program written in the Matlab language into a Kahn process network. An overview of the flow through these tools is given in Figure 5.1.

The first tool in the tool chain is MATPARSER. MATPARSER takes as input a nested loop program from the file `prog.m` and converts it into the single assignment program `prog.sac`. For details about the exact class of nested loop programs MATPARSER can handle and what these single assignment programs are, see Chapter 3. MATPARSER is not part of the research described in this dissertation. Detailed information about this tool can be found in [13].

The second tool in the tool chain is DGPARSER. DGPARSER takes as input the single assignment program from the file `prog.sac` and converts it into the file `prog.xml` that is a textual description of a polyhedral reduced dependence graph in XML format. DGPARSER can accept any single assignment program that can

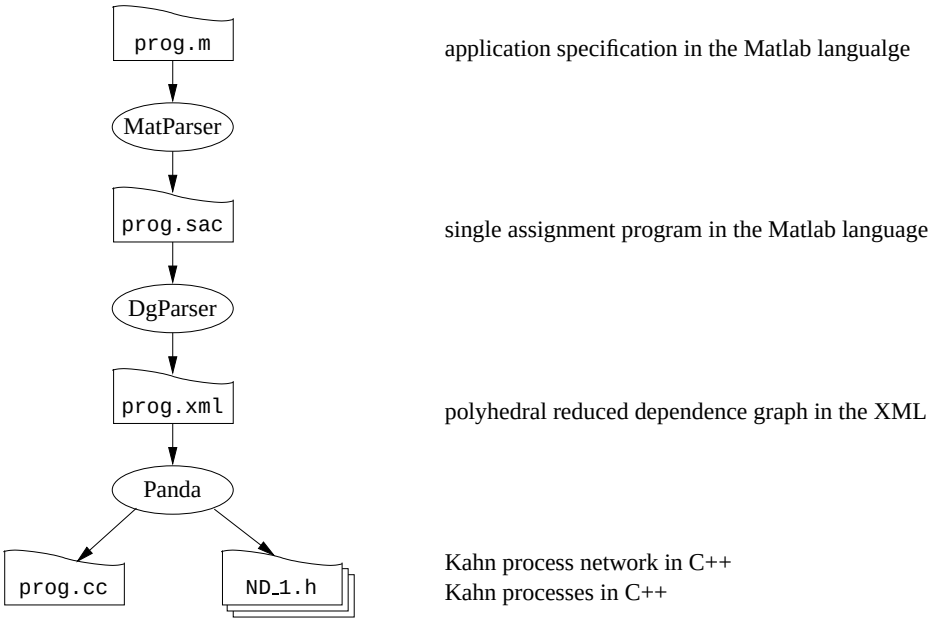
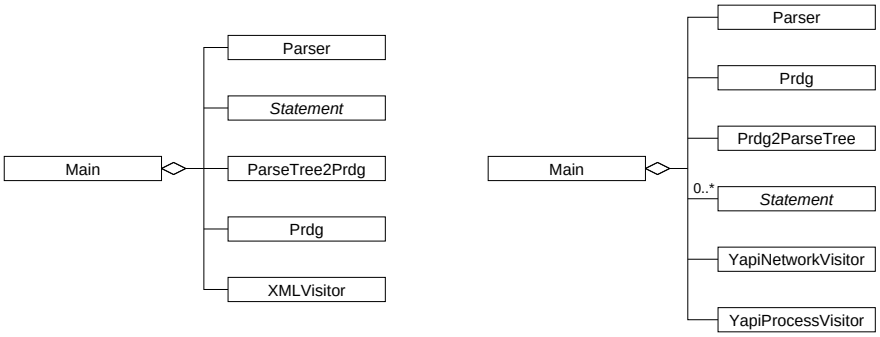


Figure 5.1: Design flow.

be generated by MATPARSER.

The third tool in the tool chain is PANDA. PANDA takes as input the file `prog.xml` and produces a number of files that define a Kahn process network. The single file `prog.cc` is the description of the network. The processes are defined in the individual files `ND_1.h`, `ND_2.h`, etc., one for each node domain of the PRDG. The processes in the files are represented by C++ classes and use the YAPI library [58].

Each of the three tools in Figure 5.1 operates in three phases, viz., *parse*, *convert*, *generate*. The first phase is to parse the textual input file into an internal data structure. The second phase is to convert the data structure into a new data structure, possibly in the same representation. The third phase is to generate the output file by visiting the second data structure. The three phases are illustrated in the top-level architecture of DGPARSER and PANDA in the UML diagrams in Figure 5.2(a) and (b), respectively. UML stands for *unified modeling language* [93]. In Figure 5.2(a) the phases parse, convert, and generate are performed by



(a) DgParser (b) Panda

Figure 5.2: Structure of DGPARSER and PANDA tools.

the *singleton* classes `Parser`, `ParseTree2Prdg`, and `XMLVisitor`, respectively. The `Parser` class produces a parse tree rooted by the root statement `Statement` of this tree. The `ParseTree2Prdg` class converts the parse tree into a PRDG represented by the `Prdg` class. Finally, the `XMLVisitor` class visits

a PRDG and produces an description of the PRDG in XML format. The logical flow inside PANDA, see Figure 5.2(b), is slightly less obvious. The single instance of the `Parser` singleton class parses an input file in XML into an instance of the `Prdg` class. Then the instance of `Prdg2ParseTree` converts the PRDG into a set of statements, each being an instance of the `Statement` class and each being the root statement of a parse tree. For every node domain in the PRDG, a parse tree is derived and every such parse tree describes a Kahn process. Moreover, `Prdg2ParseTree` also has side effects on the PRDG. The two visitor classes `YapiNetWorkVisitor` and `YapiProcessVisitor` visit this new PRDG and the set of parse trees, respectively. The former generates a description of the Kahn process network, while the latter generates a class definition for each Kahn process. The classes in the diagram that have `Visitor` in their name are structured according to the *visitor design pattern*, see [94].

The real work in DGPARSER and PANDA is done in the classes `ParseTree2Prdg` and `Prdg2ParseTree` classes, respectively. These classes are dealt with in Section 5.2 and Section 5.3, respectively.

5.2 DGPARSER

Each data structure that is described in this section is defined by a set of classes. Such a set of classes is grouped into a *package*. I use UML class diagrams to explain the data structure defined in each package.

5.2.1 Parse tree

Figure 5.3 shows the class diagram of the parse tree data structure that is used in DGPARSER. The figure shows the classes representing the statements the parse tree is composed of. Instances of the classes `IpdStatement`, `OpdStatement`, and `NodeStatement` are the **ipd** , **opd** , and **node** statements in Section 3.3.1, respectively. Instances of the classes `ParameterStatement`, `ForStatement`, `IfStatement`, and `ElseStatement` are the **parameter** , **for** , **if** , and **else** statements in Section 3.3.2, respectively. An example of an original Matlab program and the single assignment program that MATPARSER generates from it are shown Program 3.1 and Program 3.2, respectively. The parse tree that is constructed inside DGPARSER is shown in Figure 3.2. Figure 3.3 shows an example that contains an index transformation statement. Part (a) shows the SAP and part (b) shows the parse tree that is constructed from it. Figure 3.4 shows an intermediate from of the parse tree in which every node has associated constraints with it. This is starting point of the conversion to the polyhedral reduced dependence graph (PRDG) model.

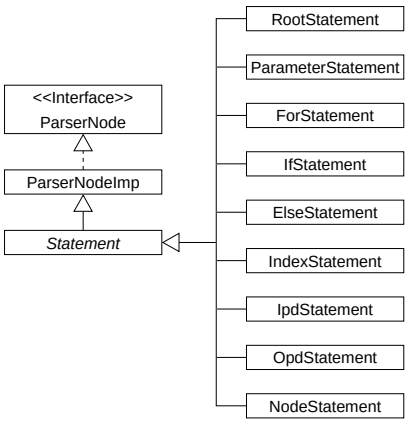


Figure 5.3: Overview of the structure of DGPARSER.

Interface inheritance is a mechanism where an interface is realized by multiple classes. In this way these classes have a common interface and by using this interface an objects can be used in place of another [94]. In DGPARSER interface inheritance is used to implement a parse tree. All leaf statement classes are specializations of the abstract class `Statement`. This means that all behavior that class `Statement` has is inherited by these leaf statements. The class `Statement` itself is a specialization of the the class `ParserNodeImp` that realizes the interface `ParserNode`. So, interface inheritance is achieved because the leaf statements inherited the realization of the interface via `ParserNodeImp` and `Statement`, see Figure 5.3. Moreover, tree-specific behavior is implemented in `ParserNodeImp` and statement-specific behavior is implemented in `Statement`. This way of structuring allows the use of the visitor design pattern.

5.2.2 Polyhedral reduced dependence graphs

The polyhedral reduced dependence graph (PRDG) is formally defined in Section 3. The PRDG data structure defines an application in terms of three characteristics, viz., topology, geometry, and behavior. Figure 5.4 gives a class diagram illustrating the geometrical and behavior part of the PRDG data structure. The figure illustrates the four classes `Node`, `Edge`, `InputPort`, and `OutputPort` representing the node

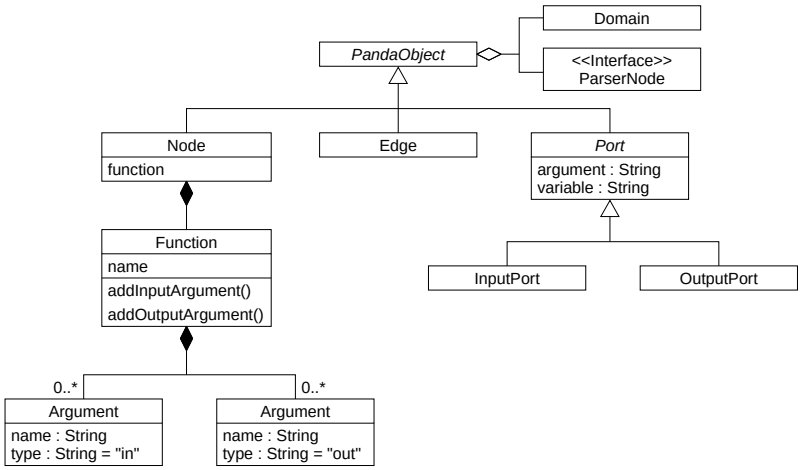


Figure 5.4: Geometrical and behavioral part of the PRDG data structure.

domain, edge domain, input port domain, and output port domain data structures in definitions 3.11, 3.12, 3.9, and 3.10, respectively. They are all derived from the basis class `PandaObject` that contains an index set `Domain` and a `ParserNode` and, therefore, contain an index set and a parse tree themselves. The use of the `ParserNode` class in the PRDG is explained in Chapter 5.3. Since `Node`, `Edge`, `InputPort`, and `OutputPort` have an index set associated with it, they are geometrically conform definitions 3.11, 3.12, 3.9, and 3.10. As shown in Table 3.1 (page 44), only the node domain (N) and the port domains (P,Q) have behavior, i.e., associations with them required for proper functional behavior. This behavior is shown in Figure 5.4 as well.

Figure 5.5 gives a class diagram illustrating the topological part of the PRDG data structure. The figure shows how the objects of the PRDG link together and directly implements the extended graph definition. The diagram clearly shows the container-contents relations. The PRDG contains one or more node domains and zero or more edge domains. A node domain contains zero or more input port domains and zero or more output port domains. An edge domain is associated with one input port domain and one output port domain.

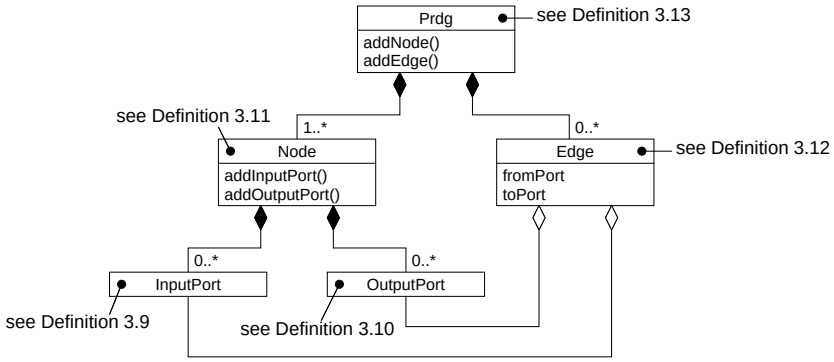


Figure 5.5: Topological part of the PRDG data structure.

5.2.3 Internal operation of DGPARSER

The actual conversion is done by the `ParseTree2Prdg` singleton class. This class converts a parse tree into a PRDG. The conversion is illustrated in Algorithm PARSETREE2PRDG. This algorithm is given in pseudo-code. The pseudo-code uses the graphical syntax presented [75]. Although the real software implementation is based on classes that can contain state, for the sake of clarity, the pseudo-code used in this section is pure functional.

PARSETREE2PRDG(T)

1 $G \leftarrow$ new `Prdg` ▷ recall that $G = (\mathcal{N}, \mathcal{E})$, see Definition 3.13
2 $H \leftarrow$ new `Hashtable` ▷ will contain all edge domains, see lines 19 and 26
3 $I_N \leftarrow$ NIL ▷ will contain all input port domains of a node domain, see lines 8, 12, and 14
4 **for all** statements $s \in T$ in DFS order
5 **do switch**
6 **case** $type[s] =$ **ipd** :
7 $P \leftarrow$ CREATE-INPUTPORT(s) ▷ domain[P] is edge domain see lines 29 and 30
8 LIST-INSERT(I_N, P)
9 **return**
10 **case** $type[s] =$ **node** :
11 $N \leftarrow$ CREATE-NODE(s)
12 **for all** input port domains $P \in I_N$
13 **do** ADD-INPUTPORT(N, P)
14 $I_N \leftarrow$ NIL
15 ADD-NODE(G, N)
16 **return**
17 **case** $type[s] =$ **opd** :
18 $Q \leftarrow$ CREATE-OUTPUTPORT(s)
19 HASH-TABLE-PUT($H, (variable[Q], Q)$)
20 ADD-OUTPUTPORT(N, Q) ▷ also sets index set of N to index set of Q
21 **return**
22 **for all** node domains $N \in \mathcal{N}[G]$
23 **do for all** input port domains $P \in N$
24 **do** $E \leftarrow$ new `Edge`
25 $toPort[E] \leftarrow P$
26 $Q \leftarrow$ HASH-TABLE-GET($H, variable[P]$)
27 **if** $Q \neq$ NIL
28 **then** $fromPort[E] \leftarrow Q$
29 $domain[E] = domain[P]$ ▷ domain[P] is still the edge domain
30 $domain[P] =$ identity mapping ▷ domain[P] is an integer polyhedral lattice
31 ADD-EDGE(G, E)

32 **return** G

The argument T passed to the PARSETREE2PRDG algorithm is a parse tree, corresponding to Figure 5.3, that is to be converted into a PRDG, corresponding to figures 5.4 and 5.5.

The algorithm consists of two parts. Lines 4–21 are the first part of the algorithm and convert the `IpdStatement`, `OpdStatement`, and `NodeStatement` statements into input port domains (`InputPorts`), output port domains (`OutputPorts`), and node domains (`Nodes`), respectively. Lines 22–31 are the second part of the algorithm and create the edge domains (`Edges`). Lines 1–3 contain three declarations. G is PRDG that is to be returned eventually (see line 32). H is a hash table data structure that is created in the first part of the algorithm and is used in the second part of the algorithm. I_N is a list that temporary stores input port domains.

In the first part, all statements of parse tree are visited in depth-first-search (DFS) order. As a consequence, all leaf nodes are visited in a left-to-right order. The leaf node is either an **ipd**, a **node**, or an **opd** statement. For each of these three cases, the corresponding element from the PRDG is created. This is illustrated by the CREATE-INPUTPORT, CREATE-NODE, and CREATE-OUTPUTPORT functions shown below. Since the leaf nodes are visited in a left-to-right order, each input port domain that is created belongs to the first following node domain that is created. Therefore the input port domains P that are created are stored in the temporary list I_N. When a node domain N is created, the elements of I_N are then added to the list of input port domains of N. Each output port domain Q that is created is added to the most recently created node domain N but is also put in the hash table H with key $variable[Q]$. The key in the hash table is used in the second part of the algorithm to find for every input port domain P the output port domain Q with the same variable name. When such Q exists, the edge domain $E = (Q, P)$ is created by setting E's $fromPort$ to Q and E's $toPort$ to P.

The CREATE-INPUTPORT function is shown below. The **ipd** statement s has two child nodes, they are indicated by $left[s]$ and $right[s]$. Take for example the first **ipd** statement in Program 3.2, $s : [in_0] = \mathbf{ipd}(a_1(i-1, j+1))$, here $left[s] = in_0$ and $right[s] = a_1(i-1, j+1)$. In line 1 the input port domain P is created. In the lines that follow, the statement s is parsed. The index set of P is determined with the GET-DOMAIN algorithm in Section 3.4.1. Note that this function is applied to the right-hand side of s. This takes the indexing function of the variable into account and constructs the index set of the edge domain, see line 29 in PARSETREE2PRDG.

CREATE-INPUTPORT(s)
1 $P \leftarrow$ new `InputPort`
2 $argument[P] \leftarrow name[right[s]]$ ▷ name of argument of function in s
3 $variable[P] \leftarrow name[left[s]]$ ▷ name of variable
4 $domain[P] \leftarrow$ GET-DOMAIN($right[s]$) ▷ see GET-DOMAIN in Section 3.4.1
5 **return** P

The CREATE-OUTPUTPORT function is shown below. It is quite similar to the CREATE-INPUTPORT function. Take for example the **opd** statement in Program 3.2, $s : [a_1(i, j)] = \mathbf{opd}(out_0)$, here $left[s] = a_1(i, j)$ and $right[s] = out_0$.

CREATE-OUTPUTPORT(s)
1 $Q \leftarrow$ new `OutputPort` ▷ create a new ouput port domain
2 $argument[Q] \leftarrow name[right[s]]$ ▷ name of argument of function in s
3 $variable[Q] \leftarrow name[left[s]]$ ▷ name of variable
4 $domain[Q] \leftarrow$ NIL ▷ domain is set in line 20 of PARSETREE2PRDG
5 **return** Q

The CREATE-NODE function is shown below. To enable the reading from and writing to a text file, the class `Node` is extended to the classes `SourceNode` and `SinkNode` respectively. In lines 1–10, based on the name of the function invoked in statement s the proper node domain is created. In lines 11–13 the `Node`

data structure is built as shown in Figure 5.5.

```
CREATE-NODE(s)
 1  switch
 2    case functionName[s] begins with "_ReadMatrix_" :   ▷ test if s is a source
 3         N ← new SourceNode
 4         return
 5    case functionName[s] begins with "_WriteMatrix_" :  ▷ test if s is a sink
 6         N ← new SinkNode
 7         return
 8    case default :                                      ▷ if s is not a source or sink,
 9         N ← new Node                                     then it is an internal node
10         return
11  F ← CREATE-FUNCTION(s)          ▷ create a new function for this node
12  function[N] ← F                 ▷ let the function of N be the function just created
13  domain[N] ← GET-DOMAIN(s)       ▷ see GET-DOMAIN in Section 3.4.1
14  return N
```

The CREATE-FUNCTION function is shown below. First a new `Function` object is created. Then its name is set to the name of statement s. The left-hand side child of s contains the list of return arguments of the function. The right-hand side child of s contains the list of input arguments of the function.

```
CREATE-FUNCTION(s)
1  F ← new Function
2  name[F] ← functionName[s]
3  for all arguments v ∈ left[s]
4  do ADD-OUTPUT-ARGUMENT(F, name[v])   ▷ creates new argument with type "out"
5  for all arguments v ∈ right[s]
6  do ADD-INPUT-ARGUMENT(F, name[v])    ▷ creates new argument with type "in"
7  return F
```

5.2.4 Operation example of DGPARSER

This section describes a simple case that illustrates the operation of DGPARSER. In the example an upper triangular $N \times N$ matrix is read from a file. For the sake of completeness the original Matlab file is shown in Program 5.1.

Program 5.1: TRANSPOSE (MATLAB)

```
 1  %parameter N 10 100;

 2  for j = 1 : 1 : N,
 3    for i = 1 : 1 : N,
 4      [ u(j,i) ] = _ReadMatrix_U();
 5    end
 6  end

 7  for i = 1 : 1 : N,
 8    for j = 1 : 1 : i,
 9      [] = _WriteMatrix_L( u(j,i) );
10    end
11  end
```

Line 1 declares N as an integral parameter with a value in the range from 10 to 100. Lines 2 – 6 read the square $N \times N$ matrix from a file. The loops iterate over all N^2 entries row by row. The prefix `_ReadMatrix_` is used to indicate that matrix U is to be read from a file. Lines 7 – 11 write all upper triangular elements from the matrix u in a column by column order. The prefix `_WriteMatrix_` is used to indicate that the values of the matrix L are to be written to a file.

The single-assignment program generated from Program 5.1 is shown in Program 5.2. It is not very different from Program 5.2, because the original Matlab program already is in single-assignment form. The difference is the additional structure added.

Program 5.2: TRANSPOSE (SAP)

```
 1  %parameter N 10 100;

 2  for j = 1 : 1 : N,
 3     for i = 1 : 1 : N,
 4        [ out_0 ] = _ReadMatrix_U(  );
 5        [ u_1( j, i) ]  = opd( out_0 );
 6     end
 7  end

 8  for i = 1 : 1 : N,
 9     for j = 1 : 1 : i,
10        [ in_0 ] = ipd( u_1( j, i ) );
11        [  ] = _WriteMatrix_L( in_0 );
12     end
13  end
```

All arguments of the functions are of the form `in_x` where x is an integer, and all return values are of the form `out_y` where y is an integer. The **opd** statement in line 5 is used to assign the return values of the functions to the two dimensional array u. The **ipd** statement in line 10 is used to assign the elements from the multi dimensional array u to the input argument of the function `_WriteMatrix_L( in_0 )`.

In DGPARSER, the single assignment program TRANSPOSE (SAP) is parsed and represented as a parse tree. Figure 5.6 shows the UML object diagram of this parse tree. The object diagram shows a number of instances of the statement classes from Figure 5.3 and how they relate with each other.

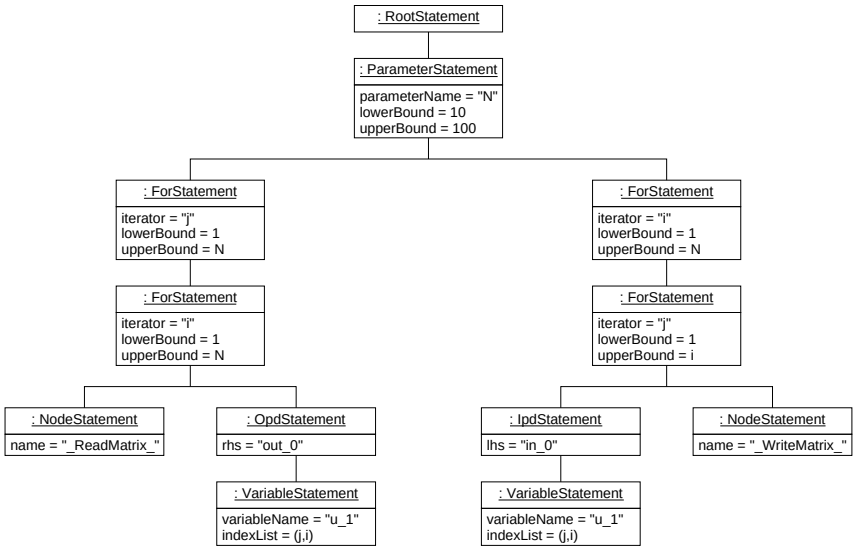


Figure 5.6: Object diagram of the parse tree.

Procedure PARSETREE2PRDG is now used to convert the parse tree of Figure 5.6 into a PRDG. Figure 5.7 shows the resulting PRDG.

Finally, the PRDG data structure is written to a file in XML format. In XML, *elements* are used to represent the classes from the PRDG. An element is defined by the `<elementName attributes> elements </elementName>` tags. The `attributes` are the attributes of the element and the `elements` are

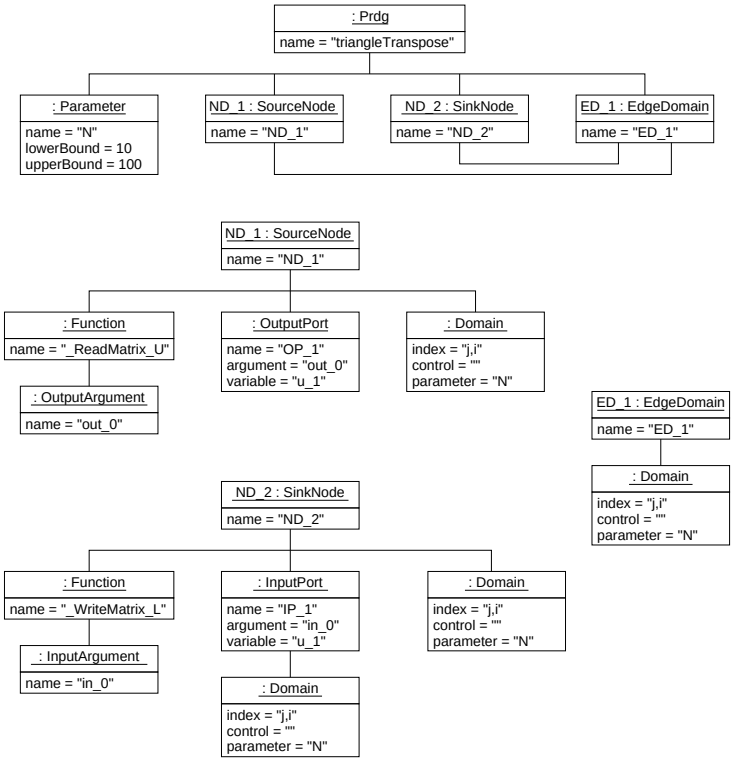


Figure 5.7: Object diagram of the PRDG.

the elements associated with the element. The complete desciption of the PRDG in XML is shown in Program 5.3.

Program 5.3: TRANSPOSE (XML)

```
 1  <?xml version="1.0" standalone="no"?>
 2  <!DOCTYPE model PUBLIC "-//UC Berkeley//DTD Panda 1//EN"
 3      "http://ptolemy.eecs.berkeley.edu/~kienhuis/dtd/panda.dtd">

 4  <model name="triangleTranspose">
 5      <parameter name="N" ub="100" lb="10" />

 6      <node name="ND_1" type="SourceNode">
 7        <function name="_ReadMatrix_U">
 8          <argument name="out_0" type="out" />
 9        </function>
10        <domain index="j, i" control="" parameter="N">
11          <constraint matrix="[1,  1,  0, 0, -1;
12                               1, -1,  0, 1,  0;
13                               1,  0,  1, 0, -1;
14                               1,  0, -1, 1,  0]" />
15          <context matrix="[1, -1, 100;
16                            1,  1, -10]" />
17          <mapping matrix="[1, 0, 0, 0;
18                            0, 1, 0, 0]" />
19        </domain>
20        <port name="OP_1" argument="out_0" variable="u_1">
21        </port>
22      </node>

23      <node name="ND_2" type="SinkNode">
24        <function name="_WriteMatrix_L">
25          <argument name="in_0" type="in" />
```

```
26        </function>
27        <domain index="i, j" control="" parameter="N">
28          <constraint matrix="[1,  1,  0, 0, -1;
29                               1, -1,  0, 1,  0;
30                               1,  0,  1, 0, -1;
31                               1,  1, -1, 0,  0]" />
32          <context matrix="[1, -1, 100;
33                            1,  1, -10]" />
34          <mapping matrix="[1, 0, 0, 0;
35                            0, 1, 0, 0]" />
36        </domain>
37        <port name="IP_1" argument="in_0" variable="u_1">
38          <domain index="i, j" control="" parameter="N">
39            <constraint matrix="[1,  1,  0, 0, -1;
40                                 1, -1,  0, 1,  0;
41                                 1,  0,  1, 0, -1;
42                                 1,  1, -1, 0,  0]" />
43            <context matrix="[1, -1, 100;
44                              1,  1, -10]" />
45            <mapping matrix="[1, 0, 0, 0;
46                              0, 1, 0, 0]" />
47          </domain>
48        </port>
49      </node>
50      <edge name="ED_1">
51        <to name="ND_2" port="IP_1" />
52        <from name="ND_1" port="OP_1" />
53        <domain index="i, j" control="" parameter="N">
54          <constraint matrix="[1,  1,  0, 0, -1;
55                               1, -1,  0, 1,  0;
56                               1,  0,  1, 0, -1;
57                               1,  1, -1, 0,  0]" />
58          <context matrix="[1, -1, 100;
59                            1,  1, -10]" />
60          <mapping matrix="[0, 1, 0, 0;
61                            1, 0, 0, 0]" />
62        </domain>
63        <doc>u_1(j,i)</doc>
64      </edge>
65  </model>
```

Lines 1–3 are the header of the document. Line 1 states the version of the XML document and that it is not a stand alone document which means that the grammar of the document is not included in the document itself. Lines 2–3 specify where this grammar is. The grammar is specified in the `panda.dtd` file. The extension DTD stands for Document Type Definition and describes what elements have what constitutes a valid XML file.

Lines 4–53 describe the actual PRDG. The `<model name="triangleTranspose">` (line 4) and `</model>` (line 65) mark the begin and end of the PRDG model definition with the name `triangle-Transpose`. The `Parameter`, `SourceNode`, `SinkNode`, and `Edge` objects shown in the top part of Figure 5.7 are defined in XML by the elements `<parameter>` (line 5), `<node>` (lines 6 and 23), and `<edge>` (line 50) elements. Now take node domain `ND_2` shown at the bottom of Figure 5.7 as an example. The element `<node name="ND_2" type="SinkNode">` (line 23) starts the definition of the node domain with name `ND_2` which is of type `SinkNode`. `ND_2` has a single input port domain `IP_1`. The definition of this input port domain starts at the `<port>` element (line 37) and its index set definition start at the `<domain>` element (line 38). The index set is described by three vectors and three matrices. The three vectors are the attributes of the `<domain>` element (line 38). The three matrices are described by the elements `<constraint>` (line 39), `<context>` (line 43), and, `<mapping>` (line 45) which are all elements of the `<domain>` element.

The matrices `constraint` and `context` each represent a polyhedron in the conic representation of parameterized polyhedra. The first column of each of these matrices encode the type of constraint. A zero or a one in the first column means that the constraint is an equality or an inequality, respectively. This first column allows the matrices $\tilde{A}$ and $\tilde{C}$ in (2.25) to be combined into a single matrix.

5.3 PANDA

In PANDA, the XML description that is produced by DGPARSER is first parsed to reconstruct the internal form of a PRDG. The PRDG data structure has been presented in Chapter 5.2.2. Inside PANDA, the PRDG is converted to a set of parse trees, one for each node domain in the PRDG. The parse tree used within PANDA is quite similar to the parse tree used within DGPARSER. Chapter 5.3.1 deals with the internal operation of PANDA immediately. Chapter 5.3.2 continues the operation example from Chapter 5.2.4.

5.3.1 Internal operation of PANDA

The conversion of the PRDG into the parse trees is done in two parts.

The first part is done by the singleton class `Prdg-Prepare` which transforms the PRDG into another PRDG. Algorithm PRDG-PREPARE shows the functionality of this class.

PRDG-PREPARE(G)
1 POINT-TO-POINT(G)
2 RECONSTRUCT-OUTPUT-PORT-DOMAINS(G)
3 **return** G

The POINT-TO-POINT method introduces new output port domains in the PRDG such that every output port domain connects to at most one input port domain. The RECONSTRUCT-OUTPUT-PORT-DOMAINS method reconstructs the index set of each output port domain such that each point in the index set of the output port domain has a consuming point in the index set of the input port domain. POINT-TO-POINT and RECONSTRUCT-OUTPUT-PORT-DOMAINS, together performing *domain matching*, are described in more detail in Chapter 4.4.

The second part is the conversion of the prepared PRDG to the set of parse trees. This is done by the singleton class `Prdg2ParseTree`. The structure of these parse trees is different from the structure of the parse tree that is used in DGPARSER, and is outlined in Figure 4.3. The function of the `Prdg2ParseTree` class is shown in Algorithm PRDG2PARSETREE. In the algorithm, every node of the PRDG is converted into a parse tree. Line 1 iterates over all node domains N of the PRDG. Figure 5.4 shows that the abstract class `PandaObject` has a class `ParserNode` associated with it. This means that every class that implements this interface, like all statements in Figure 5.3, can be associated with the Panda Objects. Here I use this property to associate a parse tree with every node domain (`Node`) of the PRDG. In the algorithm s (and s') is used to represent anything that implements the `ParserNode` interface. The statement s (and s') is a *stitch node*. A stitch node represents the node node in the parse tree currently operating on. Initially the stitch node s is the root of the parse tree. Then in line 4, a set of **parameter** statements are stitched under s. In the function STITCH every newly stitched statement becomes the new stitch node. As a result, the last added **parameter** statement becomes the stitch node at the end of line 4.

PRDG2PARSETREE(G)
1 **for all** nodes $N \in \mathcal{N}[G]$
2 **do** $s \leftarrow$ new `RootStatement`
3 $parseTree[N] \leftarrow s$ ▷ $parseTree[N]$ is the root of the parse tree
4 $s \leftarrow$ STITCH(DOMAIN2PARSTATEMENT($domain[N]$), s) ▷ see Chapter 4.3.2

```
 5    s ← STITCH(DOMAIN2FORSTATEMENT(domain[N]), s)          ▷ see Chapter 4.3.2
 6    s ← STITCH(NODEDOMAIN2GENERICIFSTATEMENT(N), s)        ▷ see Chapter 4.3.2
 7    for all input ports P of N
 8    do s' ← s
 9       s' ← STITCH(INPUTPORT2IFSTATEMENT(P), s')           ▷ see Chapter 4.4.2
10       s' ← STITCH(INPUTPORT2GENERICIFSTATEMENT(P), s')    ▷ see Chapter 4.4.2
11       s' ← STITCH(INPUTPORT2ASSIGNMENTSTATEMENT(P), s')  ▷ see Chapter 4.5.5
12    s ← STITCH(NODE2ASSIGNMENT(N), s)                      ▷ see Chapter 4.2.2
13    for all output ports Q of N
14    do s' ← s
15       s' ← STITCH(OUTPUTPORT2GENERICIFSTATEMENT(Q), s')  ▷ see Chapter 4.4.2
16       s' ← STITCH(OUTPUTPORT2ASSIGNMENT(Q), s')          ▷ see Chapter 4.5.5
```

In lines 4–16, the node domain N is converted into the corresponding parse tree in four parts. Every part corresponds to a set of nodes with the same label in Figure 4.3. Lines 4–6, 7–11, 12, and 13–16, convert the nodes labeled with a 1, 2, 3, and 4 respectively.

The statement s (or s') determines where in the parse tree the statements are to be *stitched*. When the stitch s is not allowed to be overwritten, a copy s' is made of it.

Every node in Figure 4.3 (recall that these nodes are actually a path of nodes of the same type) is obtained by converting a piece of the PRDG into a vector of statements. This conversion is done by the functions with the names DOMAIN2..., PORT2... , and NODE2.... The function STITCH is used to stitch the vector of statements in the proper way into the parse tree. This function is shown below.

```
STITCH(S, s)
1  for all statements p in S
2  do ADD-CHILD(p, s)
3     s ← p
4  return s
```

The methods called in Algorithm PRDG2PARSETREE, together performing *domain scanning* and *linearization*, are explained in previous chapter. The methods rely on several external software implementations. For the basic operations on polyhedra, the PolyLib [61] is used. This library is extended by Loechner and Clauss [64, 66, 69–71] to count the number of integral points in parameterized polytopes. To have the PolyLib functionality available in Java, a Java Native Interface (JNI) [95] to the PolyLib is implemented as part of in PANDA.

In addition PANDA contains a library to perform some basic operations on symbolic pseudo-polynomials. These operations include creation, conversion from and to strings, and operations like addition, multiplication, and substitution (for the non-periodic coefficients).

For the polynomials I use a canonical, distributive, sparse, and zero-represented representation [96], [97]. *Canonical* means that polynomials which are mathematically the same are represented in a unique way, *distributive* means that the polynomials are represented in a sum of products form, *sparse* means that only terms with non-zero coefficient are stored, and *zero-represented* means that if the exponent of a variable is zero then the zero is explicitly stored. The polynomial $\mathbf{Z}[x, y] = 2xy + x^2 + 3$ is internally stored as $1 \cdot x^2y^0 + 2 \cdot x^1y^1 + 3 \cdot x^0y^0$. I chose for this representation because of its simplicity. Since the number of variables and the degree of the polynomials is small, computation time and storage is not much of a issue is selecting the representation.

5.3.2 Operation Example of PANDA

This chapter continues with the case from Chapter 5.2.4. The XML code shown in Program 5.1 is first parsed into a PRDG representation. The Object diagram of this PRDG is the same as in Chapter 5.2.4 and is shown in Figure 5.7.

Onto the PRDG algorithm PRDG-PREPARE is applied. The implicit output port domain in Figure 5.7 is now made explicit by making it equal to the index set of the edge domain. The parse trees of the processes are derived by applying PRDGTOPARSETREE to the prepared PRDG. Since there are two node domains the conversion results in two parse tree. The generate phase in PANDA converts the parse trees into C++ class files defining the processes and converts the PRDG into into a C++ class file defining the network. The complete generated files of the processes are shown in programs 5.4 and 5.5, although they have been slightly beautified manually.

Most of the code declares and defines the variables, ports, data structure, and so on used in the program. The real work is done in `main()` methods at line 23–39 and 25–39 of programs 5.4 and 5.5, respectively.

In Program 5.4 the **for** loops in lines 25 and 26 are generated from node 1b, constructed by Algorithm DOMAIN2FORSTATEMENT, of Figure 4.3. In line 19, the matrix U is initialized by reading all elements from the file `U.sif`. In line 27, the elements are read from the matrix U. This line is generated from a parse node 3 in Figure 4.3 that was constructed by Algorithm NODE2ASSIGNMENT. Since `ND_1` is a source process, only the piece of code generated from the output port domains is present. The condition in line 29 is generated from node 4a in Figure 4.3. The condition in line 30 is generated from node 4b in Figure 4.3. This condition is always true since the output port domain is dense. These two nodes are constructed by Algorithm OUTPUTPORT2GENERICIFSTATEMENT. Finally lines 34–36 are generated from node 4c, constructed by Algorithm OUTPUTPORT2ASSIGNMENT, in Figure 4.3. Clearly, only data is written to output port `WP_1` when $i \geq j$. This are exactly the elements on and above the diagonal of U. Thus only the data that is really used by the consuming process `ND_2` is communicated.

In Program 5.5 the loop structure is similar to that in Program 5.4. Since `ND_2` is a sink process only the piece of code generated from the input port domains is present. In this program the index set of the input port domain is the same as the index set of the node domain, therefore no statements are generated from the parse nodes 2a and 2b in Figure 4.3. Lines 29–34 are generated from parse node 2c, constructed by Algorithm INPUTPORT2ASSIGNMENTSTATEMENT. The structure is similar to lines 3–7 in Program 4.8. Line 29 is the read polynomial and specifies from what element in the RAQ `u_1` is to be read. The read polynomial is expressed by a polynomial divided by an integer. In this way it is guaranteed that the expression evaluates to an integer. Lines 30–33 simulate the blocking semantics of the Kahn process, read operations from input `RP_1` are performed until there is valid data in the RAQ at the element pointed to by the read polynomial. Then, at line 34 the data is read from the RAQ and assigned to the input argument of the function. The function call at line 35 then passes this argument to the function. Like the function in process `ND_1` this function is special in that it communicates to a file. Here, when the process iterated over all points in the index set, the matrix L is written to file `L.sif`.

Program 5.4: ND_1

```
 1  #ifndef ND_1_H
 2  #define ND_1_H

 3  #include "math.h"
 4  #include "process.h"
 5  #include "port.h"

 6  #include "jac_func.h"
 7  #include "JMatrix.h"
 8  #include "Vector.h"
```

```
 9  class ND_1 : public Process {
10   private:
11    double value;         // input ports
12    OutPort<double> WP_1; // output ports
13    int N;                // parameters
14    double out_0;         // function arguments
15    JMatrix U;            // input matrix

16   public:
17    bool doWrite;
18    ND_1(Id n, Out<double>& wp_1, int parm_N) :
19      Process(n), WP_1(id("WP_1"), wp_1), N(parm_N), U( "U" ) {
20      // System.out.println( " --- Process ND_1 Created -- " );
21    };
22    const char* type() const {return "ND_1";};

23    void ND_1::main() {
24      //parameter N = [10] : [100];
25      for ( int j = 1 ; j <= N ; j += 1 ) {
26        for ( int i = 1 ; i <= N ; i += 1 ) {
27          out_0 = U.getElement(j, i);
28          doWrite = false;
29          if ( -j+i >= 0) {
30            if ( ( 1 ) / 1 >= 1) {
31              doWrite = true;
32            }
33          }
34          if ( doWrite==true) {
35            write( WP_1, out_0 );
36          }
37        } // for i
38      } // for j
39    }

40  };
41  #endif
```

Program 5.5: ND_2

```
 1  #ifndef ND_2_H
 2  #define ND_2_H

 3  #include "math.h"
 4  #include "process.h"
 5  #include "port.h"

 6  #include "jac_func.h"
 7  #include "JMatrix.h"
 8  #include "Vector.h"

 9  class ND_2 : public Process {
10   private:
11    double value;         // input ports
12    InPort<double> RP_1;
13    int N;                // parameters
14    double in_0;          // function arguments
15    double out_0;
16    JMatrix L;            // output matrix
17    Vector u_1;           // RACs
18    int readAddress_u_1;  // write addresses for RACs

19   public:
```

```
20    ND_2(Id n, In<double>& rp_1, int parm_N) : Process(n), RP_1(id("RP_1"),
21      rp_1), N(parm_N), L( parm_N, parm_N ), u_1(1000), readAddress_u_1(0){
22      // System.out.println( " --- Process ND_2 Created -- " );
23    };

24    const char* type() const {return "ND_2";};

25    void ND_2::main() {
26      //parameter N = [10] : [100];
27      for ( int i = 1 ; i <= N ; i += 1 ) {
28        for ( int j = 1 ; j <= i ; j += 1 ) {
29          readAddress_u_1 = ( 2*N*j - 2*N + 2*i - pow(j,2) + j - 2 ) / 2;
30          while ( u_1.getWriteAddress() <= readAddress_u_1 ) {
31            read( RP_1, value );
32            u_1.put( value );
33          }
34          in_0 = u_1.getElement(readAddress_u_1);
35          L.setElement( i, j, in_0 );
36        } // for j
37      } // for i
38      L.toFile( "L" );
39    }

40  };
41  #endif
```

The procedure that constructs the network itself is a visitor of the PRDG and is described in Chapter 4.6 and is implemented by the `YapiNetworkVisitor` class, see Figure 5.2. The result is the C++ code in Program 5.6. In line 11 the single edge `ED_1` is declared. The processes `nd_1` and `nd_2` are declared as instances of the classes `ND_1` and `ND_2` just described in lines 13 and 14, respectively. The real construction of the network is done in lines 18–20. Line 18 constructs the edge domain and lines 19 and 20 construct the processes `nd_1` and `nd_2`. In the construction the channel `ED_1` is connected to the output port of `nd_1` and to the input port `nd_2`.

Program 5.6: TRIANGLETRANSPOSE

```
 1  #ifndef triangleTranspose_H
 2  #define triangleTranspose_H

 3  #include "fifo.h"
 4  #include "process.h"
 5  #include "network.h"

 6  #include "ND_1.h"
 7  #include "ND_2.h"

 8  class triangleTranspose : public ProcessNetwork {
 9   private:
10    // fifos
11    Fifo<double> ED_1;

12    // processes
13    ND_1 nd_1;
14    ND_2 nd_2;

15   public:
16    triangleTranspose(Id n, int parm_N) :
17      ProcessNetwork(n),
18      ED_1(id("ED_1")),
19      nd_1(id("ND_1"), ED_1, parm_N),
20      nd_2(id("ND_2"), ED_1, parm_N)
```

```
21      {
22      };

23  };

24  #endif /* triangleTranspose_H */
```

Chapter 6

Conclusions

This dissertation focused on the compilation of imperative nested loop programs into Kahn process networks (KPNs). This approach is chosen for two reasons: on the one hand, applications developers are used to write the application specification in an imperative language (such as Matlab or C), while on the other hand, there is a tendency that the mapping of applications onto embedded systems starts with an inherently parallel model of computation. KPNs have been recognized to be one of such parallel models, specifying applications in an implementation-independent way, yet still allowing these specifications to be tuned to a particular architecture if desired.

To enable the conversion of an imperative program into the KPN model, the set of imperative programs is confined to the class of *piece-wise* affine nested loop programs. This restriction still allows to study the kernels of the applications that belong to the domain of digital signal processing.

A piece-wise affine NLP is first subjected to an exact data dependence analysis implemented by the tool MatParser (outside the scope of this dissertation). The result of this analysis is a functionally equivalent single-assignment program (SAP). In the SAP, there is only one type of data dependency, namely direct flow dependency.

I introduced the *polyhedral reduced dependence graph* (PRDG) model that describes the application in terms of *behavior*, *topology*, and *geometry*. Starting the conversion of the application to the KPN from this model has two advantages: (1) it enables us to decompose the conversion problem into a number of well defined sub-problems, and (2) this model is very useful for program transformations as shown by the systolic array and automatic parallelization communities. For this reason, it was decided to first convert the SAP into the PRDG model and derive the KPN from there.

The PRDG is a graph whose nodes represent sets of operations, and whose edges represent sets of data dependencies. The set of operations of each node is characterized by a single function and an index set that identify the set of integral points that describe the iterations where the function must be computed. Usually functions on the boundary of such an index set depend on the results of functions in some other index set, while functions internal to the index set depend on other functions inside the index set. For this reason, a set of input ports and a set of output ports is associated with a node. Similarly, an index set is associated with each port and an edge in the PRDG connects an output port of one node to an input port of another node. An affine function is associated with each edge that tells which iterations inside the index set of the input port depend on which iterations inside the index set of the output port. In this way, the behavior of the application is captured by the functions inside the nodes, the regular parts of the application are captured by the index sets associated with the nodes, input ports, and output ports, and the irregularity is captured by

the topology of the graph that is described by the PRDG.

Chapter 3 formally defines the PRDG model and deals with the conversion from SAP to to this model. The parse tree representation of the SAP is scanned and the nodes of the parse tree are converted into either nodes of the PRDG or annotations of these nodes. Nodes representing functions are converted into nodes, sets of nested loops are converted in index sets of the nodes, conditional statements (together with the nested loops) are converted into ports together with their index sets, and variable index references are converted into the edges of the PRDG.

Chapter 4 deals with the conversion of the PRDG into the KPN model. A Kahn process network is a network composed of processes which communicate with each other by writing tokens on or reading tokens from unbounded queues that connect them. A token is an abstract container of information. A write operation onto a channel is always non-blocking; a read operation from a channel blocks when no token is available. A Kahn process itself executes a sequential program, or at least, behaves like one. The generation of the KPN from the PRDG is performed in two stages, viz., the generation of the processes and their code, and the generation of the network itself.

The generation of the network is straightforward. A process is generated for every node in the PRDG and a port in the corresponding process is generated for every port in the PRDG. The edges in the PRDG, connecting the ports of the nodes in the PRDG, are mapped onto channels the connect the ports of the corresponding processes in the KPN. This means that the topological views of the KPN and PRDG are in one-to-one correspondence. However, the philosophy taken in this dissertation is that every token written onto a channel is also read from that channel and is useful for the receiver. This means that the straightforward construction of the KPN is to be preceded by a transformation that we called *domain matching*. Domain matching yields a new PRDG with the property that every output port in the new PRDG matches the input port in the PRDG that depends on it. We say that an input matches an output port when the dependence function is a bijection between the points inside their index sets. This mapping allows the mapping of the edges of the new PRDG onto channels of the KPN such that every token written onto a channel is also read and used.

The generation of the processes requires that code must be generated in such way that functions are called in the proper order, and that they operate on the proper data. The generation of the functions in the proper order requires that the index set of each node in the PRDG be scanned and that the proper loops together with guard code be synthesized. This this task is performed by *domain scanning*. The domain scanning procedure described in this dissertation relies on methods found in literature and extends these to deal with the index sets found in the PRDG model.

Letting the functions operate in the proper order is required the semantics of the channels in the KPN. Since a channel represents a queue, the order in which data is written onto a channel is the same as the order in which data is read from that queue. However, the order in which the consuming process reads the tokens from the channel may be different that the order in which the values contained in these tokens are passed to the functions. This requires that reordering of data must be performed. The reordering method described in this dissertation models the reordering problem by supporting random access to a linear (1-dimensional) arrays that can be seen as extensions to the queues of the channels. The problem of mapping the higher-dimensional indexing functions onto indexing functions of the linear array is what we call *linearization*. The linearization problem is formulated in terms of counting problems in polytopes. Such a counting problem consists in determining the number of integral points contained by a parameterized polytope. The literature describes a solution to this problem by so called *Ehrhart polynomials*, whose derivation is implemented by a software library called POLYLIB. The problem of linearization when broadcasts inside an index set are present is not yet dealt with, and left as future work. For now, we require that a procedure called localization is performed on the source program to resolve these broadcasts.

For all methods that are described in this dissertation, software versions have been written and are part of

the COMPAAN tool-set. The dependence analysis (which is not part of this dissertation) is implemented by the tool MATPARSER, the conversion of the SAP to the PRDG by the tool DGPARSER, and the conversion of the PRDG to the KPN by the tool PANDA. These tools are written in the object-oriented language Java which allows for a well structured design of compiler-like software. Throughout this dissertation special attention is paid to have the formulation of methods and algebra in close resemblance to their software versions. In this way this dissertation can be used to understand the internal structure and the operation of the implementation. Modern techniques, including design pattern, UML, and XML, have been used to keep the software well organized. The tools are tested for a small set of sample programs, each testing a different aspect of the flow, to proof the feasibility of the approach.

Bibliography

[1] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[2] Stephan Wong, Sorin Cotofana, and Stamatis Vassiliadis. Coarse reconfigurable multimedia unit extension. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing (PDP 2001)*, Mantova, Italy, 2001.

[3] Robert Schreiber, Shail Aditya (Gupta), Scott Mahlke, Vinod Kathail, Bob Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing.*, 2001.

[4] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prohid, a data-driven multi-processor architecture for high-performance dsp. In *Proc. ED&TC*, March 17-20 1997.

[5] Edwin Rijpkema, Ed F. Deprettere, and Gerben Hekstra. A strategy for determining a Jacobi specific dataflow processor. In *Proceedings ASAP'97 conference*, July 1997.

[6] Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *VLSI Signal Processing, IX*, pages 461–470, 1996.

[7] J-Y. Brunel, H.J.H.N. Kruijtzer, F. Pétrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits. Cosy communication IPs. In *Proceedings of the 37th Design Automation Conference DAC 2000*, 2000.

[8] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. The construction of a retargetable simulator for an architecture template. In *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, March 15–18 1998.

[9] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings of the 1999 IEEE Workshop in Signal Processing Systems*, Taipei, Taiwan, 1999.

[10] Edwin Rijpkema and Ed F. Deprettere. A parallel processor for fast execution of time-adaptive Jacobi algorithms. In J.P. Veen, editor, *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, pages 261–266, November 1996.

[11] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[12] Peter Held. *Functional Design of Data-Flow Networks*. PhD thesis, Dept. EE, Delft University of Technology, May 1996.

[13] Bart Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, February 2000.

[14] H. T. Kung and C. C. Leiserson. *Algorithms for VLSI Processor Arrays*, chapter 8.3, pages 271–292. Addison-Wesley, Reading, Mass., 1980 (1978).

[15] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[16] D. Moldovan. On the analysis and synthesis of vlsi algorithms. *IEEE Transactions on Computers*, C-31(11):1121–1126, November 1982.

[17] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Conference Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 208–214, Ann Arbor, Michigan, June 1984.

[18] Jean-Marc Delosme and Ilse C.F. Ipsen. Systolic array synthesis: Computability and time cones. In *In M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente, editors, Int. Workshop on Parallel Algorithms and Architectures*, pages 295–312. Elsevier Science, North Holland, April 1986, 1996.

[19] Alain Darte. Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model.

[20] Sailesh K. Rao and Thomas Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE.*, 76(3), March 1988.

[21] Vwani P. Roychowdhury. *Derivation, Extension and Parallel Implementation of Regular Iterative Algorithms*. PhD thesis, Stanford University, Dept. of Electrical Engineering, Stanford, CA, December 1988.

[22] M. van Swaaij. *Data Flow Geometry: Exploiting Regularity in System-level Synthesis*. PhD thesis, Interuniversitair Mikro-Elektronica Centrum, Leuven, Belgium, 1992.

[23] C. Mauras, P. Quinton, Sanjay Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S.Y. Kung and E. Swartzlander, editors, *Proceedings of the International Conference on Application Specific Array Processing 1990, ASAP'90, IEEE Computer Society*, pages 100–110, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[24] J. L. van Meerbergen, P. E. R. Lippens, W. F. J. Verhaegh, and A. van der Werf. PHIDEO: High-level synthesis for high throughput applications. *Journal of VLSI Signal Processing*, 9:89–104, 1995.

[25] J. Annevelink. *HIFI: A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays*. PhD thesis, Delft University of Technology, The Netherlands, 1988.

[26] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. On Manipulating $\mathbb{Z}$-polyhedra using a Canonical Representation. *Parallel Processing Letters*, 1997.

[27] Lothar Thiele. *Compiler Techniques for Massive Parallel Architectures*, pages 101–150. Kluwer Academic Publishers, P.O. Box 17, 3300 AA, Dordrecht, the Netherlands, 1992.

[28] J. Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14:297–332, February 1993.

[29] Lothar Thiele. From linear recursions to computing arrays. In *IEEE Conf. on Circuits and Systems, Nanjing, China*, pages 115–118. IEEE, January 1989.

[30] Lothar Thiele. On the hierarchical design of VLSI processor arrays. In *IEEE International Symposium on Circuits and Systems, Helsinki*, pages 2517–2520. IEEE, January 1988.

[31] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[32] John Backus. Can programming be liberated from the von neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[33] Alfred van der Hoeven. *Concepts and Implementation of a Design System for Digital Signal Processing*. PhD thesis, Delft University of Technology, Delft, The Netherlands, October 1992.

[34] Peter S. Pacheco. A user's guide to mpi.

[35] G. A. Geist, J. A. Kohla, and P.M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[36] J. Merlin and B. Chapman. High performance fortran 2.0. In *Proceedings of Sommerschule uber Moderne Programmiersprachen und Programmiermodelle*, pages 15–19, Technical University of Hamburg, Harburg, September 1997.

[37] Thomas Brandes. Adaptor programmers guide, version 7.0, 1999.

[38] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[39] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[40] M. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):542–471, October 1991.

[41] Christian Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pages 398–416, 1993.

[42] Thomas Brandes. Adaptor: A compilation system for data parallel fortran programs. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*. Vieweg, Wiesbaden, 1994.

[43] PRiSM/SCPDP. Systematic construction of parallel and distributed programs, project home page, http://www.prism.uvsq.fr/english/parallel/paf/axes_us.html.

[44] The PIPS Workbench Project. Centre de recherche en informatique/École des mines de paris, home page, http://www.cri.ensmp.fr/pips.

[45] P Boulet and Michèle DION. Code generation in bouclettes. In *Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 273–280, London, UK, January 1997. IEEE Computer Society Press.

[46] Doran K. Wilde. *From* ALPHA *to Imperative Code: A Transformational Compiler for an Array Based Functional Language*. PhD thesis, Oregon State University, July 1995.

[47] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes 1, Rennes, France, February 1993.

[48] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelisation. In *27th Hawaii International Conference on System Sciences*. IEEE, January 1994.

[49] Philippe Clauss, Vincent Loechner, and Frédéric Vivien. Program compiliation and optimization, http://icps.u-strasbg.fr/pco/.

[50] Vincent Loechner, Benoît Meister, and Philippe Clauss. Data sequence locality: a generalization of temporal locality. In *Proceedings of the Europar'2001, Manchester, UK*, aug 2001.

[51] Philippe Clauss and Benoî Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. *ACM SIGARCH Computer Architecture News*, 28(1), mar 2000.

[52] Om Prakash Gangwal, Andre Nieuwland, and Paul Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *ISSS*, pages 1–6, 2001.

[53] Kees Goossens, Jef van Meerbergen, Ad Peeters, and Paul Wielage. Networks on silicon: Combining best-effort and guaranteed services. In *Proceedings of the design automation and test conference*, March 2002.

[54] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE, 75(9)*, September 1987.

[55] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[56] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *IEEE Int. Conf. ASSP*, pages 3255–3258, Detroit, Mitchigan, May 1995.

[57] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prophid, a data-driven multi-processor architecture for high-performance DSP. In *Proc. ED&TC*, March 17-20 1997.

[58] 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. Design Automation Conference, ACM 2000*, pages 402–405, 2000.

[59] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[60] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[61] Doran K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, Dec 1993. Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.

[62] T. Mattheiss and D. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Math. of Op. Research*, 5(2):167–185, 1980.

[63] Hervé Le Verge. A note of chernikova's algorithm. Technical report, Irisa, 1992.

[64] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6), December 1997.

[65] The Polylib Team. Polylib user's manual, http://www.irisa.fr/polylib/.

[66] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.

[67] P.M. Gruber and C.G. Lekkerkerker. *Geometry of Numbers*. North-Holland, Amsterdam, 1987.

[68] E. Ehrhart. Sur les polyèdres rationnels homothétiques à n dimensions. *C.R. Acad. Sci. Paris*, 254:616–618, 1962.

[69] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96, Chicago, Illinois*, pages 415–424. 1996, August 1996.

[70] Ph. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyse and transform scientific programs. In *10th ACM International Conference on Supercomputing, ICS'96, Philadelphia*, May 1996.

[71] Vincent Loechner. *Contribution à l'Étude des Polyèdres Paramétrés et Applications en Parallélisation Automatique*. PhD thesis, Université Louis Pasteur, Strasbourg, December 1997.

[72] Richard P. Stanley. *Enumerative Combinatorics: Volume 1*. Wadsworth Inc., Belmot, California 94002, 1986.

[73] C Ancourt. *Génération Automatique de Code de Transfert pour Multiprocesseurs à Mémoires locales*. PhD thesis, Université de Paris VI, 1991.

[74] A.J. Goldman. Resolution and separation theorems for polyhedral convex sets. In H.W. Kuhn and A.W. Tucker, editors, *Linear inequalities and related systems*, Princeton, NJ, 1956, 1956. Princeton University.

[75] Thomas H. Cormen, Charles E. Leisserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1993.

[76] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061 USA, 1988.

[77] S.Y. Kung. *VLSI Array Processors*. Prentice-Hall International Editions, 1988.

[78] P. Feautrier. Compiling for massively parallel architectures: A perspective. *Algorithms and Parallel VLSI Architectures III*, pages 259–270, 1995.

[79] Ed F. Deprettere, Peter Held, and Paul Wielage. Model and methods for regular array design. *Int. J. of High Speed Electronics, Special issue on Massively Parallel Computing-Part II*, 4(2):133–201, 1993.

[80] William Barnier and Jean B. Chan. *Discrete Mathematics with Applications*. West Publishing Company, 1989.

[81] Bart Kienhuis. Parallelizing nested loop programs containing div, floor, ceil, mod and step functions. Master's thesis, Delft University of Technology, Delft, the Netherlands, 1994.

[82] W. Pugh. Counting solutions to presburger formulas: How and why. In *Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Princeton, NJ, 1956, 1994.

[83] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[84] Joseph T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Minneapolis, MN*, volume I, pages 429–432, April 1993.

[85] K. Arvind and K. P. Gostelow. *Formal Description of Programming Languages*, chapter Some Relationships Between Asynchronous Interpreters of a Dataflow Language. (E. J. Neuhold, Editor, North-Holland Publ. Co., New York, 1977.

[86] Arvind and Gostelow. The u-interpreter. *Computer*, 15(2), Februari 1982.

[87] R. Jagannathan. *Parallel and Distributed Computing Handbook*, chapter Dataflow Models. (E.Y. Zomaya, Editor), McGraw-Hill, 1995.

[88] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Dept. EECS, University of California, Berkeley, December 1995.

[89] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[90] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. of Parallel Programming*, 28(5), October 2000.

[91] Philippe Clauss. The volume of a lattice polyhedron to enumerate processors and parallelism, research report ICPS 95-11. Technical report, Université Louis Pasteur, dept. ICPS, Strasbourg, 1995.

[92] Alco O. Looye. Multiport memory and floating point cordic pipeline on Jacobium processing elements. In *Proceedings of the ProRISC/IEEE on Circuits, Systems, and Signal Processing 1997*, November 1997.

[93] Grady Booch, James RumBaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[95] S. Liang. *The Java Native Interface*. Addison-Wesley Java Series, 1999.

[96] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms For Computer Algebra*. Kluwer Academic Publishers, Kluwer Adademic Pulishers Group, Distribution Centre, Post Office Box 322, 3300 AH Dordrecht, the Netherlands, 1992.

[97] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and algorithms for algebraic computation*. Academic Press, Academic Press inc., San Diego, CA 92101, second edition, 1993.

Index

Samenvatting

Modelleren van taak niveau parallellisme in stuksgewijs regelmatige programma's.

Digitale signalen zijn stromen van informatie eenheden die *tokens* genoemd worden. Het vakgebied van de digitale signaalbewerking (of verwerking) houdt zich bezig met het bewerken en verwerken van stromen van tokens. Voorbeelden hiervan zijn het verwerken en bewerken van video, audio, multimedia, radar en sonar signalen. Dit gebeurt met behulp van procedures die stromen van tokens converteren in nieuwe stromen van tokens. Dit converteren gebeurt door middel van operatoren. Een operator opereert op tokens van binnenkomende stromen en produceert op basis hiervan nieuwe tokens die uitgaande stromen vormen.

De procedures op de stromen van tokens in signaalbewerking procedures zijn vaak operatief in een cyclische regelmaat. Door deze regelmaat kunnen deze procedures op een compacte manier beschreven worden door middel van *nested loop programs* (geneste lus programma's, afgekort NLP). Daarom worden veel toepassingen, of delen daarvan, binnen het domein van de digitale signaal bewerking vaak uitgedrukt in termen van NLP's.

NLP's zijn compact, omdat ze sequentiële ordening van operaties compact kunnen specificeren. Echter, deze specificatie is alleen passend als de computer architectuur die dit programma moet uitvoeren ook sequentieel van aard is, wat het geval is bij instructieset architecturen. Voorbeelden hiervan zijn microprocessors die in PC's gebruikt worden en DSP's (digitale signaal processors).

In toenemende mate ontstaat de vraag om stromen te bewerken waarvan de tokens steeds korter op elkaar volgen. De bovenstaande architecturen kunnen niet langer aan deze vraag voldoen en er zal moeten worden overgestapt naar een ander type architectuur. In dit proefschrift beschouwen we architecturen die meerdere seqientiële programma's, ofwel *taken*, gelijktijdig kan uitvoeren. Dit gelijktijdig uitvoeren wordt parallelle executie genoemd.

Door de sequentiële aard van de NLP's ontstaat het probleem dat programma's in dit model niet passend zijn voor architecturen die ontworpen zijn voor een parallelle executie van taken. Daarom dienen NLP's eerst vertaald te worden naar een model dat wel passend is voor dit soort architecturen.

Het model waar in dit proefschrift voor gekozen is, is het *Kahn proces netwerk* (KPN) model. Het KPN model bestaat uit processen en kanalen tussen de processen. Een proces executeert een taak, gespecificeerd als een sequentieel programma, en kan tijdens deze executie tokens lezen van binnenkomende kanalen en tokens schrijven op uitgaande kanalen. Een kanaal kan een onbegrensd aantal tokens bevatten. Een proces kan altijd tokens op een uitgaand kanaal schrijven maar zal blokkeren wanneer er gelezen wordt van een kanaal dat leeg is. Dit model is gekozen omdat het onafhankelijk is van de architectuur waarop deze afgebeeld dient te worden, maar ook omdat het een aantal handige eigenschappen heeft waaronder compositieoneerbaarheid en executie volgorde onafhankelijkheid. Deze laatste eigenschap wil zeggen dat

de uitkomst van de toepassing altijd dezelfde is, ongeacht de volgorde waarin de processen op de tokens in de stromen opereren.

Dit proefschrift beschrijft een verzameling van methoden om NLP's te vertalen naar KPN's. Deze vertaling gebeurt in drie stappen.

De eerste stap is een afhankelijkheidsanalyse die het originele NLP vertaalt naar een *single assignment program* (enkelvoudig-toekenningsprogramma, afgekort SAP). Een SAP is zelf weer een NLP maar dan met de eigenschap dat aan iedere variabele in het programma slechts één maal een waarde toegekend wordt. Het gevolg hiervan is dat de operaties in een SAP in elke willekeurige volgorde geëxecuteerd mogen worden zolang iedere variabele maar een waarde toegekend heeft gekregen voordat deze wordt gebruikt. De afhankelijkheidsanalyse is geïmplementeerd in de tool MATPARSER en is geen deel van dit proefschrift.

De tweede stap is de vertaling van de SAP naar een *polyhedral reduced dependence graph* (polyhedraal gereduceerde afhankelijkheidsgraaf, afgekort PRDG). Er zijn twee redenen voor de introductie van dit model. Ten eerste maakt dit model het mogelijk om de vertaling vanuit dit model naar het KPN model op te splitsen in een aantal goed gedefinieerde deelproblemen. Ten tweede is dit model bij uitstek geschikt voor het toepassen van transformaties zoals die worden toegepast bij het afleiden van en afbeelden op regelmatige rij-architecturen en binnen de parallelliserende compilers.

Een PRDG is een gerichte graaf bestaande uit knopen (*eng., nodes*) en kanten (*eng. edges*). Een knoop heeft een aantal ingangspoorten en uitgangspoorten. Iedere knoop beschrijft een verzameling van operaties en dus ook verzamelingen van ingangsargumenten en resultaten. Verzamelingen van ingangsargumenten en resultaten worden respectievelijk gebonden aan de in -en uitgangspoorten van de knoop. Een kant verbindt een uitgangspoort aan een ingangspoort en beschrijft daarmee een verzameling van afhankelijkheden tussen de operaties in de knopen.

De lussen in een SAP beschrijven de verzameling van *iteraties* waarop de functies geëvalueerd moeten worden. Zo'n verzameling van iteraties wordt een *index verzameling* genoemd en wordt in het PRDG model op compacte wijze beschreven door een *polytope* en een afbeelding. De verzameling van afhankelijkheden beschreven in een kant en de verzamelingen van argumenten en resultaten van de verzameling van functies worden op eenzelfde manier beschreven.

Op deze manier beschrijft iedere knoop in de PRDG een regelmatig deel van het SAP en komt de niet-regelmaat in het SAP tot uiting in het aantal knopen in de PRDG en de kanten ertussen. De vertalingmethode die SAP's vertaalt naar PRDG's is geïmplementeerd in de tool DGPARSER.

De derde stap is het vertalen van de PRDG naar een KPN en bestaat uit twee delen; het genereren van de processen en het genereren van het netwerk dat de processen met elkaar verbindt.

De generatie van het netwerk is eenvoudig. Voor iedere knoop in de PRDG wordt een proces gegenereerd. Daarbij krijgt het proces een poort voor elke poort van de knoop. De kanten in de PRDG verbinden paren van poorten en worden afgebeeld op kanalen in het KPN die de corresponderende poorten van de processen met elkaar verbinden. Dit betekent dat de topologie van de PRDG gelijk is aan de topologie van het gegenereerde KPN.

De generatie van de processen zelf gebeurt in drie stappen, namelijk, *domain matching*, *domain scanning*, en *linearization*.

Om geen overbodige communicatie in de gegenereerde KPN's te hebben is er voor gekozen alleen die tokens op een kanaal te schrijven die daadwerkelijk nodig zijn voor het proces dat van dit kanaal leest. Om dit te bewerkstelligen wordt de *domain matching* transformatie op de PRDG uitgevoerd die er voor zorgt dat iedere uitgangspoort in de PRDG verbonden is met ten hoogste één ingangspoort en dat ieder paar van poorten dat door een kant verbonden is een passend aantal resultaat-argument paren heeft. Het resultaat is

een PRDG zonder overbodige communicatie.

De generatie van de processen behelst de code generatie van het sequentiële programma dat in het Kahn proces geëxecuteerd wordt. Om deze code enigsinds compact te houden willen we dat deze programma's NLP's zijn. De index verzameling van de functie in iedere knoop in de PRDG moet dus vertaald worden naar een NLP. Deze vertaling wordt *domain scanning* genoemd. De methode die in dit proefschrift gebruikt wordt voor domain scanning berust op methoden die in de literatuur bekend zijn en breidt deze uit voor het gebruik ervan op de index verzamelingen zoals deze in het PRDG model voorkomen.

Wanneer eenmaal door de domain scanning procedure de volgorde waarin de functies in een proces executeren bepaald is, dient er voor gezorgd te worden dat deze functies op de juiste tokens opereren. Drie zaken dienen hier in beschouwing genomen worden. Ten eerste, om het communicatie model eenvoudig te houden is er voor gekozen dat de token producerende processen de tokens in dezelfde volgorde op de kanalen schrijven als waarin deze tokens geproduceerd worden. Ten tweede, de kanalen in het KPN model zijn queues (wachtrijen). Hierdoor zal de volgorde waarin de tokens die van de kanalen gelezen worden dezelfde zijn als waarin deze erop geschreven zijn. Ten derde, de token consumerende processen zullen in het algemeen de tokens in een *andere* volgorde willen bewerken als waarin deze van het kanaal gelezen zijn. Hierdoor zal in het algemeen herordening van de tokens nodig zijn. In dit proefschrift wordt een herordening methode beschreven die we *linearization* noemen. Het linearisatie probleem wordt geformuleerd in termen van telproblemen in polytopen. Zo'n telprobleem bestaat uit het bepalen van het aantal integer punten dat een geparameteriseerde polytope bevat. De literatuur beschrijft een methode voor het oplossen van dit probleem met behulp van zogenoemde *Ehrhart polynomen*. De methode is geïmplementeerd in de bibliotheek POLYLIB. Het linearisatie probleem is (nog) niet opgelost ingeval er sprake is van *broadcasts* binnen een index verzameling. Dit is het geval wanneer een resultaat van een functie gecommuniceerd dient te worden naar meerdere ingangsargumenten in een enkele ingangspoort. De vertaling van het PRDG model naar het KPN model is geïmplementeerd in de tool PANDA.

Alle methoden die in het proefschrift beschreven zijn zijn geïmplementeerd in de tools MATPARSER, DGPARSER en PANDA die tezamen de COMPAAN vertaler vormen.

Deze tools zijn geïmplementeerd in de object-georiënteerde taal Java die een goed gestructureerd ontwerp van compilerachtige software toestaat. Door het hele proefschrift heen is aandacht besteed aan de relatie tussen de algebraïsche concepten en de software implementatie. Op deze manier kan het proefschrift ook gebruikt worden om de interne structuur en werking van de implementatie te begrijpen.

Curriculum Vitae

Edwin Rijpkema was born on October 27, 1970 in Amstelveen, the Netherlands. In 1987 he received his MAVO diploma at the Hermann Wesselink College in Amstelveen after which he did the first three years of the MTS, electronics at the Christelijke MTS Patrimonium in Amsterdam. Then in 1990 he went to the HTS to do the first year of a B.Sc. program at the HTS-A in Amsterdam. In 1991 he started his study in electrical engineering at the Delft University of Technology. He joined the Information and Communication Theory group, where he received in 1995 his M.Sc. after his graduation project on fractal image coding for video sequences. In 1996 he joined the Circuits and Systems group at the same university on a Ph.D. position and conducted research in the context of mapping of digital signal processing applications onto parallel architectures. In particular, the research entailed the derivation of process networks from imperative nested loop programs. In 2000 he joined the Leiden Embedded Research Center at the Leiden Institute of Advanced Computer Science of the Leiden University, to continue his Ph.D. research. Also in 2000 he got employed at the Philips Research Laboratories (NatLab) at the embedded system architectures group. As a member of the Æthereal project, he carries out research on networks-on-chip with emphasis on router architectures.