Stellingen

behorende bij het proefschrift

Functional Design of Data-Flow Networks

Peter Held

1. Om een algoritme efficiënt op een regelmatig data-flow netwerk af te beelden, is het noodzakelijk gedetailleerde informatie te hebben van de data afhankelijkheidsstructuur van het algoritme.

   In order to map an algorithm efficiently on a regular data-flow network, it is necessary to have detailed information about the algorithm's data dependence structure.

2. Het is onvermijdelijk, dat het modelleren van een reëel object gepaard gaat met weglating, generalisering en vervorming van dat object.

   It is inevitable, that the process of modeling of a real object is accompanied by deletion, generalization, and distortion of that object.

3. De kracht van object georiënteerde methoden is dat zij kunnen worden toegepast in de analyse en de ontwerp fase, maar ook omdat er goede implementatie talen beschikbaar zijn, zoals Objective C.

   The power of object oriented methods is that they can be used in the analysis and design phase, but that there are also good implementation languages available, such as Objective C.

4. Het gebruik van database methoden, zoals versie en simultaan-toegang beheer, is essentieel wanneer meerdere personen op een zelfde project werken.

   The use of database methods, such as versioning and concurrency control, is essential when several persons are working on a single project.

5. De rekenkracht van data-flow netwerken is gebaseerd op het toepassen van gedistribueerd geheugen en besturing.

   The processing power of data-flow networks is based on the use of distributed memory and control.

6. Een goed ontwerp is een ontwerp dat net voldoet aan de ontwerpeisen.

   A good design is a design that just meets the design constraints.

7. Net als het verbreden van snelwegen vaak slechts leidt tot het verplaatsen van files, zo zal ook het gebruik van Internet slechts de files doen verplaatsen van de asfalt snelwegen naar de informatie snelwegen.

8. Het streven van de wetenschap om kennis op te bouwen, lijkt op het bouwen aan de toren van Babel. Net als toen, leidt het nu ook tot een Babylonische spraakverwarring door het toenemend gebruik van vakjargon.

9. Het uiteindelijke doel van leven is niet kennis maar actie.
Thomas Henry Huxley.

10. Men kan stilstaan in stromend water, maar niet in de wereld van mensen.
Japans spreekwoord.

11. Iemand die te zeer vasthoudt aan zijn eigen ideeën treft weinigen om met hem in te stemmen.

12. Het gebruik van de fiets in Den Haag wordt een steeds slechter alternatief voor de auto, omdat in Den Haag het onderhoud aan fietspaden bestaat uit het plaatsen van gele bordjes met het opschrift 'Slecht Wegdek'.

13. Een deel van de voedseltekorten en milieuproblemen kunnen worden opgelost wanneer meer mensen voortaan vegetarisch gaan eten.

14. In het schaakspel krijgt niet elke speler de overwinning die hij verdient, omdat een eindstelling met een koning en twee paarden voor wit, tegen alleen de zwarte koning, theoretisch remise is.

FUNCTIONAL DESIGN OF DATA-FLOW NETWORKS

Peter Held

Delft University of Technology
May 1996

FUNCTIONAL DESIGN OF DATA-FLOW NETWORKS

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op maandag 20 mei 1996 te 10.30 uur

door

Pieter Cornelis HELD

elektrotechnisch ingenieur
geboren te 's-Gravenhage

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. P.M. Dewilde

Toegevoegd promotor: Dr.ir. E.F. Deprettere.

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. P.M. Dewilde, TU Delft, promotor
Dr.ir. E.F. Deprettere, TU Delft, toegevoegd promotor
Prof.dr. F.P. Quinton, IRISA, Rennes, Frankrijk
Prof.dr. S. Vassiliadis, TU Delft
Prof.dr.ir. J.A.G. Jess, TU Eindhoven
Prof.Dr. -Ing. O.E. Herrmann, TU Twente
Dr.ir. K.A. Vissers, Philips Research, Eindhoven

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Held, Pieter Cornelis

Functional design of data-flow networks /
Pieter Cornelis Held. - [S.l. : s.n.] (Den Haag : Dryadis). - Ill.
Thesis Technische Universiteit Delft. - With ref.
ISBN 90-9008182-8
Subject headings: data-flow networks / dependence analysis / system design.

Contents

Chapter 1

Introduction

Digital signal processing is increasingly used in all kinds of scientific, industrial and consumer applications. Typical examples can be found in communication (high speed modems, wireless cellular telephony, in-door cordless communication, audio and video conferencing), multimedia (image, speech and audio coding and compression), imaging (tomography, synthetic aperture radar, underwater acoustics), computer graphics (rendering, visualization, virtual reality) and in control (system identification, plant control, robotica).

In some of these applications, the processing of signals may be very demanding. This may come from the often huge number of operations that have to be carried out. It may also come from the fact that the processing of signals has to be carried out in real time. Still other constraints may restrict implementation volume, power dissipation and storage capacity.

Here is an illustrative example. Suppose a video signal has to be processed at a rate of 50 frames per second, each frame consisting of 1024×1024 pixel elements, and each pixel requiring 128 elementary operations. This results in a throughput rate of about 6,000 million elementary operations per second. Programmable off-the-shelf processors do not have this computational capacity. Moreover, even if they would have such capacities, low-power constraints might rule them out. Indeed, the video procession example considered here could be just one component in an audio, video and data processing unit, say a wireless network node, whose software supervisor is a battery operated portable CPU. Assuming that this CPU is a laptop computer, it will be clear that all signal processing components of the wireless network node will have to be very compact and will have to operate in a low-power low-voltage mode so that they all fit on a board not larger than the size of the laptop and do not exhaust the laptop power supply whenever the node has to operate in the network.

This example is typical for the applications we have in mind. In these applications, there is a continuous supply of signals - in the form of samples, one-dimensional sequences, or vectors, two-dimensional sequences or matrices or whatever appropriate quantity measured

in amount per unit of time. These signals are processed and the processed signals are taken away at the same or a lower rate than their acquisition rate. The processing, then, is very often in the form of a repetitive invocation of a relative small number of functions that take a signal - in whatever type it is defined - as an argument and return a value that may or may not appear as an argument for another function. These functions are commonly represented as nodes in a graph in which the arcs interconnecting the nodes are the paths along which the signals flow as arguments of the functions - when flowing toward nodes - and results of the functions - when flowing away from nodes. These so-called flow graphs clearly reveal that, in most cases, some of the functions can be evaluated concurrently and - important for us - the graph has a periodic structure. Indeed, signal processing flow graphs are typically periodic and on top of that, they sometimes reveal a striking degree of regularity which make them think of crystals or cobwebs or beehives and the like. As signals may enter and leave the graph at equal and fast rates, parallelism must usually be exploited to allow the periodic execution to be faultless. Moreover, the processing of signals is typically to be implemented in field applications, that is, at locations which do not offer room for large scale implementations. And last but not least the number of applications requiring low-voltage low-power implementations is growing rapidly.

As a consequence of our focusing on such applications, the classical processor which processes the operations sequentially, which also has tremendous overhead and demands for large power supply must be ruled out as candidate implementation architecture. This is not only so for the micro processor - say the RISC processor - even the traditional signal processor - say TMS320Cx (Texas Instruments), DSP5600x (Motorola) or ADSP-210x (Analogic Devices) - is designed for too broad an application range for it to be a feasible platform for implementing our high throughput high density signal processing applications. These sequential processing systems are thus inadequate and systems based on parallel and/or pipelined processing have to be considered and designed. Such systems will inherently be application specific and most likely of type processor array - one dimensional or higher dimensional - which may or may not fit into a single chip implementation. Notice that parallel processing not only caters for higher speeds, it also provides a solution to low-power implementation requirements. Indeed, parallel processor arrays are more cost effective in terms of power consumption [46].

The conjecture that effective and efficient implementations must be application specific stems from the following observation. Digital signal processing systems are implementations of architectures into which algorithms are executed. The usefulness of a particular architectural style depends heavily on the specific features and properties of the algorithms that are to be implemented. For example, an architecture whose structure is a mesh of nearest neighbor communicating functions may not be useful when the algorithm enforces a tree-like structure as a consequence of the algorithm's specific internal dependency structure. By contrast, this type of mesh is most suited for the implementation of convolution-like algorithms such as an inner-product based algorithm for the computation of the product of two matrices. A judicious analysis of the dependency structure of an algorithm may reveal that there is a large amount of inherent parallelism in it and that the

flow of arguments toward functions and results away from functions is local. Though the algorithm dependency structure is a key issue in determining the appropriate architectural structure and style for the implementation of it, one of the major problems is to make the algorithm's dependency structure visible, that is, explicit. Indeed, almost all algorithms are given in the form of an executable program which has been developed on a workstation or PC in a common imperative language such as C. Such a program has an inherent sequential execution nature and, therefore, hides whatever parallelism it may have in itself, as much of the inherent parallelism is captured in artificial dependencies due to the strict ordering of operations in sequential programs. One can argue that one could get round this difficulty by not starting from the sequential program specification but developing a suitable algorithm and designing an accessory architecture simultaneously and at once. However, this is a difficult task and, moreover, it does not always make sense as it would ignore the tremendous amount of research efforts in algorithm design which, unfortunately, is commonly put in public domain as a sequential program. Thus, even if a problem solution can be cast in a novel algorithm and a new architecture for it, common practice is such that existing programs realizing an algorithm are first analyzed and brought into a form which is void of side effects and transparent with respect to inherent parallelism. Only when this structure appears not to lead to an efficient accompanying architecture makes it sense to conceive a redevelopment of the algorithm which will then at least be guarded by and inspired through the results obtained from the analysis of given executable specifications. Thus, the analysis of programs remains a problem and this dissertation proposes a solution to it - at least for a subclass of programs - and provides an implementation of the procedure by which the analysis of programs from the feasible class can be carried out automatically.

There is thus a strong - be it not always explicit - interplay between algorithms and architectures, and for the drawing up of the specification of the ultimate implementation there is a need for methods to provide workable reference algorithms and to derive from them accompanying architectures for their execution. For that to be achievable systematically, it is convenient to let both algorithm and architecture be expressed as flow graphs. The data flow or signal flow graph has proven to be very powerful even long before the birth of what is now called signal processing. Simulation environments and design systems that use signal flow graph concepts do exist. Examples are SPW (Cadens), COSSAP (Synopsys), DSP station (Mentor Graphics), Signal [24] and Ptolemy [14]. The latter system is special in that it knows several different models of data flow which allow the specification and manipulation of heterogeneous or so-called embedded systems. All of these environments and systems tend to be generic in that they do not distinguish between irregular and regular graphs. Regular flow graphs have special properties which can be exploited to make their specification, description and manipulation elegant and tractable. In fact, regular flow graphs have inherent structure whereas irregular graphs do not have such structure. The latter may require the introduction of structure in order for them to be turned into efficient implementations. The former, on the other hand, require special concern in order not to destroy this inherent structure during the process of converting them into imple-

mentations. Architectures that reflect algorithm regularity and are themselves regular are reminiscent of H.T. Kung's systolic architectures [44][42], introduced in the eighties. Although systolic architectures are but concepts and closer to algorithm structures than to architectures, they have undoubtedly paved the way to what are now called piecewise regular architectures [43].

Roughly speaking, these architectures are computational networks or networks of communicating functions which are local in time and space. These will be our target architectures, and algorithms that are naturally executed in such architectures will likewise be our target algorithms. Nested loop programs are, as a matter of fact, akin of those algorithms and form by far the largest class of algorithms in signal and image processing. It is a subclass of this class of algorithms to which we alluded above and for which we propose methods to analyze them and to bring them in flow graph models which do take the inherent structure into account. They are called reduced-size graphs and the challenge, then, is to develop methods to transform such graphs from the level of algorithm graph down to the level of architecture graph without breaking their property of being of reduced size, that is, without destroying the regularity structure.

The problem that we thus face is basically the following. How can signal processing tasks be cast into effective and efficient parallel algorithms and accompanying parallel architectures in such a way that regularity, if present as a structure property, is preserved as much as possible and constraints, imposed by the application and possibly also from presupposedly inferred implementation limitations, are obeyed in an as natural way as possible. We are thus seeking for a design methodology which gently fits the domain of (piecewise) regular algorithm and architecture flow graphs.

This is a rather complex subject. However, the inherent structural property of regularity suggests that models and methods can be envisaged which can be cast in rigorous mathematical terms, in particular of a linear integer algebra nature. The intended methodology has been laid down partly in two European Basic Research Action programs (BRA 3280 - NANA, and BRA 6632 - NANA2) [1]. Partners that took part in the consortium have been IMEC, Louvain, Belgium; ESAT, Katholieke Universiteit Leuven, Louvain, Belgium; LIP, Ecole Normale Supérieur de Lyon, Lyon, France; INRIA, IRISA, Campus de Beaulieu, Rennes, France; and CAS, Delft University of Technology, Delft, The Netherlands. All of them had been engaged in similar research and continued contributing in the area after the project came to an end in June, 1995. The objectives of the research were twofold. On the one hand, ENSL [63], ESAT and DUT concentrated on the development of novel algorithms for parallel architectures. On the other hand, INRIA, IMEC and also DUT focused on design methodologies and synthesis techniques for such architectures. Here, we give a concise view on the partner's contributions in the design methodology domain, leaving the algorithmic development approaches - albeit strongly influenced by and affecting in turn the design methodology concepts - undiscussed.

INRIA focussed on the design of fully pipelined hard-wired regular arrays. Their design methodology is based on an original technique contributing to the synthesis of 'systolic

arrays', proposed in 1983 [58] and is called dependence mapping or space-time mapping. The INRIA partner developed a transformational design environment called Alpha aiming at the systematization of the design and synthesis of complex ASICs [1] [87] [59] [16].

IMEC concentrated on fully customized regular arrays, with the emphasis on efficient exploitation of ASIC features in generalized regular arrays intended for front-end image and video processing applications. Specific contributions were related to control, initialization, high-level memory, I/O management and verification [65] issues. The IMEC partner developed the Cathedral-4 design system, aiming at the systematization of the design of such arrays, in particular the details related to the mentioned issues [1] [86].

TU Delft aimed at deriving parallel architectures from regular data-flow graphs obtained from executable specifications of nested loop algorithms. This dissertation reports on some of the work contributed here. The DUT partner developed the prototype environment HiFi, which contains a tool box for the systematic design of piecewise regular algorithms and architectures [1].

Surely, other researchers have contributed in this area, too. Among them, we mention Y.H. Hu (University of Wisconsin, Madison), S. Rajopadhy and D. Wilde (Oregon State University), L. Thiele (University of Saarbrucken, Germany), P. Capello (University of California, Santa Barbara), D. Baltus (MIT, Massachusetts, Cambridge), W.B. Burleson (University of Massachusetts, Amherst), R.M. Owens and M.J. Irwin (The Pennsylvania State University), J.A.B. Fortes and L. Jamieson (Perdue University, W. Lafayette), S.Y. Kung (Princeton University), M.W. Wolfe and M.S. Lam (Stanford University), J.-M. Delosme (Yale University, New Haven), T. Meng (Stanford University), U. Banerjee (Control Data Corporation, Sunnyvale), J. van Meerbergen (Philips Research, The Netherlands), and others.

Other design systems that aim at systematic designing of piecewise regular architectures are DESCARTES [7], and CASPER [73]

It is worth mentioning that one novelty in the approach of the NANA consortium is the successful strategy to design the algorithm and the architecture together. In this approach, an implementation is derived from a behavioral specification - usually in the form of an executable program - through a series of transformations on flow graphs specifying and describing both the algorithm and the architecture. This is only possible for a specific problem or at most a group of similar, coherent or otherwise related problems. These are usually called application specific or dedicated algorithm-architecture pairs.

The design environments mentioned above can be situated above the behavioral synthesis systems or as special domains within behavioral design systems. They share the following main ingredients:

1. representation models of the algorithm and of the architecture

2. basic transformation methods and tools supporting specific stages in the mapping

process from the original algorithm to the final architecture.

3. one or more synthesis scripts along architectural design methodology.

This dissertation contributes to the first-mentioned point in the context of the HiFi system, i.e. the consistent modeling of algorithms and architectures. The design systems Cathedral-4 and Alpha require an algorithm to be entered in the form of an applicative specification written in special design languages. Cathedral's design language is called Silage. Alpha-du-Centaur's design language is called Alpha, in which the algorithm is expressed as a set of affine recurrence equations. These applicative specifications express the parallelism in the algorithms.

The HiFi system has another approach and allows an algorithm to be written in a standard imperative language like C, and then to derive automatically the applicative specification from the imperative program description. More formally, the functional behavior given in the form of an algorithm is refined structurally. The resulting specification is then expressed in the HiFi design model, which is basically a *dependence graph* model. The dependence graph expresses explicitly the parallelism. The model is described below in more detail.

We can characterize the HiFi system by its model and methods or tools, although there are a lot of other aspects of the design system, software engineering aspects, such as its object-oriented database. In the following sections, we describe briefly the model and summarize the tools in HiFi system's toolbox.

1.1 Algorithm and Architecture model

1.1.1 The model

However complex an algorithm may be, it ultimately has to be executed by a collection of primitive units, which HiFi considers to be finite-state machines. We model a finite-state machine as an *AST node* [5]. This architectural model was inspired by the concept of applicative state transitions (ASTs) as published by Backus [6], and, in acknowledgement of that fact, we have named our node accordingly.

The capabilities of a primitive unit are specified by a number of behavior descriptions corresponding to each of its states. The behavior can change from state to state (e.g., an ALU may in one state execute a logic function and, in another an arithmetic operation). Therefore, we say that an AST node has a temporal behavior.

HiFi looks upon each state of a primitive unit that carries out a specific behavior as a black box with a number of input and output ports. Its behavior is defined by a relation between the data supplied at its input ports and the expected data at its output ports.

If we do this for all possible input values, the behavior, as it reveals itself to the outside world, is completely specified and is impartial to whatever realization form it may take.

In many cases, an explicit specification of the behavior in the form of an input-output table is infeasible. HiFi allows an implicit specification by means of a function, whose structure is not relevant to the ultimate architecture.

In fact, the AST node is looked upon as a model for a sequential processor. It consists of

- a set of functions that constitute the operational capabilities of the processor.
- a set of states that keep track of the memory inside the system.
- a selection mechanism that specifies the sequential ordering of function evocation and evaluation.

Thus the AST model logically separates function, state, and function ordering.

A second type of node in HiFi expresses parallelism and is called a *structure node*. It is a network of nodes, of either type, in which ports of nodes are connected by edges. The nodes in such a network are concurrent processes that are only aware of their local state and communicate asynchronously with each other over edges. The communication between the AST nodes of such a network is akin of the model of communicating sequential processes (CSP) [80]. In the context of CSP, communication is seen as a shared event between two subprocesses. This means that the production of data in one AST process is synchronized with the consumption of data in another AST.

We refer to a structure node as a functional data-flow network. [1] Nodes represent operations and the edges represent the transfer of values between the nodes. A characteristic property of data-flow networks is that each input port has precisely one edge attached to it, whereas the number of edges on an output port is left free. Data-flow networks distinguish themselves from pure data-flow graphs by the fact that the nodes may have dynamic behavior. That is to say, the function executed by an AST node may change from state to state. However, in a particular state there can only be one function active. This function is called the *current* function. A current function of an AST node fires when all its inputs contain a token and its outputs are free of tokens. After firing, the input tokens are taken away and one output token is placed on its output ports [80].

In [45], E. Lee discusses several models: dynamic data-flow (DDF), synchronous data-flow (SDF), boolean data-flow (BDF), etc. To position the HiFi model, we could say that its single token passing is a special case of BDF.

[1] We could also call them signal flow graphs or data flow graphs. Intuitively, graphs are more abstract and static than networks which are closer to technical dynamical structures.

1.1.2 Regular Dependence Graph model

So far, the functional data-flow network may appear to be closer to an architecture than an algorithm. This is at least suggested by the temporal behavior of the AST node. However, we do use the same model to represent algorithms as well. For this purpose, we assume that the algorithm is statically specified, that is, if it is given in the form of a program (an executable specification), then this program must be written in a single assignment code. Assuming, in addition, that all variable dependencies are manifest (known at compile time), the single assignment program has a graphical representation which is a special form of a data-flow network and is commonly called a (data) dependence graph (DG). A DG is special because each and every node in the graph evaluates one, and only one, function exactly once during the evaluation of the algorithm.
Now, recalling that our main interest is the designing of algorithm/architecture pairs for nested loop type problem specifications, the DG for such an algorithm has the particular property that it reveals a high degree of homogeneity, that is, it is systolic-like. To be even more specific, we shall be dealing with so-called piecewise homogeneous dependence graphs and corresponding piecewise regular flow graphs [76] which are indexed graphs within so-called linearly bounded lattice [78]. Broadly speaking, a linear bounded lattice (LBL) is characterized by a lattice and a polytope [66]. An indexed graph having an LBL support is homogeneous (or regular) if the specification of a node and its dependencies is independent of the index on the support. A graph is piecewise homogeneous (or regular) if it can be partitioned in a finite number of LBLs, independent of the size of the graph, and the convex hull of the set of individual polytopes is again a polytope.
The LBL DG model in HiFi can be compared to the Polyhedral DG model in Cathedral and the affine recurrence equations model in Alpha. The piecewise regular DG is super imposed on the data-flow model and has the advantage that an explicit timing model is part of the specification avoiding an overloading of semantics in a pure applicative specification. Of course, a real-life application never appears as a pure piecewise regular data-flow graph, there will always be irregular parts in it. Interactions of regular and irregular parts in flow graphs may influence efficiency of implementations drastically. It is, therefore, necessary to provide models for interfacing and interaction of regular and irregular parts in flow graphs, even when irregular parts are not developed or designed within the system (or the domain of the system) wherein the regular parts are. Part of the work described in this dissertation has been dealing with this interfacing problem.

Thus, the DG model allows us to express the functionality and the potential parallelism of the algorithm in detail. Simultaneously, the model represents a possible architecture, although it is unlikely that this architecture meets the requirements of the specification of the application. In addition, the model allows one to analyze and inspect properties of the algorithm/architecture. One can get good estimates of the expected computation time, speed-up, etc. and various costs of the architectures that may be derived from the DG. Based on the result of such an analysis, one may choose for an another algorithm, or one may modify the algorithm by algorithmic transformations [84].

1.2 Design trajectory

Within the whole spectrum of possible architectures, we can think of two conceptual ones at either sides of the spectrum to execute an algorithm specified by a DG. At the one end, would be the architecture based on a flow graph network (a structure node) obtained by the direct mapping of each node of the DG onto an AST node and of each dependency of the DG onto an edge. This corresponds to full parallel execution of the algorithm. In HiFi, an algorithm is actually entered as such an architecture (a DG). This is feasible because we focus on regular architectures that can be described in a reduced way. At the other end, is the architecture consisting of a single AST node that has a state for each node of the DG it originates from. This corresponds to sequential execution of the algorithm. This architecture is very costly in the number of time steps needed to execute the algorithm. The former one is expensive in number of distinct functions needed for that execution. The objective of the HiFi design trajectory is to derive an architecture between these two extreme points which is optimal in the sense that a cost function, in which the various parameters (including execution time, implementation volume) can be given weights, is minimized. That is, a continuous trade-off can be made between time, space and memory needed to execute the algorithm on an implementation. For example, a processor array will be designed in such a way that its size is independent of the "size" of the algorithm, the processors are optimally used, and its throughput rate is balanced with the I/O speed of the host processor.

1.3 The Tools

The tools in the HiFi tool box can be divided into three categories. The first set of tools is concerned with converting user specifications which do not fit the imposed model into such specifications. The second set of tools consists of transformation tools that operate on piecewise regular DGs. Finally, there is the set of support tools among which a data-flow simulator. However, we will not discuss the latter set of tools, as they are described elsewhere [80].

1.3.1 Conversion Tools

We distinguish between three types of algorithm specifications:

- **functional input-output** specification. This is an input-output map or a relation between an input and an output that must be a function. [2] As it is at present

[2]The requirement that the relation must be a function (in the mathematical meaning of the word) is important. If an input-output map cannot be specified as a function, then the specification is formally undefined for the HiFi system.

difficult, if not impossible, to design implementations directly from input-output function specifications, we allow such specifications to be replaced by executable program versions written in a subset of the C or MATLAB programming language [74].

- **temporal** specification. By temporal specification we mean a specification which is a valid AST node within the system. An AST node is a particular model for a sequential program in which there is an explicit sequencing mechanism (control function) that determines the sequence in which the functional units are to be evaluated. The functional units themselves are again executable routines, written in a subset of C or MATLAB, that must have a corresponding functional input-output specification.

- **structural** specification. This is a specification expressed as a dependence graph, that is, a structure node. In it, the leaf nodes must be AST nodes.

These three types of specifications are depicted in figure 1.1 together with their mutual relationships. In this figure, the functional input-output specification is labeled NLP (nested loop program). The central circle, labeled SAP, is the single assignment code version of the NLP specification. The SAP specification must exist as the design of an architecture for the implementation of an NLP always starts out from this specification. However, the HiFi system does not require a SAP specification to be given. Indeed, one of the tools of the system is a data dependence analysis tool which allow conversion of NLPs into SAPs.

Although a design session can start from anyone of these three types of specifications, it will often be the first one that is provided from outside. Assuming that this is the case, the first step will, then, be the procedural algorithm description to be converted to the DG model in which all parallelism is now fully explicit. A large part of this dissertation is devoted to the tool *HiPars* by which this conversion can be done automatically for the defined class of nested loop programs (NLPs). Again, characteristic of this class of program is that they have static control. Many signal-processing algorithms belong to this class. The data dependence analysis involved is exact and detailed down to the level of the iterations of the nested loop program. The output of *HiPars* is a single assignment program (SAP). We convert it with the tool *sap2dg*, see chapter 9, into the dependence graph model.
The design process itself can be characterized as a process in which one has to decide which part of the algorithm has to be processed sequentially (software) and which part of the algorithm has to be processed in parallel (hardware). Or in terms of the model, which part is to be specified by an AST-node and which part by a structure node (DGs) and how the parts are interfaced. The design process is thus a refinement process in which we decompose the initial algorithmic specification, with two fundamental refinement directions: (1) temporally and (2) structurally.

In the dissertation, we present the models and methods for supporting this refinement process. In table 1.1 we have listed several tools that we will discuss. The tools are conversion tools which convert one type of specification into another.

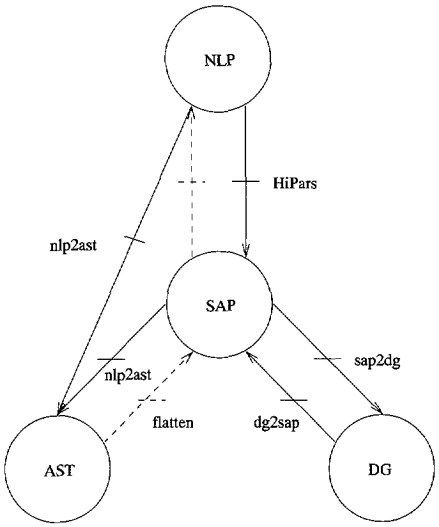

Figure 1.1: The three types of specification and their conversion. The solid lines are the tools we discuss in the dissertation.

We restrict ourselves primarily to discussing the techniques used by these tools and the way the types of specification are modeled. The choices about when and on which object to apply the tools depend to a large extent on the properties of the algorithm and the specification of the system. It is the task of the designer to make these choices based on thorough understanding of the problem.

input	tool	output
nested loop program (NLP)	HiPars	single assignment program (SAP)
single assignment program (SAP)	sap2dg	dependence graph (DG)
dependence graph (DG)	dg2sap	single assignment program (SAP)
nested loop program (NLP)	nlp2AST	applicative state transition (AST)

Table 1.1: The table of conversion tools with type of specification at input and output side.

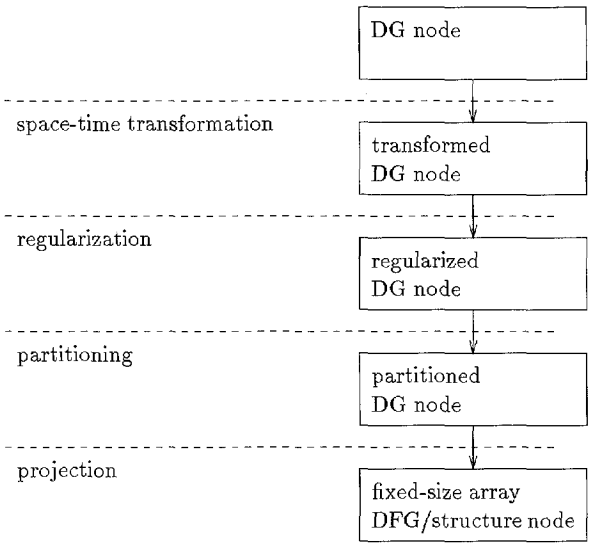

Figure 1.2: The HiFi design trajectory for piecewise regular DGs.

1.3.2 Transformation tools

The DG forms a source node in what could be called a design tree. It is a design trajectory with the objective of deriving a AST/Structure co-design architecture for the execution of piecewise homogeneous concurrent algorithms. The trajectory is schematically depicted in figure 1.2 and consists of a sequence of transformations which typically, but not necessary, take the piecewise regular DG model input to the transformation sequence. In the DG model, the elements of the network such as ports, edges and nodes are indexed and grouped in index-domains, which are defined by linearly bounded lattices [78]. The nodes of the graph are arranged into segments. Each segment is specified by an index domain and a function specifying the operation of the nodes. The edges between the ports of the nodes are defined by an index relation between indexed input and output ports of nodes. We will explain this in chapter 9.

The transformation tools are index transformations. They rearrange the elements by assigning new indices to the elements. The tools are closed operations in the sense that the result is again a DG, although a modified one.
The most important transformations are: space-time transformation, localization, regularization, partitioning and projection. Localization is applied to substitute local, propagating communication for long-distance and broadcast communication in the DG. Communication is an important cost factor in parallel processing. By choosing for regular and local

interconnects, this cost can be reduced considerably. The space-time transformation introduces space and time dimensions in the DG. The objective of regularization is to combine several regular pieces into one regular piece that is controlled by external controllers. The partitioning tool is used to partition the DG, with large and often parameterized size, into a two-level hierarchical graph of which the lower level graph (the tile) is of fixed size. The projection tool is to re-use nodes of the DG by projecting the space-time transformed DG onto a data-flow graph with memory.
As a DG is presumably piecewise regular, the tools may be applied on each and every piece separately and independently provided the whole is a feasible graph in the end. For example, overall causality constraints must be preserved. Thus piecewise linear transformations are feasible transformations and may lead to mappings that are more optimal than with global transformations [8][61]. The application of transformations on individual pieces raises the question of how to model the interfaces between pieces. This crucial and delicate issue has been disregarded too often which is one of the reasons, if not the reason, why so many 'designs' are either but conceptual or inefficient. It is not hard to imagine a DG consisting of, say three, pieces, one of which is transformed to a regular one-dimensional data-flow array, another one being mapped onto a single AST node whose behavior is implemented onto a programmable DSP, and the last one being clustered onto another AST node whose behavior is modeled by an irregular flow graph which is implemented in an ASIC. In this 'design' interfaces between three constituents will determine, for a great deal, the efficiency of the regular-irregular software-hardware implementation.
In this dissertation we will pay attention to the modeling of interfaces between the various nodes that we know: AST-nodes, structure nodes including dependence graphs and being of type regular or irregular.

1.4 The output

Eventually, we end up with a possibly hierarchical network of AST nodes, i.e. the system is expressed as a set of functions structured in time and space.
What is most important is that all network's nodes are derived from an initial functional algorithm specification by a series of algebraic transformations such that the constructed network can be shown to be input-output equivalent to the initial specification.
Even when the initial algorithm is regular, the resulting network will not be purely regular in the sense that all the nodes perform the same function/operation as is the case in pure systolic arrays. The transformations will introduce special AST-nodes to take care of the input and output distribution of data, memory buffers, and switches. The nodes make explicit and take care of additional emerging synchronization needs. One must be careful to avoid performance degradation due to the appearance of such nodes. Optimization is mandatory here in order to keep space-time product efficiency as close to a constant as possible, irrespective of transformations that are applied to turn an initial specification into an implementable data-flow graph.

The functions that are executed in the states of the AST nodes of the final network will be executed either in software or on hardware. In case these functions are not further refined, that is of the software/hardware support is not specified, then they are referred to as *elementary* functions. For example, table 1.2 lists a set of typical elementary functions. This set of elementary functions is available in a MATH library.

name	expression
cos	$y = \cos(x)$
sin	$y = \sin(x)$
tan	$y = \tan(x)$
add	$y = a + b$
sub	$y = a - b$
mul	$y = a.b$
div	$y = a/b$
floor	$y = \lfloor x \rfloor$
ceil	$y = \lceil x \rceil$
mod	$y = ab$

Table 1.2: Table of supported elementary functions.

Finally, some remarks on the realization. Although the output of our system is a network of communicating AST-nodes, whose internal functions are elementary, the HiFi model was designed with software-hardware implementations in mind.
This, of course, means that there must be an interface between the HiFi design system and, if not HiFi, another design system in which the network can be further refined to take it down to the implementation level. This system may even be an intermediate system requiring again an interface to yet another system. For example, we might see HiFi as a domain for the Ptolemy system [14][45], or might wish to link the HiFi system to any one of the currently available behavioral synthesis or high-level synthesis systems. The difficulty here is that all those systems, including HiFi, are far from compatible and VHDL [60] is often the only language (originally a hardware description language, now extended with some specification capabilities) through which various design systems can interact with each other. Anyhow, for the downstream to be unblocked, it is necessary that the elementary functions in HiFi's function library can be interpreted by the satellite design system in the sense that they are likewise in the operating system's library or can be refined in terms of components in its library. This implies that the granularity of the elementary functions must be fine enough so as to be not coarser than the coarsest objects known to the adopting design system. Thus if a communication protocol is to be implemented by a four or two phase handshake protocol [33], then the HiFi single token passing mechanism must be so refined that such a protocol can be extracted from it. Similarly, if an AST is to be implemented by a finite-state machine, HiFi must not output AST specifications which are too far away from FSMs. Finally, HiFi functions can be synthesized to circuits by means of existing synthesis tools such as DSP station [20]. The input language of of the

DSP station is called *data flow language* (DFL), which is based on the SILAGE language with some procedural extensions. By using DSP station, we can automatically translate DFL descriptions of functions to register transfer level, which in turn can be synthesized into digital circuits. An alternative to designing ICs, is to use standard digital signal processors (DSP). The system has also compilers for DSP processors such as the 'C30' and 'C40' [20].

1.5 The HiFi system

The development of the HiFi system has been a large software engineering enterprise. It includes the development of a data structure to represent the dependence and flow graph model, and the implementation of several transformation tools. Some of these are described in this dissertation either completely or partially. Other tools and the completion of the partially described ones can be found elsewhere [89].

We have set up the system conformable to some general guidelines. Firstly, we have aimed at an open design system in the sense that it is under the control of the designer who supervises, gears, and steers the path to follow, makes the design decisions. Secondly, we believe that designing is an interactive process between the system and the designer. We have implemented various user interfaces to assist the designer, for instance, to invoke the tools and to select their parameters.

We have chosen to build a toolbox being a collection of relatively small tools instead of large tools by which large design steps have to be taken in which fine-grain decisions have been automated. The tools are written in an object oriented language, in our case Objective C. In fact, the whole system has a strong object oriented flavor [68]. Thus nodes, whether of type AST or of type structure node are objects, having methods and data encapsulation, and knowing about classes, message passing, polymorphism and inheritance.
The backbone of the tools is formed by a data structure. The data structure consists of a collection of objects, representing the piecewise regular model. It contains objects to represent elements such as ports, nodes, and functions of the network. There are tools for design and separate tools for querying information about the design.

The HiFi system is built on top of the NELSIS CAD Frame [83], which offers a database consisting of a collection of design objects in which the designs are stored. In addition, NELSIS CAD Frame offers a number of features to organize the tools and to present them to the designer in a user-friendly way.

A powerful design concept is hierarchy. The system supports the hierarchical specification of designs. An important reason to use hierarchy is that it reduces the design complexity. The idea is to start with a design object specified by a procedural program in MATLAB, and to refine this specification step by step into a collection of design objects organized in the form of a *design graph*. Each design object captures a description of either a structure,

an AST, or a function. To support hierarchy, the description is split into an interface and a body specification. The hierarchical relationships between design objects are maintained by the NELSIS CAD Frame. The result of the design process is a collection of design objects specifying a network or architecture that can be synthesized.

For more information on the HiFi design system, we refer the reader to [23][37][80].

1.6 Main Contributions

The concepts and models that are pivotal to HiFi are not new. They were introduced in [5][13][39] and used to build a simulator in [80]. These basics emerged from the desire and attempts to conceive a logical and consistent functional design methodology. In it the key paradigm is the flow graph by which both behavior and structure are modeled. There can be found several flow graph models in the literature and in other design systems. Our flow graphs are networks in which functions communicate. Here functions are mathematical functions and have, therefore, precise meaning and properties. The communication is thus essentially a passing of arguments and results. The functions reside in the nodes of the network which, in contrast to most flow graph models, are not function nodes but nodes with function states. They are our AST nodes. Our flow graph thus differs from classical flow graphs, such as the Kahn model [40] (which is sometimes considered to be the mother of all flow graphs) or the various flow graphs in the Ptolemy system [45]. Being consistent in a design methodology is one thing, completeness or closedness is more of a though issue. If the ultimate goal is to produce efficient implementations of algorithms, then genericity is hard to sustain. Flow graphs may have certain specific properties - they may be annotated, they may be partially pre-structured or even indexed - which provoke certain specific approaches which exploit such properties for efficiency sake. If specific properties are neglected, then the result may be unsatisfactory. This is why most of the design systems are either biased toward application typical designs and design methodologies, or get stuck at the level of genuine simulation platform in some generic sense.
As a result, we have concentrated on a particular application domain within the originally generically thought HiFi system. The domain we have chosen is the one which is missing in most design systems, that is, the domain of regular flow graphs. When looking at the few competing sides, then, it is seen that the algorithms - homogeneous algorithms - that lend to such flow graphs, are assumed to be specified in terms of sets of recurrence equations. As this is, practically speaking, a severe handicap, we have decided to look, in the first place, closer to the problem of how to specify the algorithm or program for which a design is to be generated. This is how we arrived at the subclass of nested loop algorithms which we will introduce in chapter 3. Thus, if an initial specification is a program, then it will have to belong to a well defined class and from there on we provide transformations and refinements of the given specification.

The design methodology is, indeed, tuned to the domain of homogeneous algorithms and regular architectures, yet it is fully consistent with the realm of the original concepts and models.

The main contributions of the dissertation are:

- definition and implementation of the piecewise regular dependence graph model based on the data-flow model.
- the tool, *HiPars*, that analysis the data dependencies of static nested loop programs and outputs a functional equivalent program in single assignment form.
- the tool, *sap2dg*, that converts the single assignment program into the piecewise regular DG model.
- the tool, *HiCompose*, that creates a hierarchical or abstract DG.
- the tool, *dg2sap*, that converts pieces of the DG into procedural program descriptions.
- the tool, *nlp2ast*, that models a nested loop program as an AST.
- organization of the tools by the NELSIS CAD Frame.
- organization of the design in the form of a design graph inside the database of the NELSIS CAD frame.

It is obvious that, with these contributions, no sufficiently powerful toolbox has yet been constructed. However, other tools - that we alluded to earlier - have been built by colleagues in the HiFi context. They are space-time partitioning, graph tiling, clustering, projection, and control generation tools. They will be presented elsewhere [89]. With these two sets of tools, realistic designs can be obtained in a genuine functional way.

1.7 A design example

In this section, we briefly present an example of a real-life design. Its details can be found in [54].
The underlying problem is the following. A signal of interest (SI) is sent from some location in space and to be received at another location. The receiver has thus to point his receiving antenna (A_0) in the propagation direction of the signal (which is a plane wave at the receiver's site). See figure 1.3.
In practice, however, the receiving antenna will also pick up jammer signals (J) (again plane waves). So, some 'focusing' and 'rejecting' mechanism has to be conceived to separate SI from J. This is achieved through beam forming [75]. Thus, a set of $p-1$ auxiliary antennas are so placed that they essentially receive the jammers.

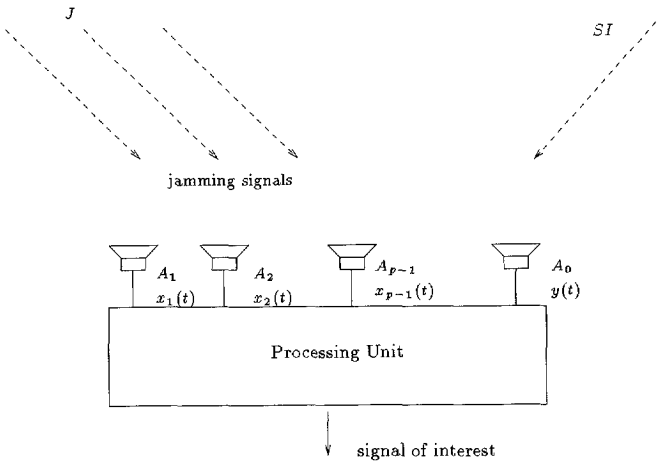

Figure 1.3: The beamforming problem. Signals from several sources are impinging on an antenna array. SI is the signal of interest. J is a jammer signal. A_0 is the main antenna and $A_1, ..., A_{p-1}$ are the auxiliary antennas.

Let $y(t)$ be the (baseband) signal at the output of the main antenna A_0. Let for $i = 1, ..., (p-1)$, $x_i(t)$ be the (baseband) signals at the outputs of the auxiliary antennas, $A_1, ..., A_{p+1}$, respectively. All these signals are assumed to be complex valued. Define the difference signal $e(t)$ as

$$e(t) = y(t) - \Sigma_{i=1}^{p-1} w_i(t) x_i(t)$$

where $w_i(t)$ are coefficients used to weigh the auxiliary signals. The task, now, is to find the $w_i(t)$ such that $\Sigma_k \mid e(t = kT) \mid^2$ is minimal, where T is a signal sampling period and k runs over a finite time span.

The implementation is in the form of a PCB board bearing four pipelined CORDIC chips [34] and five programmable gate arrays. This board is a prototyping testboard and other algorithms can be implemented in the same configuration. A single chip implementation of the beamformer is feasible and could easily sustain a throughput of the vectors $[y(t), x_1(t), ..., x_{p-1}(t)]$ of 200.000 per second with a chip clock rate of 40 MHz.

The design of this architecture went through the following steps, inside HiFi:

- A specification of the beamforming algorithm was given as a nested loop MATLAB program (The program resembles the one given as program 3.2.3 in chapter 3).
- HiFi's tool *HiPars* converted this program to a single assignment program (Chapter 7).

- HiFi'tool *sap2dg* converted the SAP program to a dependence graph (DG) (Chapter 9).
- HiFi's tool *regularize* combined the complex functions vectorize and rotate into a complex CORDIC-AST node.
- HiFi's tool *space − time* transformed the DG to obtain maximal throughput.

Outside HiFi:

- Refinement of the complex CORDIC-AST into an irregular flow graph of interconnected real CORDIC-ASTs. (Interconnected through FPGs).
- Refinement of the real CORDIC-AST into a pipeline CORDIC ASIC.
- Control generation (implementation in XILINX)
- Board connected to data buffer via VME bus.

1.8 Outline

The standard design example in this thesis is the SVD algorithm [31]. In chapter 2, we give an introduction to the SVD algorithm and come up with a nested loop program description of the algorithm, which is suited for parallel implementation.

In the following chapters, we describe the tool *HiPars* [36][35][41], which is the most important of the set of tools discussed.

Given a processing problem, HiPars inputs a procedural algorithmic description of the method chosen for solving it, and outputs a description in 'single assignment form', which is equivalent to a dependence graph. These exhibit the maximal available parallelism in the chosen algorithm. Some main principles of HiPars go back to Bu's work [13]; he analyzed the conversion mechanism to the single assignment form of a restricted class of imperative programs, and Feautrier's work [25][26] who proposed effective methods to solve the search problems involved. These methods can only handle programs with static control. We mean by this that the control variables are known at compile time as parameters, i.e. values that are independent of the actual data being processed. We do not require them to be known constants. The attraction of the method is that it can handle symbols as parameters (parameterized design).

In chapter 3, we define the class of *nested loop programs* that *HiPars* can take as input. We do that by representing programs by *parse trees*. The parse trees define the syntax of the programs but they are also used as the internal data structure of *HiPars*.

In chapter 4, we formulate the data dependency problem. Data dependencies are the result of argument passing between functions via the variables of the nested loop program. The kernel of HiPars is formed by Feautrier's Parametric Integer Programming (PIP) algorithm, which we use to find expressions for the data dependencies inside programs analytically. In chapter 5, we annotate the parse tree with linear inequalities and describe procedures for setting up the input for PIP.

In chapter 6, we explain the Parametric Integer Programming algorithm and describe how a PIP problem differs from a classical LP problem.

The output produced by PIP can directly be written in terms of parse trees. In chapter 7, we will combine the output produced by PIP and construct the parse tree corresponding to the single assignment program, which we write as a procedural program in the MATLAB programming language. The SAP exhibits the precise dependencies between individual operations. The single assignment program is an executable program and is functionally equivalent to the original nested loop program.

In chapter 9, we present the piecewise regular DG model. The DG forms the algebraic object on which the transformation can be applied. The elements of the DG are described by index domains, which we define by linearly bounded lattices. In chapter 8, we describe the relation between linearly bounded lattices and the control structure of the single assignment programs outputted by HiPars. We apply some transformations to parts of the DG of the SVD and obtain thus a virtual array.

In chapter 10, we describe the tool *HiCompose*, which we have implemented to derive so-called hierarchical graphs for a given dependence graph. By applying the concept of abstraction we obtain an overview of the often complex dependence graphs. The number of elements of the hierarchical graph is directly related to the number of regular pieces in it. In this chapter, we also present the DG of the SVD algorithm in the form of an hierarchical graph, of which the nodes are graphs themselves.

Often, it is not realistic to implement the DG directly as an architecture, simply because the number of nodes and edges is too large. To obtain an architecture of reasonable size, we have to merge or cluster parts of the DG into a single computational node, which we specify by a procedural algorithm. For this purpose, we have implemented the tool *dg2sap*, which we describe in chapter 11.

In chapter 12, we discuss how we model procedural algorithms by AST nodes. The nodes of the structures at which we arrive are generally mutually dependent, which implies that the nodes must have have temporal behavior. With the AST, we are able to model this temporal behavior. The resulting networks are thus networks of concurrently operating sequential processes.

Finally, in chapters 13 - 16, we focus on the software engineering aspects of the HiFi system and show how the system is set up and organized.

Chapter 2

Example of Specification: SVD

2.1 Introduction

Signal processing problems can often be represented in the form of a system of equations $Ax = b$ where the number of equations exceeds the number of unknowns and a least squares solution is desired. In this chapter, we present the Singular Value Decomposition algorithm which is an attractive numerical tool to solve such problems. Even if A is square, SVD is attractive if a good control of the numerical accuracy is desired. The matrix A is usually first converted to a square matrix on which subsequently SVD is updated. An application in this sense is the so-called angle of arrival or direction of arrival (DOA).

In practical situations, the matrix A and the vector b are measured quantities, for example snapshots of received signals, and these are usually corrupted by noise, interferences or quantization effects. The SVD, then, can approximately recover clean measures. The numerical properties of the SVD, such as stability and convergence, are well understood [72].

The outline of this chapter is as follows. In section 2.2, we will give a mathematical introduction to the SVD algorithm and introduce the odd-even Jacobi algorithm developed by Luk [47], which is especially suited for parallel implementation. In this dissertation, we will analyze the data dependencies of this algorithm by the tool *HiPars* [37] [22]. HiPars requires a Nested Loop Program to be written in the programming environment MATLAB [74]. In section 2.3, we present the MATLAB program for the odd-even algorithm, which we will use as design example throughout.

A well-known application of the SVD algorithm is in finding the least-squares solution x to $Ax = b$ problem. This is illustrated in section 2.4 by a practical example. In the DOA, the matrix A is build on observation data. In section 2.4, we discuss the roll of SVD in the DOA problem.

2.2 SVD

The singular value decomposition of a real square matrix $A \in \mathbf{R}^{n \times n}$ is a product decomposition, consisting of two orthogonal matrices U and V, and a non-negative diagonal matrix $\sum = diag(\sigma_1, ..., \sigma_n)$ such that:

$$A = U \Sigma V^T \tag{2.1}$$

The σ_i are the singular values of A.

There are several ways to compute the SVD. One of the most famous SVD algorithm is the Golub-Kahan-Reinsch algorithm [31], which first brings the matrix in upper bidiagonal form. We, however, take an algorithm described by Luk in [47], which is better suited for parallel implementation.

Luk's algorithm is based on the classical Jacobi algorithm and computes the SVD of a symmetric matrix A by a series of Jacobi rotations. A Jacobi rotation, denoted $J(p, q, \theta)$, is an orthogonal matrix which equals the identity matrix except for the four entries:

$$\begin{array}{ll} J_{pp} = cos(\theta) & J_{pq} = sin(\theta) \\ J_{qp} = -sin(\theta) & J_{qq} = cos(\theta) \end{array} \tag{2.2}$$

The Jacobi algorithm consists of a sequence of operations of the form:

$$A_{k+1} = J(p, q, \theta)^t A_k J(p, q, \theta) \tag{2.3}$$

where the angle θ is chosen such that the elements a_{pq} and a_{qp} of A_k are annihilated. For this to be accomplished θ must satisfy:

$$tan(2\theta) = \frac{a_{pq} + a_{qp}}{a_{qq} - a_{pp}} \tag{2.4}$$

Let $off(A)$ denote the Frobenius norm of the matrix $A - diag(A)$:

$$off(A) = \sum_{i=1}^{M} \sum_{j=1, j \neq i}^{M} a_{ij}^2 \tag{2.5}$$

Each step will reduce the $off(A_{k+1})$ so that for $k \to \infty, A_k \to \Sigma$.

The order in which the elements are annihilated matters. When the elements are annihilated in a cyclic-by-row order, convergence is quadratic [28].

Program 2.2.1 is the cyclic-by-rows Jacobi algorithm [47]. The program ends when $off(A)$ is smaller than a predefined small number ϵ.

Program 2.2.1 [Jacobi]

```
while ( off(A) > ε ) do
    for p = 1 to n-1
        for q = p+1 to n
            compute θ according to equation 2.4
            A = J(p,q,θ)^t A J(p,q,θ)
        end
    end
end
```

□

For non-symmetric matrices, the algorithm is slightly different and is known as Kogbetliantz' algorithm. In this algorithm the angle in the left-hand side rotation may be different from the angle in the right-hand side rotation. (2.3), then becomes (2.6).

$$A_{k+1} = J_1(p,q,\theta_1)^T A_k J_2(p,q,\theta_2) \tag{2.6}$$

and the angles θ_1 and θ_2 satisfy:

$$\begin{aligned} tan(\theta_1 + \theta_2) &= \frac{a_{pq} + a_{qp}}{a_{qq} - a_{pp}} \\ tan(-\theta_1 + \theta_2) &= \frac{a_{pq} - a_{qp}}{a_{qq} + a_{pp}} \end{aligned} \tag{2.7}$$

It is easy to see that for the symmetric case, $a_{pq} = a_{qp}$, the two angles are identical.

The cyclic Kogbetliantz algorithm is slightly different from the row-cyclic Kogbetliantz algorithm. In the cyclic Kogbetliantz algorithm only elements of the first upper diagonal are annihilated. To guarantee convergence, additional left and right permutations are needed. The permutations are carried out implicitly by shifting the rotation angles by $\pi/2$. This permutation scheme was suggested by Gentleman and is described in [56]. It is easy to show that the order in which the elements are annihilated is equivalent to the cyclic-by-rows ordering scheme [47]. The algorithm converges when the angle θ_2 is constrained to so-called outer rotations: i.e., $\pi/4 \leq \mid \theta_2 \mid \leq \pi/2$ [28].

Program 2.2.2 is the odd-even Kogbetliantz algorithm [47]:

Program 2.2.2 [odd-even]

```
while ( off(A) > ε ) do
    for p = 1 to n-1 step 2
        compute θ1 and θ2 according equation 2.7
        A = J(p, p+1, θ1)^t A J(p, p+1, θ2)
    end
    for p = 2 to n-2 step 2
        compute θ1 and θ2 according equation 2.7
        A = J(p, p+1, θ1)^t A J(p, p+1, θ2)
    end
end
```

□

An example may clarify the difference between the orderings.

Example 2.2.1 [Ordering]

In this example, we illustrate that the odd-even ordering corresponds with the cyclic-by-rows ordering taking a matrix A of order 6. Let (p, q) denote the index of an element of A. In the cyclic-by-rows scheme the elements are annihilated in the following order:

```
(1,2) (1,3) (1,4) (1,5) (1,6)
      (2,3) (2,4) (2,5) (2,6)
            (3,4) (3,5) (3,6)
                  (4,5) (4,6)
                        (5,6)
```

The actual ordering in which entries of the matrix are 'visited' is implicit as the loops sweep only along the first upper diagonal by taking entries $(p, p+1)$. However, due to the permutations, entries further away from this diagonal are driven toward it. Thus, entries are moved toward the diagonal and then picked up whereas in the cyclic-by-rows algorithm, the entries are picked up in-place. To make the actually picked up entry locations visible, choose θ_1 and θ_2 equal to $\pi/2$ so that the rotations matrices J become permutations ($cos(\pi/2) = 0$ and $sin(\pi/2) = 1$). The actual ordering underneath the $(p, p+1)$ ordering in the loops of program 2.2.2 are then:

```
(1,2) (3,4) (5,6) (1,4) (3,6)
(2,4) (1,6) (3,5) (2,6) (1,5)
(4,6) (2,5) (1,3) (4,5) (2,3)
```

We see that the odd-even ordering selects the same elements in a slightly different order. Yet, as far as convergence is concerned, the two programs behave equivalently [48].

□

Note that one sweep (stage) of the odd-even algorithm annihilates elements on the first upper diagonal. This diagonal has $n-1$ elements. The number of off-diagonal elements of an $n \times n$ matrix is $n(n-1)/2$. Thus $n/2$ sweeps in the odd-even algorithm correspond to one cycle in the cyclic-by-rows algorithm.

For some more information on SVD algorithms we refer to the literature [11] [30] [67] [71].

2.3 MATLAB Program

In this section we, give the nested loop program that computes the SVD according to the odd-even Kogbetliantz algorithm [47]. HiPars requires a nested loop program to be written in the MATLAB programming environment, and in a specific way. More specifically, statements or groups of statements in the body of the program appear as functions that are called by the program.

We give the MATLAB program below. First, we describe the functions that the program calls to compute the angles and to perform the row and column rotations. Function `Angle` computes the angles θ_1 and θ_2, see program 2.3.1. If the angle is smaller than $\mid \theta \mid < \pi/4$ we shift it by $\pi/2$ such that the angle becomes an outer rotation and satisfies: $\pi/4 \leq \mid \theta \mid \leq \pi/2$. This implicitly implements Gentleman's permutation scheme [55]. If the sign of the angle θ_1 (`th1` in the function) is negative, $\pi/2$ is added otherwise $\pi/2$ is subtracted.
Function `Angle` constrains angle θ_1. [1]

[1] Convergence has also been proven when the angle θ_2 is constrained instead [28].

Program 2.3.1 [Angle]

```
function [th1, th2] = Angle (A11,A12,A21,A22)

        thsum = atan((A21+A12)/(A22-A11));
        thdiff = atan((A21-A12)/(A22+A11));

        th1= (thsum-thdiff)/2;
        th2= (thdiff+thsum)/2;

        if abs(th1) < (pi/4),
            if sign(th1) == -1,
                th1=th1+(pi/2);
                th2=th2+(pi/2);
            else
                th1=th1+(pi/2);
                th2=th2+(pi/2);
            end
        end
```

□

The computation $J(p, p+1, \theta_1)^T A J(p, p+1, \theta_2)$ essentially operates on entries on the intersection of rows p and $p+1$ and columns p and $p+1$.

We introduce functions `RotRow` and `RotColumn` for the row and column rotations, respectively. See program 2.3.2 and program 2.3.3. We have written the functions in matrix form, showing clearly the row and column rotations. The functions are identical and differ only in their function header.

Program 2.3.4 shows the piece of code for the row rotations. A similar piece of code is needed for the column rotations.

Putting things together, we get MATLAB program 2.3.5 which is a valid executable specification of the SVD of an $\mathtt{M} \times \mathtt{M}$ real matrix A, according to algorithm 2.2.2. We have replaced the convergence test with a for-loop statement whose upper bound depends on a parameter `N`. As the algorithm has a quadratic convergence, the iterations can be stopped after a sufficiently large value of $\mathtt{N} = O(log(\mathtt{M}))$ of stages [47].

Program 2.3.5 starts with computing the angles of rotation for odd-indexed rows after which the corresponding row rotations and column rotations are carried out. Here, the row rotations precede the columns rotations. A vice versa ordering is valid as well. The program goes through the same steps for the even-indexed rows.

Program 2.3.2 [RotRow]

```
function [y1, y2] = RotRow(th,A1,A2)

    V2 = [cos(th) , sin(th); -sin(th) , cos(th)];
    v = [A1, A2] * V2 ;
    y1 = v(1);
    y2 = v(2);
```

□

Program 2.3.3 [RotColumn]

```
function [y1,y2] = RotColumn(A1,A2,th);

    B= [cos(th) , sin(th); -sin(th) , cos(th)];
    v = B'* [A1;A2];
    y1 = v(1);
    y2 = v(2);
```

□

Program 2.3.4 [Row Rotations]

```
for j = 1 :  1 :  M,
   [ a(i,j), a(i+1,j) ] = RotRow(th1(i), a(i,j), a(i+1,j));
end
```

□

Program 2.3.5 [odd-even SVD]
Let N and M be parameters.

```
for stage = 1 : 1 : N,
    for i = 1 : 2 : M-1,
        [th1(i),th2(i)] = Angle(a(i,i),a(i,i+1),a(i+1,i),a(i+1,i+1))
    end
    for i = 1 : 2 : M-1,
        for j = 1 : 1 : M,
            [a(i,j),a(i+1,j)] = RotRow(th1(i),a(i,j),a(i+1,j));
        end
    end
    for i = 1 : 2 : M-1,
        for j = 1 : 1 : M,
            [a(j,i),a(j,i+1)] = RotColumn(a(j,i),a(j,i+1),th2(i));
        end
    end
    for i = 2 : 2 : M-2,
        [th1(i),th2(i)] = Angle(a(i,i),a(i,i+1),a(i+1,i),a(i+1,i+1));
    end
    for i = 2 : 2 : M-2,
        for j = 1 : 1 : M,
            [a(i,j),a(i+1,j)] = RotRow(th1(i),a(i,j),a(i+1,j));
        end
    end
    for i = 2 : 2 : M-2,
        for j = 1 : 1 : M,
            [a(j,i),a(j,i+1)] = RotColumn(a(j,i),a(j,i+1),th2(i));
        end
    end
end
```

□

2.4 Applications

2.4.1 Least Squares solution to $Ax = b$

A well-known application of the SVD is to find the least-squares solution of a system of equations $Ax = b$. We illustrate this by a numerical example.

The SVD is used to compute the pseudo inverse A^+ [72] of A which implicitly projects vector b on the column space of A, yielding 'best fit' $\hat{x}$. This means that, denoting $e = b - Ax$, the solution x minimizes the length of e:

$$\hat{x} = avg\ min||e||_2$$

Let A be an $m \times n$ matrix with SVD $A = U\Sigma V^T$. Let x be a vector of size n and b a vector of size m.

The pseudo inverse of A which yields the least squares solution for the system

$$Ax = b$$

is given by,

$$x = V\Sigma^+ U^T b$$

where Σ^+ is defined as

$$\begin{bmatrix} 1/\sigma_1 & & & \\ & 1/\sigma_2 & & \\ & & \ddots & \\ & & & 0 \\ & & & & 0 \end{bmatrix}$$

when Σ is

$$\begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & 0 \\ & & & & 0 \end{bmatrix}$$

Note that small singular values will cause large singular values in the pseudo inverse. Therefore, for numerical reasons, truncation is required.

Example 2.4.1 [Least Squares]

Let the system of equations $Ax = b$ be given by:

$$\begin{pmatrix} 1 & 5 & 5 & 25 & 25 & 25 \\ 1 & 5 & 10 & 25 & 50 & 100 \\ 1 & 5 & 20 & 25 & 100 & 400 \\ 1 & 10 & 5 & 100 & 50 & 25 \\ 1 & 20 & 5 & 400 & 100 & 25 \\ 1 & 20 & 20 & 400 & 400 & 400 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 16 \\ 36 \\ 76 \\ 36 \\ 76 \\ 361 \end{pmatrix}$$

The SVD of matrix A produces matrices:

$$U = \begin{pmatrix} 0.05 & 0.00 & 0.03 & 0.53 & 0.00 & -0.84 \\ 0.12 & -0.13 & -0.01 & 0.58 & 0.69 & 0.38 \\ 0.37 & -0.69 & -0.59 & -0.06 & -0.13 & -0.04 \\ 0.12 & 0.13 & -0.01 & 0.58 & -0.69 & 0.38 \\ 0.37 & 0.69 & -0.59 & -0.06 & 0.13 & -0.04 \\ 0.83 & 0.00 & 0.53 & -0.14 & 0.00 & -0.02 \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} 832.20 & 0 & 0 & 0 & 0 & 0 \\ 0 & 382.74 & 0 & 0 & 0 & 0 \\ 0 & 0 & 110.32 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9.80 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.95 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.18 \end{pmatrix}$$

$$V^T = \begin{pmatrix} 3.3348 & 3.74 & -2.20 & 8.44 & 2.57 & -42.63 \\ 2.5916 & 2.50 & 0.61 & -23.81 & 36.39 & -106.51 \\ -10.9979 & -6.25 & 10.83 & -185.91 & 141.06 & -341.04 \\ 4.7980 & 5.58 & -5.79 & 50.26 & -38.68 & -95.60 \\ 7.1496 & 9.13 & -9.01 & 184.43 & -233.92 & -286.53 \\ 2.6851 & -2.72 & -5.87 & 141.78 & 41.88 & -677.41 \end{pmatrix}$$

and the solution $x = V\Sigma^+U^T$ is:

$$x = \begin{pmatrix} 1 \\ -1 \\ -1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

□

2.4.2 Direction of arrival

The aim of the direction of arrival (DOA) estimation problem is the determination of the angles of arrival of a number of signals impinging on a sensor array [82].

In the ESPRIT model [64][2], the antennas are arranged in a plane and form pairwise identical sensors. The displacement between the sensors in each pair is constant. We assume that the number d of impinging signals is less than the number of sensor pairs m [82].

Each row of $A \in \mathbf{Z}^{m \times n}$ contains the sampled signal received by the first sensor of the sensor pair associated with that row. We assume that the number of samples n is larger than the number of sensor pairs m. So the matrix has more columns than rows, $n > m$. For a particular time instant t, the following equation holds

$$a_t = Hs_t$$

where a_t is the t-th column of A, H is an $m \times d$ transfer matrix and s_t is the t-th column in the $d \times n$ matrix S whose rows are the sampled signals impinging on the sensor array. The SVD is used here to estimate d, and to provide the column space of H as well as the row space of S. This, however, must be done in the practical situation that the equation $a_t = Hs_t$ is actually of the form

$$\hat{a}_t = Hs_t + n_t$$

where n_t is the t-th column of an $m \times n$ noise matrix.

Thus the receiver in the DOA system makes an estimate of the number of signals by looking at the singular values of the matrix $\hat{A}$. Large singular values correspond with the signals of interest, small singular values are assumed to originate from the noise.

U and V^T are partitioned as $[U_s \mid U_n]$ and $\begin{bmatrix} V_s^T \\ V_n^T \end{bmatrix}$ where the $s-n$ seperation corresponds to a partition of Σ in a block of 'large' singular values and a block of 'small' singular values. For instance, for the matrix A of example 2.4.1 we would take U_1 and V_1^T of rank three. Generally speaking, we thus have

$$\begin{bmatrix} U_s & U_n \end{bmatrix} \begin{bmatrix} \Sigma_s & 0 \\ 0 & \Sigma_n \end{bmatrix} \begin{bmatrix} V_s^T \\ V_n^T \end{bmatrix} = U_s \Sigma_s V_s^T + U_n \Sigma_n V_n^T$$

where the number of columns of U_s is equal to the number of signals d impinging on the array. The set of second sensors in the antenna array provide a second set of equations [82].

$$A' = H\Phi S$$

(or $\hat{A}' = H\Phi S + N'$ in practice), where Φ is a diagonal matrix which contains the directions that have to be estimated. The matrix Φ, hence the directions, can be obtained by solving a generalized eigenvalue problem, see [82].

2.5 Conclusions

In this chapter we have given an example of how algorithms are specified in the *HiFi* environment. The specification $A = U\Sigma V^T$ is an input-output behavior specification, which, for reasons of validation ability, is replaced with an executable specification. This specification is a program written in the MATLAB environment and in a pre-specified way. We have worked out such a specification and chose it an odd-even ordering algorithm proposed by Luk. It is worthwhile to emphasis again that any feasible executable specification can be accepted. Thus, a cyclic-by-row version would also have done. Although this does not imply that all possible implementations for the SVD can be derived from any arbitrary executable specification, it is sometimes possible to transform an executable specification into another one by relying on certain rule-based algorithmic transformations. Such methods could be implemented in tools which might be included in *HiFi*'s transformation toolbox. An example of such algorithmic transformations in the SVD context can be found in [85] where it is shown that a cyclic-by-rows SVD program can be so transformed that an input-output equivalent program results that has a dependence graph which differs in a non-trivial way from the dependence graph of the original one. In fact a whole class of input-output equivalent programs can be so derived and this class contains as a matter of fact the odd-even ordered SVD algorithm which we took as the reference executable specification in this chapter.

Because the variable dependencies of the SVD are complex, we start with the data dependence analysis of a part of the SVD program, i.e., the loop stages responsible for the column and row rotations. This subprogram is still interesting because of the step size greater than one in the loops. The MATLAB functions will not be parsed. The resulting single assignment program is given in chapter 7. In chapter 9, the result is expressed in the dependence graph model. Finally, the complete data dependence analysis of the SVD is presented in chapter 10.

Chapter 3

A Class of Nested Loop Programs

3.1 Introduction

In the introduction, we stated that *HiPars* is a tool that can analyze certain *nested loop programs* and produce a description of the program's data dependencies. In this chapter, we define the class of *nested loop programs* that can be analyzed. We do that in terms of *parse trees*, which are also the internal data structure of *HiPars*.

Parse trees are extensively used in the domain of computer science dealing with syntax rules and language specifications. In that field, a language is written in terms of a grammar [10]. A *grammar* gives us the mechanism for translating a program description into a parse tree. Further, the grammar will check whether the syntax of the program is correct. In this chapter, we define procedures on this parse tree which are used by the tools of the *HiFi* system.

Our main objective is to define the syntax of the class of nested loop program we have in mind. Pieces of such programs are represented by specific types of nodes in the underlying parse tree.

3.2 Nested Loop Programs

To introduce the subject, consider program 3.2.1 which is an executable specification of a convolution of two vectors, a signal x and an impulse response h:

$$\forall P \leq i \leq N : y(i) = \sum_{j=1}^{P-1} x(i-j)h(j)$$

The program consists of two *loop* statements and a *function-call* statement in its loop body.

Program 3.2.1 [Convolution]

Let P and N be parameters such that N > P > 1.
Let muladd be a function computing $y = ax + b$.
Let y, x, and h be real vectors of dimension, N + P − 1, N + P − 1, and P − 1, respectively.

```
for i = P to  N,
  for j =  1 to  P-1,
       [y(i)] = muladd(y(i),x(i-j),h(j));
   end
end
```

□

It is characteristic of nested loop programs that a relatively small number of statements are repeated many times in a lexicographical ordering. Typically, the number of repetitions or iterations in a program depends on the size of the input data. To make a program description independent of the size of the input data, we introduce parameters which stand for their possible size. In program 3.2.1, the number of iterations depends linearly on the parameters P and N. These parameters and the variables in programs have to be properly declared and initialized.

Although we allow programs to be parameterized, the parameters must be manifest, that is, known at compile time. In other words, programs must have static control. Control may not depend on the value of the variables actually processed. Although this is a major restriction, the class of signal processing algorithms having static control is huge. The SVD program, given in the previous chapter, belongs to this class.

We allow the following type of control statements:

- loop statements
- conditional statements

The loop statement forces repetition of the statements inside the loop body a number of times and in a particular order. After each loop the loop *iterator* is incremented with the step size or *stride*. [1]

[1] A step size equal to 1 is commonly not given explicitly in a program. Thus in program 3.2.1, both step sizes are 1.

The conditional statement is used to conditionally execute blocks of statements. It is of the form:
`if` *condition*,
`then`-block of statements
`else`-block of statements
`end`
The condition is a Boolean expression. If the condition is true, the block of statements of the `then` part will be evaluated. If false, the block of statements of the `else` part will be evaluated.

Apart from control statements, a program may consist of *assignment* statements, which take the form of function calls. One or more variables are taken as arguments of the function, and the result of the function is assigned to one or more variables on the left of the assignment sign. We require functions to be mathematical functions. In other words, we assume that there are no side effects.

Function calls allow us to specify programs hierarchically. The function specification itself is not parsed. However, when the body of the function is also expressed as a nested loop program, *HiPars* can analyze this one as well.

Expressions inside loop bounds and inside conditions must be 'affine' (see chapter 5 for a definition) expressions on the program's iterators and parameters (for a precise definition see further).

To illustrate the class of programs that can be written in terms of the statements introduced above, we give two examples.[2]
Program 3.2.2 shows that assignment statements may be placed between loop statements.
Program 3.2.3 illustrates the use of conditional statements. [3]

Program 3.2.2 [Assignment statements]

Let `F1` and `F2` be functions.
Let `a` and `h` be variable arrays.
Let `N` be a parameter.

```
for k = 1 to  N-1,
    for j = k+1 to N,
        [h(k,j)]  = F1(a(k,j),a(k,k));
        for i = k+1 to N,
            [a(i,j)] = F2(a(i,j),a(i,k),h(k,j));
        end
    end
end
```

[2] A program does not have to be a perfect nested loop. It may consists of several loop stages.
[3] The two example programs are derived from the LDU and QR algorithms [31], respectively

□

Program 3.2.3 [Conditional Statements]

Let F1 and F2 be functions.
Let A and phi be variable arrays.
Let M and N be parameters.

```
for i = 1 to N-1,
   for j = i+1 to N,
      for k = i to M,
         if k <= i,
            [A(i,k),A(j,k), phi(i,j)] = F1(A(i,k),A(j,k));
         else
            [A(i,k),A(j,k)] = F2(A(i,k),A(j,k),phi(i,j));
         end
      end
   end
end
```

□

3.2.1 Variables

The memory inside the nested loop programs consists of variables. We assume that variables with different names do not overlap in memory. Again, variables have to be declared and initialized.

Variables are referenced in the function-call statements. In a function call the variables on the right-hand side (RHS) of the assignment statement are read and taken as input arguments. The result of the evaluation of the function is assigned to the variables on the left-hand side (LHS).

We require variables to be of type *array*. A variable array is a collection of variables. Each variable in the array is identified by the name of the array and a unique index.

Let v be a variable array of dimension d. Let $(l_1, .., l_d)$ be an index vector. Then a generic variable of the array is $v(l_1, .., l_d)$. The value of a variable is of a certain data type. Examples of data types are `Float` and `Integer`.

The variables appearing in the program are of the form $v[f(I, P)]$, where I is the vector build on the iterators and P the vector build on the parameters. The function f is called the indexing function. We require the indexing function to be 'affine', see chapter 4, section 4.3, for a definition.

Typically, variables are assigned several times in nested loop programs. Each time a new value is assigned to a variable, the old value will be lost.

3.3 Index Transformation Statements

Programs may contain *transformation* statements by which nonlinear index transformations can be specified, which are often called quasi-linear operators.

Let a stand for a linear expression.
Let b be a positive integer.
The transformation statements that can be accepted by *HiPars* are:

- q = `div(`a`,`b`);`
  with `div` the integer division operator, and b the divisor.
- q = `mod(`a`,`b`);`
  with `mod` the modulo operator, and b the remainder.
- q = `floor(`a`);`
  with `floor` the floor operator, where a may have fractional coefficients.
- q = `ceil(`a`);`
  with `ceil` the ceil operator, where a may have fractional coefficients.
- q = `equal(`a`);`
  with `equal` the identity operator, used to specify linear transformations.

Observe that, apart from the `equal` operator, all the operators are nonlinear operators.

The variable q may be used inside expressions of a program. They are, just as the loop iterators, control variables inside the program. The scope of a control variable is the block of statement in which it is introduced. Expressions may be nested. This means that the expression a of an operator may depend on the control variables defined by other transformation statements.

As said, with these statements we can specify nonlinear index expressions. For example, the index transformation $q = \lfloor (2 * i - j)/3 \rfloor$ is specified by:

```
q = floor(2/3*i-j/3)
```

Remark The appearance of non-linear index transformations does not violate the condition, introduced in the previous subsection, that expressions must be affine. This is clearly illustrated in the following example, program 3.3.1.

Program 3.3.1 [Index Transformation Statements]
Let F1 and F2 be functions.

```
for i = 1 to M step 2,
    for j = 1 to M step 2,
        P4 = mod(i,3);
        if P4==0,
           [a(i,j)] = F1();
        end
        P6 = mod(j,3);
        if P6==0,
           [ ] = F2(a(i,j));
        end
    end
end
```

□

3.3.1 Transformation Operators in Expressions

For ease of programming, we accept that the nonlinear index transformations appear within expressions, including in indexing functions of variables. This violates the earlier affine conditions. However, *HiPars* will automatically restore these conditions.

For instance, we can rewrite program 3.3.1 as program 3.3.2, listed below, with the modulo operators in the conditional expressions.

Program 3.3.2 [Operators in Expressions]
Let F1 and F2 be functions.

```
for i = 1 to M step 2,
    for j = 1 to M step 2,
        if mod(i,3) == 0,
           [a(i,j)] = F1();
        end
        if mod(j,3) == 0,
           [] = F2(a(i,j));
        end
    end
end
```

□

If transformation operators occur in expressions, *HiPars* preprocesses the program to replace transformation operations in expressions by variables that are produced by transformation statements, that is, *HiPars* converts program 3.3.2 to program 3.3.1. Thus,

HiPars substitutes each operator in an expression by a control variable and inserts an appropriate transformation statement defining the control variable just before the expression. After the substitution, affine expressions on the control variables result. For instance, the expression `2*mod(j,3)-1` will be preprocessed to `2*p-1` and an additional transformation statement `p = mod(j,3)`.

3.4 Parse Trees

We represent a nested loop program by a *parse tree*. In this subsection, we introduce some parse tree definitions and notions. We adopt the definitions from [10] which we quote literally, for ease of reference.

We define trees as graphs with special properties.

Definition 3.4.1 [Graph]

A *graph* $G = (V, E)$ consists of a nonempty set V of *nodes* and a set E of *edges* such that each edge corresponds to a unique unordered pair of distinct nodes $\{u, v\}$ and no more than one edge corresponds to $\{u, v\}$. The sets V and E are assumed to be finite.

□

Definition 3.4.2 [Path]

A *path* from node v to node w in a graph is an alternating sequence of nodes and edges $(v_0, e_1, v_1, e_2, v_2, ..., e_n, v_n)$ where $v_0 = v$, $v_n = w$, and each edge e_i joins nodes v_{i-1} and v_i for $i = 1, ..., n$.

□

Two nodes u and v of the graph are connected if a *path* exists between the two nodes. A path is called a *simple* path if there are no repeated edges.

Definition 3.4.3 [Tree]

A *tree* is a graph such that there is a unique simple path between each pair of nodes.

□

If a graph consists of only simple paths, then the graph is a *tree*. Trees are very useful to represent hierarchical data structures and are used quite often to analyzing algorithms [10]. A tree with a unique node designated as the *root*, is called a *rooted tree*. We will denote the root by the symbol @. Henceforth, every tree is a rooted tree. An example of a tree is depicted in figure 3.1.

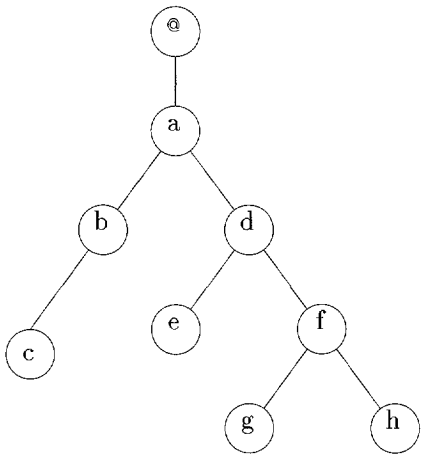

Figure 3.1: Example of a rooted tree.

The *level* of a node is the number of edges in the path to the root. For example, node d is at level 2. We say that we go down the tree when we go to a node with a higher level. When we go to a node with a lower level, we say that we go up the tree.

We say that node v_1 is a *child* of node v_2 if the level of v_1 is one higher than the level of v_2. Node v_2 is the *parent* of v_1 if v_1 is a child of v_2. The *descendants* of a node are all the nodes of higher level that can be reached by a path from this node. The descendants of d in figure 3.1 are the nodes e, f, g, h. The *ancestors* of a node are the nodes in its path to the root. The ancestors of b in figure 3.1 are $@, a$. If two nodes have the same parent, we call them *siblings*. A node with no children is called a *leaf*.

Observe that according to definition 3.4.3 there is a unique path from the root to any node v in the tree. From this follows that each node has exactly one parent.

3.5 Modeling Programs as Parse Trees

We model programs by representing them as parse trees. Nodes of the parse tree represent pieces of the program, such as statements, functions, variables, and parameters. Edges represent logical relations between the nodes according to the structure of the program.

The following types of nodes are used to represent statements:

- a *For-node* to represent a loop statement
- an *If-node* to represent the **then** part of a conditional statement

- an *Else-node* to represent the `else` part of a conditional statement
- a *Function-node* to represent a function in a function-call statement
- an *End-node* to represent an end statement, which indicate the end of a block of statements. [4]

The rules to construct the parse tree are as follows. Each statement of the program is parsed and a node of the appropriate type is introduced. The parent of a statement node is the node representing the control statement of the block of statements in which the statement appears. When there is no control statement, the node becomes a child of the root. After processing all the statements in this manner, we have obtained the parse tree.

Observe that the level of a node corresponds with the nesting level at which the statement appears. Functions are represented by Function-nodes, which are leaves of the tree.[5] The nodes in the path from a Function-node to the root represent the active control statements for the function.

According to these rules, *HiPars* derives parse trees from nested loop programs. Figure 3.2 shows the parse tree derived from the SVD program 2.3.5. The `stage` loop statement is the first statement and becomes a child of the root. Its loop body consists of six loop stages. Two loop stages for computing the angles, and four loop stages for carrying out the rotations.

The parse tree reveals precisely the control structure of the program. For instance, Function-node `RotRow` has as ancestors two For-nodes that are different from the ancestors of function-node `RotColumn`, but share the For-node `stage`. This resembles exactly the control structure of the program, in which function `RotRow` is nested in other loops than function `RotColumn`, but they have in common the loop statement with iterator `stage`.

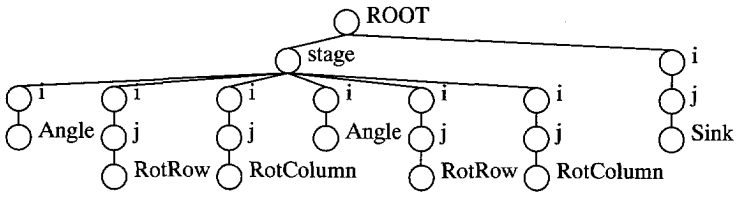

Figure 3.2: The parse tree of the SVD program.

[4]End statements are modeled by End-nodes in parse trees. However, we do not draw End-nodes in figures of parse trees as they are implicitly defined by the tree structure.

[5]If variables that are used in functions are considered explicitly than they will become leaves. See next section.

Expressions in a program must be linear expressions in which only the operators *addition*, *multiplication* and *subtraction* are allowed. In chapter 5, we precisely formulate the type of expressions that can be used. Expressions are represented by the nodes corresponding to the parts of the program in which the expressions occur. This means that For-nodes model the lower and upper bound expressions, If-nodes model the conditional expressions, etc.

3.5.1 Variable-nodes

We represent each occurrence of a variable in the program by a *Variable-node*. To make a distinction between the variables on the left-hand side (LHS) and those on the right-hand side (RHS) in a function-call statement, we introduce the *LHS-node* and the *RHS-node*. They are siblings and children of a Function-node.

A Variables-node has as parent node either a RHS-node or a LHS-node, depending on the position of the variable in the function-call statement. Figure 3.3 shows a subtree modeling a function-call statement.

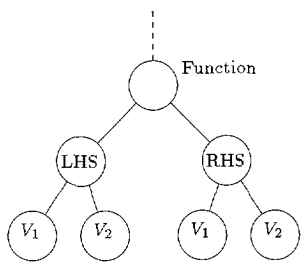

Figure 3.3: Tree structure of a function-call statement. Variable-nodes are children of LHS/RHS-nodes.

3.5.2 Parameter-nodes

In order to use *HiPars*, we have to declare the parameters used in the program by means of declaration statements, which are of the form:

```
%parameter NAME lb ub ;
```

In each declaration statement, we declare the name of a parameter and its range which is specified by two integer constants `lb` and `ub`, with `lb` the lower bound and `ub` the upper

bound. We represent a parameter by a *Parameter-node* which is inserted as child of the root of the parse tree. Figure 3.4 shows the root of a tree with parameter nodes as children.

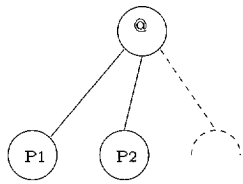

Figure 3.4: Parameter nodes are children of the root node.

To summarize, we have introduced so far the following types of nodes:

Types of Nodes
For-node
If-node
Else-node
End-node
Function-node
RHS-node
LHS-node
Variable-node
Parameter-node

3.5.3 Index Transformation Statement nodes

We model transformation statements by employing the following types of nodes:

- DIV-node
- MOD-node
- FLOOR-node
- CEIL-node
- EQUAL-node

Transformation nodes are control nodes. This means that *HiPars* treats them as such in the construction of a parse tree. Typically, they are internal nodes of the parse tree. A control node represents a control variable and its definition. The scope of the control

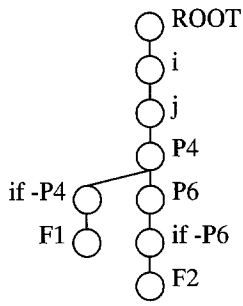

Figure 3.5: The parse tree of program 3.3.1.

variable is the block of statements in which it is declared. The parse tree of program 3.3.1 containing two modulo operations is shown in figure 3.5. The nodes labeled $P4$ and $P6$ are the nodes that represent the modulo operators. Observe that control node $P6$ appears only in the control path of Function-node $F2$.

3.6 Conclusion

We have defined the subclass of programs that *HiPars* can take as input. Many signal-processing algorithms can be expressed by programs that belong to this subclass. The control of these programs must be static but may depend on manifest size parameters. Expressions inside programs must - at least implicitly - be affine, which is required because of the method *HiPars* uses to find the data dependencies. Thus to support more complex index expressions we allow non-linear index transformations, which are interpreted by *HiPars* in such a way that expressions become explicitly affine; see chapter 5. Programs are written in a procedural programming language in the MATLAB environment.

By insisting on the use of function calls, programs are hierarchical. *HiPars* does not automatically parse the body of a function. However, when a function is also specified by a nested loop program, *HiPars* can analyze the body of the function as well.

Programs have been represented as parse trees. The structure of a parse tree resembles clearly the control structure of a nested loop program. Typically, the internal nodes of the parse tree are the nodes representing the control statements and the leaves are subtrees, consisting of function and variable nodes representing the function-call statements. The parse tree forms the internal data structure of *HiPars*.

Chapter 4

Data Dependencies

4.1 Introduction

In nested loop programs the order of execution of functions is sequential and is completely determined by the program's control statements.

The given sequential ordering is not the only possible ordering in which the functions can possibly be evaluated. There may be, and in most cases there actually are, other orderings of the functions for which the algorithm has the same input-output behavior. In particular, functions are ready to evaluate and could do so as soon as their arguments are available. Arguments are passed between functions via the variables. Argument passing imposes precedence relations on the functions and define a partial order. We call these precedence relations *data dependencies.*[1] Once the data dependencies are known, the parallelism inside the program can be exploited.

The outline of this chapter is as follows. In section 4.2, we describe formally the sequential order in which the functions are evaluated in nested loop programs. In section 4.3, we give a formal definition of the notion of a data dependency. Finally, in section 4.4, we formulate the data dependency problem and outlining the procedure to solve it.

4.2 Lexicographical Ordering

We define the ordering of functions through the ordering of the iterations at which they are evaluated and their textual positions in the program.

The ordering of the iterations is determined by the loop statements and is called *lexico-*

[1] In general data dependency may include data-dependent conditions. As we do not allow such dependencies to occur, confusion is excluded.

graphical order [9]. We denote the *lexicographical order* by $\prec$. The vector consisting of the control variables, typically loop iterators, is called *iteration-vector*. A particular instance of an iteration-vector is called an iteration. The set of iterations is called the index or iteration space.

Let $I = (i_1, ..., i_n)^T$ and $J = (j_1, ..., j_n)^T$ be two iterations in the same index space. The relation $I \prec J$, means that iteration I precedes iteration J.
Formally,

$$(\forall 1 \leq q \leq p : i_q = j_q) \wedge i_{p+1} < j_{p+1} \Rightarrow I \prec J \tag{4.1}$$

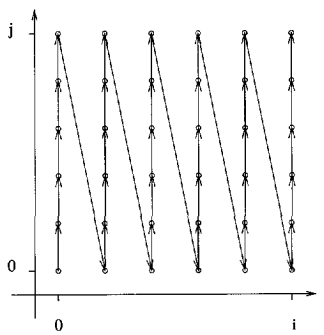

Figure 4.1: The lexicographic ordering of the iterations.

Example 4.2.1 [Lexicographic Ordering]

Consider the nested loop program 4.2.1. The iteration-vector of this program is $I = (i, j)^T$. Figure 4.1 shows the lexicographical ordering in which the iteration-vector passes through all the iterations. Just follow the arrows starting at $(0, 0)$.

□

Program 4.2.1 [Lexicographic Ordering]

Let F1 and F2 be arbitrary functions.
Let a and b be array variables.

```
for i = 1 to 5,
    for j = 1 to 5,
          [a(i,j)] = F1(a(i,j));
          [b(i,j)] = F2(b(i,j));
    end
end
```

□

When functions are called within a single loop body, the iterations at which they are evaluated can be equal. For instance, in program 4.2.1, F1 and F2 are both evaluated at $I = (4, 4)^T$. This means that equation (4.1) is not sufficient to determine the ordering of functions. We have to take into account, in addition, the position of the functions in the text of the program.

Let $A(I)$ and $B(J)$ be two arbitrary statements of a nested loop program. We define a Boolean function $T(A, B)$ to determine the textual order of statements. T is true when statement A precedes statement B in the text, else T is false. Thus when $p = n$ in (4.1), the precedence relation between the two statements is resolved by $T(A, B)$.

When function-call statements appear in different loop bodies, they will have different iteration-vectors, that is, they appear in different iteration spaces. When two iteration spaces say $\mathcal{I}$ and $\mathcal{J}$ have an intersection, then the two corresponding iteration-vectors, say I and J, have a common sub-vector. This sub-vector is called the *common nesting vector* of I and J. The dimension of this vector is called the *common nesting level* c. The value of c is composed by means of procedure 4.2.1 below.

Procedure 4.2.1 [Find Common Nesting Vector]

Let A and B be two nodes of the parse tree. To determine the common nesting vector of A and B, we derive the path from node A to the root and the path from node B to the root. Then, the procedure searches for common for-nodes in the two paths, starting from the root-node. The *common nesting vector* is formed by the loop iterators annotated by the For-nodes that are in both paths. The size of this vector is the *common nesting level* c. The procedure ends when two For-nodes are found that are not the same. The procedure is outlined below.

```
find path from node A to the root
find path from node B to the root
while nodes in both paths are equal {
      if node is For-node {
        update common iteration vector
      }
      select the next node in both paths
}
```

□

With the iterator lexicographical ordering (4.1) and the notion of a common nesting level of function-call statements, we can define the lexicographical ordering of functions.

Definition 4.2.1 [Lexicographic Order]

Let $I = (i_1, ..., i_n)^T$ and $J = (j_1, ..., j_m)^T$ be two iterations, and $A(I)$ and $B(J)$ two function-call statements.
Let c be the common nesting level of A and B.

Then, we say that $A(I)$ precedes $B(J)$ lexicographically, denoted $A(I) \prec B(J)$, if one of the following hold

- $c = 0 \wedge T(A, B)$
- $(\forall 1 \leq q \leq p : i_q = j_q) \wedge i_{p+1} < j_{p+1}$
- $(\forall 1 \leq q \leq p : i_q = j_q) \wedge p = c \wedge T(A, B)$

□

When $c = 0$ the two statements have no iterators in common and the order is determined solely by the textual order $T(A, B)$.
$T(A, B)$ also determines the ordering when $p = c$.

By applying definition 4.2.1 for each value of p in the range $[1, ..., c]$, we get all together $c + 1$ disjoint cases. We call this process *lexicographical expansion.*

4.3 Data Dependency

In the introduction to this chapter we noticed that a data dependency originates from argument passing between functions. Data dependencies thus result from referencing variables during the function calls.

A function call involves three things. First, variables on the right of the assignment statement are read. Secondly, the function operates on these values and produces output data. Thirdly, the output data is written to the variables on the left of the assignment statement. A data dependency means that a function operates on the value that is the result of the evaluation of another function.

We speak of a *write access* when assigning a variable a value. We speak of a *read access* when reading the value of a variable. A data dependency can only result from a read and a write access to a variable of the *same* name, since we assume that variables with different names do not overlap in memory. LHS variables are read, RHS variables are written to. We refer to a LHS variable and a RHS variable of the same name as a *read-write pair.*

We denote a variable by $v(f(I))$, with v the name of the variable, and $f(I)$ a function on iteration-vector I defining the index of the variable. This function is called the *indexing*

function. The dimension of the variable does not have to be the same as the dimension of the iteration-space. Let A be an integer matrix and c an integer constant vector [2] of appropriate size. We require the indexing functions to be affine functions:

$$f(I, P) = AI + c \tag{4.2}$$

Example 4.3.1 [Indexing Function]

Let the iteration-vector be $(i, j, k)^T$. The indexing function $f(I)$ of the variable $a(j+1, k)$ is

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{4.3}$$

□

Now we formulate the conditions under which two functions are said to be directly data (variable) dependent.

Definition 4.3.1 [Direct Data Dependency]

Let v be a variable array. Let A be the function reading variable $v(g(J))$ with indexing function $g()$ at iteration J. Let B be a function writing to variable $v(f(I))$ with indexing function $f()$ at iteration I.

Function A is *direct data dependent* on function B if the following three conditions hold:

1. $f(I) = g(J)$
2. $I \prec J$
3. I is the *lexicographical largest* iteration satisfying the first two conditions.

□

In the sequel, we mean direct dependency when we say dependency, except when stated otherwise explicitly. The first condition in definition 4.3.1 says that the functions must reference the same element of the array. The second condition says that the function writing to the variable must precede the function reading it. The third condition says that an iteration I' lexicographical farther away than I that satisfies the first two conditions does not exist.

[2] The constant vector may be parametrerized. See also chapter 5

Program 4.3.1 [Data Dependency]
Let F1 and F2 be functions.

```
for i = 1 to 5,
    [A(i+2)] = F1(B(i+1));
    [B(i)] = F2(A(i));
end
```

□

The next example analyzes this program with respect to data depedendencies.

Example 4.3.2 [Data Dependency]

We investigate by enumeration if there are data dependencies inside program 4.3.1. Function F1 reads RHS variable B(i+1) and writes LHS-variable A(i+2). Function F2 reads variable A(i) and writes variable B(i). So there are two read-write pairs: <A(i),A(i+2)> and <B(i+1),B(i)>.

i	A(i+2)	A(i)	B(i)	B(i+1)
1	A(3)	A(1)	B(1)	B(2)
2	A(4)	A(2)	B(2)	B(3)
3	A(5)	A(3)	B(3)	B(4)
4	A(6)	A(4)	B(4)	B(5)
5	A(7)	A(5)	B(5)	B(6)

The table lists the variables that are referenced for each value of the loop iterator i, $1 \leq i \leq 5$. The iteration vector $I = (i)$ is one dimensional. For instance, at iteration $I = (1)$ variable A(3) is written and variable A(1) is read. At iteration $I = (3)$ the variable A(3) is read. Thus the two iterations are data dependent. Similarly, we find that the iterations $I = (2)$ and $I = (4)$ and the iterations $I = (3)$ and $I = (5)$ are data dependent.

There are no data dependencies for variable array B because the read accesses to B always precede the write accesses. This violates the second condition for data dependency.

□

To illustrate the third condition of definition 4.3.1, we use program 4.3.2.

Program 4.3.2 [Third Condition of Data Dependency]
Let F1 be a function.

```
for i = 0 to N,
    for j = 0 to N,
        [a(i+j)] = F1(a(i+j));
    end
end
```

□

This program has the read-write pair <a(i+j),a(i+j)>. Assume the program is at iteration $I = (2,1)^T$, where the function reads variable a(3). We want to determine at which iteration this variable was produced.

Figure 4.2 shows all the iterations of program 4.3.2, in which we have also drawn the line $i+j = 3$ defining the iterations $(i,j)^T$ writing to variable a(3). To find the lexicographical largest iteration on the line $i + j = 3$, we have depicted the lexicographical order of the iterations of the program given in figure 4.3.

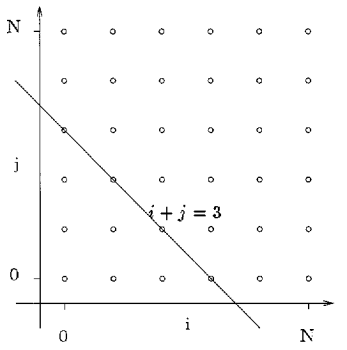

Figure 4.2: Iteration space of program 4.3.2 with superimposed on it the line $i + j = 3$.

By enumeration, we find that the variable a(3) is written at the iterations: $(3,0)^T$, $(1,2)^T$ and $(0,3)^T$. By inspecting figure 4.3, we find that iteration $(1,2)^T$ is the lexicographical largest iteration writing a(3). We conclude that iteration $(2,1)^T$ is dependent on iteration $(1,2)^T$.

4.4 The Data Dependency Problem

The objective is to make the data dependence structure of the program explicit and to model it by a dependence graph. This requires that we know exactly for any iteration

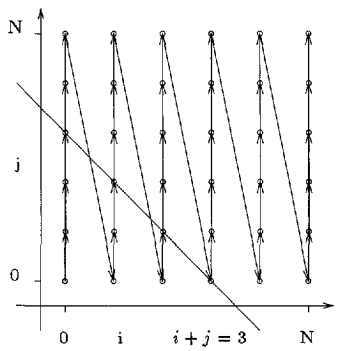

Figure 4.3: Lexicographical ordering of iterations in program 4.3.2.

J in the program on which iteration I it is dependent. Observe that this requires more than a dependency test. In a dependency test, we only check whether two iterations are dependent but do not provide detailed information about the dependency.

It turns out that the data dependency problem can be formulated as an integer linear programming (ILP) problem. The first two conditions for data dependency define a set of feasible iterations I on which J may be dependent.

The solution is the lexicographical largest iteration of this set. However, the problem is more complex as the solution will in general be depending on J. We can solve this problem by a parameterized integer linear programming procedure (PIP), which finds the solution as an affine expression on J [25] [26]. When no solution is found, there is no data dependency.

A complete data dependence analysis of a program involves finding the dependencies for all the RHS variables appearing in the function-call statements. When there are several LHS variables with the same name as the RHS variable, there are several read-write pairs of the same RHS variable. *HiPars* calls PIP to solve the data dependency problem for each read-write pair separately. After that, the solutions found for each read-write pair are combined in order to satisfy the third condition of data dependency.

HiPars uses procedure 4.4.1 to find all the read-write pairs. For example, according to this procedure, *HiPars* finds 120 read-write pairs for variable a in the SVD program 2.3.5.

Procedure 4.4.1 [Find read-write pairs]
This procedure returns the read-write pairs of a program by doing a Depth First Search *(DFS)* of the parse tree of the program. In a DFS search procedure the variables are found in the order in which they appear in the text. For each RHS variable, the procedure applies a DFS search for LHS variables. If the name of a LHS-variable is equal to the name of the RHS variable, we have found a read-write pair. Procedure 4.4.1 is outlined below.

```
for each RHS variable in DFS order {
    for each LHS variable in DFS order {
        if LHS variable name = RHS variable name {
            add read-write pair
        }
    }
}
```

□

4.5 Conclusion

We have formulated the conditions under which functions in nested loop programs are depending on each other through read-write operations on variables. We have taken into account the fact that functions may appear in different loop bodies. In such a case it is necessary to determine the common nesting level. When statements appear in the same loop body, the textual position of the statements to resolves their ordering.

We have also mentioned that the analysis involves finding closed expressions for the dependencies. The analysis is thus much more than a data-dependency test. The data dependency problem can be formulated as an LP problem, which can be solved by PIP that outputs the expressions for the data dependencies.

In chapter 6 we explain PIP, which is used to find the dependencies for each read-write pair separately. In chapter 7, we combine the solutions found for the individual read-write pairs of a RHS variable and come up with the overall solution of the data dependency problem.

Chapter 5

Annotating the Parse Tree

5.1 Introduction

We solve the data dependencies inside nested loop programs by the linear programming algorithm (PIP). PIP's input must be in the form of system of inequalities, which specify a parameterized set of iterations.

In chapter 3, we explained how a nested loop program is modeled by a parse tree. In this chapter, we annotate the nodes of a parse tree with linear inequalities in order to specify the input for PIP.

The inequalities annotated by the nodes in the path from a particular node to the root define the set of iterations for which the corresponding statement in the nested loop program is reached. We call these sets *iteration domains*. By doing this for the function-call statements, we define the sets of iterations at which the read and write accesses to variables take place in a nested loop program.

As PIP's input consists of a set of linear inequalities, we have to model the non-linear transformation operators inside nested loop programs by linear inequalities. In addition, we have to do this when the stride of the loop statements is greater than one. We give the linear inequalities for each of these cases.

We begin this chapter by introducing some notations and definitions of the field of Linear Programming [52].

5.2 Notation and Terminology

First of all, we denote the set of integers by $\mathbf{Z}$ and the set of reals by $\mathbf{R}$.

Let $a = (a_1, a_2, ..., a_n)^T \in \mathbf{R}^n$ be a vector and $c \in \mathbf{R}$ be a constant, and let $I = (i_1, i_2, ..., i_n)^T$ be a vector of variables taking values in $\mathbf{R}^n$. Then an *affine expression* is an expression of the form:

$$a_1 i_1 + a_2 i_2 + ... + a_n i_n + c \tag{5.1}$$

Or, in vector notation:

$$aI + c \tag{5.2}$$

Based on this definition, we define a *linear inequality* as:

$$aI + c \geq 0 \tag{5.3}$$

An inequality with a smaller than or equal to operator, $\leq$, can be transformed to the form of equation 5.3 by taking its negative.

We define an *integral polytope* as a bounded set of points k taking values in $\mathbf{Z}^n$ specified by a system of linear inequalities. A polytope defines a complete set of integer index points. Let m be the number of inequalities. With A an $m \times n$ integer matrix and C a vector in $\mathbf{Z}^m$, we define an integral *polytope* e by [52]:

$$e = \{k \in \mathbf{Z}^n \mid A\ k + C \geq 0\} \tag{5.4}$$

We allow the constant vector b to depend on the program's size parameters. More precisely, b may be an affine expression on these parameters.
Let P be the vector of parameters and let p be the number of parameters. With $D \in \mathbf{Z}^m$ a constant vector and a matrix $B \in \mathbf{Z}^{m \times p}$, we write C as:

$$C = BP + D \tag{5.5}$$

We use the term *domain* when we require the points I to lie on a *lattice*. With L an $n \times m$ integral matrix and $O \in \mathbf{Z}^n$ an integral vector, we define a lattice as:

$$I = L\ k + O \tag{5.6}$$

We call O the *offset* of the lattice and the columns of L the *lattice vectors*.

A domain is defined by a lattice and a polytope:

$$\{I \in \mathbf{Z}^n \mid I = L\ k + O, A\ k + C \geq 0\} \tag{5.7}$$

We say that the lattice points I are generated by the columns of L, with the variables of k bounded by the polytope.

A domain is *dense* when it includes all integer points $I \in \mathbf{Z}^n$ inside the polytope. When not all integer points inside the polytope are included, we call the domain *sparse*. This means that the lattice constraints filter out points of the polytope.

5.3 The iteration domain

We refer to the set of iterations at which a statement is reached in a program as the *iteration domain* of the statement. In particular, we want to specify the iterations at which read and write accesses to variables take place.

We specify sets of iterations by polytopes of the form of equation (5.4).

To obtain these polytopes, we annotate the nodes of the parse tree representing the nested loop program by linear inequalities. In the following two sections, we define these linear inequalities for each type of node. After we have annotated the nodes of the parse tree, we easily derive the polytopes from the parse tree by procedure 5.3.1.

The polytope of a node's iteration domain is formed by the inequalities of the nodes in its path to the root. This procedure can be used to derive the polytope for any node of the parse tree, but is mainly used for finding the polytopes for the LHS and RHS variables.

Procedure 5.3.1 [Derive Polytope]

Let N be an arbitrary node of the parse tree. The procedure finds the iteration domain of N by collecting the inequalities of the nodes in the path from N to the root. Together the inequalities form the polytope e of the iteration domain. The procedure is outlined below.

for each node in the path from node N to the root {
    add linear inequality to polytope e
}

□

Each point in a polytope represents an iteration of the nested loop program. The number of points in a polytope is equal to the number of times the corresponding statement of the program is evaluated.

The dimension of the iteration domain depends on the number of For-nodes in the path. When there are t For-nodes in the path, we say that the nesting level of the node is t. The iteration vector $I = (i_1, ..., i_t)$ of the iteration domain consists of the iterators of the loop statements.

5.4 Annotating the parse tree

Below, we annotate types of nodes by inequalities that belong to dense iteration domains. In section 5.5, we annotate the remaining types of nodes belonging to nondense iteration domains.

The types of nodes are annotated as follows:

- **Parameter-node**

  Let p be a parameter represented by a parameter-node. The value of p must be an integer between integer constants c_1 and c_2, with $1 \leq c_1 \leq c_2$.

  A parameter-node is annotated with the inequalities:

$$c_1 \leq p \leq c_2 \tag{5.8}$$

- **For-node, stride equal to one**

  Let i be the iterator of a loop statement represented by a For-node. Iterator i is bounded by the lower bound expression l and upper bound expression u, with l and u affine expressions on the parameters and the outer loop iterators.

  When the stride is equal to one, a For-node is annotated with the inequalities:

$$l \leq i \leq u \tag{5.9}$$

- **If-node**

  An If-node is annotated with its conditional expression, which is of the form:

$$a\ I + C \geq 0 \tag{5.10}$$

- **Else-node**

  An Else-node has always a corresponding If-node in the parse tree. Let the inequality of the If-node be given by equation (5.10). Then, an Else-node is annotated with the inequality:

$$aI + C < 0 \tag{5.11}$$

Since every expression in nested loop programs must be an affine expression, we call them *affine nested loop programs*. Below, NLP stands for affine nested loop programs.

Example 5.4.1 [Iteration domain]

We derive the iteration domain of the function-call statement in program 5.4.1. The nesting level of the statement is two. Each loop statement results in two inequalities, one for the lower bound and one for the upper bound. With the inequalities for the range of parameter M, polytope e is specified by:

$$\begin{array}{rrrrl} i & & & -1 & \geq 0 \\ -i & & +M & & \geq 0 \\ -i & +j & & & \geq 0 \\ & -j & +M & & \geq 0 \\ & & M & & \geq 0 \\ & & -M & +25 & \geq 0 \end{array} \tag{5.12}$$

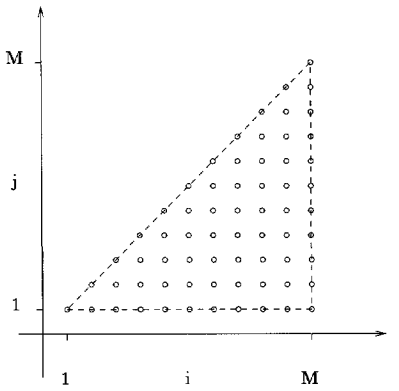

Figure 5.1: Iteration domain of program 5.4.1

We have depicted the iteration domain in figure 5.1.

□

Program 5.4.1 [Iteration domain]

Let F1 be a function.
Let M be parameter with range $0 \leq M \leq 25$.

```
for i = 1 to M,
    for j = i to M,
       [a(i,j)] = F1(a(i,j));
    end
end
```

□

5.5 Nondense Domains

When a nested loop program contains loop statements with a stride greater than one or has non-linear transformation operators in its expressions, the iteration domains become nondense. However, the sets of iterations can still be described by polytopes. In chapter 8, we discuss the relation between the points k of the polytope and the points of the iteration domain defined by the lattice.

5.5.1 Stride greater than one

For a loop statement with stride greater than one, the lower and upper bound expressions are not sufficient to describe the values that the loop iterator takes on. We have to add an additional constraint specifying that the difference between values of the iterator is a multiple of the stride.

The syntax of a loop statement with stride b is:

`for` i `=` l `to` u `step` b

Its semantics is as follows. First the iterator is initialized with the value of l. Then as long as $i \leq u$, it is incremented by the value of b after each loop.

Example 5.5.1 [Stride]

Consider the following loop statement with stride three:
`for` i `=` 1 `to` 10 `step` 3

If we applied equation (5.9) to model this statements, we would get the polytope:

$$1 \leq i \leq 10 \tag{5.13}$$

However, according to the semantics of the loop statement, the iterator i of the loop statement takes on the values $1, 4, 7, 10$ only. Therefore, the polytope includes too many points and does not model correctly the loop statement.

□

The example shows that we cannot specify the iteration domain in terms of the loop iterator only. Therefore, we introduce an additional control variable to account for the stride. Let q be this variable, which is an integral variable. Then, we model the stride by the equation:

$$i = l + q * b \tag{5.14}$$

By taking also the loop bounds into account, we annotate a For-node as follows.

- **For-node with Stride greater than one** Let i be the iterator of a loop statement represented by a For-node. Let l and u be the lower and upper bound expression, respectively. Let b be the stride. With q an integer variable, a For-node with stride $b > 1$ is annotated by:

$$\begin{aligned} i &= l + b * q \\ i &\leq u \end{aligned} \tag{5.15}$$

The equation together with the inequality of the upper bound form a polytope in the (i, q) space. The polytope is closed and defines the values of iterator i. Thus the nondense iteration domain is decribed by a polytope in a higher dimensional space. Each additional variable introduces an extra dimension.

5.5.2 Non-linear Index Transformations

In this section, we model the non-linear operators that may appear in the expressions of the program: *integer division*, *ceil*, *floor*, and *modulo*. We define them by linear inequalities for which we also have to introduce new variables as we did for the stride of loop statements.

In general, non-linear operators will make the data dependence analysis very complex. It is only because we can define these non-linear operators by linear inequalities that they are allowed inside nested loop programs.

Definition 5.5.1 [Integer Division]

Let a be an integer and let b be a positive integer, called the *divisor*. The result of *integer division* of a and b are integers q and r, such that:

$$a = b * q + r, \quad 0 \leq r \leq (b-1) \tag{5.16}$$

We call b the *divisor* and r the *remainder* of the division.

□

We denote the operator computing the integer division by $div(a, b)$ in which the value of a may be defined by an affine expression. However, b must be a positive integer constant.

When a $div()$ operator is used in conditional statements or in the bounds of loop statements, we cannot describe the polytope of the domain in terms of the loop iterators solely. We have to introduce a variable q defined by equation (5.16) as additional variable in the polytope.

With a little rewriting of equation 5.16 we obtain two linear inequalities:

$$0 \leq a - b * q \leq (b-1) \tag{5.17}$$

These inequalities define the value of the variable q. As $q = div(a, b)$, a div operation is defined by these two linear inequalities.

Observe that for each value of a and b the integer division of a and b results in a unique value of q. Therefore, we can replace each $div()$ operator by q in the expression of the nested loop program.

Example 5.5.2 [Integer Division]

The values of $q = div(i+3, 4)$ with $i \in [2, 14]$ are defined by the inequalities:

$$\begin{array}{l} 2 \leq i \leq 14 \\ 0 \leq i + 3 - 4q \leq 3 \end{array} \tag{5.18}$$

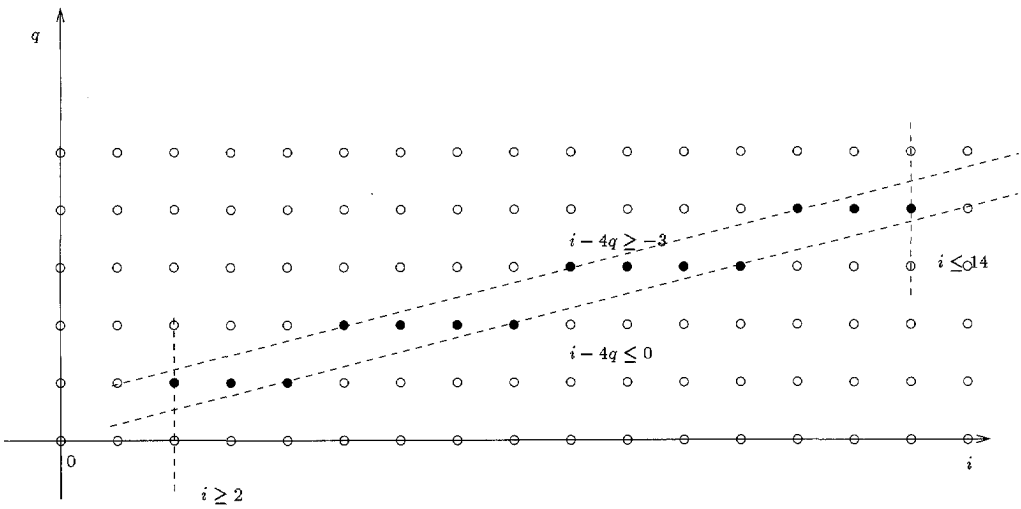

Figure 5.2: The index points defined by $q = div(i + 3, 4)$

These inequalities define a polytope in the (i, q) space, which we have drawn in figure 5.2. Each point $(i, q)^T$ in the polytope stands for a result of the division operator.

□

Thus *HiPars* annotates a Div-node as follows:

- **Annotating the *Div*-node**
  Let q be a variable.
  Given an affine expression a and an integer constant b.

  A Div-node of the parse tree is annotated with the inequalities:

$$\begin{aligned} a - b * q &\geq 0 \\ (b - 1) - a + b * q &\geq 0 \end{aligned} \tag{5.19}$$

The fact that we can define the operator $div()$ by two inequalities makes this operator special. Below we define the operators $mod()$, $floor()$, and $ceil()$ in terms of a $div()$ operator. This implies that these operators can also be defined by linear inequalities. For details we refer to [36]. With a/b standing for the real division of a and b, we define:

- $floor(a/b) = div(a, b)$
  rounds the result of a/b to the greatest integer smaller than or equal to a/b.

- $ceil(a/b) = -div(-a, b)$
  rounds the result of a/b to the smallest integer greater than or equal to a/b.
- $mod(a, b) = a - b * div(a, b)$
  returns the value of the remainder of the integer division.

According to these formulas, *HiPars* rewrites nested loop programs in terms of the $div()$ operators only.

Below, we have listed program 5.5.1 which is obtained from program 3.3.2 after substitution of the modulo operators by integer division operators inside the expressions of the conditional statements.

Program 5.5.1 [Program with Integer Divisions]

Let M be a parameter.
Let F1 and F2 be two functions.
Let a be a two-dimensional variable array.

```
for i = 1 to  M step 2,
   for j = 1 to M step 2,
        if  i - 3*div(i,3) <= 0,
                [a(i,j)] = F1();
        end
        if j - 3*div(j,3) <= 0,
                F2(a(i,j));
        end
   end
end
```

□

By applying procedure 5.3.1 in order to derive a polytope, *HiPars* will not only find iterators, but also additional control variables introduced to model strides and non-linear index transformations.

Let $\mathbf{q} = (q_1, .., q_r)$ be the vector of these additional control variables.

Then the control vector of the polytope is $k = (i_1, ..., i_n, q_1, ..., q_r)^T$, not necessarily in this order, and takes values in $\mathbf{Z}^{n+r}$. In a way, we have extended the dimension of the iteration space, making it possible to define the set of iterations by polytopes. Or, in other words, we have transformed the nondense domain into a dense domain in a higher dimensional space. The number of points in the nondense domain is equal to the number of points of the polytope. The transformation is thus one-to-one.

5.6 Formulating the Data Dependency Problem

In this section, we formulate the conditions of data dependency given in chapter 4 as PIP input. The input of PIP consists of a *problem* and a *context* specification, which are both specified by a polytope.

As said, *HiPars* calls PIP for each read-write pais. Let the read-write pair be $(v(g(J)), v(f(I)))$, with $v(g(J))$ a RHS variable and $v(f(I))$ the LHS variable. I, J are the control vectors of the left-hand side and right-hand side variable, respectively.

The values that control vector I takes on are specified by a polytope e_w defined by the inequalities annotated by the nodes in the control path of the LHS variable:

$$e_w = \{I \mid A_w I + b_w \geq 0\} \tag{5.20}$$

Polytope e_w describes the set of iterations at which write accesses to the LHS variable take place.

The equations resulting from $f(I) = g(J)$, which are both affine functions, are:

$$A_f I + b_f = A_g J + b_g \tag{5.21}$$

The vectors b_w, b_f, and b_g are affine expression of the size parameters of the nested loop program.

Note that by the equation the variables of I as well as the variables of J enter the system. These equations are known as Diophantine equations [9]. As this system is underdetermined, solutions will generally be expressed in the free variables. In our problem, we choose the variables of J to be free and we keep them in the system as parameters. The solution for I will then be expressed in terms of these parameters [79].

Thirdly, the inequalities corresponding to each case of the lexicographical expansions of $I \prec J$ are specified. The number of cases depends on the common nesting level as defined by procedure 4.2.1.

In addition to the above inequalities, *HiPars* adds non-negativity constraints on the variables of the problem. Together the linear inequalities form a parametrized polytope. We refer to this polytope as the *problem* of PIP and denote it by e_p.

The lexicographical largest vector of polytope e_p

$$d = max_{\prec}(e_p) \tag{5.22}$$

is the vector on which J is dependent. We use PIP to find this solution vector, if it exists, and call it the *dependency-vector*. The set of feasible solutions is parametrized with respect to vector J and the solution vector will be expressed in terms of the elements of J. Because of this parametrization, PIP branches internally into a number of cases and returns for each branch a solution-vector. The result produced by PIP is, in general, a solution tree.

We will now describe the *context*, which is also specified by a polytype. Because the data dependency problem is parametrized, we have to specify the values that the parameters can take on. The parameters of the context are the size parameters of the program and the elements of the vector J. The values of J are defined by the polytope e_r of the RHS variable.

The value of a size parameter, say p, is between the range defined by two integer constants c_1 and c_2: $c_1 \leq p \leq c_2$ with $c_1 \geq 1$. In addition, we add non-negativity constraints on the parameters of the context.

The context is thus defined by:

1. the polytope e_r of the RHS variable
2. the ranges of the size parameters.

We represent the context by the polytope e_c.

We denote the input of PIP by the tuple $< problem, context >$. In procedure 5.6.1, we outline the procedure to formulate the PIP input.

Procedure 5.6.1 [Formulate PIP input]

use procedure 4.2.1 to find common nesting vector
for each lexicographical expansion {
    formulate a PIP Problem e_w
        use procedure 5.3.1 to find polytope e_w of the LHS variable
        add inequalities of the lexicographical expansion
        add index equations defined by the indexing functions
    formulate a PIP Context e_r
        use procedure 5.3.1 to find polytope e_r of the RHS variable
        add constraints on size parameters
}

□

As an example, we analyze the data dependencies of the part of the SVD algorithm that performs the row and column rotations inside the odd stage. We have listed this part, called 'Rotate', in program 5.6.1. In this chapter, we formulate the PIP input for a particular read-write pair in this program. In chapter 7, we present the result of the complete data dependence analysis for the program 'Rotate' in the form of a single assignment program.

The parse tree of program 'Rotate' is shown in Figure 5.3. The program consists of two loop stages. In the first loop stage the row rotations are carried out. In the second loop stage the column rotations.

Both function-call statements access variable array a. There are four write accesses to variable a and four read accesses to variable a, which result in 16 read-write pairs for which we have to formulate the PIP input.

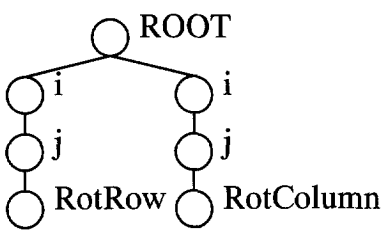

Figure 5.3: Parse tree of program 'Rotate'.

Program 5.6.1 [Rotate]

Let `M` $\in [1, 10]$ be a parameter.
Let `RotRow` and `RotColumn` be functions.
Let `a` be a two-dimensional variable array.

```
for i = 1 : 2 : M-1,
   for j = 1 : 1 : M,
      [a(i,j),a(i+1,j)] = RotRow(th1(i),a(i,j),a(i+1,j));
   end
end
for i = 1 : 2 : M-1,
   for j = 1 : 1 : M,
      [a(j,i),a(j,i+1)] = RotColumn(a(j,i),a(j,i+1),th2(i));
   end
end
```

□

Example 5.6.1 [PIP Input]

We specify the context and the problem for the read-write pair `<a(j,i+1),a(i,j)>` of program 5.6.1. The RHS variable `a(j,i+1)` is the variable appearing as second argument inside the function call to `RotColumn`. The LHS variable `a(i,j)` is the first variable appearing inside the function call to `RotRow`.

We first describe the *context*. The control vector of the RHS variable is $J = (i, j)^T$. For the sake of clarity, we give the variables of J the subscript r. So $J = (i_r, j_r)^T$. In addition, the program has size parameter `M`.

By applying procedure 5.3.1 to the RHS variable `a(j,i+1)`, we find its polytope e_r, in which variable q_r models the stride:

$$
\begin{aligned}
& i_r = 1 + 2q_r \\
& i_r \leq M - 1 \\
& j_r \geq 1 \\
& j_r \leq M
\end{aligned}
$$

We complete the specification of the context by adding the inequalities specifying the range of parameter `M`:

$$
\begin{aligned}
& M \geq 1 \\
& M \leq 10
\end{aligned}
$$

Next we specify the *problem*. First, we apply procedure 5.3.1 to find polytope e_w defining the iterations of the LHS variable:

$$
\begin{aligned}
& i = 1 + 2q_w \\
& M - i \geq 0 \\
& j - 1 \geq 0 \\
& M - j \geq 0
\end{aligned}
$$

The indexing function of the LHS variable is $f(I) = (i, j)^T$ and the indexing function of the RHS is $g(J) = (j_r, i_r + 1)^T$. To satisfy the second condition for data dependency $f(I) = g(J)$, we add:

$$
\begin{aligned}
& i = j_r \\
& j = i_r + 1
\end{aligned}
$$

Finally, we determine the number of the lexicographic cases we have to solve. By applying procedure 4.2.1, we find that the common nesting level is zero. However, the Boolean function T is true because the function call to `RotRow` precedes function call to `RotColumn` in the program. Thus there is only one case to solve.

At this point, we have specified the data dependency problem completely.

Note that the number of cases is equal to two when the LHS variable is taken from the

same statement as the RHS variable. Then we have to consider two cases:

case 1). $-i + i_r - 1 \geq 0$

case 2). $i - i_r = 0$
$-j + j_r - 1 \geq 0$

and to define a separate PIP problem for each case.

□

5.7 Conclusion

We have formulated the data dependency problem as a PIP problem, in the form of a *parametrized polytope* defining the set of iterations to which a LHS variable is written. The lexicographic largest iteration of this set is the *dependent* iteration.

We have shown the procedure used by *HiPars* to annotate each node of the parse tree of a nested loop program by linear inequalities. This requires the introduction of additional control variables when the stride of loop statements is greater than one and when the program contains non-linear index transformations. Extending the dimension of the polytope beyond the dimension of the domain can be regarded as an up transformation of the nondense domain into a higher dimensional space. By traversing the path from a node to the root of a parse tree, a polytope can be constructed that describes the set of iterations at which the corresponding statement is reached.

Chapter 6

Parametric Integer Programming

6.1 Introduction

When we restrict ourselves to the class of nested loop programs and require the programs to have static control and linear expressions, we can analyze the data dependence by a linear programming technique (PIP = parametric integer programming) due to Feautrier [25] [26]. In the previous chapter, we showed that the class of nested loop programs which contains the non-linear operators *integer division*, *modulo*, *ceil* and *floor* in its expressions can be converted to the previous. Based on the definition of integer division, we can express these operators in terms of linear control statements, allowing us to use PIP for the analysis of these programs too. PIP was introduced and implemented by [25] and later improved [18].

The outline of this chapter is as follows. We describe the linear programming techniques used by PIP. After introducing some terminology and notations of linear programming, we explain the classical *dual simplex algorithm* for finding an optimal solution of a system of linear inequalities. Then, we describe how Feautrier modified the dual simplex algorithm in order to find the lexicographic largest solution of the system. The solution returned by PIP is a closed expression on the parameters of the system.

6.2 Systems of equations

To introduce linear programming, we consider first a system of simultaneous linear equations. Let C be an m by n matrix and let $\mathbf{d}$ be an m-dimensional vector called the *constant vector*. With $\mathbf{x}$ standing for the vector of variables $x_1, ..., x_n$, we denote the system by:

$$C\mathbf{x} = \mathbf{d} \tag{6.1}$$

Or, equivalently, in expanded format:

$$\begin{array}{c} c_{11}x_1 + c_{12}x_2 + ... + c_{1n}x_n = d_1 \\ c_{21}x_1 + c_{22}x_2 + ... + c_{2n}x_n = d_2 \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ c_{m1}x_1 + c_{m2}x_2 + ... + c_{mn}x_n = d_m \end{array} \tag{6.2}$$

We assume that there are fewer equations than variables, $m < n$. This means that the system has either no solution or infinitely many solutions. A solution is any n-tuple s of real values for which the m equations are valid statements. A system for which no solution exists is *infeasible.*

Example 6.2.1 [System]

A system with $m = 2$ and $n = 4$ is:

$$\begin{array}{c} x_1 + x_2 + 2x_3 = 22 \\ x_1 - x_2 + x_4 = 4 \end{array} \tag{6.3}$$

For instance, tuples $s_1 = (11, 7, 2, 0)$ and $s_2 = (10, 8, 2, 2)$ are solutions.

□

In order to find system solutions, we need a generic representation of a solution. Since at most m variables in the system are free, we rewrite it into *basic* form in order to exhibit a set of such independent variables, called the *basis* of the solution. We only consider systems that can be written in this from. We refer to [12] for proofs that show a system in basic form exists which is equivalent to the original system.

Definition 6.2.1 [Basic Form]

Let S be a system of m linear equations in $m + p$ variables $x_1, ..., x_{m+p}$, with $m \geq 2$ and $p \geq 1$ and $n = m + p$.
Let I be an m x m identity matrix and A be an m x p matrix.
Let vector $\mathbf{x}'$ be a permutation of $\mathbf{x} = (x_1, ...x_n)$ and $\mathbf{b}$ a constant vector.
System S is in **basic form** if there is a permutation $\mathbf{x}'$ of $\mathbf{x}$ such that

$$[IA]\ \mathbf{x}' = \mathbf{b} \tag{6.4}$$

is equivalent to the original system.

□

We call the variables forming the first m elements of $\mathbf{x}'$ *basic variables* and we say that the set of these variables is a *basis* for S. The remaining p elements of $\mathbf{x}'$ are *nonbasic* variables. By definition, we call the solution of which all nonbasic variables are equal to zero the *basic solution*. The *basic solution* is unique for a system in basic form. From equation 6.4 follows that the value of the basic variables are equal to the elements of constant vector $\mathbf{b}$.

Example 6.2.2 [Basic Form]

We can write the system of example 6.2.1 in basic form with x_3 and x_4 as basic variables by dividing the first equation by the coefficient of x_3, resulting in the equations:

$$\begin{cases} x_3 + 1/2x_1 + 1/2x_2 = 11 \\ x_4 + x_1 - x_2 = 4 \end{cases} \tag{6.5}$$

Or in matrix notation:

$$\begin{pmatrix} 1 & 0 & 1/2 & 1/2 \\ 0 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 11 \\ 4 \end{pmatrix} \tag{6.6}$$

By setting the nonbasic variables x_1 and x_2 to zero, we obtain the tuple $(x_1, x_2, x_3, x_4) = (0, 0, 11, 4)$ as a basic solution.

□

The basis of a system is, in general, not unique if it exists. If the m equations of the system are independent, we can choose as basis any set of m variables, with a nonzero coefficient, from the set of n variables. Let ! denote the *factorial* operator. Then the number of choices for a basis is typically equal to $C_m^n = n!/m!(n-m)!$. In our example, there are $4!/2!2! = 6$ bases:

$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_2, x_3\}, \{x_2, x_4\}, \{x_3, x_4\}$.

Some bases may appear more convenient for specific treatment. Therefore, we may want to change the basis in a systematic way by *pivoting*. Suppose that we would like to enter nonbasic variable x'_{m+s} to the basis. Then we must find a nonzero coefficient of variable x'_{m+s}. Let element a_{rs} of matrix A be this coefficient, which we call the *pivot*. The pivot operation on a system of equation involves simple linear operations:

- divide equation r by the pivot a_{rs}. This makes the coefficient of the variable x'_{m+s} equal to one.

- add $-a_{is}/a_{rs}$ times equation r to all other equations $i \neq s$ of the system. This makes all remaining elements a_{is} in the column of matrix A zero.

- Finally, apply row and column permutations to write the resulting system in *basic* form.

By pivoting with pivot a_{rs}, variable x'_{m+s} will enter the basis and variable x'_r will leave the basis. Note that the basis after one pivot operation differs only in one variable. The other $m-1$ basic variables will stay in the basis. It is easy to prove that systems obtained by pivoting are equivalent, i.e., they have the same solution.

We can make the identity matrix implicit and write the system as a *condensed tableau.* This is a more pictorial way of representing systems, and it is very often used in LP. The condensed tableau is defined as follows, with $p = n - m$:

	k_1	k_2	...	k_p	
j_1	a_{11}	a_{12}	...	a_{1p}	b_1
j_2	a_{21}	a_{22}	...	a_{2p}	b_2
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
j_m	a_{m1}	a_{m2}	...	a_{mp}	b_m

The box has two fields that are filled with the elements of matrix A and vector $\mathbf{b}$, respectively. The numbers on the left margin of the box are the subscripts of the basic variables. The numbers k_1 , ... k_p on the top margin of the box are the subscripts of the nonbasic variables.

Example 6.2.3 [Condensed Tableau]

The condensed tableau of example 6.2.2 is

	1	2	
3	1/2	1/2	11
4	1	−1	4

□

6.3 LP problems and methods

At this point, it is appropriate to define the *linear programming* (LP) problem in *primal form*. Let f be a linear function defined by the inner product of a constant row vector $\mathbf{c}$ and variable vector $\mathbf{x} = (x_1, ..., x_n)^T$. We refer to function f as the *objective function*. The primal LP problem is to maximize

$$f(\mathbf{x}) = \mathbf{cx} \tag{6.7}$$

subject to the inequalities:

$$A\mathbf{x} \leq \mathbf{b} \tag{6.8}$$

and non-negativity constraints:

$$\forall j, 1 \leq j \leq n : x_j \geq 0 \tag{6.9}$$

To solve the LP problem, we put this problem in the form of a system of linear equations. Let $\mathbf{a}_r$ be a row of matrix A, with $1 \leq r \leq m$. We write each inequality $\mathbf{a}_r\mathbf{x} - b_r \leq 0$ as an equation $a_r\mathbf{x} - b_r + x_{n+r} = 0$ under the constraint $x_{n+r} \geq 0$. As matrix A has m rows, there will be m such variables, which we call *slack variables*. Note that the resulting system of equations is in basic form, with the m slack variables as basic variables and the variables of $\mathbf{x}$ as nonbasic variables. In addition, we add the objective function $F = cx$ as equation $F - cx = 0$ to the system.

When we put everything together in a condensed tableau, called the *primal tableau*, we get:

	k_1	k_2	...	k_p	
j_1	a_{11}	a_{12}	...	a_{1p}	b_1
j_2	a_{21}	a_{22}	...	a_{2p}	b_2
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
j_m	a_{m1}	a_{m2}	...	a_{mp}	b_m
F	$-c_1$	$-c_2$	...	$-c_n$	0

Two famous methods to solve the primal LP problem are:

- the *simplex method*
- the *dual simplex method*

Both methods consist of a sequence of pivoting operations resulting in at most C_m^n steps to a final tableau, which corresponds with one of three possible outcomes:

1. if all elements of $\mathbf{b}$ and $-\mathbf{c}$ are nonnegative, the basic solution is the *optimal feasible solution.*

2. the problem is *infeasible*, i.e. there is no solution at all.

3. the objective function is unbounded above.

We consider only the first two cases. In the problems we are considering, the unboundedness of the cost function will not appear.

The *simplex method* can be applied only when the elements b_i are greater than zero, $b_i \geq 0, 1 \leq i \leq n$, in the initial tableau. The *dual simplex algorithm* can be applied when all entries $-c_i$ in the objective row are nonnegative. The latter situation will always be the case in our problems. In the following, we consider only the dual simplex algorithm.

The dual simplex algorithm pivots in such a way that all entries in the objective row remain nonnegative and will lead, after a finite number of pivot operations, to a tableau with also nonnegative elements of the constant vector $\mathbf{b}$, if it exists. For more information on this algorithm we refer to [12] [52]. The rules for selecting the pivot are:

1. select the rows of the tableau of which the constant b_r is negative and which contain negative elements a_{rs}.

2. select element a_{rs} of these rows for which the absolute value of the ratio c_s/a_{rs} is the smallest.

In other words, introduce in the basis a variable causing the highest increase in the value of F and remove from the basis a variable which is not sufficiently constrained by its b term ($\ldots - x_i \ldots \leq b_i < 0$).

The primal LP problem searches the maximum of F in the domain defined by the system of inequalities. We can find the minimum value of F by taking the negative of the cost function and searching for the maximum value of $-F$. As will become clear in later sections, we are especially interested in finding the minimum.

Example 6.3.1 [Dual Simplex Algorithm]

Suppose we want to find the minimum value of the objective function $F(x_1, x_2) = 10x_1 + x_2$ for nonnegative variables x_1 and x_2, satisfying the following two linear inequalities:

$$\begin{aligned} -x_1 - x_2 &\leq -12 \\ -x_1 + x_2 &\leq -4 \end{aligned} \tag{6.10}$$

Figure 6.1 shows the feasible solutions in the (x_1, x_2) space.

As we are searching for the minimum we use as cost function $F' = -F = -10x_1 - x_2$. Next, we introduce slack variables x_3 and x_4 to obtain the equations:

$$\begin{aligned} x_3 - x_1 - x_2 &= -12 \\ x_4 - x_1 + x_2 &= -4 \end{aligned} \tag{6.11}$$

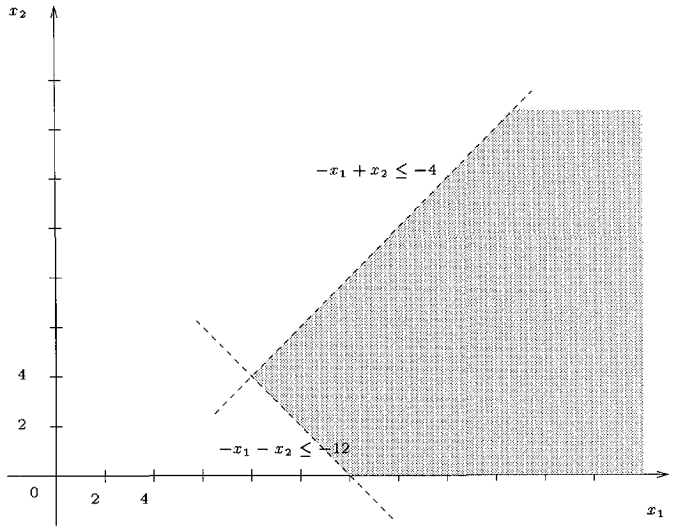

Figure 6.1: The shaded area indicates the feasible solutions defined by the inequalities $-x_1 + x_2 \leq -4$ and $-x_1 - x_2 \leq -12$ and non-negativity constraints $x_1 \geq 0$ and $x_2 \geq 0$. The optimal feasible solution is $(8, 4)$.

This system is in basic form with basis $\{x_3, x_4\}$ and its condensed tableau is:

	1	2	
3	-1	$\mathbf{-1}$	-12
4	-1	1	-4
F'	10	1	0

We see that all the elements of the cost vector c are positive. Therefore, we can apply the dual simplex algorithm. Both rows have a negative constant, and the negative elements in these rows are: $a_{11} = -1$, $a_{12} = -1$ and $a_{21} = -1$. Next we determine the ratio c_s/a_{rs}: $\lfloor c_1/a_{11} \rfloor = 10$, $\lfloor c_1/a_{21} \rfloor = 10$ and $\lfloor c_2/a_{12} \rfloor = 1$. Thus we choose a_{12} as pivot.

After pivoting with a_{12}, we get the tableau:

	1	3	
2	1	1	12
4	$\mathbf{-2}$	1	-16
F'	9	1	-12

In this tableau, we see that only the second row has a negative b constant and that element a_{21} is the only negative negative element in this row. Thus we pivot with -2 and we obtain the next tableau:

	4	3	
2	1/2	3/2	4
1	**−1/2**	−1/2	8
F'	9/2	11/2	−84

This tableau is the final tableau because it does not have negative elements b. The optimal feasible solution is $s_{opt} = (4, 8, 0, 0)$. Therefore, the minimum value of $F = 84$ for $x_1 = 4$ and $x_2 = 8$.

□

6.4 Lexicographic Dual Simplex Algorithm

In example 6.3.1, the objective coefficient c_1 is ten times bigger than c_2. This means that an increase in the value of variable x_1 weights ten times as much as an increase in the value of variable x_2. Suppose we let c_1 go to infinity, and suppose that we have found an optimal solution $s = (s_1, s_2)$. Then the value of the objective function at this optimum is less than or equal to the value of F for any other arbitrary feasible solution (p_1, p_2):

$$\begin{aligned} c_1 s_1 + s_2 \leq c_1 p_1 + p_2 \Leftrightarrow \\ c_1(p_1 - s_1) + (p_2 - s_2) \geq 0 \end{aligned} \tag{6.12}$$

Let us further assume that the values of the variables of feasible solutions are bounded, i.e. $\lim_{c_1 \to \infty} p_i/c_1 = 0$. Then there are two cases for which equation 6.12 is true:

- case $p_1 - s_1 > 0$, regardless of the values of p_2 and s_2
- case $p_1 - s_1 = 0$ and $p_2 - s_2 > 0$

In other words, the optimal feasible solution (s_1, s_2) is the smallest lexicographical solution.

Before we generalize this result, we consider another property of the pivoting operation, which is that the computation of values of matrix A and vector b of consecutive tableaus does not involve elements of the objective row.
As the optimal feasible solution is the value of b in the final tableau, the optimum value is determined by calculations on values of elements of A and b only.

In the dual simplex algorithm, we select the pivot on basis of the smallest ratio $\lfloor c_s/a_{rs} \rfloor$. Now we introduce another pivot selection rule for the dual simplex algorithm which will result in the lexicographically smallest feasible solution of the system.

First, we extend the tableau with trivial equations $x_1 = x_1, \ldots, x_m = x_m$ and place them at the first m rows of the tableau, in the order given. In other words we put $x_1, \ldots, x_m$ into the basic set. Initially $b_1 = 0, \ldots, b_m = 0$. When we do not disturb the order of the first m rows, the solution for the vector $(x_1, \ldots, x_m)$ is equal to the first m elements of $\mathbf{b}$. We maintain this order by not selecting pivots from the first m rows. In the case where elements of b in these first rows are negative, negative elements of $\mathbf{b}$ will always exist in the rows below. This reflects the fact that the first m equations are trivial.

Let $\mathbf{a}_k$ stand for an arbitrary column vector of the extended A matrix, $1 \leq k \leq m$.

The lexicographic pivot rules are:

- select the rows with row number $r > m$ with negative elements of b.
- Let K be the set of column numbers k of columns $\mathbf{a}_k$ with element $a_{rk} < 0$.
- Determine the smallest lexicographical vector of the vectors:

$$k \in K : \frac{\mathbf{a}_k}{a_{rk}} \tag{6.13}$$

- Let $\frac{\mathbf{a}_s}{a_{rs}}$ be this vector. Then we choose element a_{rs} as a pivot.

Observe that the column pivot is fully determined by the subscripts r and s of the negative elements of a_{rs} in the rows with negative constant b_r.

We conclude that when we modify the rules for selecting the pivot as indicated above, we find the *smallest lexicographical solution.* Furthermore, the objective row can be removed from the tableau. This algorithm is known as the *lexicographic dual simplex algorithm.* Proofs can be found in [52].

Example 6.4.1 [Lexicographic Dual Simplex]

We apply the *lexicographic dual simplex algorithm* in order to find the lexicographically smallest solution (x_1, x_2) that satisfies:

$$\begin{aligned} -x_1 - x_2 &\leq -12 \\ -x_1 + x_2 &\leq -4 \end{aligned} \tag{6.14}$$

We add slack variables x_3 and x_4 as we did in example 6.3.1 and we add the trivial equations $x_1 - x_1 = 0$ and $x_2 - x_2 = 0$ resulting in the tableau:

	1	2	
1	−1	0	0
2	0	−1	0
3	−1	**−1**	−12
4	−1	1	−4

Observe that the tableau no longer has an objective row.

There are three possible candidates for the pivot: a_{11}, a_{12}, a_{21}.
To determine the pivot column we compute the vectors:

$$\frac{\mathbf{a}_1}{a_{11}} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \frac{\mathbf{a}_1}{a_{21}} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \frac{\mathbf{a}_2}{a_{12}} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ -1 \\ 0 \end{pmatrix}.$$

Clearly, the last vector is the lexicographically smallest one among the three vectors. So the pivot is a_{12}. After pivoting, we obtain the following tableau:

	1	3	
1	−1	0	0
2	1	−1	12
2	1	1	12
4	**−2**	1	−16

And after pivoting with element a_{21} we reach the final tableau:

	4	3	
1	−1/2	−1/2	8
2	4	−1/2	4
2	1/2	3/2	4
1	**−1/2**	−1/2	8

From the tableau, we read that $(x_1, x_2) = (8, 4)$ is the lexicographically smallest feasible solution.

□

In the preceding sections, we have explained the kernel of the algorithm used to find data dependencies. In the sections to come, we will solve the LP problem when:

- the optimal solution must be an integral solution.
- the constant vector **b** is symbolic

In the following section, we address the first topic and enter the field of *integer linear programming.* Then, we deal with the second topic which will lead us to the symbolic evaluation of expressions.

6.5 Integral solution

So far, we have assumed that the solutions of the system are tuples of real values. In this section, we restrict the solution to being integer. In particular, we are interested in finding the smallest lexicographical solution in the domain defined by the system $A\mathbf{x} \leq \mathbf{b}$ and nonnegativity constraints.

The general procedure is to solve the system first without the integer constraint. If the result is integral we have also solved the integer case. If the result is not integral, we add an inequality contraint to the system, which makes this solution infeasible but is satisfied by any feasible integral solution. We say that the inequality cuts off the nonintegral solution and call it a *cut plane.*

The nonintegral solution is found with the lexicographic dual simplex algorithm explained in the preceding section.

Example 6.5.1 [Nonintegral]

Suppose that we change in example 6.4.1 the constant b_2 of the second inequality from 4 to 5, the final tableau that results after the same pivot operations is:

	4	3	
1	$-1/2$	$-1/2$	$8\frac{1}{2}$
2	4	$-1/2$	$3\frac{1}{2}$
2	$1/2$	$3/2$	$3\frac{1}{2}$
1	$\mathbf{-1/2}$	$-1/2$	$8\frac{1}{2}$

The solution $(x_1, x_2) = (8\frac{1}{2}, 3\frac{1}{2})$ is the lexicographically smallest solution but is not an integral solution.

□

We need a cut that cuts off the optimal nonintegral solution but not any integer solution. The method that defines such a cut is the *Gomory's cutting plane method.*

Definition 6.5.1 [Gomory Fractional Cut]

Suppose that the optimal feasible solution is vector $\mathbf{b}$, with element b_r not an integer. Let f_{rj} and f_r denote the fractional parts of a_{rj} and b_r, respectively, that is,

$$\begin{aligned} f_{rj} &= a_{rj} - \lfloor a_{ij} \rfloor \\ f_r &= b_r - \lfloor b_r \rfloor \end{aligned} \tag{6.15}$$

With $x'_{m+j}, 1 \leq j \leq m+p$ as nonbasic variables and with x_{n+1} as additional integral variable, we define a *Gomory fractional cut* as:

$$x_{n+1} - \sum_{j=1}^{p} f_{jr} x'_j = -f_r \tag{6.16}$$

□

The fractions f_{fr} and f_r that appear in equation 6.16 are all positive. For instance, $-\frac{1}{3} - \lfloor \frac{1}{3} \rfloor = -\frac{1}{3} - (-1) = \frac{2}{3}$. This means that when we add this equation to the system, the optimal solution becomes infeasible, since $b_{n+1} = -f_r$ is negative.

Example 6.5.2 [Gomory Fractional Cut]

We apply the Gomory cutting plane method to the final system of example 6.5.1 of which the optimal feasible solution $(b_1, b_2, b_3, b_4) = (8\frac{1}{2}, 3\frac{1}{2}, 8\frac{1}{2}, 3\frac{1}{2})$ is not integral. The nonbasic variables are x_3 and x_4. Because every element of $\mathbf{b}$ is non integral, we can select any row r for constructing the cut. We choose b_3. We apply formula 6.15 to compute the fractional parts: $f_{31} = \frac{1}{2}, f_{32} = \frac{1}{2}$ and $f_3 = \frac{1}{2}$. This gives the Gomory cut, with variable x_5 as a slack variable:

$$x_5 - \frac{1}{2}x_3 - \frac{1}{2}x_4 = -\frac{1}{2} \tag{6.17}$$

□

We add a cut, as the last row, to the tableau and apply the lexicographical dual simplex method to get another optimal solution. If this solution is again non integral, we add another cut. This process continues until we reach an integral solution. The process is proven to be convergent, so that it reaches the integral optimal solution after a finite number of cuts [25].

Example 6.5.3 [Continued]

We add the cut of equation 6.17 as the last row to the tableau of example 6.5.1:

	4	3	
1	−1/2	−1/2	$8\frac{1}{2}$
2	4	−1/2	$3\frac{1}{2}$
2	1/2	3/2	$3\frac{1}{2}$
1	**−1/2**	−1/2	$8\frac{1}{2}$
5	**−1/2**	−1/2	−1/2

The pivot is a_{51} and after pivoting we get the tableau:

	5	3	
1	−1	0	9
2	1	−1	3
2	1	−1	3
1	−1	0	9
4	−2	1	1

The solution $(x_1, x_2, x_3, x_4, x_5) = (9, 3, 0, 1, 0)$ is integral and optimal. In this example, we have obtained the integral solution after adding only one cut. In general, we may have to add a number of cuts.

□

6.6 Parameterized constant vector

We now consider the case that vector $\mathbf{b}$ of the system is parameterized. It is a function of parameters whose values we do not know. However, we make some assumptions. First of all we assume that the parameters are nonnegative and, second, that the value of each element of $\mathbf{b}$ is defined by an affine expression of the parameters. Let there be q parameters. Let P stand for the vector of parameters, $\mathbf{e}_i$ for an integral vector of length q and d_i for an integer constant. Then each element b_i is defined by:

$$b_i(P) = \mathbf{e}_i P + d_i \tag{6.18}$$

The dual algorithm applies the operations *multiplication* and *addition* to compute the value of $\mathbf{b}$. Now the multiplication of $b_i(P)$ with a constant α is given by $\alpha\mathbf{e}_i P + \alpha d_i$. The addition of two elements $b_i(P)$ and $b_j(P)$ is equal to $(\mathbf{e}_i + \mathbf{e}_j)P + (d_i + d_j)$. This means that we can evaluate vector $\mathbf{b}$ symbolically and that after each pivot step vector $\mathbf{b}$ is of the form of equation 6.18.

There is, however, a problem. In the dual simplex algorithm, we must know the sign of the elements of $\mathbf{b}$ in order to select the pivot row. When the vector is parametrized, the sign depends on the values of the parameters, which we do not know. For example, suppose that N and M are nonnegative parameters and that element b_1 is defined by:

$$b_1 = 2N - M \tag{6.19}$$

Then the sign of b_1 depends on the values of the parameters N and M. Element b_1 is negative when $2N \leq M$, otherwise it is nonnegative.

Therefore when we have a parametrized constant vector, we need additional assertions about the value of the parameters in order to determine the sign of the elements of $\mathbf{b}$. We call this set of assertions the *context* of the system. The context is a system of affine linear inequalities of the parameters.

After each pivot operation new expressions $b_i(P)$ appear, of which we do not know the sign. Therefore, we split the problem into two cases. In the first case we assume $b_i(P)$ to be negative and add the inequality $b_i(P) < 0$ to the context. In the second case we assume b_i to be nonnegative and add $b_i(P) \geq 0$ to the context. Thus the problem branches into a number of mutually disjoint cases each with its own context defining the sign of $\mathbf{b}$. We solve the LP problem by the dual simplex algorithm for each case, separately, resulting in either an optimal feasible solution or the conclusion that the problem is infeasible. We denote the infeasible solution by $\perp$. The optimal feasible solution is the value of $\mathbf{b}$ in the final tableau. As $\mathbf{b}$ is an affine expression of the parameters, the optimal solution is thus an affine expression of the parameters. The context corresponding to each final tableau defines the values of the parameters under which the solution is valid.

We present the complete solution of the parametrized integer problem as a tree. The internal nodes of the tree are the inequalities of the contexts. The leaves of the tree are the solutions of the final tableau, which is either a vector of which the elements are affine expressions of the parameters or is infeasible (denoted by $\perp$). The context of a leaf is defined by the inequalities of the nodes of the path up to the root node of the tree.

By pivoting, we take the integer division of b_r. As integer division is not a linear operation, we linearize this term by introducing a new parameter p. Parameter p is equal to $div(b_r, pivot)$. We define p by two inequalities, according to procedure 5.19, which we add to the context.

In general, if the absolute value of the pivot is greater than one and the constant term of the pivot row is parametrized, we introduce a new parameter.

Procedure 6.6.1 [New Parameter]

Let r be the pivot row, and let the pivot element selected from this row be a_{rs}. Suppose that the absolute value of a_{rs} is greater than one and the b_r is an affine expression of the parameters.

Then we introduce a *new parameter* p that is defined by the integer division of b_r and a_{rs}:

$$p = div(b_r, a_{rs}) \tag{6.20}$$

The new parameter is added to the parameter vector P of the system and the following inequalities are added to the context:

$$\begin{array}{c} b_r - \|a_{rs}\|\ p \geq 0 \\ (\|a_r s\| - 1) - b_r + a_{rs} p \geq 0 \end{array} \tag{6.21}$$

□

Example 6.6.1 [Parameterized Constant Vector]

Let N be a nonnegative parameter, $N \geq 0$. Suppose that in the second inequality of example 6.4.1 the constant is not -4 but equal to $-N$:

$$-x_1 + x_2 \leq -N \tag{6.22}$$

As usual, we introduce slack variables x_3 and x_4 and construct the tableau:

	1	2	
1	−1	0	0
2	0	−1	0
3	**−1**	−1	−12
4	−1	1	$-N$

The context of this tableau is $N \geq 0$.

We pivot with $a_{32} = -1$ and obtain:

	1	3	
1	−1	0	0
2	1	−1	12
2	1	1	12
4	**−2**	1	$-N - 12$

In the case where $-N - 12 \geq 0$ all elements of **b** are nonnegative and the tableau is final. But this inequality can never be true when $N \geq 0$. So the context for thia case is empty. Remains the case $-N - 12 < 0$. We add this inequality to the context. As b_4 is negative we have to select a pivot from this row. The only negative element is a_{41}. After pivoting with this element we get the following tableau:

	4	3	
1	−1/2	−1/2	$(N + 12)/2$
2	1/2	−1/2	$12 - (N + 12)/2$
2	1/2	−1/2	$12 - (N + 12)/2$
1	−1/2	−1/2	$(N + 12)/2$

We see that the pivot operation causes an *integer division* of the expression $(N + 12)$ by 2 and that a multiple of the result is added to other elements of **b**. Now let p_1 be the *new*

parameter and its value equal to $div(N+12,2)$. When we substitute this parameter in the tableau we get:

	4	3	
1	$-1/2$	$-1/2$	p_1
2	$1/2$	$-1/2$	$12-p_1$
2	$1/2$	$-1/2$	$12-p_1$
1	$-1/2$	$-1/2$	p_1

Again we have to branch. When $12-p_1 \geq 0$, we are finished because element of $\mathbf{b}$ is nonnegative. When $12-p_1 \leq 0$ we have to do another pivot step.

This gives the following tableau:

	4	2	
1	-1	-1	12
2	0	-1	0
3	-1	-2	$-12+2p_1$
1	-1	-1	12

The solution is $(12,0)$ if $-12+N \geq 0$. There are no other cases left to solve.

Figure 6.2 gives the complete solution tree. The leaves of this trees are the solutions in the context defined by the path from a leaf node to the root of the tree.

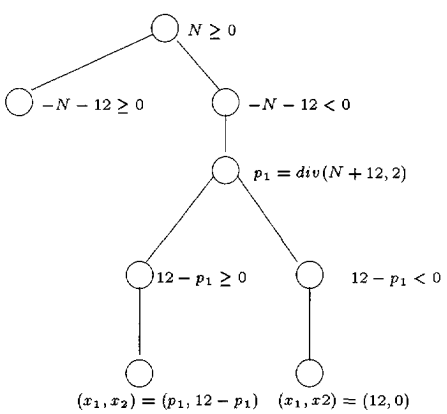

Figure 6.2: The solution tree in which the internal nodes model the inequalities of the context and the leaves the solution-vectors.

□

6.7 Conclusion

We have described how PIP finds the lexicographic minimum solution of a feasible set. However, to find data dependencies, we will need to find the lexicographical *maximum*. This is achieved by solving the problem for the negative values of $\mathbf{x}$, i.e. $-\mathbf{x}$, and introducing an unbounded positive *parameter* $\mathbf{P}$ to satisfy the non-negativity constraints after the substitution (solving for minimum of $(\mathbf{P} - \mathbf{x})$. Henceforth, we assume that PIP finds the *lexicographical maximum* of the problem.

The input of PIP is formed by a parameterized polytope. Because of the parameterization, the optimum solution is a vector of which the elements are affine expressions on the parameters. Because of internal branching, the solution returned by PIP is, in general, a solution tree in which the leaves are the solution vectors and the internal nodes define the context for which the solutions are valid.

A linear programming technique may sometimes turn out to be an unrealistic solution because of the huge amount of computation time needed to solve the LP problem. However, PIP is very well applicable, as for realistic problems the number of unknowns, which are the loop iterators and parameters of a nested loop program, are typically small.

As *HiPars* uses PIP to solve the data dependencies, the nested loop programs are limited to the affine ones. Another technique for analyzing data dependencies is described in [51]. This technique is based upon the Omega test and finds the lexicographical maximum of dependence relations by means of projections. The class of nested loop programs is also affine although some extensions are made to handle non-affine program fragments. However, they do not express the solution as functions as PIP does.

Chapter 7

Single Assignment Programs

7.1 Introduction

An application of PIP, *Parametric Integer Programming*, is to find expressions for the data dependencies inside static nested loop programs as described by Feautrier in [26]. As iterations are integer valued, the problem of data dependency is an *Integer Linear Programming* problem.

We solve separately the data dependency problem for each single read-write pair. PIP finds closed expression for the data dependencies. We represent the output of PIP by parse trees, which we call *solution trees.*

When a nested program has several LHS variables of the same name, we have to combine the solution trees for each individual read-write pair of an RHS variable. We call the process of combining trees *grafting.*

Grafting can result in large trees. To limit the size of the tree, *HiPars* removes dead branches and redundant nodes from the parse tree. We call this process *pruning. HiPars* prunes the tree after each graft operation.

Finally, *HiPars* adds the solution trees of the RHS variables to the parse tree of the original nested loop program and writes the resulting parse tree as a single assignment program (SAP), showing the data dependencies inside the nested loop program at the level of iterations.

7.2 PIP's Output

As explained in the previous chapter, the solution returned by PIP is, in general, a solution tree. The fact that the solution is a tree, derives from the branching inside PIP.

The solution vectors, represented by the leaves of the tree, are affine expressions in the parameters of the final tableau. We represent these parameters by the parameter vector P standing for the size parameters and a vector k_J standing for the remaining parameters. The elements of k_J are the control variable of the RHS variable that are entered as parameters to the context, and the additional parameters that PIP possibly has introduced to find an integral solution.

Thus a solution-vector is of the form:

$$d = A_d k_J + b_d \tag{7.1}$$

with b_d a parameterized constant with respect to the size parameters. Often, we write this equation in the form $d = D(k_J)$, with D an affine function, called the *dependency function.*

We define the *solution tree* produced by PIP.

Definition 7.2.1 [Solution Tree]

A *solution tree* T is a parse tree. The internal nodes of T are nodes of the types: 'If', 'Else' and 'Div'. These nodes represent the linear inequalities of the context.

Each leaf node of T represents a solution-vector d, which may be undefined (denoted by $\perp$).

The *domain* under which a solution-vector is valid is defined by a polytope e_d consisting of the inequalities represented by the nodes in its path to the root.

□

Next, we define a *dependency* by a solution-vector d and the polytope e_d under which the solution-vector is valid. We denote a *dependency* by the tuple $< e_d, d >$.

From the tree structure follows that the domains of dependence-vectors do not intersect, i.e. the domains are mutually disjoint. Moreover, they form a partition of the domain of the right-hand side variable. This means that a solution-vector exists for each iteration inside the iteration domain of the RHS variable.

We combine the solution trees for all cases of the lexicographical expansion into a single solution tree, which is a simple procedure because all cases of the expansion are mutually disjoint.

Example 7.2.1 [Output of PIP]

As illustration, we give the solution-trees produced by PIP for the second RHS variable `a(j,i+1)` inside the function call to `RotColumn` (program 2.3.3) inside program 5.6.1 'Rotate' listed in chapter 5.

We start by solving the data dependency problem for the LHS variable `a(i,j)` which first appears in the function call to `RotRow` (program 2.3.2). Figure 7.1 shows the solution-tree, which we denote by T_1.

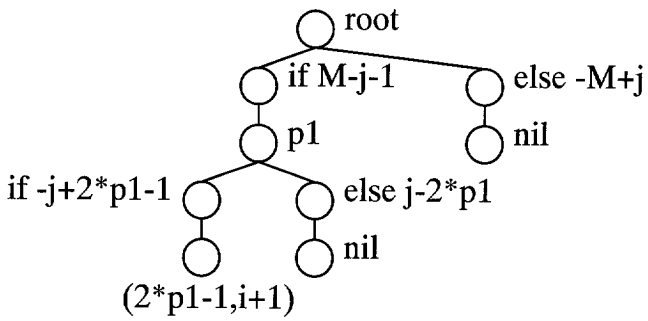

Figure 7.1: Solution-tree T_1.

Below, we have written the solution in pseudo MATLAB format:

```
if M-j-1>=0,
  p1=div(j+1,2);
  if -j+2*p1-1>=0,
      (2*p1 -1,i+1)
  else
      nil
  end
else
    nil
end
```

The tree has three solution-vectors $d_1 = (2p_1 - 1, i + 1)^T$, $d_2 = \perp$ and $d_3 = \perp$. The internal nodes are of types If-node, Else-node and Div-node and specify the domains under which the vectors are valid.

For instance, the solution vector d_1 is valid under the following constraints:

$$\begin{aligned} M - j - 1 &\geq 0 \\ j + 1 - 2p_1 &\geq 0 \\ -j + 2p_1 &\geq 0 \\ j + 1 - 2p_1 &\leq 0 \end{aligned}$$

The solution tree for the second LHS variable `a(i+1,j)` in the function call to `RotRow` is shown in figure 7.2.

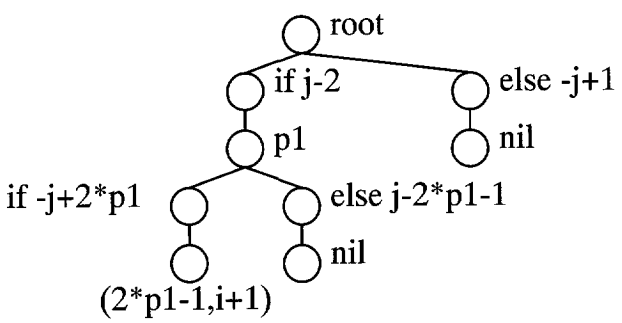

Figure 7.2: Solution-tree T_2.

This tree has three solution-vectors of which two are undefined. The solution-tree in pseudo MATLAB format is:

```
if j-2>=0,
  p1=div(j,2);
  if -j+2*p1>=0,
     (2*p1-1,i+1)
  else
    nil
  end
else
   nil
end
```

The solutions for the third and fourth LHS variable are both undefined ($\perp$).

□

7.3 Grafting Trees

In this section, we combine the solution trees belonging to a particular RHS variable.

Suppose that there are n LHS variables with the same name as the RHS variable. Then we solve first the data dependency problem for each read-write pair separately by PIP. Let the resulting solution-tree for each pair $< RHS, LHS_i >$ be denoted by $T_i, 1 \leq i \leq n$.

According to the third condition of data dependency, we must find the lexicographic largest solution-vector among the solution-vectors of the solution-trees T_i.

Each tree partitions the domain of the RHS variable. However, domains of vectors of the various trees may have a nonempty intersection. When there is a nonempty intersection, we have to select the lexicographically largest solution vector from the vectors valid in the intersection. In other words, we have to combine the partial solutions T_i into a single solution-tree for the RHS variable. In order to construct this solution-tree, we have to take the intersection of the domains among all the trees T_i and compare all the vectors defined in each intersection.

First, we describe how we select the largest of two solution-vectors of the verious trees.

Let T_1 and T_2 be solution trees of an RHS variable.
Let $s_1 = < e_1, d_1 >$ and $s_2 = < e_2, d_2 >$ be solutions of the trees T_1 and T_2, respectively. When $e_1 \cap e_2 \neq \emptyset$, if $d_1 \prec d_2$ we select d_2 else we select d_1:

$$\begin{array}{l} \textbf{if } d_1 \prec d_2 \textbf{ then} \\ \quad d_2 \\ \textbf{else} \\ \quad d_1 \end{array} \tag{7.2}$$

By definition, the undefined vector is the lexicographical smallest solution-vector. Thus $\perp \prec d$ is always true.

In order to represent the condition in equation 7.2 as a parse tree, we expand the lexicographical operator in all its cases. The number of cases is equal to the common nesting level of the iteration vectors corresponding to the left-hand side variables. We call the resulting tree the *expansion tree*, and denote it by E. The expansion tree is a solution-tree that has dependency vectors d_1 and d_2 as leaves. We call the operator that returns the expansion tree E of two solution-vectors the *expand* operator:

$$E = expand(d_1, d_2) \tag{7.3}$$

Example 7.3.1 [Expand Operator]

We derive the expansion tree of the dependence-vectors of the program 'Rotate' (5.6.1).

Both T_1 and T_2 define dependence-vectors for the RHS variable `a(j,i+1)`. The dependence-vector of T_1 is $d_1 = (2p_1-1, i+1)^T$. The dependence-vector of T_2 is $d_2 = (2p_2-1, i+1)^T$. We first determine the common nesting level of the iteration-vectors I_1 and I_2. The elements of I_1 are the loop iterators of the first loop stage as are the elements of I_2. So the common nesting level is two. The first case is:

$$2p_1 - 1 > 2p_2 - 1$$

The inequalities of the domains define the equations $2p_1 = j + 1$ and $2p_2 = j$, by which we substitute the variables p_1 and p_2. This gives: $j > j - 1$. As this is always true, d_1 is the largest solution.
The second case is:

$$\begin{aligned} 2p_w - 1 &= 2p_r - 1 \\ i + 1 &< i + 1 \end{aligned}$$

which is always false. Thus the result of the graft operator on d_1 and d_2 is a tree with just one leaf: $E = d_1$.

□

Now, we give the procedure for combining two solution-trees T_1 and T_2 of a right-hand side variable. Let d_1 be an arbitrary solution-vector of T_1 and d_2 be an arbitrary solution-vector of T_2. We apply the expand operator on all the possible pairs of solution-vectors (d_1, d_2). The resulting expansion trees will become subtrees in the combined tree.

The procedure in terms of solution-trees is given in procedure 7.3.1. We call this procedure the *graft* operator. Thus given two solution-trees T_1 and T_2, the combined tree T is given by:

$$T = graft(T_1, T_2) \tag{7.4}$$

Procedure 7.3.1 [Graft Operator]

The procedure to *graft* two solutions-trees T_1 and T_2 starts with a DFS to find all the leaves of T_1. Each leaf d_1 is substituted by the solution-tree T_2. After which, the procedure searches the leaves of the subtree T_2. The procedure substitutes each leaf d_2 by adding the expansion tree returned by the *expand* operator on d_1 and d_2. The procedure is outlined below.

Procedure Graft Solution Tree
for each leaf d_1 in T_1 {
    graft the tree with T_2
    for each leaf d_2 in T_2 {
        graft the tree with $expand(d_1, d_2)$
    }
}

□

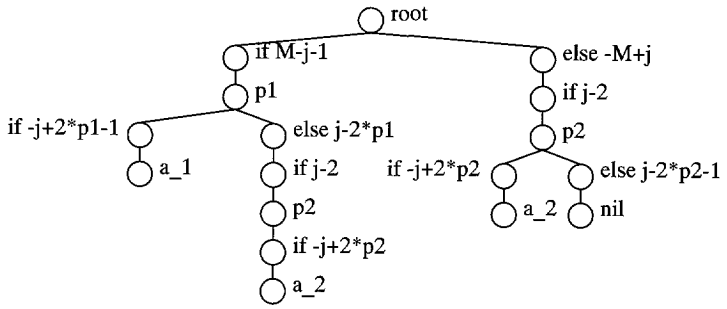

Figure 7.3: The tree after grafting and pruning of the solution trees T_1 and T_2.

Figure 7.3 shows the tree after applying the graft operator on the solution-trees T_1 and T_2 given in the example.

A tree returned by the graft operator is again a solution tree. So we may apply the graft operator on the tree again, which gives us the following procedure for combining several trees.

Let T_0 be initially be defined as $\perp$. Let $T_i, 1 \leq i \leq n$ be solution-trees of an RHS variable. Then we get the combined tree by applying the graft operator recursively [26].

$$T_n = graft(T_{n-1}, T_i) \tag{7.5}$$

The graft operator is associative. This means that we may graft the trees T_i in any order although the number of branches of the final tree may depend on the ordering.

After applying the recursive procedure to all solution-trees T_i, we obtain the *complete solution tree* for an RHS variable. We graft this tree in place of the RHS variable-node in

the parse tree of the original program. Figure 7.5 shows the parse tree of program 'rotate' with the solution-trees found for all the RHS variables grafted on.

7.4 Pruning Trees

Straightforward grafting of the solution-trees may lead to there being many branches. Suppose that there are n solution-trees and that each of them has m leaves. Each time we apply the graft operator we replace every leaf by the next solution tree. This goes on for n times, resulting in at most:

$$m^n \tag{7.6}$$

branches in the final solution tree. The number of branches grows thus exponentially.

We reduce the number of branches considerably by eliminating dead leaves. A dead leaf is a solution that will never be reached because of conflicting inequalities in its control path, i.e., its domain is *empty*. The dead leaves do not contribute and may be pruned from the tree. We use the Omega test to determine whether a domain is empty [57].

By pruning the tree after *each* graft operation, *HiPars* expands only branches that define valid domains.

Apart from the dead leaves, *HiPars* removes also all the nodes in the path or branch until the first *sibling* in the path to the root of the tree is reached.

Procedure 7.4.1 [Prune Operator]

Let T be a parse tree.

The procedure to prune a tree starts by searching for all leaves in T. The procedure calls for each leaf the procedure 5.3.1 to find its domain (polytope) and test whether the domain is empty. If the domain is empty, the procedure removes the nodes of the dead branch.

```
for each leaf l in T {
    derive domain by procedure 5.3.1
    check domain with the Omega Test
    if empty domain {
      prune tree from dead branch
    }
}
```

□

HiPars also prunes the nodes that are *redundant*. A node is redundant when its linear inequalities are redundant of all the dependence-vectors that are descendants of that node. *HiPars* uses three simple procedures to check for redundancies. First, it prunes If-nodes when the same inequality is annotated by a node at a lower level in the tree.
Second, it prunes identical integer division operators introduced by PIP. As the solution trees are solved independently, it may happen that new parameters in different solution-trees are the same.
Third, it prunes unnecessary branching in the parse tree. A branch is represented in the tree by an If-node and a corresponding Else-node. When the subtree of the If-node is identical to the subtree of the Else-node, the branch is unnecessary and can be removed.

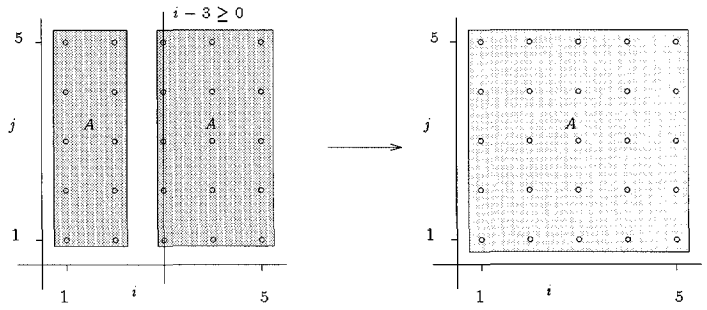

Figure 7.4: Removing a redundant branch.

Figure 7.4 shows, on the left side, a domain cut in two parts by the inequality $i - 3 \geq 0$. The subtrees of both parts are equal to A. Therefore, the branch is redundant and we can remove it. This is depicted on the right-hand side of figure 7.4. Thus the branch as well as one subtree is removed.

7.5 Single assignment program

In the previous section, we constructed a parse tree from a nested loop program that describes all the data dependencies. In this section, we write this parse tree as a *single assignment program*.
The definition of a Single Assignment Program (SAP) is given in [92].

Definition 7.5.1 [Single Assignment Program]

A **Single Assignment Program** is a program where every variable is assigned one value only during execution of the algorithm.

□

In order to obtain the single assignment program, we substitute LHS variables, with name v and indexing function f, for *single assignment* variables [13], with unique name v_α and the identity as indexing function.

Next, we obtain the single assignment program by traversing the parse tree, in DFS order, and writing down the MATLAB statements referred to by the nodes.

Nodes representing the solution-vectors are written down as follows. When arriving at a solution-vector $D(k_J)$ of a particular RHS variable with name v, we write the assignment statement, with $\beta \in \mathbf{Z}$ indicating the argument position:

$$[in_\beta] = \texttt{ipd}(v_\alpha(D(k_J)))$$

where ipd is an identity function.

When the solution-vector is undefined, we write down the same assignment statement but with variable $v_\alpha(D(k_J))$ replaced by the original RHS variable $v(g(J))$.

Typically, these statements are placed in the body of a block of nested conditional and transformation statements, which define the iteration domain for which the dependency is valid. The operators in the transformation statements are integer division operators.

The single assignment program is a convenient output format because we can directly read the dependencies from the code. In addition, the SAP program is an executable program and is functionally equivalent to the nested loop program from which it is derived.

When we write down the solution tree of program 5.6.1 'Rotate', we get the single assignment program 7.6.1. This program has been generated automatically by the tool 'HiPars' [36].

The names of the single assignment variables of the row rotations are a_1 and a_2. The names of the single assignment variables of the column rotations are a_3 and a_4. Observe that the indexing function of each LHS variable is the identity function. This means that the variables of these arrays are assigned a value only once, which is in agreement with the definition of a SAP. Note also that the original variables named a have become input variables.

There are no data dependencies for the row rotations as all inputs of the function RotRow are input variables. The function RotColumn is dependent on RotRow as it takes values of the single assignment variables a_1 and a_2 as inputs. For instance, the value of input argument in1 of function RotColumn is the value of variable a_1 or a_2 depending on the conditional statements.

7.6 Conclusion

A complete dependence analysis of the program involves finding the dependencies for all RHS variables appearing in the function call statements. An RHS variable can only be dependent on a left-hand side (LHS) variable of the same name. If there are more LHS variables of the same name, *HiPars* finds first the solution of the RHS variable with each of the LHS variables separately. *HiPars* calls PIP for each read-write pair, which returns the solution in the form of index vectors and the domains for which the solutions are valid [25] [26]. The index vector may be undefined, which means that the RHS variable does not depend on the LHS variable. After applying PIP for all the LHS variables, *HiPars* determines the lexicographically largest index vector among the solution-vectors, which is the *dependency*. Dependencies are linear functions on the iterators, parameters and variables standing for the integer divisions. The complete solution of the dependence analysis of a single RHS variable may consists of multiple dependencies defined on mutually exclusive iteration domains [37].

The output of HiPars is a single assignment program, which is a functional equivalent with the original nested loop program. The indexing functions of the RHS variables inside the SAP are the data dependencies, which we wanted to find.

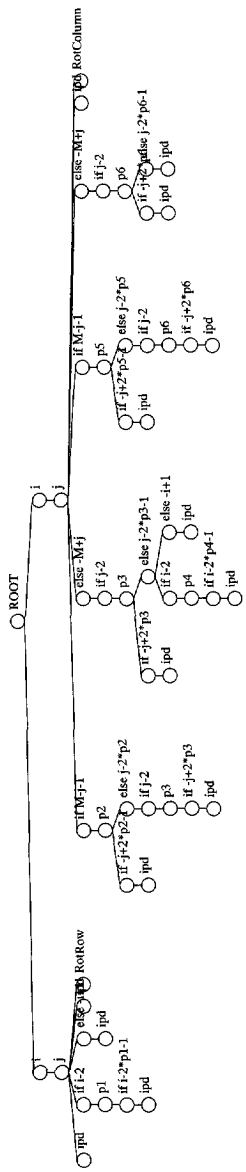

Figure 7.5: The parse tree of program 5.6.1 'Rotate' onto which the solution-trees for all the read-write pairs are grafted.

Program 7.6.1 [Single Assignment Program of 'Rotate']

```
for i=1 : 2 : M-1,
  for j=1 : 1 : M,

    [in0] = ipd(th1(i));

    if i-2>=0,
      p1=div(i,2);
      if i-2*p1-1>=0,
        [in1] = ipd(a(i,j));
      end
    else
      [in1] = ipd(a(i,j));
    end

    [in2] = ipd(a(i+1,j));

    [a_1(i,j),a_2(i,j)   ] = RotRow(in0,in1,in2);

  end
end
for i=1 : 2 : M-1,
  for j=1 : 1 : M,

    if M-j-1>=0,
      p2=div(j+1,2);
      if -j+2*p2-1>=0,
        [in0] = ipd(a_1(2*p2-1,i));
      else
        if j-2>=0,
          p3=div(j,2);
          if -j+2*p3>=0,
            [in0] = ipd(a_2(2*p3-1,i));
          end
        end
      end
    else
      if j-2>=0,
        p3=div(j,2);
        if -j+2*p3>=0,
          [in0] = ipd(a_2(2*p3-1,i));
        else
```

```
          if i-2>=0,
            p4=div(i,2);
            if i-2*p4-1>=0,
              [in0] = ipd(a(j,i));
            end
          else
            [in0] = ipd(a(j,i));
          end
        end
      end
    end
    if M-j-1>=0,
      p5=div(j+1,2);
      if -j+2*p5-1>=0,
        [in1] = ipd(a_1(2*p5-1,i+1));
      else
        if j-2>=0,
          p6=div(j,2);
          if -j+2*p6>=0,
            [in1] = ipd(a_2(2*p6-1,i+1));
          end
        end
      end
    else
      if j-2>=0,
        p6=div(j,2);
        if -j+2*p6>=0,
          [in1] = ipd(a_2(2*p6-1,i+1));
        else
          [in1] = ipd(a(j,i+1));
        end
      end
    end
    [in2] = ipd( th2(i));

    [a_3(i,j),a_4(i,j)] = RotColumn(in0,in1,in2);

  end
end
```

□

Chapter 8

Linearly Bounded Lattices

8.1 Introduction

In this chapter, we take a closer look at the *single assignment program* (SAP) produced by the tool HiPars, whose working we have explained in the preceding chapters. From the user point of view, *HiPars* is a tool that takes as input a nested loop program and outputs a program that is in single assignment form. Here we explain the output of *HiPars* as it presents itself to the user.

We express the control statements inside the single assignment program, specifying the domains of dependencies as linearly bounded lattices [78]. Using linearly bounded lattices, we not only have a sound mathematical specification of the domains but also a way to interpret the result geometrically, making the data dependence structure easier to understand.

In the next chapter, we convert the single assignment program into a piecewise regular dependence graph whose description is based on linearly bounded lattices.

The outline of this chapter is as follows. In section 8.2, we convert an example nested loop program into single assignment form and give the resulting single assignment program, specifying the data dependencies.
In section 8.3, we give the definition of linearly bounded lattices. In the previous chapters, we will specified sets of iterations by polytopes. This may lead to high dimensional polytopes, which are hard to interpreted. In this chapter, we specify the sets of iterations by linearly bounded lattices. By using lattices the iteration domains are easier to interpret. In sections 8.4 and 8.5, we optimize the description of the domain by using the Hermite Normal Form, leading to a domain description with a lower dimensional polytope, and specify the lattice corresponding to this polytope. The result is a more compact description of the single assignment program, which is easier to comprehend.

8.2 Single Assignment Program

As example, we convert program 8.2.1 into a single assignment program by *HiPars*. The example program is just a demonstrator and is not taken from the SVD algorithm.

The program has two loop statements both with stride two and has modulo operators in its conditional expressions.

Program 8.2.1 [Demonstrator]

Let M be a parameter.
Let a be a variable array.
Let F1 and F2 be functions.

```
for i = 1 to M step 2,
    for j = 1 to M step 2,
        if mod(i,3) ==0,
           [a(i,j)] = F1();
        end
        if mod(j,3) ==0,
           [ ] = F2(a(i,j));
        end
    end
end
```

□

We will convert it in single assignment form by *HiPars*. Before looking in detail at the SAP program, we make some general remarks about the format in which *HiPars* writes a single assignment program.

LHS variables have unique names and are fully indexed. So the indexing functions of these variables are identity functions.

RHS variables are either single assignment variables or are program input variables. The indexing functions of the single assignment variables are the dependence functions. The indexing functions of the input variables do not define dependence relations between the program iterations. We assume the input variables to be initialized. The iteration domains of the dependencies are represented in the SAP in the form of control statements, which take the form of conditional and integer division statements.

The SAP produced by HiPars is program 8.2.2. It has one single assignment variable a_1, which is a two-dimensional array.

Program 8.2.2 [Single Assignment Program]
Let M be a parameter.
Let F1 and F2 be two functions.
Let a_1 be a two-dimensional variable and in0 a temp variable.
Let q1, q2, q3 and q4 be control variables.

```
for i=1 to  M step 2,
  for j=1 to M step 2,

    if mod(i,3) == 0,
      [ a_1( i,j ) ] = F1( );

    end

    if mod(j,3) == 0,

      q1=div(i+1,2);
      if -i+2*q1-1>= 0,
        q2=div(j+1,2);
        if -j+2*q2-1>= 0,
          q3=div(2*q1+2,3);
          if -i+3*q3-3 >= 0,
             q4 = div(q3,2);
             in0 = a_1(i,j);
          else
             in0 = a(i,j);
          end
        else
          in0 = a(i,j);
        end
      else
           in0 = a(i,j);
      end

      F2(in0);

    end
  end
end
```

□

The input argument of function F2 is the value of the temp variable in0, which is conditionally assigned in the block of control statements preceding the function call. The control structure is quite complicated and consists of three conditional statements and four integer division transformation statements, introducing the additional control variables q1,q2,q3 and q4. Note that the index transformations are nested.
Figure 8.1 shows the iterations domains of F1 and F2.

The *div* operators are produced by PIP, which *HiPars* calls to solve the integer programming problem in order to find expressions for the data dependencies inside the nested loop program.

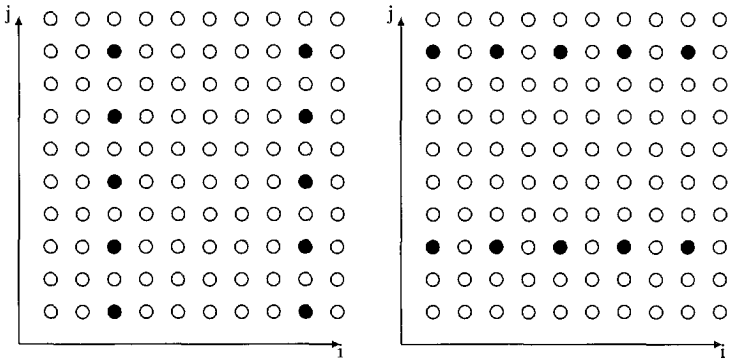

Figure 8.1: The iteration domains of F1 (left) and F2. (right)

The assignment to variable in0 with the value of a_1(i,j) defines a data dependency between the function F2 and the function F1.

We specify the set of iterations at which F2 depends on F1. This set is equal to the set of iterations for which the variable a_1(i,j) is reached in the SAP.

Because a single assignment program belongs to the class of nested loop programs we defined in chapter 3, we can apply procedure 5.3.1 for deriving the polytope for variable a_1(i,j). As explained, the procedure sets up a parse tree, annotates the nodes with linear inequalities, and constructs the polytope.

For ease of reference, we give here once more the linear inequalities by which an integer division operator $q = div(a, b)$ is annotated in the parse tree:

$$0 \leq a - b * q \leq (b - 1) \tag{8.1}$$

Example 8.2.1 [Index Transformation]

In program 8.2.2, the value of control variable q3 is defined by the assignment statement:

```
q3 = div(2*q1+2,3);
```

In inequality 8.1, we substitute $a = 2q_1 + 3$, $b = 3$, $q = q_3$, and obtain:

$$0 \leq (2q_1 + 2) - 3q_3 \leq 2 \tag{8.2}$$

Figure 8.2 shows the points $(q_1, q_3)^T$ of the polytope for $1 \leq q_1 \leq 10$. The points are:

$$(1,1), (2,2), (3,2), (4,3), (5,4), (6,4), (7,5), (8,6), (9,6), (10,7)$$

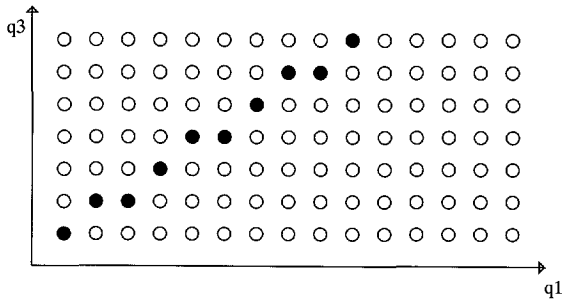

Figure 8.2: The values of control variable q3 defined by `div(2*q1+2,3)`.

□

The piece of code defining the domain of variable `a_1(i,j)` has four control variables q1, q2, q3 and q4, each defined by an integer division operator. In addition, the program has two loop iterators: i and j. So the resulting polytope has six variables and lies thus in a six-dimensional space $\mathbf{Z}^6$. The polytope is:

$$\begin{aligned} i + 1 - 2q_1 &\geq 0 \\ i + 1 - 2q_1 &\leq 1 \\ -i + 2q_1 - 1 &\geq 0 \\ j + 1 - 2q_2 &\geq 0 \\ j + 1 - 2q_2 &\leq 1 \\ -j + 2q_2 - 1 &\geq 0 \end{aligned}$$

$$
\begin{aligned}
2q_1 + 2 - 3q_3 &\geq 0 \\
2q_1 + 2 - 3q_3 &\leq 2 \\
-i + 3q_3 - 3 &\geq 0 \\
q_3 - 2q_4 &\geq 0 \\
q_3 - 2q_4 &\leq 1
\end{aligned}
$$

8.3 Linearly Bounded Lattices

We are interested in the values of the loop variables for which statements inside a program are reached. The example above shows that polytopes can be complicated, which makes it not easy to derive this information. In this section, we distinguish, therefore, between the loop variables and control variables standing for the integer division.

To specify the values of the loop iterators, we define domains by *linearly bounded lattices* [23] [77] [78].

Definition 8.3.1 [Linearly Bounded Lattice]

Let e be a polytope in $\mathbf{Z}^m$. Let k be the vector of control variables ranging over polytope e. Let I be a vector taking values in $\mathbf{Z}^n$.

With L a matrix of size $n \times m$ and O an integer vector, we define a *linearly bounded lattice* D as the set of points $I = Lk + O$ with $k \in e$:

$$D = \{\ I \mid I = Lk + O \ \wedge \ k \in e\ \} \tag{8.3}$$

□

We say that the lattice points I are generated by the columns of L, with the variables of k bounded by the polytope. We call O the *offset* of the lattice and the columns of L the *lattice vectors.*

Now let the elements of I be the loop variables. We split the lattice matrix L as $[\ L_1 L_2\ ]$, with L_1 an $n \times n$ identity matrix and L_2 a $n \times (m - n)$ zero matrix,

$$L = \begin{pmatrix} 1 & & & & 0 & 0 \\ & 1 & & & 0 & 0 \\ & & . & & . & . \\ & & & . & . & . \\ & & & 1 & 0 & 0 \end{pmatrix} \tag{8.4}$$

and specify the values of the iterators by:

$$I = [L_1 L_2]k + O$$

The lattice specifies the relation between the index points I in $\mathbf{Z}^n$ and the points k in $\mathbf{Z}^m$, with $m \geq n$. The decomposition is valid when the variable of k belonging to L_2 are defined by integer division operators. Each null column in L corresponds to a variable introduced by an integer division operator. When there are integer division operators, the lattice specification takes the form of a projection. The points I are obtained by projecting the points in the polytope onto the iteration space. The fact that we may project the polytope follows from the definition of integer division. The number of points in polytope e equals the number of points of domain D. Thus for each iteration in D corresponds only one point in polytope e.

Example 8.3.1 [Lattice]

The lattice of the iteration domain for variable `a_1(i,j)` is:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{8.5}$$

Figure 8.3 shows the iteration points for which function F2 depends on F1.

□

8.4 Reducing the dimension of the polytope

By exploring the fact that some of the control variables of the polytope originate from integer division transformations, we can reduce the number of variables of the polytope considerably. Often, we need no more variables than the nesting level of the original statement in the program.
The method is based on the *Hermite normal decomposition* [52]. Other approaches can be found in [49] [57].

Let $Q = (q_1, .., q_m)^T$ be the vector of variables defined by the division operators. To find the lattice defined by the integer divisions and other inequalities involving the variables of Q, we write the inequalities defining *div*'s as equations by setting the constant $(b - 1)$ in

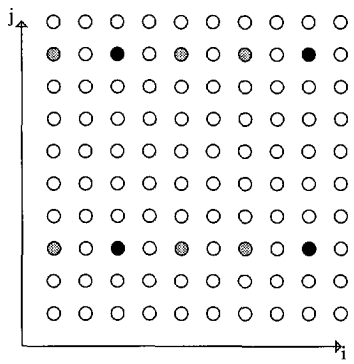

Figure 8.3: F2 depends on F1 at the iterations indicated by the black dots.

equation 8.1 to zero.
Let N be the matrix of which the rows are the normals of these equations. Let I be the vector of the loop iterators. We write the system of equations defined by the *div* operators as:

$$N \begin{pmatrix} I \\ Q \end{pmatrix} = 0$$

Example 8.4.1 [System of Equations]

With $I = (i, j)^T$ and $Q = (q_1, q_2, q_3, q_4)^T$, the system of equations corresponding to program 8.2.2 is:

$$\begin{aligned} i - 2q_1 &= 0 \\ j - 2 * q_2 &= 0 \\ 2q_1 - 3q_3 &= 0 \\ q_3 - 2q_4 &= 0 \end{aligned}$$

Thus matrix N is

$$N = \begin{pmatrix} 1 & 0 & -2 & 0 & 0 & 0 \\ 0 & 1 & 0 & -2 & 0 & 0 \\ 0 & 0 & 2 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

□

We assume that the system has a solution. Otherwise, we would have removed this piece of code from the program by dead code elimination procedures.

The system has m equations in $n + m$ variables. Because each row introduces a variable q_k it follows that the rows of N are independent. The null-space of the system is thus n-dimensional, equal to the dimension of the iteration space. We call the variables corresponding to the null-space the *free variables* of the system.

To find the solution, we use the *Hermite normal decomposition* [52]. This procedure gives us two unimodular matrices C_1 and C_2 such that:

$$N[C_1 C_2] = [H0]$$

in which the matrix H is called the Hermite normal form of N. Matrix H is nonsingular and has thus an inverse, because the rows of N are linearly independent. Observe that matrix C_2 consists of the vectors of the n null-space vectors of N as $NC_2 = 0$. Therefore, any linear combination of the vectors of C_2 added to a given solution s will also be a solution of the system. Because we are only interested in the values of I, we decompose matrix C_1 into C_{11}, size n by n, and C_{12} and decompose matrix C_2 into matrices C_{21} and C_{22} as follows:

$$C = \begin{bmatrix} C_1 & C_2 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{21} \\ C_{12} & C_{22} \end{bmatrix}$$

Now, the columns of matrix C_{21} are the lattice vectors. Thus the Hermite normal form gives us directly lattice matrix L defined by the *divs*.

Example 8.4.2 [Hermite Normal Form]

The Hermite normal decomposition of matrix N gives:

$$C_1 = \begin{pmatrix} 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

and matrix C_2:

$$C_2 = \begin{pmatrix} 6 & 0 \\ 0 & 2 \\ 3 & 0 \\ 0 & 1 \\ 2 & 0 \\ 1 & 0 \end{pmatrix}$$

The values of the iteration vector $I = (i, j)^T$ are generated by the matrix C_{21}. With K_f the vector of free variables, we write I, with offset O still to be determined, as:

$$I = \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix} K_f + O$$

□

8.5 Lattice Offset

Next, we have to find the lattice offsets. Let $B = (b_1, .., b_m)^T$ be a constant vector consisting of the divisors of the integer divisions, with remainder r_k between $0 \leq r_k < b_m$. An offset O must first of all be an integral solution of the system:

$$\mathbf{0} \leq N \begin{pmatrix} O \\ Q \end{pmatrix} < B \tag{8.6}$$

Apart from these inequalities, there may be other inequalities in the program that restrict the value of Q. Inequalities not involving Q are disregarded as they do not affect the lattice offset.

Let $< N_q, B_q >$ be the system of all inequalities involving Q. We assume that $N_q C_2 = 0$. When this assumption is satisfied, we use the vectors of C_{21} as lattice vectors because the variables corresponding to C_{21} are free.

Let K_b be the vector of variables corresponding to matrix C_1 and let K_f be the vector of variables corresponding to matrix C_2.

We define $(O, Q)^T$ as

$$\begin{pmatrix} O \\ Q \end{pmatrix} = [C_1 C_2] \begin{pmatrix} K_b \\ K_f \end{pmatrix} \tag{8.7}$$

and substitute it in the polytope:

$$N_q \begin{pmatrix} O \\ Q \end{pmatrix} \geq B_q \tag{8.8}$$

resulting in the polytope:

$$N_q C_1 K_b \geq B_q \tag{8.9}$$

This polytope defines all the lattice offsets $O = C_{11}K_b$ and we call it the *lattice offset domain.* The number of offsets depend on the value of the divisors $b_j, 1 \leq j \leq m$. The lattice corresponding to the polytope is defined by:

$$I = C_{21}K_f + O \tag{8.10}$$
$$O = C_{11}K_b \tag{8.11}$$
$$N_q C_1 K_b \geq B_q \tag{8.12}$$

The lattice is bounded by the other inequalities in the nested loop program, such as loop bounds. These inequalities together form a polytope and define with the lattice an iteration domain.
A special case is when the offset domain contains a single point. Then the lattice descriptions reduces to $I = C_{21}K_f + O$, and we do not have to enumerate the lattice offset domain.

Example 8.5.1 [Lattice Offset]

In program 8.2.2 there are three conditional statements defining inequalities that depend on Q:

$$-i + 2 * q1 - 1 \geq 0$$
$$-j + 2 * q2 - 1 \geq 0$$
$$-i + 3 * q3 - 3 \geq 0$$

After the substitution $I = C_{11}K_b$ and $Q = C_{12}K_b$, we get inequalities in variables of K_b:

$$-k_1 \geq 1$$

$$-k_2 \geq 1$$

$$-k_1 - k_3 \geq 3$$

By applying the same substitution in equation 8.6, we obtain the inequalities:

$$-1 \leq k_1 \leq 0$$

$$-1 \leq k_2 \leq 0$$

$$-2 \leq k_3 \leq 0$$

$$-1 \leq -k_3 + 2k_4 \leq 1$$

After some computation we find that $K_b = (-1, -1, -2, -1)^T$ is the only solution. So that the offset is

$$O = C_{11} K_b = (3, -1)^T$$

.

□

8.6 Result

When the column vectors of a lattice L of an iteration domain form a basis of the iteration space, the iteration domain becomes easier to interpret.

In the case of the example, the values that iterator vector $I = (i, j)^T$ takes on are defined by the lattice, bounded by the polytope e_d:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} + \begin{pmatrix} 3 \\ -1 \end{pmatrix} \tag{8.13}$$

Below, we have listed the program 8.6.1, which we obtained from single assignment program 8.2.2 by specifying the lattice with two modulo operations.

Program 8.6.1 [Program with modulo operators]

Let `M` be a parameter.
Let `F1` and `F2` be two functions.
Let `a_1` be a two-dimensional variable and `in0` a temp variable.

```
for i=1 to  M step 2,
  for j=1 to M step 2,

    if mod(i,3) ==0,
      [a_1(i,j)] = F1( );
    end

    if mod(j,3) == 0,

      if mod(i+3,6) == 0,
       if mod(j+1,2) == 0,
             in0 = a_1(i,j);
```

```
        else
            in0 = a(i,j);
        else
            in0 = a(i,j);
        end

        F2(in0);

    end
  end
end
```

□

8.7 Conclusion

In this chapter, we have explained the relation between integer divisions inside the single assignment programs generated by *HiPars* and linearly bounded lattices. The SAP code produced by *HiPars* can be complicated and hard to interpret because of possibly many additional control variables introduced in the domain descriptions. We can optimize the code by reducing the number of variables by applying the Hermite normal form on the system of equations on the these variables, which are defined by integer division operators. This decomposition leads to matrices C_1 and C_2, with corresponding variable vectors K_b and K_f. Matrix C_2 defines the lattice vectors with the variables of K_f as free variables. The domain of lattice offsets is formed by a polytope in variables of K_b. The polytope is characterized by matrix C_1 and inequalities on the variables standing for the integer divisions.

As a result, we can transform polytopes of iteration domains into linearly bounded lattices characterized by polytopes of lower dimensions and a lattice forming the basis of the iteration domain. Finally, the result can be used to simplify the single assignment program by using modulo operations.

Chapter 9

Piecewise Regular Dependence Graphs

9.1 Introduction

The single assignment form in which the result of the data dependence analysis is written, is, in fact, an intermediate format. In this chapter, we convert single assignment programs into dependence graphs (DG). A dependence graph consists of a set of nodes and a set of edges. Nodes stand for function evaluations. Edges define precedence relations between the nodes and represent the argument passing between functions inside the single assignment program. A DG is an applicative specification of the functional behavior of the single assignment program, which implies that the model is free of memory and control.

Below, we first consider programs consisting of a sequence of assignment statements only and derive a DG by enumerating the function calls. After that, we consider programs with control statements. We restrict ourselves to the class of nested loop programs as defined in chapter 3. Because the number of function evaluations of such programs may be large and may depend on size parameters, we can no longer use enumeration. Instead, we exploit the regularity of the DG and group edges and nodes in regular pieces, which we derive systematically from the control structure of the program. We refer to DGs specified in this way as *piecewise regular dependence graphs.*

The derivation of the DG from an SAP can be regarded as a refinement step. A functional specification in the form of a single assignment program is refined structurally into a graph of nodes and edges. In chapter 15, we show that this refinement may be applied on a node at any level of hierarchy. Here we assume the refinement to take place on a certain level of hierarchy and treat elements of the DG, i.e., edges, nodes and ports as elementary elements, without bothering about what is inside them.

The outline of this chapter is as follows. In section 9.2, we give formal definitions of the

elements of a dependence graph, its nodes, ports and edges. As example, we specify a DG by enumeration.
In section 9.3, we derive piecewise regular DGs from programs which contain also control statements. For this purpose, we have implemented the tool *sap2dg* that converts a SAP into a DG. The tool takes as input a SAP, typically generated by *HiPars*, and outputs the equivalent DG description. As example, we convert a part of the SVD algorithm 2.3.5 into a dependence graph. The description is written in the HiFi design language [80], which is based on Objective C [69] and describes a data structure in which elements of the DG are represented by objects. The transformation tools of the HiFi system operate on this data structure. In the appendix, we have listed several objects representing dependence graph elements.
In section 9.4, we make some remarks about the optimization of the DG description.

9.2 Dependence Graphs

According to its definition, a single assignment program is a program in which every variable is assigned one value only during its execution [92].
Because of the single assignment property, we can derive the data dependencies directly from the program description. The tool *HiPars* outputs SAPs with the left-hand side variables fully indexed. As a result, read accesses to these single assignment variables correspond with the data dependencies inside the program.

To make the dependence structure more explicit, we represent the single assignment program as a dependence graph. The elements of the dependence graph are *nodes*, *ports* and edges. The dependence graph is functionally equivalent to the single assignment program.

In [80] a primitive or elementary node is defined by input and output ports and a functional behavior specifying the input-output behavior of the node.
To specify functional behavior, more formally, we introduce the notion of *type*. A *type* is a set of values. By declaring an object of *type* T, we specify that the object takes only values ranging over the set of values T. Examples of data types are 'Integer' and 'Float'. Structured types are formed by product sets of types, denoted in the familiar tuple notation.

Definition 9.2.1 [Port]

Let l be a name.
Let T be an arbitrary type with $t \in T$ an arbitrary value.

We define a *port* p by the name-type pair: $< l, T >$. The value of port p is the name-value pair $< l, t >$. We T the *type* and t the *data value* of port p.

□

The name of a port must be a unique identifier among the set of the node's input or output ports.

Based on definition 9.2.1, we define an elementary node.

Definition 9.2.2 [Elementary Node]

A node N consists of input and output ports and a *functional behavior.*

Let I be the *input domain* formed by the product set of the types of the input ports of node N. Similarly, let O be the *output domain* formed by the product set of the types of the output ports of N.

We define the function f_b from I to O, with $i \in I$ and $o \in O$:

$$I \mapsto O : o = f_b(i) \tag{9.1}$$

as the *functional behavior* of node N.

□

The behavior of a node according to this definition is deterministic. Values of the output ports of a node are completely determined by the current input values. In other words, the output of an elementary node does not depend on the input of the past (no hidden states).

For instance, we can model a multiplication operation by a node with two input ports a and b and one output port y and define f_b by the expression:

$$y = a.b$$

In this example the functional behavior is specified by a simple expression. In general, we specify the functional behavior f_b of a node by an algorithm, in the form of a MATLAB program, which gives the constructive procedure to compute the output from the input of the node. In program 9.2.1 below, we have listed the algorithm of the function 'RotRow' called by the SVD algorithm. It is the same algorithm as the one given in chapter 2 but here expressed in the functions of the math library listed in table 1.2 in the introductory chapter. It illustrates that a node may stand for complete algorithms as long as the algorithm has a functional behavior.

Example 9.2.1 [Node 'RotRow']

We specify node 'RotRow' by three input ports, p_{i1}, p_{i2}, and p_{i3},and two output ports p_{o1} and p_{o2} all of data type $Float$. We name the input ports according to the input variables in the algorithm: $A1$, $A2$ and th. Thus, the input ports are:
$p_{i1} =< A1, Float >, p_{i2} =< A2, Float >$ and $p_{i3} =< th, Float >$.

Figure 9.1: Node 'RotRow' with its input and output ports. The labels are the name-type pairs of the ports. All ports have data type 'Float'.

Similarly, we name the two output ports $y1$ and $y2$ of data type 'Float':
$p_{o1} =< y1, Float >, p_{o2} =< y2, Float >$.

Figure 9.1 shows elementary node 'RotRow' with its input and output ports.

□

Program 9.2.1 [Algorithm of node 'RotRow']

We refer to the math library for definitions of the functions used.
Variables `A1`, `A2` and `th` are input variables.
Variables `y1` and `y2` are output variables.
Variable `h1`, `h2`,...,`h8` are temp variables.

```
h1 =  cos(th);
h2 =  mul(A1,h1);
h3 =  sin(th);
h4 =  mul(A2,h3);
y1 =  sub(h2,h4);
h5 =  sin(th);
h6 =  mul(A1,h5);
h7 =  cos(th);
h8 =  mul(A2,h7);
y2 =  add(h6,h8);
```

□

We represent argument passing between functions by edges.

Definition 9.2.3 [edge]

Let p_i be an input port of a node.
Let p_o be an output port of another node.

We define an *edge* e by the tuple $< p_i, p_o >$.

□

The data type of the token carried by an edge is defined implicitly by the data types of the ports it connects. The values of the tokens that flow along the edges are the values of the variables in the underlying single assignment program.

An edge defines a precedence relation between the nodes that it connects. The node of the output port must be evaluated before the node of the input port. The ordering of the nodes by edges is a partial ordering.

Now we give a formal definition of *dependence graphs*.

Definition 9.2.4 [Dependence Graph]

A *dependence graph*, *DG*, is an acyclic graph and consists of a set of nodes N and a set of edges E. The nodes stand for function evaluations and the edges stand for precedence relations between the functions of the underlying single assignment program.

□

Observe that a DG is an applicative description and is thus free of variables (memory). It is an alternative specification of the functional behavior of nodes, which we may interpreted as a structural refinement of the node initially specified by a single assignment program.

When the number of function evaluations is small, we can specify the DG by enumeration. We introduce a node for each function evaluation and an edge for each argument passing between functions via the single assignment variables.

Example 9.2.2 [Dependence Graph of function 'RotRow']

We give the dependence graph for program 9.2.1 specifying the functional behavior of the node 'RotRow'. It is easy to verify that the program is in single assignment form.
The first function is function `cos`, which we represent by node N_1. The second function is `mul`, which we represent by node N_2. In total there are 10 nodes.

Assume that the input and output ports of the nodes are named $in1$, $in2$,... and $out1$, $out2$,..., respectively. To make the names of ports unique, we refer to ports by their full name, which is the concatenation of the name of the node and the name of the port.

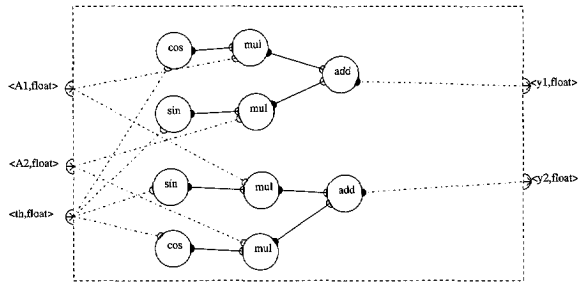

Figure 9.2: The dependence graph of the node 'RotRow'.

The second argument of the first function `mul` is the output of function `cos` via variable `h1`, which we represent by the edge:

$$d_1 = (N_2.in1, N_1.out1)$$

In total there are 8 edges.

Thus the *DG* is:

$< \{N_1, N_2, N_3, N_4, N_5, N_6, N_7, N_8, N_9, N_{10}\}, \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\} >$.

Figure 9.2 shows the dependence graph. The dashed lines represent edges (links) between unconnected input and output ports of the nodes to the input and output ports of node 'RotRow' itself. These edges (links) correspond to the input and output variables `A1`, `A2`, `th`, `y1`, and `y2`.

□

9.3 Piecewise Regular DG

The programs we consider may have control statements and size parameters. This means that the number of functions evaluations may be large and may depend on the size parameters, which makes specification of dependence graphs by enumeration impractical, or even impossible.

In order to specify dependence graphs for our class of nested loop programs, we introduce the concept of *node-domains*, *edge-domains* and *port-domains*. These domain definitions are also the basis of an hierarchical data structure on which design transformation tools, such as space-time transformation, operate. In chapter 14, we discuss this data structure in detail.

9.3.1 Node-Domains

The number of function evaluations of a program depends on the number of function-call statements and the size of the iteration domains of these statements. Each iteration in the iteration domain of a function-call statement corresponds to a function evaluation. The control statements active for the function-call statement determine the shape of the iteration domain.

We group the set of nodes associated with a function-call statement as *node-domains*. All the nodes within a node-domain have by definition the same functional behavior. [1]

We identify each node by an index, which is equal to the value of the corresponding iteration vector.

The iteration domain is characterized by a *linearly bounded lattice* [77]. Let k be the vector of control variables and e be the polytope corresponding to the function-call statement. With lattice matrix L defined by $[\, L_1 L_2 \,]$, L_1 an $n \times n$ identity matrix and L_2 a $n \times (m-n)$ zero matrix, as shown in chapter 8, the iteration domain $\mathcal{I}$ is defined as:

$$\mathcal{I} = \{\ I \mid I = Lk + O \ \wedge \ k \in e\ \} \tag{9.2}$$

Definition 9.3.1 [Node-domain]

Let $\mathcal{I}_{N\mathcal{D}}$ be an iteration domain
Let N be a node.
A *node-domain* ND is a set of indexed nodes all having the same functional behavior as node N:

$$ND = \{N_I | I \in \mathcal{I}_{ND}\}$$

□

Example 9.3.1 [Node-Domain]

In chapter 7, we derived the single assignment program of program 'Rotate'. In program 9.3.1 below, we have striped this program to show only the statements which are relevant to the derivation of node-domains.

We specify the node-domain corresponding to the function-call statement `RotRow`. The behavior of the nodes is specified by node 'RotRow', which we have defined in the previous section.

[1] A node may have its own parameter setting.

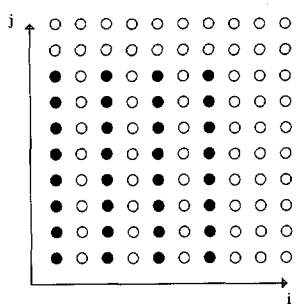

Figure 9.3: The node-domain corresponding to the function-call statement 'RotRow'. The black dots are the nodes computing the function 'RotRow'.

The control vector of the statement is $k = (i, j, p)^T$, where variable p is introduced to account for the loop stride. The polytope defining the values of k is defined by:

$$\begin{aligned} i &\geq 1 \\ i &\leq M - 1 \\ i &= 2 * p \\ j &\geq 1 \\ j &\leq M \end{aligned}$$

Next, we specify the lattice. The iteration vector I is $(i, j)^T$. Thus, matrix L_1 is a 2×2 identity matrix. The polytope has one additional variable p. So matrix L_2 has one zero column:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ p \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{9.3}$$

Figure 9.3 shows the iteration domain for $M = 8$.

□

Program 9.3.1 [Part of SAP of 'Rotate']
Let M be a parameter.
Let RotRow and RotColumn be functions.
Variable a_1,a_2,a_3,a_4 are single assignment variables.

```
for i = 1 : 2 : M-1,
   for j = 1 : 1 : M,
       [ a_1(i,j),a_2(i,j)] = RotRow(in0,in1,in2);
   end
end
for i = 1 : 2 : M-1,
   for j = 1 : 1 : M,
       [ a_3(i,j),a_4(i,j)] = RotColumn(in0,in1,in2);
    end
end
```

□

9.3.2 Edge-Domains

The structure of a single assignment program corresponds one-to-one to the structure of the original nested loop as far as the function call statements are concerned. Arguments of a function are assigned in assignment statements preceding the function-call statement. As shown in chapter 7, these statements are of the form:

$$\mathtt{in}_\beta = \mathtt{ipd}(v_\alpha(D(k)))$$

In this chapter, we also fully index the *argument* variables like $\mathtt{in}_\beta$.

As an example, an assignment statement from SAP program 7.6.1 becomes:

```
[in1(i,j)] = ipd(a_2(2*p6-1,i+1));
```

We assume that the names of the argument variables match the names of the input ports of the node representing the function.

In a way similar to that done for nodes, we group ports into *port-domains*. We define a port-domain by a port p and an iteration domain $\mathcal{I}_{PD}$:
$PD =< p, \mathcal{I}_{PD} >$.

The polytope of iteration domain $\mathcal{I}_{PD}$ is the polytope corresponding to the assignment statement. Let this polytope be denoted by e. With the lattice specified by a lattice

matrix L_i and an offset vector O_i, we define a port-domain PD_i by:

$$PD_i = \{p_{I_i} \mid I_i = L_i k + O_i \ \wedge k \in e \ \}$$

The lattice matrix L_i can be decomposed into an L_1 and an L_2 matrix as we did for the lattice matrices of node-domains. This means that the lattice points are obtained by projecting the points inside the polytope on the iteration space.

The indexing function D of a right-hand side variable inside an assignment statements is the *dependency function*. This function is an affine function on the control vector k of the assignment statement, and specifies the index I_o of the output port:

$$I_o = D(k_i) \tag{9.4}$$

Thus, there is an edge $d = (p_{k_i}, p_{I_o})$ between the input port p_{k_i} and the output port p_{I_o}.

The domain of the port-domain defining the output ports is formed by the same polytope e but now with dependence function $D()$ specifying the lattice:

$$PD_o = \{p_{I_o} \mid I_o = D(k) \ \wedge k \in e \ \}$$

Next, we introduce *edge-domains* as constructors for edges. By edge-domains we have a reduced specification of the edges of a DG.

We define an *edge-domain* by a *dependence function* and a *polytope* on which the function is defined.

Definition 9.3.2 [Edge-domain]

Let e be a polytope and k_i the vector ranging over polytope e.
Let $D()$ be an affine function on k_i.

An *edge-domain* ED is a set of edges of the type:

$$ED = \{d = (p_{k_i}, p_{k_o}) \mid k_o = D(k_i) \ \wedge \ k_i \in e \ \} \tag{9.5}$$

□

Observe that the polytope of the edge-domain is the same as the polytope of the input port-domain.

In general, polytopes of edge-domains have more variables and inequalities than polytopes of node-domains, because the control structure for dependencies inside programs is generally more complicated than the control structure of function calls.

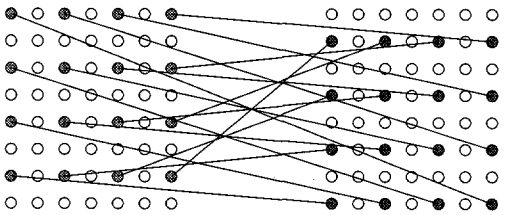

Figure 9.4: The set of edges specified by an edge-domain for the second input port of the 'RotColumn' nodes.

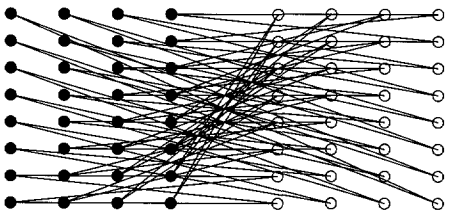

Figure 9.5: All the edges for the second input port of the 'RotColumn' nodes.

All input ports of a node must be connected. As an input port can have at most one incoming edge, port-domains of a particular input port are mutually disjoint.
Note that output port-domains can overlap, as output ports may have multiple outgoing edges (broadcasts).

Example 9.3.2 [Edge-Domain]

In program 9.3.2 below, we have listed the part of the single assignment program of 'Rotate' specifying the input for the second argument of the function `RotColumn`. There are three assignments to the input variable `in1(i,j)`.

We specify the edge-domain corresponding to the first assignment to input variable `in1(i,j)`:

```
[ in1(i,j) ] = a_1(2*p5-1,i+1);
```

The polytope of the edge-domain is defined by:

$$\begin{aligned} i &= 1 + 2p \\ i &\leq M - 1 \\ j &\geq 1 \\ j &\leq M \\ M - j - 1 &\geq 0 \\ j + 1 - 2p_5 &= 0 \end{aligned}$$

The input port-domain corresponding to this edge-domain is the set of input ports named `in1` of the `RotColumn` nodes. We specify the indices of these input ports by the polytope of the edge-domain, listed above, and the lattice $I = L_i k + O$:

$$\begin{pmatrix} i_i \\ j_i \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ p \\ p_5 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{9.6}$$

The output port-domain of the edge-domain is the set of output ports named `out1` of the `RotRow` nodes corresponding to the variable `a_1`. Its iteration domain is defined by the polytope of the edge-domain, but now with the dependence function $D()$ defining the lattice:

$$\begin{pmatrix} i_o \\ j_o \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ p \\ p_5 \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \tag{9.7}$$

Figure 9.4 shows the set of edges specified by the edge-domain. There are two other assignment statements to variable `in_1(i,j)` in the code. Their edge-domains can be specified similarly. In figure 9.5, we have enumerated the edges between nodes `RotColumn` and `RotCol` specified by the three edge-domains.

□

Program 9.3.2 [Part of SAP for the second argument of `RotColumn`]
Let p5 and p6 be control variables.
Let a_1 and a_2 be fully indexed single assignment variables.

```
for i=1 : 2 : M-1,
  for j=1 : 1 : M,

    if M-j-1>=0,
      p5=div(j+1,2);
      if -j+2*p5-1>=0,
        [in1(i,j)] = a_1(2*p5-1,i+1);
      else
        if j-2>=0,
          p6=div(j,2);
          if -j+2*p6>=0,
            [in1(i,j)] = a_2(2*p6-1,i+1);
          end
        end
      end
    else
      if j-2>=0,
        p6=div(j,2);
        if -j+2*p6>=0,
          [in1(i,j)] = a_2(2*p6-1,i+1);
        end
      end
    end
    [out0,out1] = RotColumn(in0(i,j),in1(i,j),in2(i,j));
```

□

9.3.3 Piecewise Regular DG

To summarize, a piecewise regular dependence graph (PRDG) is modeled by:

- linearly bounded **lattices** $\mathcal{I}$,
  $\mathcal{I} = \{I \in \mathbf{Z}^n | I = LK + O, K \in \mathbf{Z}^m, AK \leq B\}$
  with O and B integer constants, which may be parameterized.

forming the support of:

- **node-domains** (ND) defining sets of indexed nodes all of the same type:

$$ND = \{n_I | I \in \mathcal{I}_{ND}\}$$

- **port-domains** (PD) defining sets of indexed ports:

$$PD = \{p_I | I \in \mathcal{I}_{PD}\}$$

- **edge-domains** (ED) defining sets of edges characterized by an affine mapping D from an input port-domain, PD_i, to an output port-domain, PD_o:

$$ED = \{(p_I, p_J) | p_I \in PD_i, p_J \in PD_o, J = D(I)\}$$

Definition 9.3.3 [Piecewise Regular DG]

A piecewise regular $DG(\mathcal{N}, \mathcal{E})$ is a directed graph $G(N, E)$ with
$\mathcal{N} = \{ND_1, ND_2, \ldots\}$ a set of node-domains defining the set of nodes N and
$\mathcal{E} = \{ED_1, ED_2, \ldots\}$ a set of edge-domains defining the set of edges E.

□

A piecewise regular DG is a reduced description of a dependence graph.

We now discuss the piecewise regular DG corresponding to SAP program 'rotate'. The DG of 'rotate' consists of two node-domains ND_1 and ND_2 which represents the nodes 'RotRow' and 'RotColumn', respectively. The dimension of the index domains of both node-domains is two. In the SAP produced by *HiPars*, there are six edge-domains specifying edges between the node-domains. In addition, there are edge-domains specifying edges between the input and output ports of the DG itself.

To simplify the edge structure, we apply an index transformation on the node-domain of 'RotColumn': $(i', j') = (j, i)$ [86]. This results in the DG depicted in figure 9.6. It shows clusters of 2×2 rotations that are independent of each other.

To show a more complicated example, we have derived the DG for the loop body of the loop statement of `stage` inside the SVD program. This loop body consists of six loop stages with function calls to `Angle`, `RotRow` and `RotColumn`. Three loop stages correspond to the odd-indexed row and column rotations and the other three to the even indexed. We have added two additional loop stages. In the first loop stage, array `A` is initialized and in the last loop stage the result is read out. Thus, the complete program consists of eight loop stages.

Figure 9.7 shows the dependence graph of the loop body without the dependencies for the angles `th1` and `th2`. In the upper right corner of the figure, the set of nodes initializes the array `A`. The nodes in the lower right corner of the figure read out the result. We have applied a similar transformation on the second node-domain of the function `RotColumn`. Note that each node-domain still has its own index space.

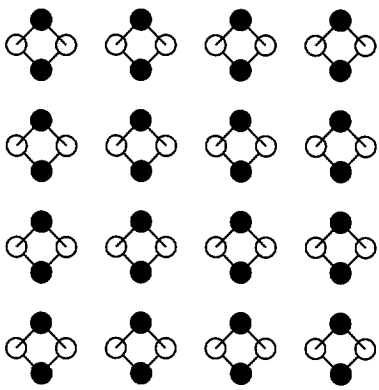

Figure 9.6: The dependencies between the node-domains of DG 'rotate' after index transformation.

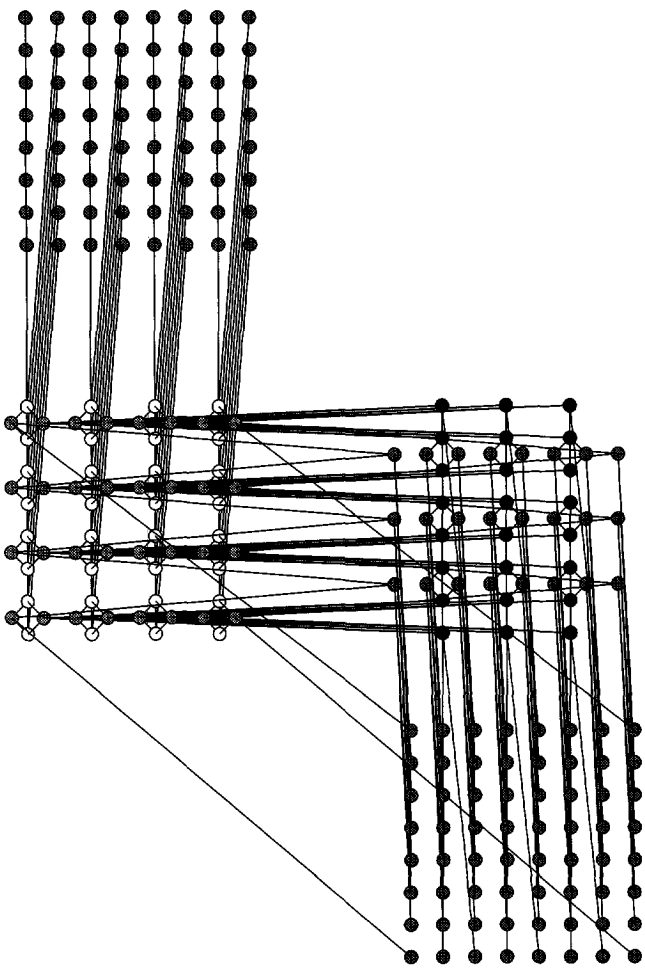

Figure 9.7: The dependence graph of the loop body of the loop statement `stage` of the SVD program for `M = 8`, extended with an input and an output stage. The edges and nodes of the angles are not drawn.

The angles are broadcasted into the array, see figure 9.8, which can be localized in order to obtain a locally connected DG. This DG can be regarded as a full-sized array. We can transform it further by the transformation HiFi system's tools.

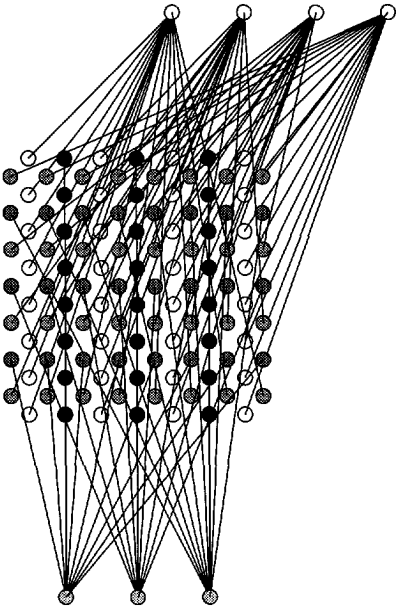

Figure 9.8: The nodes of the function `Angle` and the edges broadcasting the angles to the nodes computing the rotations.

9.4 Optimization

An important optimization is the removal of redundant inequalities and variables from the polytopes of edge-domains and node-domains.

For this purpose, we use existing methods: (1) the Chernikova routine [27] and (2) the omega test [57]. The Chernikova computes the extreme boundary points of a polytope, which is the dual representation of a polytope.
The omega test is a software routine that is based on the Fourier-Motzkin elimination method. It tries to derive equations from the polytope. Variables that are defined by equations can, in certain cases, be removed.

Both methods treat parameters as variables. As a result, parameterized polytopes will generally consist of more inequalities than nonparameterized polytopes. It depends on the context whether an inequality is redundant or not. An inequality may be redundant for

certain parameter values but may be not for other parameter values. From this follows that we can optimize HiFi descriptions further once we have set the value of the parameters for a specific application.

9.5 The tool 'sap2dg'

We have implemented the tool *sap2dg*. The tool inputs a single assignment program and converts it into a piecewise regular DG described in the HiFi design language.

The output of the DG is a data structure in which objects represent the node-, edge-, and port-domains defining the elements of the DG. In the appendix, we have listed several example objects.

The data structure has a tree structure. The child objects of the root are the node-domain objects and the edge-domain objects. The port-domains are children of the node-domains to which they belong.

9.6 Conclusions

We have explained how nested loop programs in single assignment form are represented by piecewise regular DGs, which are applicative specifications of the single assignment programs.

We have obtained a compact description of DGs by specifying the elements by index domains. By using index domains we have an algebraic well-defined specification on which we can apply transformations such as space-time transformation, partitioning and projection [23].

The piecewise regular DG can be used for the whole class of single assignment programs that *HiPars* outputs. The number of node-domains is equal to the number of function-call statements and the number of edge domains is equal to the number of RHS variables inside the single assignment program. However, a DG may consist of irregular parts as well, as we have illustrated by the example given at the beginning of this chapter.

By means of port-domains we specify for each input port and output port of a node the edge to which it is connected. This is necessary because in a data-flow architecture the communication between the nodes is asynchronous. As a consequence, we have to specify when a node must read its input ports and when it must write to its output ports, i.e., we have to specify control for each individual port.

9.7 Appendix: HiFi Objects

Below, we give example objects, which belong to the piecewise regular DG description of 'rotate'. The objects are written in the HiFi design language, which is based on objective C. The objects are generated by the conversion tool *sap2dg*.

HiFi Code 9.7.1 [Polytope]

We create a polytope object by the method `name` of the class `Polytope`. We set matrices A and B with the methods `setAMatrix` and `setBMAtrix`, respectively. We specify the parameters of the polytope by the method `setParVector`. Below, we have listed the polytope object for the function call to `RotColumn`. This polytope has one parameter M and three control variables: `'i'`, `'j'`, `'p'` . It lies thus in $\mathbf{Z}^3$.

```
polyRotColumn = [Polytope create];
[polyRotColumn setKVector: ('i'; 'j'; 'p')];
[polyRotColumn setAMatrix: (
0,-1,0;
0,1,0;
-1,0,0;
1,0,0;
1,0,-2;
-1,0,2)];
[polyRotColumn setBMatrix: (
0,-1;
1,0;
0,-1;
1,-1;
0,1;
0,-1)];
[polyRotColumn setParVector: ('M')];
```

□

HiFi Code 9.7.2 [Iteration Domain]

We create a iteration domain object by the method `create` of the class `Domain`. We set the polytope of the domain by the method `polyRotColumn`. We set the lattice by the method `setLattice`. The lattice is defined by an `AffineMap` object.

Below, we have listed the object for the node-domain of the function call to `RotColumn`. The iteration-vector is $I = (\mathrm{i}, \mathrm{j})^T$. Thus matrix L_1 is a 2×2 identity matrix. In addition, the polytope has one control variable p, to model the stride. So lattice matrix L is extended by addition of a zero column.

```
aDomain = [Domain create];

[aDomain setPolytope: polyRotColumn];

lattice = [AffineMap create];
[anAffineMap setAMatrix: ( 1,0,0;
                           0,1,0)];
[anAffineMap setBMatrix: ( 0,0,0;
                           0,0,0)];

[aDomain setLattice: lattice];
```

□

HiFi Code 9.7.3 [Node-domain]

We create a node-domain object by the method `name` of the class `NodeDomain`. We set the type of the node and the domain with the methods `setNodeType` and `setDomain`, respectively. The code for the node-domain of `RotColumn` is:

```
ND_RotColumn = [NodeDomain name: 'ND_RotColumn'];
[ND_RotColumn setNodeType: 'RotColumn' ];
[ND_RotColumn setDomain: aDomain ];
```

□

HiFi Code 9.7.4 [Edge-Domain]

We create an edge-domain object by the method `name` of the class `EdgeDomain`. In addition, we have to specify the names of an input port domain and an output port-domain.
An edge-domain corresponding with the input argument of the function `RotColumn` in program 9.3.2 is specified below. We set its attribute `polytope` to `polyIn1`. We have not listed the HiFi code for this polytope. The port-domains `IPD` and `OPD` are specified in the HiFi codes below.

```
ED = [EdgeType name: 'ED' ];
[ED setToNodeName: 'ND_Column' ];
[ED setToPortName: 'IPD' ];
[ED setFromNodeName: 'ND_RotRow' ];
[ED setFromPortName: 'OPD' ];
[ED setPolytope: polyIn1 ];
```

□

HiFi Code 9.7.5 [Input Port-Domain]

We create an input port-domain by the `input:type` method of the class `PortDomain`, by which, in addition, the name and type is set. The type is a reference by name to another port.
The input port-domain belonging to edge-domain `ED` is specified by input port `a_1` of node `RotColumn` and a domain. We specify the domain by the polytope of `ED_10` and a lattice matrix `Li`, which takes the form of a projection matrix. We set lattice matrix `Li` with method `setToMap`.

```
IPD = [PortDomain input: 'IPD' type: 'in1' ];
[IPD setVariable: 'a_1' ];
[ND_RotColumn addPortDomain: IPD ];

Li = [AffineMap create ];
[Li setAMatrix: ( 1,0,0,0;
                  0,1,0,0)];
[Li setBMatrix: ( 0,0; 0,0)];
[Li setParVector: ( 'M')];
[ED setToMap: Li ];
```

□

HiFi Code 9.7.6 [Output Port-Domain]

We create an output port-domain by the method `PortDomain`. We have to specify a name and a type. The type is a reference by name to another port.
The output port-domain belonging to edge-domain `ED` is formed by the ports named `out1` of the `RotColumn` nodes. The domain is defined by the polytope of `ED` and a lattice matrix `Lo`, which is equal to dependence function $F = (2p6 - 1, i + 1)^T$. We set lattice `Lo` with the method `setFromMap`.

```
OPD = [PortDomain output: 'OPD' type: 'out0' ];
[OPD setVariable: 'a_1' ];
[ND_RotRow addPortDomain: OPD ];

Lo = [AffineMap create];
[Lo setAMatrix: ( 0,0,0,2;
                  1,0,0,0)];
[Lo setBMatrix: ( 0,-1;0,1)];
[Lo setParVector: ( 'M')];
[ED setFromMap: Lo ];
```

□

Chapter 10

Hierarchical Graphs

10.1 Introduction

In the previous chapter, we presented the model for piecewise regular dependence graphs. Although they are described in a reduced way, the descriptions can still be complex, even for a small nested loop program, such as the SVD program 2.3.5.

The reason for this complexity is that parallelism is expressed at the level of iterations. As a result, the DG is often so detailed that it offers few insights to a designer. To retain overview, we use the concept of abstraction, which leads us logically to *hierarchical graphs*. By applying abstraction we hide detail without losing it.

The definitions of the elements of hierarchical graphs are based on the well-known recursive definition of sets. The elements of a set are either singletons or sets themselves. Singletons can be viewed as the most elementary kind of set. For example, in classical set theory there is just one singleton, namely $\emptyset$, see [32]. An hierarchical element is a set of elements, which may be hierarchical themselves. We refer to 'singleton' elements as *elementary*.

We use *port-domains*, *node-domains* and *edge-domains* of DG descriptions as set constructors. Note that there are many other ways to specify sets.

We have already applied abstraction when we modeled function evaluations by nodes of dependence graphs. There too, we are only interested in what a node computes, i.e., its functional behavior. Of course, each node computes according to a certain algorithm but we consider this algorithm only as a specification of the functional behavior.

In contrast to elementary nodes, we know the structural refinement of hierarchical nodes, which we specify by dependence graphs too. Along the same line of argument, we can also temporally refine nodes. Temporal decomposition of nodes is discussed in chapter 12.

Just as elementary nodes, hierarchical nodes have input and output ports. Ports of (hierarchical) nodes can be connected by (hierarchical) edges. A set of nodes interconnected by (hierarchical) edges is called an *hierarchical graph*. Often, we will not explicitly mention that an element of a graph is hierarchical when it is clear from the context or when it is not relevant.

More formally, we consider an hierarchical graph as a composition or structuring of the underlying DG into segments specifying the structure of hierarchical nodes. The composition of a dependence graph into segments is arbitrary. Any composition of the underlying DG into segments, which are connected in some way, can be the basis of the hierarchical graph. We compose the graph according to the structure of the DG description in terms of port-, node- and edge-domains.

We create for each DG node-domain an hierarchical node. The number of assignment statements in the nested loop program is usually small and equal to the number of node-domains. As a result, the graph will consists of a few number of nodes. The number of edges then equals the number of data dependencies between the segments.

Other rules apply for hierarchical graphs. They may be cyclic, and hierarchical ports may have multiple incoming edges. The main rule is that the underlying dependence graph, obtained by flattening the hierarchical graph, must be a valid one.

In section 10.2, we derive the hierarchical graph from the DG of 'Rotate'. In section 10.3, we present the tool 'HiCompose' that derives hierarchical graphs from a DG description. Then, we present the hierarchical graph of the SVD algorithm and discuss its complexity.

The drawings in this dissertation were made with the tool 'HiView', which we implemented to display graphs. The advantage of hierarchical graphs is that they are two dimensional, regardless of the dimension of the iteration spaces of the underlying DG. We believe that a geometrical view on graphs is helpful to the designer in coping with the design complexity.

10.2 Hierarchical Graphs

In this section, we give definitions of hierarchical ports, nodes and edges and compose, as an example, an hierarchical graph from the dependence graph corresponding to the SAP program 'rotate' listed in the appendix of chapter 7. The structure of a hierarchical graph corresponds one-to-one with the structure of the DG in terms of port-, node-, and edge-domains, which we will use as constructors of hierarchical elements.

First we define a hierarchical port.

Definition 10.2.1 [Hierarchical Port]

Let l be a label.
Let T_{PD} be a port-domain.
An hierarchical port AP is defined by the name-type pair $< l, T_{PD} >$ and is a set of ports, which may be hierarchical themselves:

$$AP = \{ p \mid p \in T_{PD} \} \tag{10.1}$$

We call l the name and T_{PD} the *type* of port AP.

□

Observe the resemblance to definition 9.2.1 of elementary ports. Here, elementary ports are considered to be singleton. The type of an hierarchical port is not a set of data values but a set of ports. Hence, we cannot really speak about the value of an hierarchical port because the port is a set of individual ports each having its own value. The grouping of ports into sets does not have a functional meaning, although it is sometimes useful to think of the value as the composition of the values of its individual ports.

To introduce the notion of hierarchical nodes, suppose that we cut a dependence graph into a number of segments. Further, assume that we use as cuts the inequalities (half-planes) of the node-domains of the DG description. Then, we obtain as many segments as there are node-domains in the DG description. Our objective is to represent each segment of the graph by a *segment node* whose structure is defined by a segment.

Depending on the way the edges of a segment cross cuts, we classify them into three kinds: *local*, *incoming* and *outgoing*. Local edges are edges connecting node ports of the same segment, they are edges that do not cross a cut. These edges are of the local structure of the segment node. Incoming edges depart from nodes of other segments. Outgoing edges arrive at nodes of other segments.

According to definition 9.3.2, an edge-domain is specified by a dependence function, an input port-domain PD_i, and an output port-domain PD_o. We use the port-domains of non-local edges as the type of the hierarchical ports of the segment nodes. A PD_i of an

incoming edge-domain becomes the type of an input port of a segment node. An PD_o of an outgoing edge becomes the type of the output port of a segment node.

Definition 10.2.2 [Segment Node]

Let ND be a node-domain and E_i, E_o, and E_l be the sets of incoming, outgoing, and local edge-domains of ND, respectively. Let $V_i = \{PD_{i,1}, PD_{i,2}, ..., PD_{i,n_i}\}$ be the set of input port-domains of the incoming edge-domains. Let $V_o = \{PD_{o,1}, PD_{o,2}, ..., PD_{o,n_o}\}$ be the set of output port-domains of the outgoing edge-domains.

A *segment node* consists of n_i input ports of which the types are the port-domains of V_i, n_o output ports of which the types are the port-domains of V_o, and a segment comprising the nodes of ND and local edges. The elements of the segment may be hierarchical themselves.

□

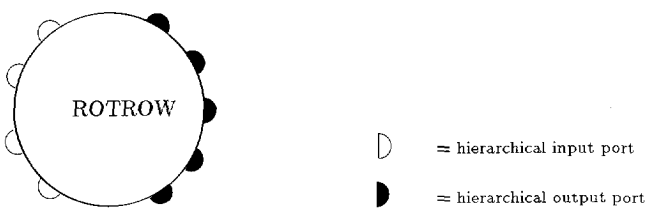

Figure 10.1: Segment node 'ROTROW' with its (hierarchical) ports, which are sets of ports.

In the previous chapter, we specified the DG of 'rotate' by two node-domains and six edge-domains between the node-domains. Figure 10.1 shows the external view of segment node 'ROTROW'. The node itself contains a graph segment and its ports are sets of ports. In a way, the segment is the structural refinement of node 'ROTROW'.

More generally, we speak of a DG node when we specify the structure of a node by a dependence graph, instead of a single segment. The DG node can be regarded as the result of merging a number of segment nodes.

As stated in the introductory chapter, we can also temporally refine nodes. When the type of refinement, structural or temporal, is not relevant to the discussion, we speak about *hierarchical-nodes.*

At the higher level graph, we connect the segment nodes with hierarchical edges.

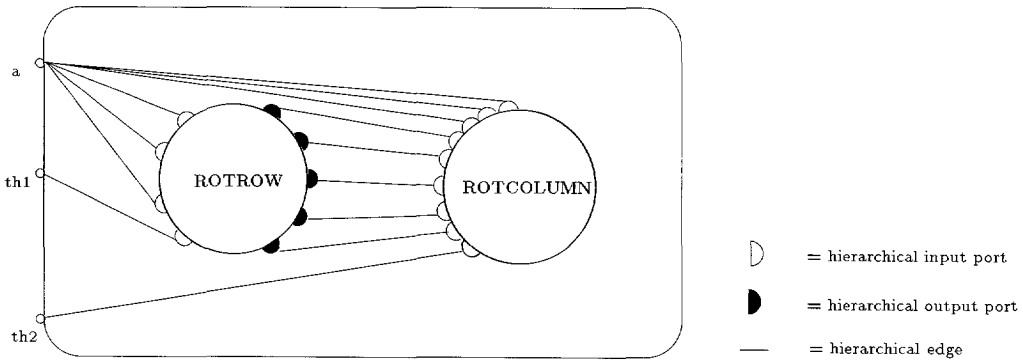

Figure 10.2: The hierarchical graph of 'rotate' comprising two segment nodes interconnected by hierarchical edges.

Definition 10.2.3 [Hierarchical Edge]

Let $T_{ED} =< PD_i, PD_o, D >$ be an edge-domain.
Let l be a label.

An *hierarchical edge* $AE =< l, T_{ED} >$ is the set of edges, which may be hierarchical themselves, defined by:

$$AE = \{ e \mid e \in T_{ED} \} \tag{10.2}$$

□

Note that each individual edge of the set must be an element of type T_{ED}, i.e., $e = (p_i, p_o) \wedge p_i \in PD_i \wedge p_o \in PD_o$. The individual edges are connected according to the dependency function of the edge-domain. Ports connected by edges must match. This means that the type of the input port must match the type of the output port. [1] However, an hierarchical edge may connect a subset of the ports of an hierarchical input port. This implies that, in contrast to elementary ports, hierarchical input ports may have multiple incoming edges, as long as after expansion this will not cause a conflict at the elementary port level.

The graph obtained after we have connected segment nodes by their edges has two *levels of hierarchy* as its nodes are graphs themselves. By applying abstraction on the graph again, we can add a third level of hierarchy. It is an recursive construction. In principal, there is no limit to the level of graph hierarchy.

[1] The level of hierarchy of ports may differ from the level of hierarchy of their nodes.

Example 10.2.1 [The hierarchical graph of 'rotate']

Figure 10.2 shows the hierarchical graph of 'rotate' consisting of the segment nodes 'ROTROW' and 'ROTCOLUMN' and hierarchical edges connecting the ports of the nodes. The graph itself has three input by which it may be connected at a higher level graph.

□

10.3 The tool 'HiCompose'

We have implemented the tool 'HiCompose' for composing hierarchical graphs. The tool takes as input a dependence graph description and outputs a graph of segment nodes and hierarchical edges.

The procedure of 'HiCompose' consists of two steps.

Let $\mathcal{N}$ be the set of node-domains and $\mathcal{E}$ be the set of edge-domains of a DG.

Step 1 of the procedure creates for each node-domain $ND \in \mathcal{N}$ a segment node SN:

- specify the structure of SN by the node-domain ND and local edges-domains $ED \in E_l$.
- specify an input port of SN for each incoming edge-domain $ED \in E_i$ and declare the PD_i of ED to be the type of the port.
- specify an output port of SN for each outgoing edge-domain $ED \in E_o$ and declare the PD_o of ED to be the type of the port.

Thus the original port-domains of the nonlocal edges are used as types of ports. A PD_o of each nonlocal edge-domain is used as the type of the output-port and a PD_i is used as the type of the input port of the corresponding segment nodes. This is another application of port-domains. Observe that hierarchy does not introduce new elementary elements.

Step 2 of the procedure interconnects the segment nodes with hierarchical edges. We add an (hierarchical) edge for each nonlocal edge-domain and connect the corresponding hierarchical input port and output port. The type of each (hierarchical) edge is specified by an edge-domain.

To summarize: the hierarchical graph is an irregular graph consisting of

- segment nodes for each node-domain in $\mathcal{N}$
- hierarchical edges for each edge-domain in $\mathcal{E}$.

10.4 Data Dependence Analysis of SVD

In this section we present the data dependence graph of the Singular Value Decomposition algorithm generated by *HiPars* [37] [22]. First we discuss the complexity of the DG of the SVD algorithm. Next, we present the DG in the form of an hierarchical graph.

10.4.1 DG of SVD

The SVD algorithm has 12 LHS variables and 20 RHS variables and has a quite complicated dependence structure. Table 10.1 gives an overview of the data dependencies between the variables of program 2.3.5. We have placed the LHS variables of program 2.3.5 at the top margin of the table and the RHS variables at the left margin of the table. We have indicated dependent variables with the character d. The table shows that HiPars has found in total 51 dependencies.

	th1(i)	th2(i)	a(i,j)	a(i+1,j)	a(j,i)	a(j,i+1)	th1(i)	th2(i)	a(i,j)	a(i+1,j)	a(j,i)	a(j,i+1)
a(i,i)					d							d
a(i,i+1)						d				d	d	
a(i+1,i)					d				d			d
a(i+1,i+1)						d					d	
th1(i)	d											
a(i,j)			d		d	d				d	d	d
a(i+1,j)				d	d	d			d		d	d
a(j,i)			d	d	d							d
a(j,i+1)			d	d		d					d	
th2(i)		d										
a(i,i)						d						
a(i,i+1)					d							
a(i+1,i)						d						
a(i+1,i+1)					d							
th1(i)							d					
a(i,j)				d	d	d						
a(i+1,j)			d		d	d						
a(j,i)					d	d			d	d		
a(j,i+1)					d				d	d		
th2(i)								d				

Table 10.1: Table of dependencies between RHS and LHS variables.

Each row of the table shows on which LHS variables an RHS variable depends. From the table, we see that an RHS variable may be dependent on more than one LHS variable.

Program 10.4.1 shows a piece of the SAP describing the data dependencies for the variable `a(i+1,i+1)` which is the fourth argument of the function `Angle`.

Program 10.4.1 [SAP code for input variable `in3`]

```
if stage-2>=0,
  p56=div(i+1,2);
  if -i+2*p56-1>=0,
    if M-i-3>=0,
      [in3] = ipd(a_8(stage-1,2*p56,i+1));
    else
      [in3] = ipd(a_5(stage-1,2*p56-1,i+1));
    end
  end
else
  [in3] = ipd(a_1(i+1,i+1] ));
end

[out0,out1] = Angle(in0, in1,in2,in3);
```

□

From the code, we derive that the function `Angle` is dependent on the function `RotColumn` inside the odd part and on the function `RotColumn` inside the even part of the algorithm, via the single assignment variables `a_4` and `a_8`, respectively. Further, it depends on the initialization loop stage via variable `a_1(i+1,i+1)`.

The control variables are `stage`, p56, and `i`.

The dependence functions are:
$D_{a8} = (stage - 1, 2 * p56, i + 1)^T$,
$D_{a5} = (stage - 1, 2 * p56 - 1, i + 1)^T$ and
$D_a = (i + 1, i + 1)^T$.

The domain of the dependence functions are specified by the active control statements.

The other data dependencies are specified in a similar way.

10.4.2 Hierarchical SVD Graph

Figure 10.3 shows the hierarchical graph for the SVD algorithm. The graph consists of 8 segment nodes called `input`, `output`, `AngOdd`, `RowOdd`, `ColOdd`, `AngEvn`, `RowEvn` and `ColEvn`. The segment nodes performing the rotations contain three-dimensional graph

segments. The segment nodes computing the angle and the input-output segments contain two-dimensional graph segments.

There are in total 90 hierarchical edges between the segment nodes. Not all the edges are visible because the line segments of the edges overlap in the figure.

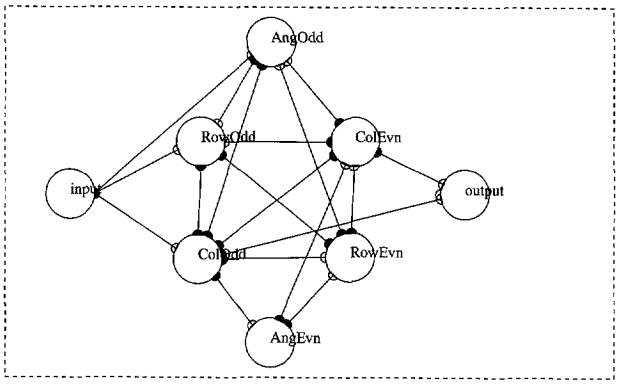

Figure 10.3: Hierarchical Graph of SVD showing the hierarchical edges between the segment nodes.

Table 10.2 shows the number of edges between the segments. The diagonal elements of the table are the number of local edge-domains of a segment node. Each row specifies for a segment node the number of incoming edges and the segment node from which they come. For instance, segment node `AngEvn` has 8 incoming edges coming from segment node `ColOdd`.

	in	AngOdd	RowOdd	ColOdd	AngEvn	RowEvn	ColEvn	Out
Input	0	0	0	0	0	0	0	0
AngOdd	4	0	0	3	0	2	6	0
RowOdd	3	2	2	7	0	8	10	0
ColOdd	2	2	12	2	0	0	2	0
AngEvn	0	0	0	8	0	0	0	0
RowEvn	0	0	4	12	2	0	0	0
ColEvn	0	0	0	14	0	8	0	0
Output	0	0	0	3	0	0	2	0

Table 10.2: Table of edge-domains between segments of the DG of SVD.

10.5 Conclusion

Our main objective has been to regain overview of complex DG descriptions. We have shown how hierarchy helps to reduce the design complexity. A rather complex DG is simplified to a small irregular graph where the nodes represent segments and hierarchical edges the data dependencies between the segments. The number of edges depends on the complexity of the data-dependence pattern.

Our model allows the level of hierarchy of a graph to be greater than two. Moreover, the level of hierarchy may be different for each element of a graph. In fact, a graph may be a mixture of hierarchical and elementary elements. An important observation about hierarchical graphs is that they do not introduce elementary elements. No matter what kind of hierarchical graph we compose from a DG, after flattening they all should give back the same underlying DG.

As stated in the introduction, many alternative DG compositions are possible. The tool 'HiCompose' creates segment nodes but this could easily be extended to DG nodes. We have decided not to do this because we regard merging of segments as design operations. For instance, we can merge two segment nodes into one DG node.

Graphs, hierarchical or not, are still algorithmical specifications. There is no notion of time yet. However, it is a small and often desirable step to regard the graphs as networks of concurrently operating nodes with each node executing a part of the algorithm.

Chapter 11

Clustering a Graph Segment

11.1 Introduction

In chapters 7 and 9 we derived piecewise regular DGs from nested loop programs. We now describe a tool that does the inverse operation, it converts dependence graphs into single assignment programs. For the design system as a whole, this is an important conversion. For instance, after applying linear scheduling transformations on a graph, we want to obtain a procedural code for the control programs that govern the scheduling. Or, we want to cluster a set of nodes in order to reduce communication costs.

The kernel of the conversion is a well-known technique called *scanning*. It has been described in [3]. Scanning means that we visit the points of the polytope in the lexicographical order defined by loop statements. To obtain an executable nested loop program, the loop bounds may depend only on the loop variables of outer loops.

In this chapter, we describe the tool 'dg2sap' that derives *cluster* programs from segments of a dependence graph. The tool does this in two steps. First, it derives the control structure. Then, it converts DG's data dependencies into read and write accesses to single assignment variables of the cluster program. The tool outputs the *cluster* program in procedural MATLAB code. Further, the cluster program has the single assignment property and is functionally equivalent to the dependence graph.

In this chapter, we do not discuss the choice of basis in which we scan the segment. We assume that a proper choice of basis has been made by our transformation tool [90]. Also, we avoid questions about scannability by assuming each segment of the DG to be scannable, i.e., there is a valid sequential ordering of the nodes. This is a fair assumption in the context of HiFi, because the design is initially specified by an executable, thus scannable, nested loop program. We can prove by construction that the design tools maintain this property.

The procedure can easily be extended for dependence graphs consisting of multiple segments. This requires that after deriving the cluster program for each individual segment of the DG, we have to apply *loop-merging* in order to construct the cluster program for the DG. Because *HiPars* gives us the data dependencies between the segments, we can use existing loop merging techniques as described in [86].

11.2 Scanning Dense Domains

Scanning a domain means visiting each and every point of the domain in a lexicographical order specified by a nested loop structure.

In this section, we assume domains to be dense. The scanning of a dense domain is similar to scanning its polytope because the domain's lattice is the identity. As a consequence, the stride of the loops of the cluster program will be equal to one. In section 11.4, we deal with nondense or sparse domains.

Let E be the polytope of a dense domain and let $k = (k_1, ..., k_m)^T$ be its vector of control variables. The objective is to scan E lexicographically in the order defined by k. To simplify the explanation, we annotate each control variable with a *level* l_v, which equals the position of the variable in vector k. Eventually, these control variables will appear as loop iterators of the loop statements at corresponding nesting levels in the cluster program.

Further, we define the level l_e of an inequality to be the highest level among its variables with non-zero coefficients. When an expression contains only parameters, its level is zero.

The kernel of the procedure is the classical Fourier-Motzkin elimination method [52]. This technique is well known and we refer to the literature for more a elaborate explanations of the technique [3]. The idea is to find for each loop variable a set of constraints that does not contain higher level variables. In other words, the loop bounds of each for-statement may not depend on the loop variables of its inner loops. It is important to remark that the elimination procedure is based on real projection. It finds the set of constraints that bound a loop variable k_l by projecting the polytope E on a subspace set up by all the variables of lower and equal levels, i.e, the $(k_1, ..., k_l)$ - space. Thus, constraints on variable k_2 are obtained by projection on the (k_1, k_2)-space. Constraints on k_3 by projecting on (k_1, k_2, k_3)-space, and so on.
After applying the Fourier-Motzkin procedure on polytope E, we obtain a new polytope E_{FM} of which the inequalities are sorted on level. This property enables us to scan the polytope by loop statements.

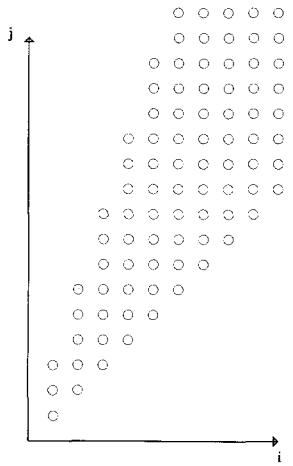

Figure 11.1: Index points of polytope (11.1) for $M = 10$.

Example 11.2.1 [Scanning a polytope]

Below we have listed, as example, the inequalities of a polytope of a dense domain with index vector $k = (i, j)^T$:

$$\begin{array}{rrcrr} -3i & -j & \geq & & 0 \\ -i & +j & \geq & & 0 \\ & -j & \geq & -2M & +3 \\ i & & \geq & & 1 \\ -i & & \geq & -M & \end{array} \qquad (11.1)$$

Figure 11.1 shows the enumerated index points of the polytope (11.1) for $M = 10$.
To derive the loop structure of the cluster program, we apply the Fourier-Motzkin procedure that returns an equivalent set of inequalities. The level of the inequalities is in parenthesis.

$$\begin{array}{lrrcrr} (0) & & 0 & \geq & -2M & +3 \\ (1) & -i & & \geq & M & \\ (1) & i & & \geq & & 1 \\ (2) & -i & +j & \geq & & 1 \\ (2) & & -j & \geq & -2M & +3 \\ (2) & 3i & -j & \geq & & 0 \end{array}$$

□

11.3 Writing the cluster program in procedural code

We use procedure 11.3.1 to derive the control structure of the program which scans a polytope. The procedure we use is a slightly adapted version of the one described in [91].

Below we have outlined the procedure. Procedure 11.3.1 constructs a parse tree of the cluster program. It first replaces the greater than operators, $\geq$, by equal to operators, $=$, and sorts equations to level. Then it splits each set of equations of the same level l, with $0 \leq l \leq m$, into a lower bound set L_l and an upper bound set U_l. Equations of L_l and U_l will appear in the bounds of the for-statement with loop variable k_l at nesting level l. The procedure adds an equation to the upper bound set U_l when the coefficient of variable k_l is negative. When the coefficient is positive, it adds the equation to the lower bound set L_l.

According to the syntax of for-loop statements, the coefficient of loop iterators must be equal to one. To achieve this we divide the inequality by the coefficient of the loop iterator. Further, as loop bounds must be integers, we round the expression to integer by *ceil* or *floor* operators. In the case of a lower bound we take the *ceil* of the expression. In the case of an upper bound we take the *floor*.

To illustrate this step of the procedure, consider the following example.

Example 11.3.1 [Floor and Ceil operator]

Suppose that i will become a loop iterator and must satisfy the following inequality:

$$-3i \geq -M + 5$$

To make its coefficient equal to one, we divide the expression -3:

$$i \leq (M-5)/3$$

As the sign of the coefficient was negative, it is an upper bound expression. So we floor it to the nearest integer:

$$i = floor((M-5)/3)$$

□

The lower bound of a loop variable k_l is the *minimum* of the ceiled expressions of L_l and its upper bound is the *maximum* of the floored upper bound expressions of U_l.

For this purpose, we need *min* and *max* operators in the cluster program. Of course, these operators can be left out when the number of inequalities of a bound set is equal to one. We omit this optimization for demonstration purposes.

Example 11.3.2 [Max and Min operators]

Suppose we have a loop iterator k with a lower bound set $L = \{\lceil (i-j) \rceil, \lceil (3i-5) \rceil\}$ and an upper bound set $U = \{\lfloor (i-j) \rfloor, \lfloor (3i-5) \rfloor\}$.

Then the lower bound of k is the maximum of the expressions of L and the upper bound is the minimum of of the expressions of U, which we write in MATLAB by `min()` and `max()` operators as follows:

```
for k = max(ceil(i-j),ceil(3*i-5)) to min(floor(i-j),floor(3*i-5)),

end
```

□

Procedure 11.3.1 [Deriving Loop Structure]

Let l denote the level of an equation. Let U_l and L_l be sets containing upper and lower bound equations, respectively.

Let E_{FM} be the polytope returned by the Fourier-Motzkin procedure and sorted to level.

The procedure constructs a parse tree from E_{FM}, corresponding with loop statement at nesting l.
The steps of the procedure are:

1. add If-nodes for inequalities on parameters only (level 0).
2. add *ceil* operator for expressions of L_l.
3. add *floor* operator for expressions U_l.
4. add *min* and *max* nodes to evaluate the maximum of the ceiled expressions of L_l and the minimum of the floored expression of L_l, respectively.
5. add For-node with loop iterator k_l and the *min* and *max* operations as lower and upper bounds, respectively.

By repeating the steps for all nesting levels we obtain the parse tree defining the loop structure of the cluster program.

□

Program 11.3.1 shows the control structure derived from the polytope (domain) of example 11.2.1 by procedure 11.3.1. In section 11.6, we complete the cluster program by adding the single assignment variables.

Example 11.3.3 [Control Structure]

The first statement of program 11.3.1 is a conditional statement specifying a constraint on parameter M. It contains *ceil* statements for each expression used in the lower bound and *floor* statements for upper bound expressions. The tool 'dg2sap' introduces temp variables, such as `lb0`. The temp variables `ma0` and `mi0` are assigned to the minimum and maximum value of floored and ceiled expressions and define the lower and upper bound, respectively.

After the first loop statement, a similar piece of code follows for the second loop of the cluster program.

□

Program 11.3.1 [Control Structure]

```
if M-2>=0,
  lb0 = equal(1);
  lb1 = equal(0);
  ma0 = max(lb0,lb1);
  ub2 = equal(M);
  ub3 = equal(2*M-3);
  mi0 = min(ub2,ub3);
  for i=ma0 : 1 : mi0,
    lb4 = equal(i);
    ma1 = max(lb4);
    ub5 = equal(3*i);
    ub6 = equal(2*M-3);
    mi1 = min(ub5,ub6);
    for j=ma1 : 1 : mi1,
      [ out0 ] = Fn();
    end
  end
end
```

□

11.4 Nondense domains

In this section, we address nondense domains, i.e., the lattice is not the identity. A naive approach would be to scan the polytope as we did for dense domains. This will lead to unnecessarily high nesting levels. Instead, we split the problem into two subproblems. First we assume that the domain is dense and derive the loop structure by procedure 11.3.1 given in the previous section. Second, we filter out the values of the loop iterator not on the lattice by means of additional constraints, which we derive from the lattice of the domain. These constraints are expressed in the form of conditional statements and transformation statements containing integer division operators.

We believe that by this approach the control structure of the cluster program resembles logically the nondense domains. For instance, the nesting level of the loop nest will be equal to the dimension of the domain and the lattice will be defined by integer division (modulo) operators.

Thus, we separate inequalities of the polytope into the ones that define the bounds of the loop statements of the cluster program and into the ones taking account of the lattice.

Recall that the index vector I of a domain is given by: $I = Lk + O$. To separate the variables of I into loop and nonloop variables, we use the property, see equation 8.4, that the lattice matrix L may be decomposed into an $n \times n$ matrix L_1 and zero matrix L_2 : $L = [L_1\ L_2]$.

Accordingly, we decompose I into I_1 and I_2, with I_1 the vector of loop variables and I_2 the vector of the non-loop variables.

From the inequalities depending only on I_1, we complete the control structure of the cluster program, listed in the previous section.

Inequalities in variables of I_2 are added in the form of conditional statements to the control structure of the cluster program and filter out instances of index-vector I that are not on the lattice. We further add integer-division transformation statements to the control structure to introduce the control variables needed in this filtering process.

We first have to sort equations to level, because the index transformations may be nested, in the case of complex lattices. Therefore, to obtain procedural code, we have to insert the div-statements into the cluster program according to some partial order.

After we have found the sets of inequalities in the nonloop variables, we order the sets such that the inequalities can be expressed directly in terms of conditional statements and index transformations. The transformation statements are placed before the block of conditional statements. Together they define the lattice constraints inside the cluster program.

11.5 Deriving the Div statements

We add index transformation statements containing integer division operators in order to introduce the non-loop control variables of the cluster program needed for the lattice constraints. By adding the div-statements, we may not make the constraints on the index tighter. As the inequalities of the block of conditional statements already specify the lattice, the inequalities associated with the division operators should not tighten the constraint on the iteration domain.

To explain this more formally, recall that, according to procedure 5.19, we may replace an integer division by its two defining inequalities, but now with the affine expression a written as $a = a_n + c$, with c an integer constant and a_n the linear part: $a_n + c - b\ q \geq 0$ and $a_n + c - b\ q \leq b - 1$.

These two inequalities must be redundant to inequalities defining the lattice, which we derived in the previous section. We check this explicitly by performing a redundancy test. Geometrically, the lattice points should lie in between the hyper-planes associated by the integer division.

Example 11.5.1 [Extract div-operator]

Suppose we have the inequalities:

$$\begin{array}{lll} i & -10 * t & \geq & 5 \\ i & -10 * t & \leq & 7 \end{array}$$

and we must introduce the control variable t by an integer division. With the divisor b equal to 10 and $a_n = i$, we may choose any constant c in the range $5 \leq c \leq 9$.

□

The example shows that there can be an ambiguity in the choice of the constant c in the expression of an integer division. Apparently, there can exist several integer divisions that are redundant for the inequalities. By default, the procedure selects the smallest value of c for the constant inside the expression. In case of the example c is 5.

Due to this ambiguity, the procedural algorithm of the cluster program may differ from the original program from which we generated the graph segment by the tool 'sap2dg'. Although they may have different constants in the expression inside div-operators, it is easy to verify that both programs are functionally equivalent.

Program 11.5.1 [The control structure of segment-node 'ROTROW']

Below, we give the loop structure of the cluster program for segment 'ROTROW'. It contains a *div*-transformation statement and and two conditional statements on control variable i_F to filter out the even values of i. Note that in the original program this was specified by stride 2.

```
if M-2>=0,
  lb0 = equal(1);
  ma0 = max(lb0);
  ub1 = equal(M-1);
  mi0 = min(ub1);
  for i=ma0 : 1 : mi0,
    lb3 = equal(1);
    ma1 = max(lb3);
    ub4 = equal(M);
    mi1 = min(ub4);
    for j=ma1 : 1 : mi1,
      i_F=div(i-1,2);
      if -i+2*i_F+1>=0,
        if i-2*i_F-1>=0,

          [ out0, out1 ] = RotRow(in0, in1, in2);

        end
      end
    end
  end
end
```

□

11.6 Adding the Single Assignment variables

In the construction of the cluster program, so far, we have only used the node-domain of the segment. We now complete the cluster program by converting the segment's port and edge-domains.

An edge-domain $E =< PD_i, PD_o, F_d >$ stands for a set of edges between segment nodes, and corresponds to read and write accesses to variables of the cluster program. We assume that variables are properly declared.

Input port-domains correspond to the read accesses to variables whose values are the arguments of the function of the segment, with the dependency function as indexing function of the variable. Output port-domains correspond to write accesses and assign the outputs of the function to the variables. The left-hand side variables are fully indexed, which implies that the cluster program is a single assignment program.

We insert the polytope of the edge-domain in the form of a block of conditional statements before the function-call statement. Note that this is a simple conversion. It does not require loop scanning. However, the code will typically contain integer division operator statements, which are derived according to the procedure outlined in the previous section.

Example 11.6.1 [Cluster program of segment-node 'ROTROW']

Below we have listed the complete cluster program that scans the segment-node 'ROTROW'. Its control structure, clearly resembles the structure of the graph segment in terms of node-domain, port-domain and edge-domains.

```
lb0 = ceil(1);
ma0 = max(lb0);
ub1 = floor(M-1);
mi0 = min(ub1);
for i= ma0  : 1 :  mi0,
  lb3 = ceil(1);
  ma1 = max(lb3);
  ub4 = floor(M);
  mi1 = min(ub4);
  for j=ma1 : 1 : mi1,
    i_F=div(i-1,2);
    if -i+2*i_F+1>=0,
      if i-2*i_F-1>=0,
        if j-1>=0,
          if -j+M>=0,
            if i-1>=0,
              if -i+M-1>=0,
                [in0] = ipd(th1(i));
              end
            end
          end
        end
      end
    end
    p1=div(i-1,2);
    if i-2*p1-1>=0,
       if i-2*p1>=0,
```

```
            if -i+2*p1+1>=0,
              if i-2>=0,
                   [in1] = ipd(a(i,j));
              end
            end
          end
        end
        if -i+1>=0,
          if j-1>=0,
            if -j+M>=0,
              if i-1>=0,
                if -i+M-1>=0,
                    [in1] = ipd(a(i,j));
                end
              end
            end
          end
        end
        if j-1>=0,
          if -j+M>=0,
            if i-1>=0,
              if -i+M-1>=0,
                   [in2] = ipd(a(i+1,j));
              end
            end
          end
        end
        [out0,out1] = RotRow(in0,in1,in2);
      end
    end
  end
end
```

□

11.7 Conclusions

We have shown how to derive a nested loop program from dense as well as nondense domains. We used the Fourier-Motzkin procedure for the derivation of the loop bounds. Another technique is described by Collard [62] who uses PIP instead of Fourier-Motzkin to construct loops from systems of affine constraints. Also Chamski uses PIP for the generation of loop bounds [17] and for enumeration of dense non-convex iteration sets [19].

By separating variables of a polytope into loop and nonloop variables, we obtain a cluster program of which the control structure resembles logically the domains defined by linearly bounded lattices.

We have implemented the tool 'dg2sap' that takes as input a segment of a dependence graph and outputs a functional equivalent single assignment program, which we refer to as the *cluster* program, since a set of computations (nodes) have been clustered into a single computational node.

With *loop-merging* techniques, we can merge several cluster programs of a DG's graph segments together into a single program [86]. This technique can be applied because the data dependencies between the graph segments are known too.

Chapter 12

Specifying Temporal Behavior

12.1 Introduction

Even in effectively parallel algorithms, there are parts of the algorithm that do not lend themselves well to parallel computation. This may be for several reasons. For instance, the data dependencies are not static, or are very irregular, or the computations involved are difficult to compute directly on hardware.

Obviously, we could merely specify these parts functionally, but this leaves out important aspects of the design, in particular local memory and control. We can hardly neglect these aspects in the design of distributed parallel networks for cost and performance reasons.

As our algorithms are written in a procedural languages, e.g. MATLAB, a processor can process these parts sequentially without difficulty. Sequential computation means that the functions of the algorithm are evaluated one after another.

In this section, we study precisely what it means to run a procedural algorithm in terms of the processor's

- memory and
- control.

For this purpose, we need a processor model. In [4], the concept of the AST node model in our context was formally presented and worked out further in [5] and [81]. It was motivated on the one hand by Hoare's work on CSP [38] and on the other by Backus' critical paper on functional programming [6].

The choice for the AST node is fundamental [38]. It logically separates function, state, and function ordering, which in procedural languages are so intertwined [21]. By modeling

procedural algorithms as AST nodes, we completely specify the order in which the functions are evaluated. In other words, we specify temporal behavior.

An important design aspect which we also model with AST nodes is the memory required by the processor network. This memory ranges from simple registers to memory buffers in the interface between the processor network and the host.

The AST concept can also model procedural algorithms containing 'while' loops, conditional branching, and data dependent branching. This gives another motivation for the AST model. Parts of the algorithm not satisfying the constraints imposed for regular array design can stay part of the design. This means that we can design systems for a much broader class of algorithms than the nested loop programs discussed earlier, however, with the regular parts treated in the more effective way described there.

By using ASTs, we also address the synchronization effects between the sequentially computed and the parallelly computed parts of the algorithm. In section 12.3 we discuss the timing behavior of AST nodes.

Below we model the procedural interpretation of our class of nested loop programs by AST nodes. This kind of modeling resembles the data-flow modeling of programs as found in [70]. Observe, however, that in our case it is a temporal specification instead of a structural one.

In section 12.4, we model assignment statements. Then, we model the semantic of control statements: (1) the loop control statement and (2) the conditional statement. Once we have established this, we give a general procedure for modeling nested loop programs as ASTs.

In this chapter we will mainly describe the internal behavior of the AST, i.e. how it selects its functions and accesses its local memory. In section 12.3, we deal with how ASTs communicate with each other.

12.2 AST nodes

The capabilities of a processor can be described by the set of functions it can compute. Let this set of, say n, functions be:

$$P = \{f_1, f_2, ..., f_n\}$$

These functions may be simple, like an addition of two numbers, but can also be complex and represented by an algorithm. We specify the functions as MATLAB programs, as usual.

A processor computes functions one after another in time. It starts, after a reset, with some particular function, which we call the *initial function*.

When we write down the functions that the processors executes, we obtain a sequence of function evaluations such as:

$$f_{init} f_3 f_8 f_4 ... f_6 f_7 f_4 ... f_{init}$$

At each moment, the processor performs one of the functions of P. The processor has thus some state. In AST terminology, we say that an AST node has a *function state*. In each of its function states, the AST evaluates a function which we assume to be an element of P. In general, the number of function states will be greater than the number of functions, because functions may have multiple port bindings. For the sake of convenience, we assume that each function state has a unique label.

An AST node has input and output ports, through which it can communicate with other AST nodes in the network. We explain the communication protocol in section 12.3.

We model local memory of a processor by a special set of *state variables*. We specify these variables in the same way we specify ports, by a name and a type. We denote the set of state variables by V. Ports as well as state variables may be of type array.

When referencing or addressing an array state variable other state variables may be used in its index expression.

An AST node has local control specifying the ordering of the function states. In addition, the AST node has a state variable which contains the current function state. We denote this state variable by s and call it the *current state*. We call the AST's set of function states the state space, denoted by S. The value of the current state variable must be an element of the state space S. We can compare the current state variable with the program counter of a sequential processor.

To evaluate a function f of a function state, the inputs and the outputs of the function have to be bound to ports and/or state variables of the AST node. To be more specific, inputs of f are bound to some or all input ports, and/or some or all state variables and outputs of f are bound to some or all output ports, and/or some or all state variables. Note that a function state may bind a subset of the ports of the AST node.

Let p_{in} be an arbitrary input port and v_s be an arbitrary state variable.
With v_{in} standing for a formal input of the function f_i of function state F_i, we define an input binding by the pair (p_{in}, v_{in}) or by the pair (v_s, v_{in}).
And, similarly, with p_{out} an arbitrary output port of the AST and v_{out} a formal output of function f_i, we define an output binding by the pair (v_{out}, p_{out}) or (v_{out}, v_s).

Example 12.2.1 [Function Binding]

Suppose we want to evaluate the addition operator *add* in a function state, computing $sum = a + b$. Thus the formal inputs are a and b and the formal output is sum.

Let us further assume that the AST has two input ports in_1 and in_2 and one output port out. We specify the binding function add as follows:

$$(in_1, a), (in_2, b), (sum, out)$$

□

Formally, a function state F is a mapping from the input port domain I^x, the variable space V^x, and the state space S, to the output port domain O^x and back to the $V^x \times S$:

$$F : I^x \times S \times V^x \mapsto O^x \times S \times V^x \tag{12.1}$$

Observe that a function state assigns a new value to the current state variable s and selects thus the next function state to be evaluated by the AST. To separate the part of the function that determines the value of state variable s, we decompose the function f of F in two parts. We call the first part the *application* part, and denote it by f_a. We call the second part the *control* part, which we denote by f_c. Similarly, we split the variables into a set of *data* variables, V_d and a set of *control* variables V_c.

f_a maps from input ports and data variables V_d to output ports and V_d:

$$f_a : I^x \times V_d^x \mapsto O^x \times V_d^x \tag{12.2}$$

and f_c maps from and to the state space S and control variable space V_c^x:

$$f_c : S \times V_c^x \mapsto S \times V_c^x \tag{12.3}$$

So f_c determines the value of the next state variable and may update other control variables. In this way, we have separated data operation from control. This corresponds with the classical separation of control and data path in digital systems.

Observe that the application part may not change the control variables. The AST model is more general, but here we restrict ourselves to the class of nested loop programs that have static control.

To summarize, we characterize an AST node by:

- a set of input and output ports I and O
- a set of functions $P = \{f_1, ..., f_n\}$
- a set of state variables V and the *current state* variable s
- a set of function states binding functions of P.

An AST is always in one function state at any given period. It starts up in the initial state.

We restrict ourselves to those types of AST nodes that return to the initial function state after a finite sequence of function states. They have cyclic behavior.

Bindings stand for read and write accesses to state variables (local memory). Each time an AST node goes to the next function state, it reads the variables bound to the inputs of f and, after completion of the algorithm, writes the output to the bound state variables and bound output ports. The index of array type variables may depend on other state variables, whose values are assigned at earlier function states.

The AST node does not model the data paths needed to pass the values from the ports and states to the functions (circuits). Also the evaluation of the index expressions of variables is not modeled in detail (memory address generation). However, it is clear that the switching between function states and the addressing of local memory can cause considerable overhead.

12.3 Timing Behavior

In this section, we describe the timing behavior of AST nodes. We distinguish between the internal timing behavior and the way the AST nodes communicate with other nodes of the network.

AST nodes can be connected to other AST nodes in the network by edges and communicate with each other according to the single token pass protocol. As the AST nodes synchronize on token passing between each other, we say that the network has data-flow control. We do not allow nondeterministic behavior.
The single token pass mechanism can be described and implemented by a four-phase handshake protocol. We refer to [33] for details.

Ports of AST nodes can be in two states: the *active* state and the *passive* state. A port is active when it contains a token. It is passive when it is free of a token. When the output port is active, the input port of another node that is connected to it is also active. By definition, state variables are always active.

A function f_i fires when all the currently bound inputs ports of the AST are active and the currently bound output ports are passive. Functional behavior does not depend on the arrival times at the ports. After completion of the function, the current input ports are passivated and the output data is written to the state variables and the current output ports, which become active. The AST goes to the next function state after completion of the function.

The control part f_c fires after the application part f_a. The reason for this is that during the computation of the application part, the state space may not be changed.

12.4 Assignments

In this section, we model an algorithm consisting of assignment statements by an AST node. We make a distinction between statements assigning to data variables and statements assigning to control variables.

12.4.1 Assignments to data variables

We consider a block of consecutive assignment statements. Each statement is a function call statement of the following form:

$$[LHS_1, LHS_2, .., LHS_m] = g(RHS_1, RHS_2, ..., LHS_n);$$

where g is the function called.

Procedural execution of an assignment statement can be divided into the following steps:

- reading the values of the RHS variables.
- evaluation of the function g on these values.
- writing the output values to the LHS variables.

We identify an assignment statement by a function state whose application part f_a is the function g.

Let the formal inputs of g be defined by the sequence $[i_1, ..., i_n]$ and let the formal outputs of g be defined by the sequence $[o_1, ..., o_m]$.

Assume that the variables referenced in the function call statement have been modeled by state variables.

Then we bind the inputs to state variables representing the RHS variables in the order as they appear in the function call, and bind the outputs to the state variables representing the LHS variables.

A block of statements is evaluated statement by statement in the order in which they appear in the code. The order of function evaluations is thus fixed. The control part f_c of a function state is thus a constant function and refers to the function state corresponding to the next assignment statement in the block. The function state modeling the last assignment statement returns to the initial function state.

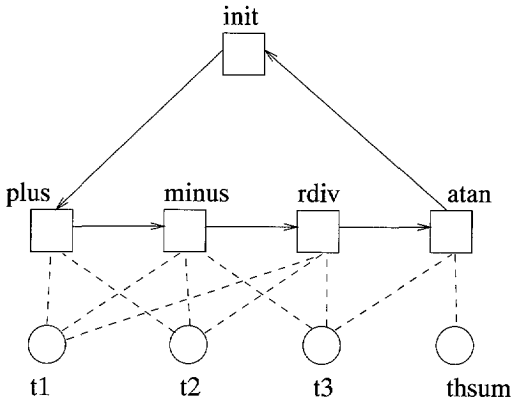

Figure 12.1: The function states and state variables of the AST node modeling a block of assignment statements. The square symbols represent function states, the circles represent state variables. The dashed line segment represent the function bindings.

Example 12.4.1 [Modeling Assignment Statements]

Below, we model the block of assignment statements listed in program 12.4.1. We label the four statements s1,s2,s3,s4 and represent them by the four function states F_{s1}, F_{s2}, F_{s3} and F_{s4}, respectively. Apart from these states, the AST node has an initial function state. The application functions f_a of these function states are:

$$\begin{aligned} f_{a,init} &= noop() \\ f_{a,s1} &= plus() \\ f_{a,s2} &= minus() \\ f_{a,s3} &= plus() \\ f_{a,s4} &= atan() \end{aligned}$$

The control parts are constant functions:

$$\begin{aligned} f_{c,init} &= s1 \\ f_{c,s1} &= s2 \\ f_{c,s2} &= s3 \\ f_{c,s3} &= s4 \\ f_{c,s4} &= init \end{aligned}$$

Note that the last function state F_{s4} goes back to the initial function state.

Figure 12.1 shows the state transition graph of the AST node in which the circles represent variables and the square boxes represent function states. The arrows indicate the ordering of the functions.

□

Program 12.4.1 [Assignments to data variables]

Let `plus` , `minus`, `rdiv` and `atan` be functions.
Let `t1`, `t2`, `t3`, and `thsum` be variables.

```
[ t1 ] = plus(t1,t2);      (s1)
[ t2 ] = minus(t1,t3);     (s2)
[ t3 ] = rdiv(t1,t2);      (s3)
[ thsum ] = atan(t3);      (s4)
```

□

12.4.2 Assignments to control variables

These assignment statements are the index transformation statements, which are of the form:

```
q  = t(index-expression) ;
```

with the function `t`: $equal(), div(), mod(), floor()$, or $ceil()$.

We represent these statements also by function states. The corresponding control variable is represented by a state variable. These state variables are special because they may be used in the index expressions.
Because the function assigns to a control state variable, the control part f_c of the function state evaluates the function t. In in addition f_c determines the next state. The application part f_a is void.

Example 12.4.2 [Assignment statement with modulo function]

Consider the following assignment statement containing a modulo function:

```
p1 = mod(i-j, 3);
```

We model this statement by a function state with $f_a = void$ and $f_c = mod()$. Variable $p1$ is modeled by a state variable. The function state has bindings with the state variables i, j and the state variable representing the constant 3.

□

12.5 Control Statements

In this section. we model the control statements and describe the procedural interpretation of these statements in terms of functions, state, and function ordering. First we model loop statements. Then we model conditional statements.

12.5.1 Loop Statements

The semantics of a loop statement is as follows. Upon entering the loop, the loop iterator is initialized with the value of the lower bound. Then, the statements of the loop body are repeatedly executed for as long as the value of the iterator stays between the loop bounds. After each loop, the value of the iterator is incremented by the stride.

We model a loop statement by three function states and a state variable as follows:

- the loop variable is modeled by a state variable.
- the *for* function state initializes the loop variable i with the value of the lower bound expression. It has the *test* state as the next state, $f_c = test$.
- the *test* function state evaluates the lower and upper bound expressions and tests whether the value of i is in between the lower and the upper bound. If true, the next function state will be the function state corresponding to the first statement of the loop body, if it exists. If false, the next function state will be the function state corresponding to the first statement following the loop.
- the *step* function state increments the loop variable i with the value of the stride. Its predecessor is the function state corresponding to the last statement of the loop body. Its next state is always the *test* state.

The application parts of these function states are void. All the expressions are thus evaluated by the control part f_c of the function states.
Each loop statement introduces thus its own control variable. We can optimize the number of control variables of an AST because the scope of the loop variable is limited to the loop body.

Example 12.5.1 [Loop Statement]

Figure 12.2 shows the function states and state variables of the AST corresponding to program 12.5.1, which consists of a double nested loop and an assignment statement in which the function `Angle` is called. The AST has two control state variables to model the loop iterators `stage` and `i`, and six data variable states modeling the variables: `in0`, `in1`, `in2`, `in3`, `out0`, `out1`.

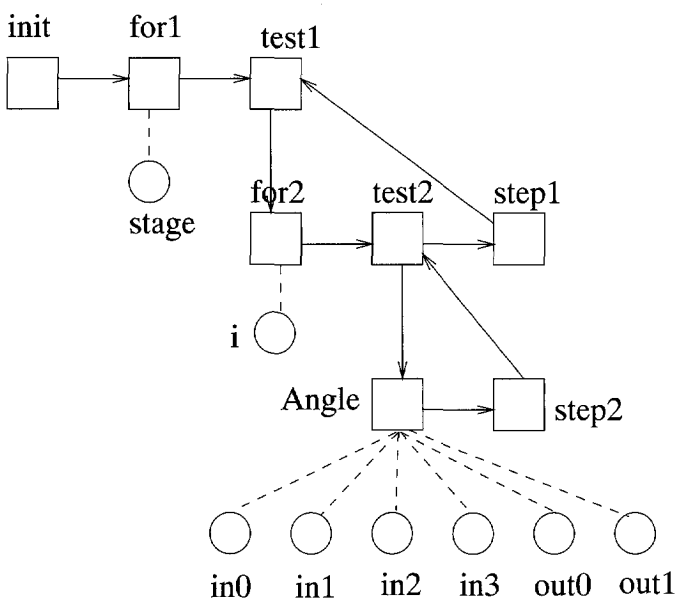

Figure 12.2: The AST node modeling the loop statement.

Each loop statement is modeled by three function states. The function state modeling function `Angle` is bound to the variable states, which is indicated by the dashed lines in figure 12.2.

□

Program 12.5.1 [Modeling loop statements]

```
for stage=1 : 1 : N,
   for i=1 : 2 : M-1,

      [out0,out1] = Angle(in0,in1,in2,in3);

   end
end
```

□

12.5.2 Conditional Statements

We represent conditional statements by function states, which we call *branch states*. The control part of a branch state evaluates the condition and selects one out of two possible next function states.

If the condition evaluates to true, the next function state will be the function state corresponding to the first statement of the block of statements of the `then` part. Otherwise, the next state will be the function state of the first statement of the `else` block.

If the `then` or `else` part does not exist, the branch state will branch to the function state corresponding to the first statement following the `then` or `else` part. The condition is evaluated by the control part f_c of the branch state. The application part f_a is void. Thus, when the AST is in a branch state, its next state is determined by the evaluation of the conditional expression, which may depend on state variables. Consecutive branch states can be merged into a single branch state with multiple next states.

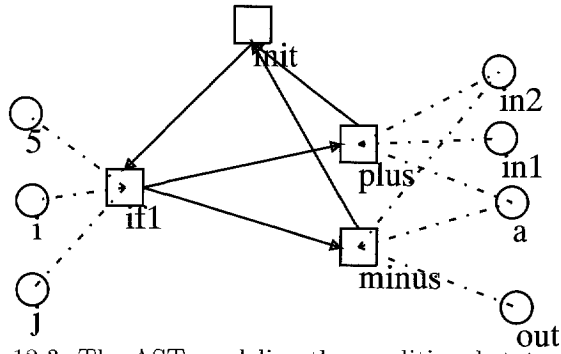

Figure 12.3: The AST modeling the conditional statement.

Example 12.5.2 [Conditional Statement]

Consider program 12.5.2 containing a conditional statement.

We represent this block by one branch state and two function states `plus` and `minus` for the assignment statements. See figure 12.3. Observe that the branch state has two possible next states. The control part f_c of the branch state evaluates $i - j \geq 5$.

□

Program 12.5.2 [Conditional statement]

```
if i-j > 5 then,
   [a] = plus(in1,in2);
else
   [out] = minus(a,in2);
end
```

□

12.6 The Tool 'nlp2AST'

We have implemented the tool called 'nlp2AST' that converts a program written in procedural MATLAB code into an AST specification according the steps outlined above.

The tool sets up a parse tree of the MATLAB program. The internal nodes of the tree are control nodes, which are modeled by the control function states as discussed above. The leaves of the parse tree are the function nodes which are modeled by data function states.

The tool creates as many data function states as there are function call statements. It also defines the control part f_c for each function state. The application part, however, is specified by name only and must be defined separately by another node in the system.

The ordering of the functions inside the AST node can be derived directly from the structure of the parse tree by traversing the tree in depth first order. It is completely determined by the parse tree.

The number of states can be reduced by merging states. In particular, conditional statements can be combined. Then it becomes a branch state with multiple next function states. By merging function states, however, the functions evaluating the conditions become more complex.

We have applied the tool on the segment node `AngOdd` of the SVD algorithm. Figure 12.4 shows the state transition graph of the resulting AST node.

To illustrate that AST nodes can also model nested loop programs, we have shown the state transition graph of the AST node representing the complete SVD program in figure 12.5. The SAP program corresponding to this AST node is listed in the appendix of this chapter.

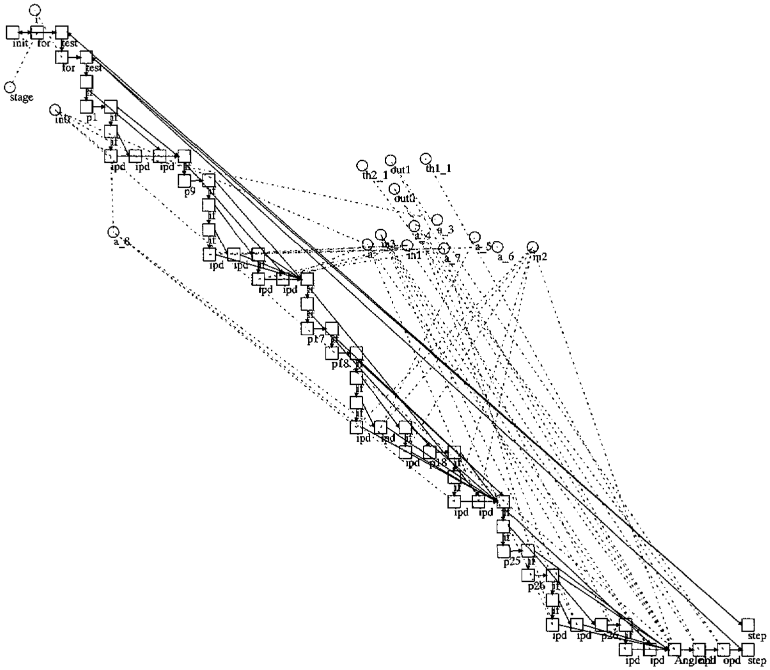

Figure 12.4: The AST modeling the single assignment program corresponding to segment node AngOdd.

12.7 Conclusion

We have modeled procedural algorithms by using AST nodes. The function states and state transitions inside an AST follow logically from the control structure of the program it represents.
In modeling programs by using AST nodes, we clearly separate function, state, and function ordering. In addition, we separate the functions that are responsible for the control from those operating on the data. Similarly, we have divided the variable space into a control space and a data space.

Our objective has been to describe clearly what procedural execution involves. The AST model shows precisely the local memory and the control needed to determine the function ordering.

We have implemented the tool `nlp2ast` that converts a nested loop program into an AST node. In other words, the tools converts a functional specification into a temporal specification.

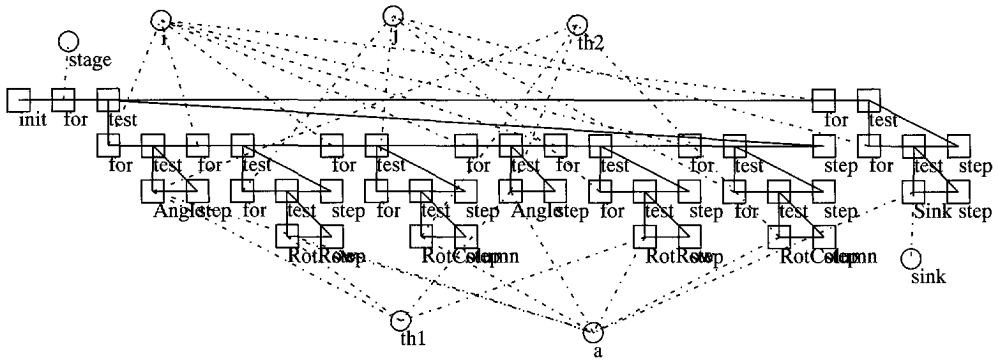

Figure 12.5: The AST modeling of the SVD algorithm.

In the previous chapter, we presented the hierarchical graph of the SVD algorithm. This graph is cyclic and its segment nodes cannot be specified by functions. However, we can convert each segment node into an AST node and thus create a network of AST nodes. AST nodes in a network can communicate with each other and synchronize on token passing. They execute thus in a lock-step mode.

12.8 Appendix: The SAP of segment node 'AngOdd'

Program 12.8.1 [SAP of segment node 'AngOdd']

```
for stage=1 : 1 : N,
  for i=1 : 2 : M-1,

    if stage-2>=0,
      p1=div(i+1,2);
      if -i+2*p1-1>=0,
        if i-3>=0,
          [in0] = ipd(a_8(stage-1,2*p1-2,i));
        else
          [in0] = ipd(a_3(stage-1,2*p1-1,i));
        end
      end
    else
      [in0] = ipd(a(i,i));
    end

    if stage-2>=0,
      p9=div(i+1,2);
      if -i+2*p9-1>=0,
        if i-3>=0,
          if M-i-3>=0,
            [in1] = ipd(a_7(stage-1,2*p9,i));
          else
            [in1] = ipd(a_6(stage-1,2*p9-2,i+1));
          end
        else
          if M-i-3>=0,
            [in1] = ipd(a_7(stage-1,2*p9,i));
          end
        end
      end
    else
      [in1] = ipd(a(i,i+1));
    end

    if stage-2>=0,
      if M-i-2>=0,
        p17=div(i,2);
```

```
      if i-2*p17-1>=0,
        p18=div(i+1,2);
        if -i+2*p18-1>=0,
          if M-i-3>=0,
            if i-3>=0,
              [in2] = ipd(a_8(stage-1,2*p18-2,i+1));
            else
              [in2] = ipd(a_5(stage-1,2*p18,i));
            end
          else
            if i-3>=0,
              [in2] = ipd(a_8(stage-1,2*p18-2,i+1));
            end
          end
        end
      end
    else
      p18=div(i+1,2);
      if -i+2*p18-1>=0,
        if i-3>=0,
          [in2] = ipd(a_8(stage-1,2*p18-2,i+1));
        end
      end
    end
  else
    [in2] = ipd(a(i+1,i));
  end

  if stage-2>=0,
    if M-i-2>=0,
      p25=div(i,2);
      if i-2*p25-1>=0,
        p26=div(i+1,2);
        if -i+2*p26-1>=0,
          if M-i-3>=0,
            [in3] = ipd(a_7(stage-1,2*p26,i+1));
          else
            [in3] = ipd(a_4(stage-1,2*p26-1,i+1));
          end
        end
      end
    else
      p26=div(i+1,2);
```

```
        if -i+2*p26-1>=0,
          [in3] = ipd(a_4(stage-1,2*p26-1,i+1));
        end
      end
    else
      [ in3 ] = ipd(a(i+1,i+1));
    end

    [out0,out1] = Angle(in0,in1,in2,in3);
    [th1_1(stage,i)] = opd(out0);
    [th2_1(stage,i )] = opd(out1);
    end
end
```

□

Chapter 13

The HiFi Design System

13.1 Introduction

We have build the HiFi system on top of the NELSIS CAD Framework [83], which we use for data management and to organize the HiFi system's tools. NELSIS is a flexible environment, which we have configured for the HiFi system.

We chose the CAD Frame for several reasons. Firstly, the framework features a database consisting of a collection of *design objects*. The data management of NELSIS supports versioning, concurrency control, and the structuring of the design objects by hierarchy relations.

In our case, a design specification is a hierarchical and parametrized specification. We capture the design specification as a *design graph*, consisting of a collection of design objects storing types of nodes and hierarchy relations between the design objects.

Secondly, we use the framework to organize the HiFi system's tools. By encapsulating the tools inside the framework, we obtain a coherent set of tools instead of a collection of individual tools. In addition, we arrange the tools in design flows through which we logically group sets of tools and specify constraints on the ordering in which the tools can be invoked.

Thirdly, we have used the framework for its graphical user interfaces. It features several browsers, such as contents browsers, hierarchy browsers, and version browsers. The tools can be invoked by the mouse, and the design objects on which the tools operate can be specified by a drag-and-drop mechanism.

In section 13.2, we introduce briefly some NELSIS terminology and describe how the hierarchical network description is captured as a *design graph*. In section 13.3, we describe how HiFi-tools are encapsulated inside NELSIS and organized by flow graphs.

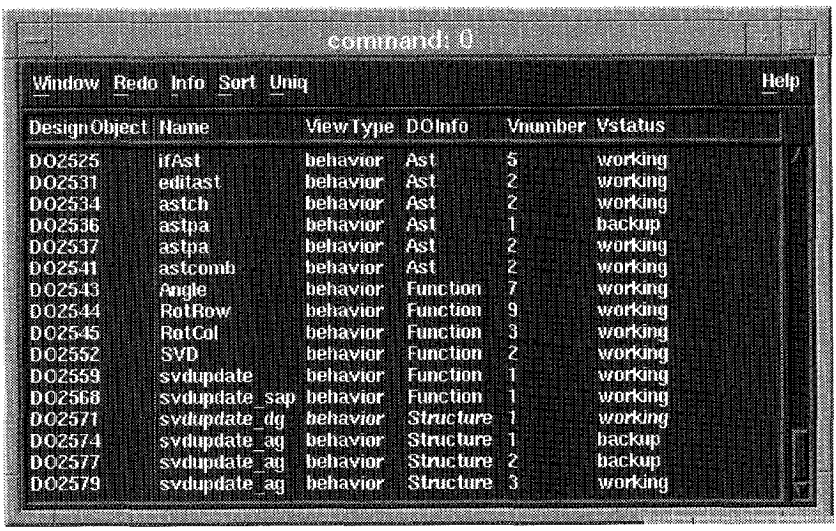

Figure 13.1: The contents browser showing the design objects stored inside the database together with their attributes.

13.2 Design Graph

The database consists of a set of design objects. The data structure of a HiFi network is a hierarchical structure consisting of node type objects. Each description of a node type is stored inside a separate design object inside the database. When a node type refers to, instantiates, other node types, we represent this by hierarchy relationships between the design objects.

A design object contains a description of a node type, which can be divided into:

- the description of the node type's interface.
- the description of the node type's body.
- the interface symbol table.
- the body symbol table.

We can list the design objects stored inside the database by a contents browser. See figure 13.1.

The hierarchy relationships between the design objects are represented by acyclic graphs,

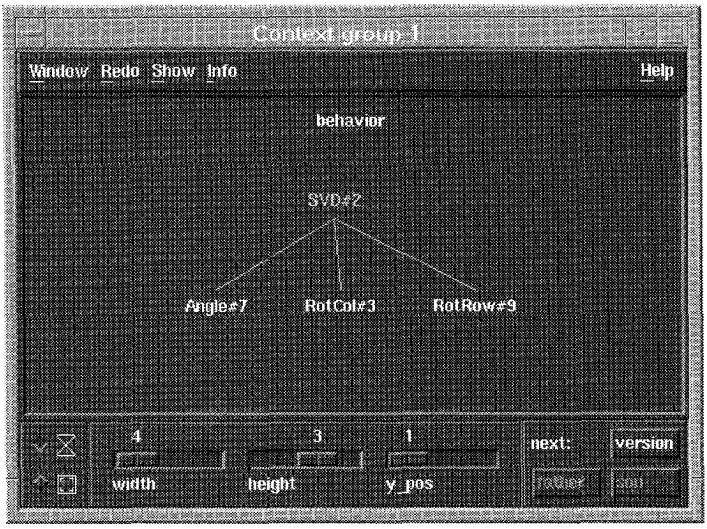

Figure 13.2: The design graph of the 'SVD'

in which the nodes are the design objects and the edges represent the relationships between the design objects.

Let N_1 and N_2 be design objects. We specify a hierarchy relationship between N_1 and N_2, denoted by the pair $< N_1, N_2 >$, when N_2 is part of N_1. Design object N_1 is called the parent and N_2 is called the child.

We use the hierarchy relationships to indicate nesting of functions. For example, the program 'SVD' calls the functions 'Angle', 'RotRow', and 'RotColumn'. The hierarchy relationships are:

$$(SVD, Angle), (SVD, RotRow), (SVD, RotColumn)$$

The SVD algorithm's design graph is shown in figure 13.2. In the window, the names of the design objects are concatenated with their version numbers.

The HiFi system requires that the type of the instances inside a node type be declared explicitly by means of 'use' declaration statements. The hierarchy relationships between the design objects correspond to these declaration statements.
For instance, the 'use' declaration statements of the node type 'SVD' are:

```
NT = [NodeType name: 'SVD'];
[NT use: 'Angle'  ];
[NT use: 'RotRow'  ];
[NT use: 'RotColumn'  ];
```

They declare the functions 'Angle', 'RotRow', and 'RotColumn' being used in the specification of the body of 'SVD'.

The design graph forms, in a way, the state of the design process. The tools may create design objects, introduce levels of hierarchy, flatten hierarchy levels, etc. The advantage of hierarchy is that tools operate only on parts of the design. Most of the tools, such as the conversion tools, modify only a single design object.

Apart from node type specifications, other design-related data is stored in the database, too. More precisely, we store (1) input and output data of the HiFi simulator and (2) tool parameter specifications. NELSIS allows the categorizing of design objects by *view-types* to indicate the type of data stored inside design objects. To this end, we have configured the view-types: *behavior*, *data* and *command*. Design objects of type *behavior* contain description of node types. Design objects of type *data* contain the stimuli data and, finally, tool parameter settings are stored in design objects of type *command*.

13.2.1 Design Alternatives

Often, we want to try out several design alternatives. NELSIS supports this by means of versions. Several design objects with the same name and view-type are called *versions*. The functional behavior of the versions should be the same. We obtain different design alternatives by substituting design objects inside the design graph for functional equivalent versions.

The framework assigns to a version a *version status*, such as WORKING, BACKUP. [1] A WORKING design object can be modified or updated. A design object with status BACKUP is write protected. Among a set of versions, there can be at most one version with status WORKING. We can change the status of versions interactively via the version browser.

The tools operate on the design objects and produce new design objects. However, the design objects are not placed automatically inside the design graph. For instance, converting design object *RotRow* into a dependence graph with the tool *HiPars* will lead to a new version of design object *RotRow*, which is not yet part of the design graph.

It is the task of the designer to substitute versions inside the design graph. To this end, the designer makes the version status of the design object that is to become part of the design hierarchy WORKING and substitutes it for the actual version inside the design graph. Note that a design graph can contain only a unique version of a particular design object.

For this purpose, we have implemented several tools by which hierarchy relationships between design objects can easily be added and removed. The design objects that are added

[1] Other version statuses are ACTUAL, DERIVED and IMPORTED.

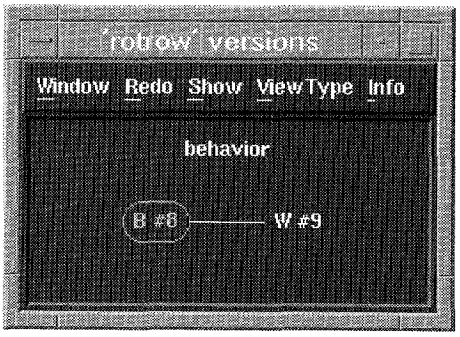

Figure 13.3: Version Browser showing the version of design object 'RotRow'.

to the design graph must have version status WORKING. We can set the status of a design object via the version browser. Thus, we can select the version that must become part of the design graph.

Let P denote a root design object.
Let C denote a child design object.

The hierarchy tools are:

- **addHierarchy** adds a hierarchy relationship between P and C. The child must be declared inside the parent and must have WORKING status.
- **removeHierarchy** removes the hierarchy relationship between the parent and the child.
- **buildDesignGraph**, builds the design graph of which P is the root. The design graph contains all the design objects that are declared inside the root design object and its descendants.
- **flattenDesignGraph**, flattens the design graph of which P is the root by removing all the hierarchy relationships between them. Further, the tool assigns all the design objects the WORKING status.

NELSIS features automatic version control for maintaining design consistency. When we edit a particular version, NELSIS creates a new version of the design object in case:

- its status is BACKUP
- it is a child.

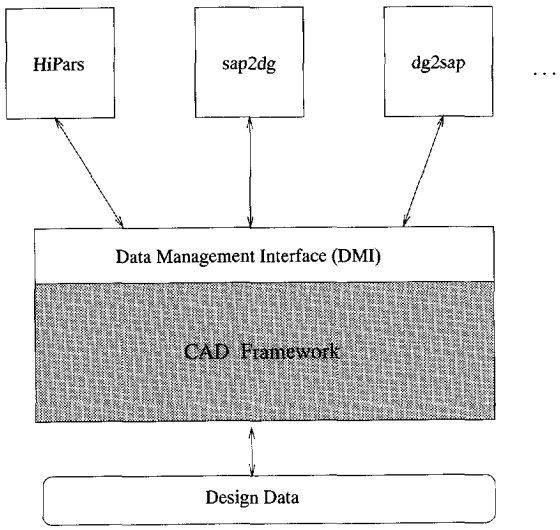

Figure 13.4: The HiFi tools interact with the NELSIS CAD Framework via the Data Management Interface (DMI).

Therefore the contents of these objects cannot be changed. A child design object is considered part of its parent design object. Therefore, edits on a child design object will result in a new version in order to keep the parent unchanged.

Only a root design object with status WORKING can be edited without being the number of versions increased. This gives another application for the hierarchy tools. Upon editing a design graph, we call the tool 'flattenDesignGraph' to create a set of working versions. Then we can apply a couple of edits on the design objects. Because the design objects have no parent and have WORKING status, this will not lead to extra versions. When we have finished editing, we call the tool 'buildDesignGraph' to build the design graph consisting of the working versions.

13.3 Organizing the Tools

In this section, we describe how the HiFi tools are encapsulated and how they interact with the CAD Framework. Further, we show how the tools can be invoked interactively via flow and option browsers.

13.3.1 Interfacing the Tools

NELSIS has a Data Management Interface (DMI) by which the tools can access the database. See figure 13.4. The DMI routines, such as check-in and check-out of design objects, can be called by the tools directly. This is called tool *integration*. We have decided not to do this, but to build a "wrapper" program that passes the input and output data to and from the tools by files. This is called tool *encapsulation*. The advantage of tool encapsulation over tool integration is that the tools remain independent of the database, i.e., the tools can operate stand-alone without the frame. A disadvantage is that encapsulation is slower and that database access can only take place at the start and end of a tool run. However, for a system in development, the disadvantages of tool encapsulation do not weigh up against the advantages.

We have implemented several routines to encapsulate the HiFi tools into NELSIS. The routines to check out a design object are:

- **getDesignObject** checks out a design object in UPDATE mode and returns the data structure of the node type.
- **getDesignObjectReadOnly** checks out the design object in READONLY mode.

When a design object is checked out it is locked by the database. [2] The locking scheme of NELSIS guarantees that at most one tool has write permission. This concurrency control allows several designers to work simultaneously on a design project.

The wrapper tool writes the node type description to a file, invokes a HiFi tool that creates one or more files containing new node type descriptions, and stores the contents of these files to the database by the routine:

- **putDesignObject**

This routine checks in and unlocks the design object.

In addition, the routine sets the attribute 'DOinfo' of the design object according to the type of specification of the node type: 'Function', 'AST', 'Structure', or 'NodeType'. The latter indicates that the body has not been specified yet.

13.3.2 Encapsulating tools with TES

To support the invocation of tools, we encapsulate the tools according to the Tool Encapsulation Specification (TES) defined by the CAD Framework Initiative [15].

[2] The contents will indicate locked design objects by creating a version with version number equal to -1.

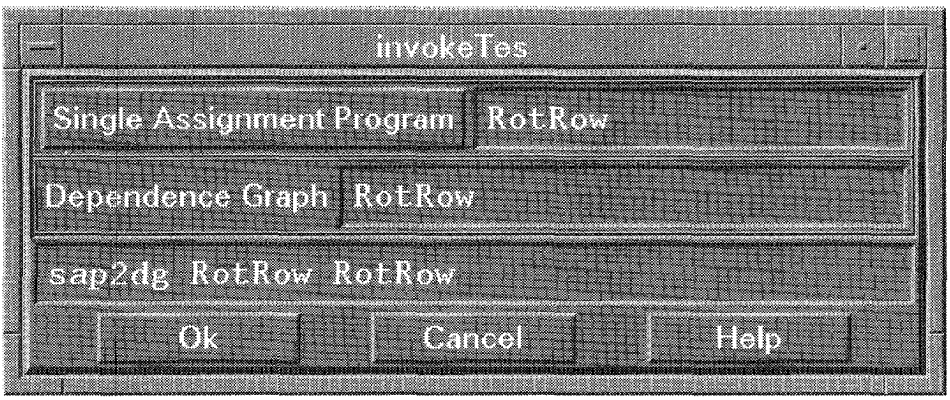

Figure 13.5: The option browser of the tool 'sap2dg'.

With TES, we can formally specify how a tool is to be invoked, i.e, its command line arguments and options are specified, with possible constraints between them.

We have written TES description for the tools 'HiPars', 'sap2dg', 'dg2sap' and 'nlp2ast'. When we invoke a tool for which a TES is defined, NELSIS will start up an option browser. Figure 13.5 shows the option browser for the tool 'sap2dg'. The option browser presents an invocation form, according to the TES description, showing the arguments and options that can be set. To invoke the tool, we simply fill in the argument fields and set the options. After pressing on the 'OK' button, the option browser builds the command line and invokes the tool.

13.3.3 Flow graphs

We have arranged the HiFi tools in flow graphs. A flow graph can be compound, or an activity. An activity is a design operation that can be performed by one of the tools encapsulated into the framework. A compound flow graph is a flow graph containing other flow graphs.

We have configured the tools in several compound flow graphs: 'convert', 'transform' and 'edit'. The tool activities of the 'Convert' flow graph are:

- HiPars, inputs a design object containing a nested loop program; produces a design object containing the single assignment program.
- sap2dg, inputs a design object containing a single assignment program ; produces a design object containing the dependence graph.

- HiCompose, inputs a design object containing a dependence graph; produces a design object containing a hierarchical graph.
- dg2sap, inputs a design object containing a dependence graph; produces a design object containing the program that scans one of the segments of the dependence graph.
- nlp2ast, inputs a design object containing a nested loop program, produces a design object containing an AST node description.

Flow graphs have ports, which specify the tool's (activity) kind of database access. Among other things, the ports specify the view types and number of design objects that may be checked out and checked in.

The ports of flow graphs are connected by channels. An activity has only access to design objects if one of its input ports is connected to the output port of the flow graph that produced the design object.

The flow graphs can be displayed by flow graph browsers. Figure 13.6 shows the flow graph 'convert'. The rectangles represent the flow graphs (activities). Flow graphs also show the status of a design object. When we drop a design object in a flow graph browser, it will highlight the channel connected to the flow graph's output port that produced the design object. With "balloons", design object attributes can be displayed. We have configured the browser to display the version name and version number of the design object inside the balloon.

The tools to edit, view, and print design objects are arranged in the compound flow graph 'edit':

- editText, invokes a standard text editor for editing a design description.
- viewText, invokes a standard text editor to view the contents of a design object.
- parseText, checks the syntax of the design description of the input design object.
- importFile, imports the contents of a file into a new design object.
- exportFile, exports the contents of a design object to a file.

Via the flow graph browser, we can invoke a tool by clicking on its flow graph symbol (rectangle) by the mouse. A tool can only be invoked if the required input design object has been specified. This is indicated by the flow browser by highlighting the flow graph symbol corresponding to the tool.

For instance, when we want to view design object 'SVD', we drag a version from the contents browser or from the hierarchy browser and drop it into the command flow graph 'edit'. See figure 13.7. We can invoke 'viewText' by clicking on its symbol.

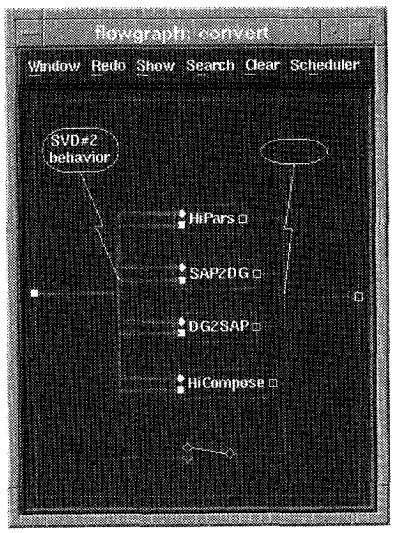

Figure 13.6: The compound flow graph 'convert'.

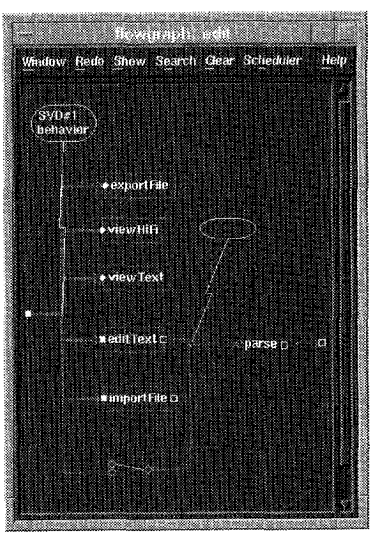

Figure 13.7: The compound flow graph 'edit' consisting of activities, channels and balloons. Design objects can be dropped into the balloons, after which the channel to which it is connected will be highlighted.

The transformation tools are arranged in the flow graph 'transform'. These tools operate on design objects containing dependence graph descriptions. They may produce several design objects containing the description of the transformed dependence graph. We describe the output produced by some of these tools in chapter 16. For more information on these transformation we refer to [23][90].

- SpaceTime, performs a space-time transformation of one or more segments of the dependence graph. The space-time transformation is an affine transformation, which can be specified by the tool invocation form.
- Partition, partitions the input DG into two dependence-graphs: (1) the tile graph and (2) the tile node. These node types are stored in separate design objects. The tile graph instantiates the tile node type. We can use the tool 'buildDesignGraph' to create a hierarchy relationship between them.
- Projection, performs a linear projection of a space-time transformed and strictly regular DG segment and produces a structure-node specifying the resulting data-flow network.
- Regularize, introduces control to make a DG more uniform so that it can be projected.

13.4 Conclusion

We have configured the NELSIS CAD Frame for the HiFi system.

The design description is captured by a design graph inside the database. Each design object stores a description of a node type. Hierarchy relationship between design objects correspond to node type instantiations. Design consistency is maintained by version and concurrency control of NELSIS.

The tools are encapsulated and interface with the framework via a wrapper tool. This keeps the software of the tools independent of the CAD frame's software. In addition, the tools can operate stand-alone. The arguments and options of tools are formally specified by TES. The database access of tools and the allowed tool invocation ordering is specified by flow graphs. In addition, the tools can be invoked interactively via the flow graph and option browsers.

Chapter 14

Specifying Node Types

14.1 Introduction

As we explained in the previous chapter, a design specification is stored in several design objects inside the database. Each design object contains the specification of a parametrized node type. Node types may refer to each other thus forming a hierarchical design specification. In this chapter, we explain the specification of the node type.

To support hierarchical specifications, a specification of a node type is separated into an interface and a body specification. The interface consists of input and output ports of nodes and the parameters.
The body can specify:

- a function
- an AST
- a structure

We have implemented the tool 'HiEdit' to enter body specifications into the system. The tool 'HiEdit' allows top-down design and features a graphical user interface. Figure 14.1 shows its main menu.

The specifications are written in the HiFi design language, which is based on Objective C. They constitute a data structure in which the node type is represented by objects. In the remainder of this work, we often use the term *node* instead of node type when the difference is clear from the context, in order to make the text easier to read. A node may be a mixture of regular and irregular parts. The HiFi language allows the specification of special objects, based on the definition of linearly bounded lattices defined in chapter 5, to specify regular parts. Irregular parts are specified by enumeration.

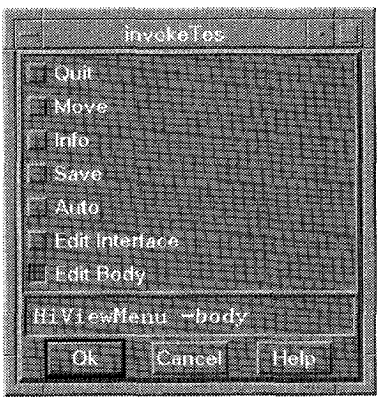

Figure 14.1: Main Menu of the 'HiEdit'

The outline of this chapter is as follows. After introducing in section 14.2 the notations to be used, we describe the data structure in a grammar-like fashion. We give also the classes to specify regular parts of a node. In section 14.3, we describe the node object. In section 14.4, we discuss the input and output port objects. These are the interface objects. In section 14.5 - 14.7, we give the specifications of functions, ASTs and structures, respectively. These are the three types of bodies. Finally, we briefly consider the tool 'HiEdit'.

14.2 Notation

The design specifications are written in the HiFi design language which is based on the object-oriented programming language Objective C [80]. An *object* consists of attributes constituting the object's state, and methods/functions that may operate on that state. Objects are defined by classes.

In this chapter, we mainly focus on the relationships between these objects. As said in the introduction, we specify these relations by a grammar. The precise syntax is illustrated by examples.

Each grammar rule specifies the number of attributes of a class. The attributes themselves are instances of other classes, which may be defined by other grammar rules. The syntax of a grammar rule is as follows. A rule starts with the name of a class followed by the symbol := after which the attributes are specified by a tuple delimited by the characters '<' and '>'.

The rule

```
Class_A := <Class_1, Class_2, ...,  Class_n> ;
```

specifies that an object of `Class_A` has n attributes that are objects of the classes $\texttt{Class}_i$, $i = 1, .., n$.

A class name in the tuple enclosed by curly brackets, such as $\{A_x\}$, means that the attribute is a set of objects of that class.

The character | separating the class names inside a tuple such as

```
Class_A := <Class_1 | Class_2 | ... |  Class_n> ;
```

denotes selection, i.e. the attribute is an object of one of the classes of the tuple.

We denote a reference by name to an object by appending `_ref` to the name of the class.

Terminals of the grammar are the classes `Name`, `String`, and `Integer`.

In chapter 5, we used lattices and polytopes to specify regular parts of the design. See equations 5.4 and 5.7. We have implemented the classes `Domain`, `AffineMap`, and `Polytope` to represent domains and polytopes in the data structure. These classes correspond one-to-one to these formal definitions and are terminals of the grammar too.

As an example, we give objects with which the domain of example 9.3.1, given in chapter 9, are represented. Recall that a domain is characterized by a lattice and a polytope. The control variables of the control vector k in the example are i, j and p, with $I = (\mathrm{i}, \mathrm{j})^T$ the index vector of the domain, and there is one parameter M.

A polytope object represents the polytope's defining system of r linear inequalities. Let m be the number of control variables and q the number of size parameters.
The attributes of the polytope object are an integer matrix A, size $r \times m$, and an integer matrix B, size $r \times (q + 1)$, with constant last column.

A `polytope` object is created with the method `create` of the class `Polytope`. We set matrices A and B with the methods `setAMatrix` and `setBMatrix`, respectively. We specify the parameter vector by the method `setParVector` and the vector k of control variables by the method `setKVector`. The polytope object for the function-call statement `RotRow` is

```
aPoly = [Polytope create];
[aPoly  setKVector: ( 'i'; 'j'; 'p')];
[aPoly  setAMatrix: (
0,-1,0;
0,1,0;
```

```
-1,0,0;
1,0,0;
1,0,-2;
-1,0,2)];
[aPoly  setBMatrix: (
0,-1;
1,0;
0,-1;
1,-1;
0,1;
0,-1)];
[aPoly setParVector: ( 'M')];
```

A lattice is an affine relation between the vector $k \in \mathbf{Z}^m$ and the index vector $I \in \mathbf{Z}^n$ of the domain and is represented by an `AffineMap` object. Its attributes are a matrix A of size $n \times m$ and a matrix B of size $n \times (q+1)$.

For our example, we have $n = 2, m = 3$ and $q = 1$ and where the zero column column of the matrix A goes with the variable p.

```
aLattice = [AffineMap create ];
[anAffineMap setAMatrix: ( 1,0,0;
                           0,1,0)];
[anAffineMap setBMatrix: ( 0,0;
                           0,0)];
```

We create a domain object by the method `create` of the class `Domain`. We set the polytope of the domain by the method `setPolytope`. We set the lattice by the method `setLattice`.

The domain object for the function-call statement `RotRow` is specified by the following code, with the polytope and lattice objects as specified above.

```
domain = [Domain create ];
[domain setIndexVector: ( 'i'; 'j')];
[domain setPolytope: aPoly ];

[domain setLattice: aLattice ];
```

14.3 The Node Type Class

The specification consists of `node type` objects and relationships between the node types. The relationships stand for instantiation of node types by another node types.

Node types are identified by an unique name. The grammar rule for the declaration of the node type class is

```
NodeType := <Name,  {ChildType}, {Parameter}, Context, Interface, Body > ;
ChildType := <NodeType_ref> ;
```

The `ChildType` set contains the nodes that are referred to in the body. A child type object refers to a node by name. This node type must be stored in another design object inside the database.

Nodes may be parametrized. Parameters may occur in inequalities specifying domains and in the constant terms of affine expressions.

A parameter is declared by a name and a range.

```
Parameter  := <Name, range>;
range := <c1, c2>;
c1 := <Integer>;
c2 := <Integer>;
```

Figure 14.2 shows the sub menu of the tool 'HiEdit' for declaring the parameters. The range of a parameter is the parameter support on which the node is valid. Additional constraints on a parameter can be specified by the context, defined by a polytope. For instance, parameter M can be restricted to even values using the constraint $mod(M, 2) = 0$.

```
Context :=  < <Polytope> | <> >;
```

Further, the node consists of an interface specification and a body specification. Separation between interface and body allows for hierarchical network specification. In section 14.4, we describe the interface specification.

Recall that the body of a node can be either a function, an AST, or a structure object.

```
Body =  < Function | AST | Structure > ;
```

In accordance with these types of bodies, we refer to the node as being a *function*, an *AST*, and a *structure*, respectively. These specifications are defined in sections 14.5, 14.6 and 14.7, respectively.

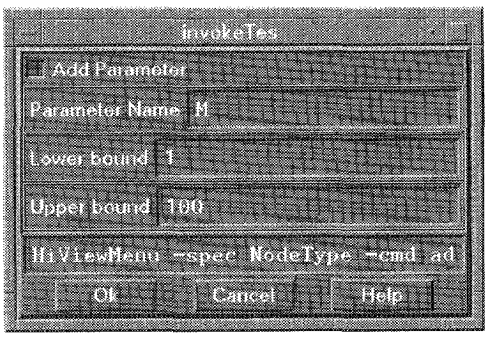

Figure 14.2: Menu for editing parameter objects.

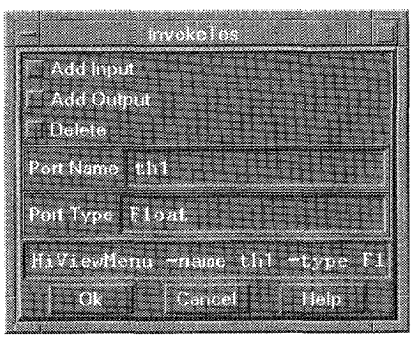

Figure 14.3: Menu for creating port-type objects.

14.4 Interface Specification

The grammar rule for the declaration of the `interface` class is

```
Interface  := < {Port} > ;
```

Port objects are specified by a name and a type. In addition to which, the port is also specified to be either an input or an output port.

```
Port := <Name, Kind, pType>;
Kind := <Integer>;
pType := < DataType | HierarchyType>
```

Obviously, we distinguish between elementary and hierarchical ports. We call a port elementary when its type is a data type, which may be of type domain and may be nested. For instance, a port of size $M \times M$ is specified by a domain consisting of the inequalities $1 \leq i \leq M, 1 \leq j \leq M$. Figure 14.3 shows the menu for editing ports.

```
DataType := < Integer | Float | DataTypeDomain >;
DataTypeDomain := < DataType, Domain> ;
```

A hierarchical port, however, refers to another port or a set of ports specified by a port-domain.

```
HierarchyType :=  < Port_ref | PortDomain >;
PortDomain := <Port_ref, Domain>
```

The interface specification of a node is generic for the three types of bodies. However, the semantics of a port depends on the type of body. Ports of structures are hierarchical or abstract and disappear after flattening of the data structure. Ports of functions declare the formal input and output arguments of the functions. Only the ports of ASTs become ports of the network after expansion of the network description.

14.5 Function

The body of a node can be a function. We capture functions by objects of the `Function` class. An executable specification of the function is contained in the attribute `code`. It is a procedural program written in the programming languages MATLAB or C. The attribute `code` is a string object.

```
Function := <Name, Code>;
Code := <String>;
```

When the function calls functions that are captured by another node, this node must be declared by a `ChildType` object.

HiFi Code 14.5.1 [Function-Type]

Below is the description of the node 'RotRow' of which the body is specified by a MATLAB program. The names of the ports match the names of the input and output variables of the program.

```
NT = [NodeType name: 'RotRow'];
A1 = [NT addPort: [PortType input: 'A1' type: 'float']];
A2 = [NT addPort: [PortType input: 'A2' type: 'float']];
th = [NT addPort: [PortType input: 'th' type: 'float']];
y1 = [NT addPort: [PortType output: 'y1' type: 'float']];
y2 = [NT addPort: [PortType output: 'y2' type: 'float']];

f = [Function name: 'rotrow'];
code = @{
 y1 = A1 * cos(th) - A2 * sin(th);
 y2 = A1 * sin(th) + A2 * cos(th);
    }@;
[f setCode: code ];
[NT setBody: f ];
```

□

14.6 AST

The body of a node can be an AST. We capture ASTs by objects of the class AST. The attributes of the AST class are a set of state variables and a set of function states in accordance with the definition of the AST given in chapter 12.

```
AstType := < {StateVariable}, {FunctionState} >;
```

Figure 14.8 shows the menu for editing AST nodes.

We specify a state variable by a name and a type which may be a domain. Figure 14.4 shows the sub menu for editing state variables. The current state variable of an AST is automatically predefined.

```
VariableState := <Name,DataType>;
```

The grammar rule for a function state is

```
FunctionState := <Name,ApplicationPart,ControlPart,{Binding},NextStateSet>
ApplicationPart := <NodeType_ref | <> >;
ControlPart := < NodeType_ref >;
```

Function states can be added by the sub menu shown in figure 14.5.

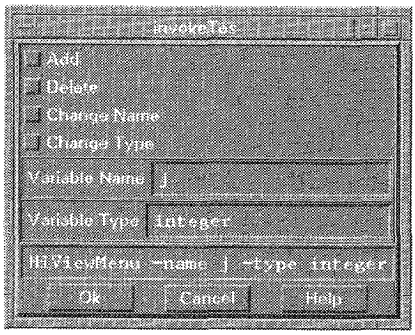

Figure 14.4: Menu for editing state variables of an AST.

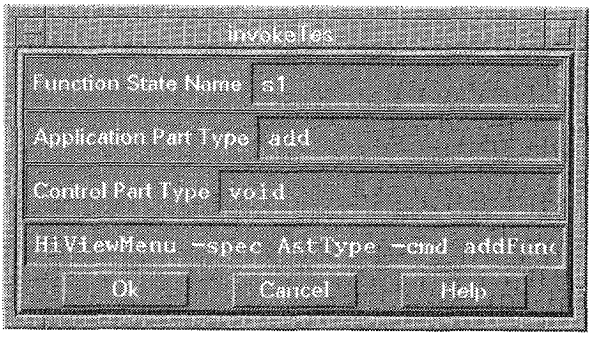

Figure 14.5: Menu for editing function states of an AST.

The application and control parts refer to other nodes. This introduces hierarchy in the specification of AST nodes as well. The application part and the control part may be void.

For each function state, we specify the values that the control part of the function state may assign to the current state variable s of an AST by a set consisting of the names of the possible next function states. The control part of the function state selects one out of the set of next function states.

```
NextStateSet :=  < {ToFunctionState} >;
ToFunctionState := <FunctionState_ref> ;
```

Finally, we bind the formal inputs and outputs of the functions of the function states to the state variables or ports of the AST node. Each binding is defined by the name of the

formal input, the name of the actual port or state variable, and a possible affine index expression. In the index expression other control AST variables may appear.

```
Binding     := < FormalPort, Actual, Index >
FormalPort  := <PortType_ref> ;
Actual      := <PortType_ref | VariableState_ref>;
Index       := <AffineMap>;
```

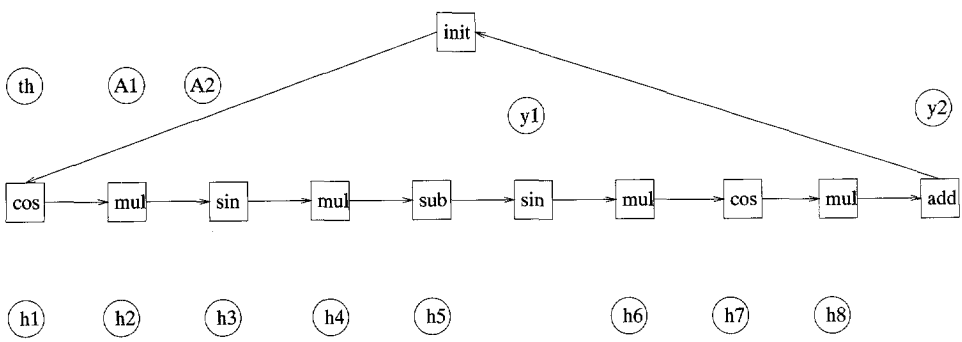

Figure 14.6: The 'RotRow' AST.

Example 14.6.1 ['RotRow' as AST type]

The 'RotRow' AST node is shown in figure 14.6. A part of the description of this AST is listed in program 14.6.1.

□

HiFi Code 14.6.1 [AST type 'RotRow']

```
NT = [NodeType name: 'RotRow'];
A1 = [NT addPort: [PortType input: 'A1' type: 'float']];
A2 = [NT addPort: [PortType input: 'A2' type: 'float']];
th = [NT addPort: [PortType input: 'th' type: 'float']];
y1 = [NT addPort: [PortType output: 'y1' type: 'float']];
y2 = [NT addPort: [PortType output: 'y2' type: 'float']];

ast = [AstType name: 'rotrow'];
[NT setImplementation: ast ];
```

```
init = [FunctionState name: 'init' ];
[init setKind: 0 ];
[init setApplicationType: 'void' ];
[init setControlType: 'Constant' ];
[ast addFunctionState: init ];
[init addNextState: [NextState name: 'nx' toState: 'cos2' ]];
cos2 = [FunctionState name: 'cos2' ];
[cos2 setKind: 5 ];
[cos2 setApplicationType: 'cos' ];
[cos2 setControlType: 'Constant' ];
[ast addFunctionState: cos2 ];
[ast addVariableState: [VariableState name: 'h1' type: 'int' ]];
[cos2 addBinding: [Binding formalOutput: 'O0' actual: 'h1' index: '']];
[ast addVariableState: [VariableState name: 'th' type: 'int' ]];
[cos2 addBinding: [Binding formalInput: 'I0' actual: 'th' index: '']];
[cos2 addNextState: [NextState name: 'nx' toState: 'mul4' ]];
mul4 = [FunctionState name: 'mul4' ];
[mul4 setKind: 5 ];
[mul4 setApplicationType: 'mul' ];
[mul4 setControlType: 'Constant' ];
[ast addFunctionState: mul4 ];
[ast addVariableState: [VariableState name: 'h2' type: 'int' ]];
[mul4 addBinding: [Binding formalOutput: 'O0' actual: 'h2' index: '']];
[ast addVariableState: [VariableState name: 'A1' type: 'int' ]];
[mul4 addBinding: [Binding formalInput: 'I0' actual: 'A1' index: '']];
[mul4 addBinding: [Binding formalInput: 'I1' actual: 'h1' index: '']];
[mul4 addNextState: [NextState name: 'nx' toState: 'sin6' ]];
...
...
...
```

□

14.7 Structure

The body of a node can be a structure. We capture structures by objects of the structure class. A structure consists of a set of nodes and a set of edges. We specify regular parts of the structure by edge- and node-domains. Irregular parts are specified by enumeration.

```
Structure := <Name,{Node},{Edge},{NodeDomain},{EdgeDomain}>;
```

Nodes are instances of other node types. When we instantiate a node type, we have to set the values of its parameters. Parameter values can be set to constants or can be defined by affine expressions in the parameters declared by the structure's node type. In this way, the values of parameters at higher hierarchical levels can propagate down to lower levels.

```
Node := <ChildType, ParameterSetting> ;
ParameterSetting := <AffineMap>;
```

When the node type referred to by the child type is a structure itself, the node is a hierarchical node.

Edges refer by name to an input port and an output port of node instances or refer to ports of the structure.

```
Edge := <Name , InputPort, OutputPort>;
InputPort :=  <<Port_ref, Node_ref > | <Port_ref>>;
OutputPort := <<Port_ref, Node_ref > | <Port_ref>>;
```

Node-domains are used to instantiate sets of nodes of the same node type. Here we also specify the value of parameters of the instances by an expression. In this expression, variables of the index-vector of the node-domain may appear, too.

The node-domain object has also the set of port-domains as attribute.

```
NodeDomain := <NodeType_ref, Domain, ParameterSetting, {PortDomain}>;
```

A port-domain groups a set of ports. It is specified by a reference to a port of the node type of the node-domain and an index domain.

```
PortDomain := < Port_ref, Domain > ;
```

The port-domain may depend on the parameters of the structure. These port-domains can also be used as types of ports.

Edge-domains are defined by an input port domain, an output port domain, and a dependence relation which is an affine map.

The dependence function is defined on the polytope of the input port domain and specifies the index of the output port, which must lie in the domain of the output port-domain.

```
EdgeDomain := <InputPortDomain, OutputPortDomain, DepenenceFunction>;
InputPortDomain := <<PortDomain_ref> | <Node_ref,Port_ref>> ;
OutputPortDomain := <<PortDomain_ref> | <Node_ref,Port_ref> > ;
DependenceFunction := <AffineMap>;
```

Observe that we may connect a port of a single node, provided that this port is a hierarchical port with an appropriate index domain. This allows to interconnect the regular parts of the network to ports of single nodes of the structure.

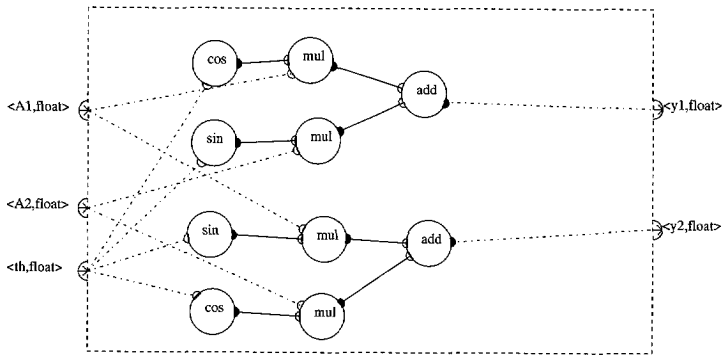

Figure 14.7: 'RotRow' specified as structure-type.

Example 14.7.1 ['RotRow' as Structure-type]

Figure 14.7 shows the structure modeling the dependence graph of 'RotRow' defined in example 9.2.1. The structure consists of 10 nodes and 18 edges. The nodes are instances of the node types 'add', 'mul', 'cos' and 'sin'. Ten edges are connected to the ports of the structure itself. A part of the HiFi code for this structure is listed hereafter.

```
NT = [NodeType name: 'RotRow'];
A1 = [NT addPort: [PortType input: 'A1' type: 'float']];
A2 = [NT addPort: [PortType input: 'A2' type: 'float']];
th = [NT addPort: [PortType input: 'th' type: 'float']];
y1 = [NT addPort: [PortType output: 'y1' type: 'float']];
y2 = [NT addPort: [PortType output: 'y2' type: 'float']];

S = [Structure name: 'S'];
cos1 = [Node name: 'cos1' type: 'cos' ];
[S addInstance: cos1 ];
id cos2 = [Node name: 'cos2' type: 'cos' ];
[S addInstance: cos2 ];
sin1 = [Node name: 'sin1' type: 'sin' ];
[S addInstance: sin1 ];
sin2 = [Node name: 'sin2' type: 'sin' ];
[S addInstance: sin2 ];
...
```

```
...
...
[S addEdge: [Edge name: 'e1' fromInput: 'th'
   toPort: 'in0'  ofNode: 'cos1']];

[S addEdge:[Edge name: 'e2' fromInput: 'th'
              toPort: 'in0' ofNode: 'cos2']];

[S addEdge:[Edge name: 'e3'
            fromInput: 'th'
              toPort: 'in0' ofNode: 'sin1']];

[S addEdge:[Edge name: 'e4' fromInput:'th' toPort:'in0' ofNode:'sin2']];

[S addEdge:[Edge name: 'e5'  fromPort: 'out0'  ofNode: 'cos1'
              toPort: 'in0'   ofNode: 'mul1']];

[S addEdge:[Edge name: 'e6'
            fromInput: 'A1'
              toPort: 'in1'  ofNode: 'mul1']];
[NT setBody: S ];
```

□

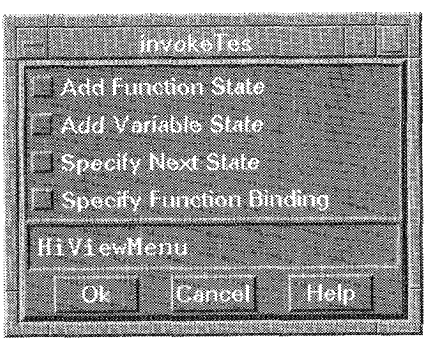

Figure 14.8: Menu for editing AST nodes.

14.8 The Tool 'HiEdit'

We have implemented the tool 'HiEdit' for node editing. The tool allows us to enter a node via various menus and displays a graphical view of the node inside a window. The symbols used to compose the graph are listed in figure 14.9.

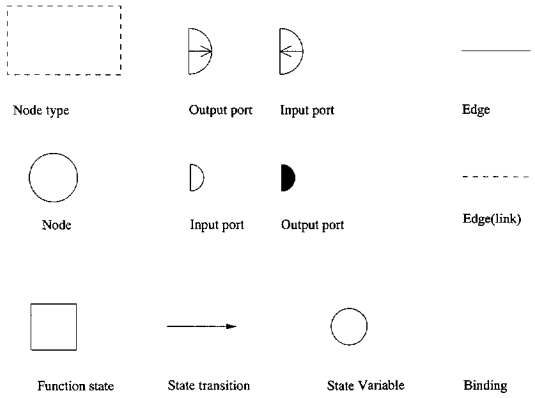

Figure 14.9: The symbols representing HiFi design objects.

The position and size of a symbol on the screen is determined by the attributes of the symbol object. The symbols are stored inside the database in a symbol table. To maintain consistency, the tool *HiEdit* (1) removes automatically a symbol from the symbol table when the real object does not exist and (2) creates a symbol for a real object when it is missing.

Some features of *HiEdit* are:

- moving of symbols through mouse clicking.
- info about symbols through mouse clicking.
- automatic placement of symbols.

The current implementation of *HiEdit* generates a PostScript description of the node type object [88] [29]. *HiEdit* invokes a standard tool to display the generated PostScript descriptions. However, the current implementation should be considered as a prototype and other user interface packages should be taken into consideration, such as 'tk' [53].

14.9 Conclusion

We have presented the data structure representing the data-flow model. It supports important design concepts such as parametrization and hierarchy by specifying a node as a tuple consisting of an interface and a body. Moreover, the specification of the node types fits into the design hierarchy support of the NELSIS CAD framework.

The node type forms the unifying object for the three types of bodies: function, AST, and structure. The conversion tools operate only on the body of node types and keep the interface invariant. In database parlance, the tools produce new versions that may be substituted in the design graph on the basis of functional equivalence, so as to obtain several design alternatives.

Dependence graphs are represented as structures in which the regular parts are specified by lattice and polytope objects. The fact that the data structure is a collection of objects is of particular importance for applying transformations on dependence graphs. It allows us to specify transformations on pieces (segments) of the dependence graph. For instance, the projection tool can be applied on each individual segment of the graph separately.

Chapter 15

The Design Graph

15.1 Introduction

In chapter 13, we captured the SVD specification by a design graph consisting of four design objects. The root design object stores the program of the 'SVD' algorithm. It has hierarchy relationships with the other three design objects which store the functions 'Angle', 'RotRow', and 'RotColumn', respectively.

In earlier chapters, we presented conversion tools by which specifications of functions, ASTs, and structures can be converted into each other. A conversion tool takes as input a design object and returns another design object containing another type of specification. In this chapter, we assume that all nodes have a functional input-output behavior. This means that structure nodes and that AST nodes consume exactly one token from all their (active) input ports and produce exactly one token at all their (active) output ports. Moreover, we assume that structures have only dependence graph specifications. We restrict ourselves thus to nodes with single token rate. The class of nodes that can be built with HiFi is larger, i.e. nodes may have multiple token rates. In the next chapter, we consider the latter class of nodes.
We may substitute these design objects inside the design graph because they have the same functional behavior (input-output behavior).

As a result, the design graph is, in general, a mixture of design objects of any of the three types of specifications. Because both the parent and the child object can be any of the three types, there are nine kinds of hierarchy relationships, each with a different semantics.

The design objects of a design graph store node-types. The data-flow network is obtained by flattening the design graph and by instantiating its node-types. It is a structure of communicating AST nodes that evaluate functions. Such a network is represented by a design graph, which is a directed acyclic graph. The root of the graph is a design object containing a structure specification. This specification has hierarchy relationships with

design objects containing AST specifications, which in turn have hierarchy relationships with design objects containing function specifications. By definition, we call a design graph of this form *realizable*.

A design graph can be transformed into another by transformations. In this chapter, we consider three kinds of transformations: (1) conversion of the type of the design objects by the conversion tools, (2) flattening of hierarchy levels, and (3) introducing structure nodes. The main objective of this chapter is to illustrate how we capture the complete design. The design graph is altered step-by-step and forms, in a way, the state of the design process. Parts of the network are specified temporally, other parts structurally. We illustrate this by a network that consists of the DG structure corresponding to the inner-loops of the SVD algorithm and an AST node that evaluates the outermost loop of the SVD algorithm sequentially.

In section 15.2 we give an overview of the hierarchy relationships and classify them into *realizable*, *flattenable*, and *nonrealizable* relationships. Then each relationship will be discussed in more detail in separate sections.

15.2 Overview Hierarchy relationships

Depending on the type of the parent node and the child nodes there are nine types of hierarchy relationships between structures (S), ASTs (A) and functions (F):

- S-S relationship: parent and child are structures
- S-A relationship: parent is structure, and child is AST
- S-F relationship: parent is structure, and child is function
- A-F relationship: parent is AST, and child is function
- A-S relationship: parent is AST, and child is structure
- A-A relationship: parent and child are ASTs
- F-F relationship: parent and child are functions
- F-S relationship: parent is function, and child is structure
- F-A relationship: parent is function, and child is AST.

Each hierarchy relationship has a different semantics. In general, a data-flow network is modeled by a structure of communicating AST nodes which evaluate functions. This is

represented by hierarchy relationships between structures and ASTs and between ASTs and functions. We call these relationships, the S-A and A-F relationships, *realizable*.

The S-S, A-A, and F-F relationships are *flattenable*, which means that we remove hierarchy. Often we do not flatten hierarchy relationships in order to keep the design specification structured, to cope with the design complexity and to allow operations on individual nodes. Note that a node-node relationship can only be flattened when both nodes are of the same type.

All other relationships are called *nonrealizable*. They allow us to capture the complete design in the form of a design graph. These relationships can be given a meaningful semantics because all nodes are assumed to have a functional input-output behavior. However, when there are nonrealizable relationships, we have to transform the design graph to make it realizable.

We discuss three transformations: conversion between nodes by the conversion tools, such as *HiPars*, flattening of nodes and introducing structures in a systematical way. The latter transformation replaces a parent node and its child nodes by a structure that becomes the parent of the parent node and the child nodes. By doing so, the types of the relationships are changed.

15.3 Parent is function

15.3.1 The F-F relationship

Often, the initial design graph consists only of design objects containing functions, i.e., design objects containing MATLAB programs. The hierarchy relationships between the design objects are, then, of the kind F-F and correspond with function calls.

The F-F relationship is flattenable. Flattening of this relationship means that each function call statement of the parent calling the child is replaced by the child function, with its formal variables substituted by the actual variables of the function call.

Example 15.3.1 [Flattening the F-F relationship]

Program 15.3.1 consists of a loop and a function call. The called function `Angle` is listed in program 15.3.2.

After flattening we obtain program 15.3.3, in which the actual variables are substituted.

□

Program 15.3.1 [Loop Angle]

```
NT = [NodeType name: 'LoopAngle'];
[NT use: 'Angle'];
[Port input: 'a' type: 'Array'];
[Port output: 'th1_1' type: 'Array'];
[Port output: 'th2_2' type: 'Array'];
Function = {

  for i=1 : 2 : M-1,

    [th1_1( i ),th2_1( i ) ] = Angle( a(i,i),a(i,i+1),a(i+1,i),a(i+1,i+1) );

  end
}
```

□

Program 15.3.2 [Angle]

```
NT = [NodeType name: 'Angle'];
[Port input: 'A11' type: 'float']];
[Port input: 'A12' type: 'float']];
[Port input: 'A21' type: 'float']];
[Port input: 'A22' type: 'float']];
[Port output: 'th1' type: 'float']];
[Port output: 'th2' type: 'float']];
Function = {

[tmp1, tmp2] = computeTh(A21,A12,A22,A11);

if abs(tmp2) < (pi/4),
   if sign(tmp1) == -1,
      th1= add( tmp1, pi/2 );
      th2= add( tmp2, pi/2 );
   else
      th1= sub( tmp1, pi/2 );
      th2= sub( tmp2, pi/2 );
   end;
end;
}
```

□

Program 15.3.3 [Flattened]

```
NT = [NodeType name: 'StageAngle'];
[Port input: 'a' type: 'Array']];
[Port output: 'th1_1' type: 'Array']];
[Port output: 'th2_2' type: 'Array']];
Function = {

for i=1 : 2 : M-1,

   [tmp1, tmp2] = computeTh(a(i+1,i),a(i,i+1)),a(i+1,i+1),a(i,i) );

   if abs(tmp2) < (pi/4),
      if sign(tmp1) == -1,
         [th1_1( i )] = add( tmp1, pi/2);
         [th2_1( i )] = add( tmp2, pi/2);
      else
         [th1_1( i )] = sub( tmp1, pi/2 );
         [th2_1( i )] = sub( tmp2, pi/2 );
      end
   end
end
```

□

15.3.2 The F-A and F-S relationships

The F-A and F-S relationships are nonrealizable. These relationships may occur inside the design graph as an intermediate state of the design process. They are not realizable because, by definition, no assertions are made about the way a function is refined either temporally or structurally. The fact that the parent is specified as a function overrules the refinement of the child.
There are several options to remove such relationships, if desired. They are: (1) converting the parent into a structure to obtain a realizable S-A or a flattenable S-S relation. (2) converting the parent into an AST resulting in an A-A or A-S relationship. (3) converting the child to a function type.

15.4 Parent is AST

15.4.1 The A-F relationship

As defined in chapter 12, an AST evaluates in each of its function states a function, which is decomposed into an application part f_a and a control part f_c. These functions are captured in separate node-types in the design graph. The A-F relationships are hierarchy relationships between the parent AST node and the functions that are referred to in its function states. Thus the design objects capturing the functions are children of the design object capturing the AST.
By definition, an A-F relationship is realizable.

15.4.2 The A-A relationship

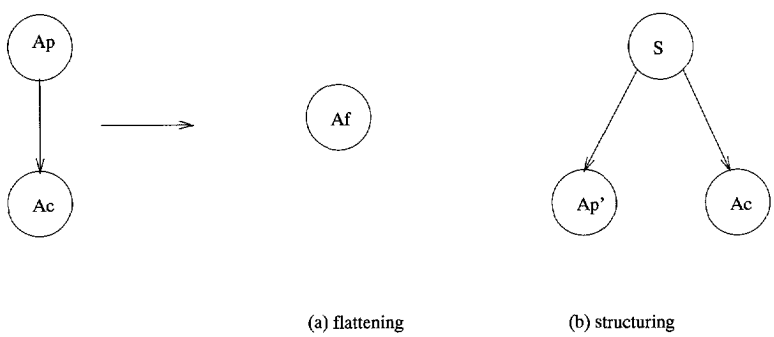

Figure 15.1: Two transformations to remove the A-A relationship. (a) flattening (b) introducing a parent structure.

When the function bound in a function state of an AST is not specified by a function but by another AST, the hierarchical relationship between the parent AST and child AST is an A-A relationship. This relationship is only valid when the child AST has a functional input-output behavior. The A-A relationship is flattenable.

There are several transformations conceivable to remove this relationship, if desired. Apart from conversions to function or structure, we can flatten the two ASTs into a single AST. Another option is to introduce a structure in which both ASTs become instances. As a result, the A-A relationship is transformed into two S-A relationships. In the following subsections, we describe these two transformations in more detail. They are illustrated in figure 15.1.

15.4.3 Flattening AST nodes

Let A_p be the parent AST and A_c be the child AST. Flattening results in a new AST type in which the child A_c is integrated into the parent A_p. We denote the new AST by A_f. As a result the hierarchy relationship disappears. See figure 15.1 (a). A_f can be substituted for the two ASTs in the design graph.

The AST A_f is obtained as follows. The ports and the parameters of A_f are the same as those of A_p. The function states and variable states of A_f are the unions of the states of A_p and A_c, with the following exceptions. Firstly, the current state variable of the child is not represented in A_f.
Secondly, a function state F' of which the f_a part refers to A_c is replaced by a function state F' of which the application part f_a is void, since the application part is now carried out by the function states corresponding to the child AST. The function state F' is only needed for the control. The control function of F' is the same as the one of F. Thus f_c of F' is identical to f_c of F.
Thirdly, the interface of A_c, its formal ports and parameters, becomes redundant. Thus, the bindings of function states of A_c to (formal) ports of A_c become bindings to the (actual) ports or variables of A_f according to the binding of f_a in A_p. Parameters of A_c are substituted by their parameter expressions defined in A_p.
Finally, state transitions in A_f must be defined to and from the function states originating from the child AST. The state transitions in A_p to function state F become transitions to the function state in A_f corresponding to the initial function state of A_c. The transitions going to the initial function state of A_c become transitions to F' in A_f.

Example 15.4.1 [Flattening AST]

Figure 15.2(a) shows the AST corresponding to the loop statements of program 15.3.1. Figure 15.2(b) shows the AST of modeling the function 'Angle'. After flattening of the child AST and the parent AST figure 15.2(c) result.

□

15.4.4 The Structuring transformation

Another transformation is to introduce a structure S that becomes the parent of the two AST nodes. See figure 15.1(b).
Both the parent AST and the child AST are instantiated in structure S. In terms of the design graph, this means that the original design graph with the `A-A` relationship is transformed into a design graph with `S-A` relationships.

The interface of the structure type is identical to that of the parent AST. The child AST remains unchanged.
The parent AST is modified as follows. Let F be the function state of A_p of which the

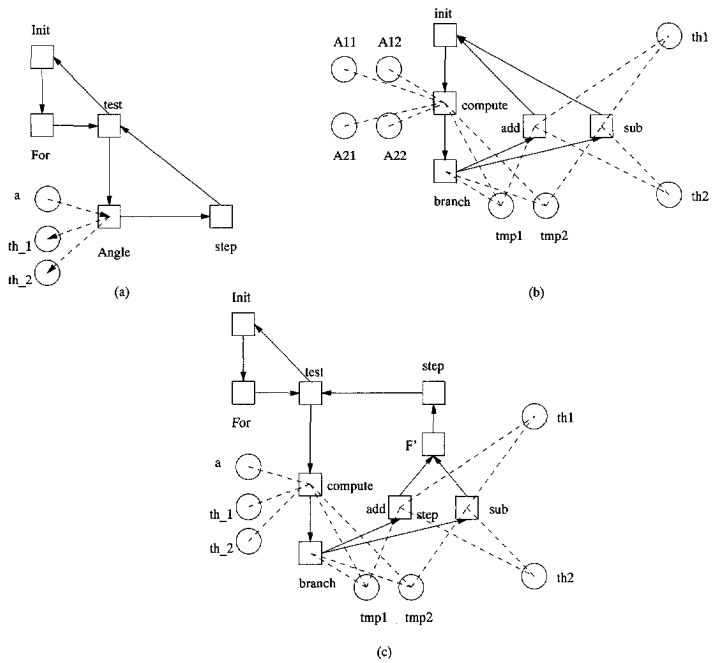

Figure 15.2: (a) the AST node modeling the loop statement, (b) the (simplified) AST node modeling the function `Angle`, and (c) the result after flattening.

application part f_a refers to the child A_c. The function bindings of F are transformed to external ports. The input bindings for f_a become output ports, the output bindings become new input ports of the modified AST. In the structure, edges connect these input and output ports with the appropriate port of the child node.
The function state F is replaced by two function states: F_w and F_r. The function state F_w maps the inputs of f_a to the new output ports. F_r reads the tokens produced by the child AST and maps their values to the original bound ports and variable states of f_a.

The child AST communicates only with the parent AST node. We call it a co-AST. Note that the function states F_w and F_r synchronize on token passing between the co-AST. They fire only when all input tokens and output tokens of the co-AST are available. This synchronization can cause considerable overhead. We can reduce the overhead by reordering the function states such that in between the function state F_r and F_w other function states are evaluated. These function states are then evaluated in parallel with the function states being carried out by co-AST. However, this requires detailed timing analysis of the function states as well.

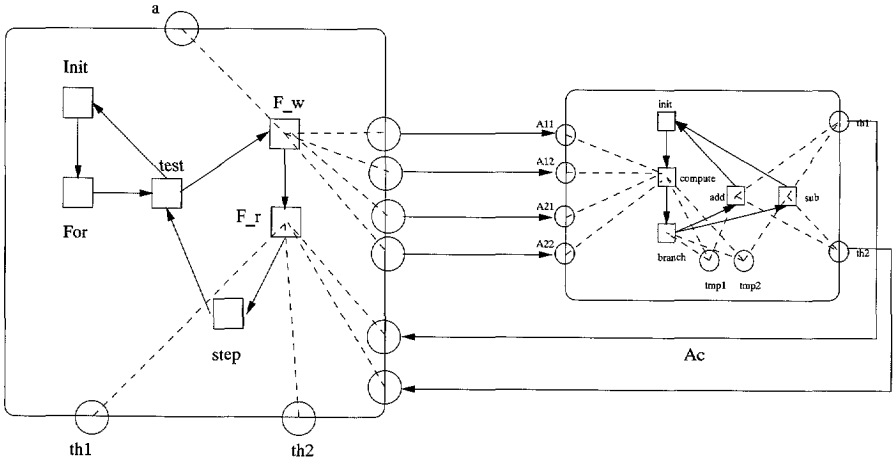

Figure 15.3: The structure resulting after transforming an A-A relationship.

Example 15.4.2 [co-AST]

Figure 15.3 shows the structure in which the AST 'Angle' appears as co-AST. The original function state F has four input bindings, which become four ports. In function state F_w tokens are placed on these ports. The ports are connected to the co-AST by four edges. The value of the angle variables $th1$ and $th2$ computed by the co-AST are the values of the tokens send back via two edges and are read by the function state F_r.

□

15.4.5 A-S relationship

The A-S relationship occurs when a function of a function state of the parent AST is specified by a structure. We assume that the structure is a DG, which has been obtained, for instance, by *HiPars*. This relationship is also nonrealizable.

Here, we can also alter the design graph by applying the structuring transformation. This transformation introduces a structure node that has as children the modified parent AST and the child structure (DG). So a function of the AST is evaluated in parallel with a structure of nodes (co-processor array).

The parent AST node is modified in the same way as discussed in the previous section. Let F be the function state of which the part f_a is specified as a structure (DG). The bindings of f_a become ports which are connected to the ports of the structure by edges. F is split

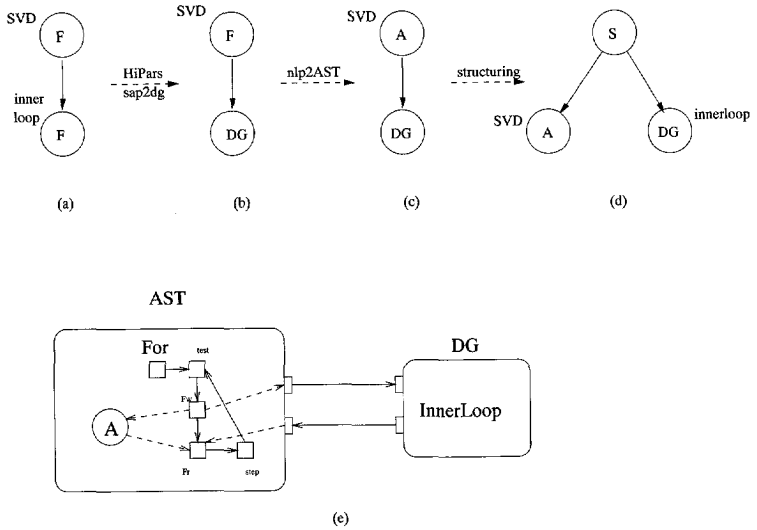

Figure 15.4: (a)..(d) chain of the design graph transformations leading to the structure (e) with the AST node modeling the stage loop and the DG-node representing the DG of the inner-loops of the SVD algorithm.

into two function states F_w and F_r that writes the tokens to, and reads the tokens from the structure, respectively.

To illustrate the transformations on a design graph, consider figure 15.4 (a). It shows a design graph consisting of two nodes representing the SVD algorithm. The node labeled 'SVD' contains the algorithm:

```
for stage = 1 to N,
    [A]= innerloop(A);
end
```

The node labeled 'innerloop' contains the function `innerloop`, which is the body of the stage loop of program 2.3.5.

Figure 15.4(b) is obtained by applying the tool *HiPars* on node 'innerloop' and then the tool *sap2dg* to produce the dependence graph, as described in chapter 9.
Next, the function 'SVD' is converted in an AST by the tool *nlp2ast*. The result is shown in figure 15.4(c).
Finally, figure 15.4(d) is the design graph after applying the structuring transformation.
Figure 15.4(e) shows the structure S containing the AST node computing the stage loop, and the (hierarchical) node containing the DG. The input and output port of the AST node

are of type domain, which size corresponds with the size of the matrix A. The ports of the structure are connected to the AST node by hierarchical edges. They are hierarchical ports and their types are defined by port-domains.

This realization has large synchronization effects, because all output tokens must be present at the ports before the function state F_r fires. This means that the structure (DG) is free of tokens when function state F_w puts new tokens at its input ports. As a result, there is no pipelining in the structure. This can be optimized by splitting the function states F_r and F_w, which causes the synchronization, into several function states of which each one partly reads or writes the ports of the structure.

15.5 Parent is Structure

15.5.1 S-S relationship

Nodes of a structure (DG) may be structures (DG) themselves. This leads to an S-S relationship. S-S relationships are flattenable.

Let S_p and S_c be the parent structure and the child structure, respectively.
By flattening, we obtain a new structure, say S_f. The set of nodes of S_f is the union of the sets of nodes of S_p and S_c. The ports of S_c disappear as a result of the flattening. Edges connected to these ports are directly connected to the ports of the nodes of S_f.

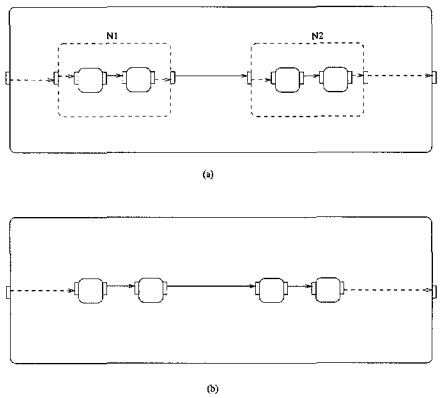

Figure 15.5: Flattening of structures. (a) a structure of two instances of another structure type, (b) the structure obtained by flattening the structure nodes $N1$ and $N2$.

Example 15.5.1 [flattening]

Figure 15.5 illustrates the flattening of structures. Figure 15.5(a) shows a structure consisting of two structure nodes $N1$ and $N2$. Figure 15.5(b) shows the structure resulting after flattening. Observe that the ports of the structures disappear.

□

15.5.2 The S-A and S-F relationships

The structures that represent data-flow networks are structures of AST nodes. This is represented by the the S-A relationship, which is a realizable relationship.

The S-F relationship is nonrealizable. By definition a function is not a node. We can convert the function type into an AST type which results in the S-A relationship. A simple conversion is to create an AST type that has a single function state of which the application part is specified by the function type.

Another option is to convert the function type into a structure type resulting in an S-S relationship.

15.6 Conclusion

When design objects contain functional behaviors, we can capture a design as an acyclic graph. Its vertices represent design objects and its edges represent hierarchy relationships. We have divided the relationships into realizable, flattenable, and nonrealizable. A realizable design graph may only contain flattenable and realizable relationships. The nonrealizable relationships can be removed by node-type conversion or by other transformations. In this chapter, we have shown the structuring transformation. There are other transformations possible. In the next chapter, we describe the AST splitting transformation.

We transform step-by-step a design graph into a design graph that can be realized as a network of AST nodes. In the next chapter, we present a set of tools with which we can design data-flow networks from dependence graph descriptions. As we will show, these transformations can be expressed in terms of design graph transformations too.

Chapter 16

Dependence Graph Transformations

16.1 Introduction

In this chapter we discuss several transformation tools that operate on dependence graphs. We do not discuss the working of these tools, for which we refer to [23], but rather show how these tools can be applied on a DG node at a certain level of hierarchy in the design graph. The transformation tools we discuss are:

- linear projection
- regularization
- partitioning.

In our model, a DG is regarded as a special kind of network, in which each edge passes a single token and each node fires only once. In order to reduce the number of network elements, we can project the DG onto a data-flow network. The nodes of the data-flow network are reused and memory buffers are specified to store data to be processed by the nodes in the flow graph.

Linear projection cannot be applied to all DGs directly. Nodes with different behavior cannot be mapped onto a single node in the data flow network, without introducing additional control to select between the different behaviors in time. Also, the resulting network must be free of port conflicts that result when different kinds of edges become connected to the same port. We call a graph *strictly regular* when it can be described by a single node-domain of which all nodes have identical behavior, and by a set of shift-invariant dependence vectors [23]. A strictly regular graph can be linearly projected. In this chapter, we only project strictly regular graph segments although projection can also be applied on graph segments as long as this does not result in node or port conflicts. The regularization

tool is a tool that can make a graph or segments of it strictly regular. The tool essentially hides irregularity by introducing distributed control.

The regularization tool replaces nodes by AST nodes that conditionally evaluate them. Note that the nodes that are controlled by these AST nodes may be AST nodes themselves. The control is thus specified hierarchically in the sense that the nodes between which a selection is made are not affected. The control for the AST nodes is generated by control generator nodes.

The partitioning tool is used to partition a dependence graph into subgraphs, called *tiles*. The size of these tiles is independent of the size parameters of the algorithms. However, the size of the tile does not have to be fixed. It may be parametrized.

The transformation tools perform functional invariant transformations. This means that the transformation tools preserve the input-output behavior of the nodes on which they operate. Each tool has its own set of parameters. For instance, the parameters of the projection tool are a schedule vector and a projection vector. Typically, the input design object on which a tool operates is part of a design graph. The output of a transformation tool can be several design objects, which are organized in the form of a sub design graph. By using functional equivalence, the subdesign graph output of a tool can be substituted for the input design object in the original design graph.

To keep the functional behavior of a node type invariant, we have to introduce *port-adaptors*. These port-adaptors are characterized by the tool parameters and perform a type conversion of the input and output ports.

As a result, we can apply the transformations in any order at any level of hierarchy. However, we have to distinguish between acyclic and cyclic graphs. By definition, non hierarchical DGs are acyclic. Cyclic DGs may appear as a result of the partitioning tool at some settings of the partitioning parameters.

When the graph is cyclic, more levels of hierarchy have to be taken into account when choosing the parameters of the transformation tools. This means that the transformation must satisfy the constraints imposed by the cycles in the graph at the higher levels. When the higher level graphs are acyclic, we can disregard these levels and operate on the DG alone. In this chapter, we do not further consider the problem of tool parameter setting and assume that a proper parameter setting has been chosen.

The outline of this chapter is as follows. In section 16.2, we discuss the regularization transformation and present its output in terms of node type objects. In section 16.4, we discuss the projection of a DG onto a structure and consider the port-adaptors that go with it.
In section 16.5, we discuss the partitioning tool. In section 16.6, we consider some combinations of transformations and describe how the design can be optimized.

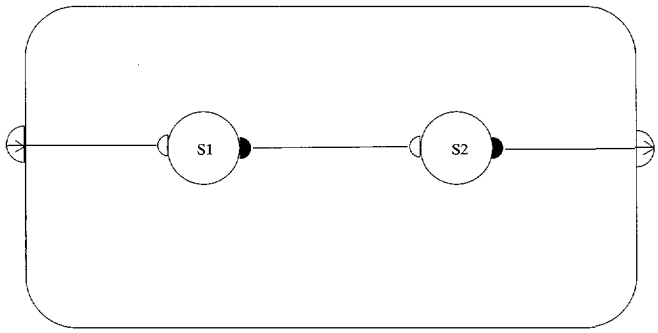

Figure 16.1: The DG node consisting of two segments of which the nodes are of type NT_1 and NT_2, respectively. The input and output ports of the DG node are indicated by the small arrows.

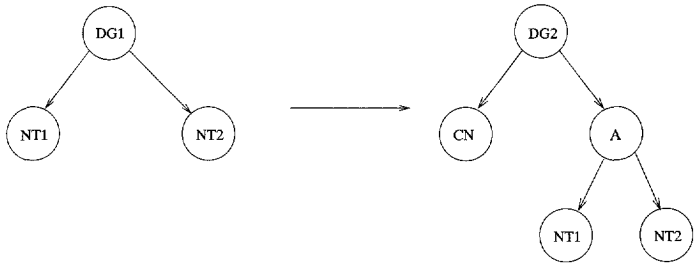

Figure 16.2: The design graph of the example before and after regularization.

16.2 Regularization

One of the tools of the design system is the 'regularization' tool [23]. To be able to project a dependence graph, or segments of it, we have to make the graph or segments strictly regular. This means that nodes in the graph or segments having different behavior and to be mapped on the same node of the resulting or target data-flow network have to be replaced by AST nodes that select between the behaviors by interpreting control signals on their control inputs. Also, we resolve potential port conflicts, which may occur through projection, in the data-flow network by introducing AST nodes that select ports by interpreting control signals on their control inputs.

The control is generated by control generating nodes and distributed into the regular network through local edges carrying the control tokens. The number of control edges that is needed depends on the number of inequalities defining the port and node-domains of

the segment. We illustrate the method by regularizing a graph consisting of two segments See figure 16.1. Observe that the two segment nodes are graphs themselves. The DG node is thus an hierarchical node [45]. The method can readily be extended to DG nodes consisting of more segments.

Let the node-domains ND_1 and ND_2 of the segments S_1 and S_2 instantiate nodes of type NT_1 and NT_2, respectively.

Figure 16.2 shows the design graph for the example, before (left) and after (right) regularization. The node labeled 'A' is an AST type selecting between the two functional behaviors of NT_1 and NT_2. The node CN generates the control. 'A' and 'CN' are instantiated in the root of the graph, a structure type DG_2. Below we describe both node types in more detail.

First we specify the AST type 'A'. See also figure 16.3. The interface of 'A' consists of the union of the sets of input ports I_1 and I_2 and the union of the sets of output ports O_1 and O_2 of NT_1 and NT_2, respectively, augmented with additional control input and output ports. It has three function states F_1, F_2 and F_c. The application part of the function states F_1 and F_2 are specified by NT_1 and NT_2, respectively. The control is evaluated in F_c, which selects either F_1 or F_2 as the next function state. F_c also assigns the value of the input control token to the output control port in order to propagate the control. F_1 and F_2 always give control back to F_c. Note that by specifying the control in this way the functions referred to in the function states F_1 and F_2 are not affected. The control is thus specified hierarchically.

After regularization, the DG node consists of a single segment which is defined by the AST node 'A' and the union of the domains of ND_1 and ND_2. The DG node is shown in figure 16.4.

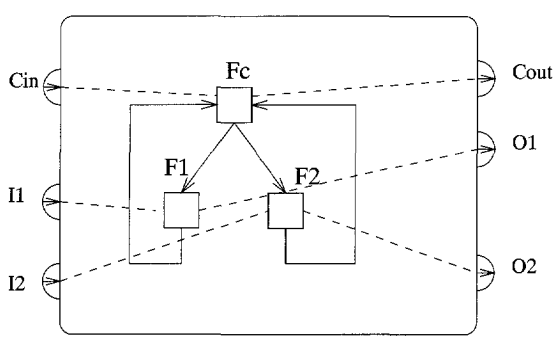

Figure 16.3: The AST node switching between the function states $F1$ and $F2$ of which the application parts are specified by the node types NT_1 and NT_2.

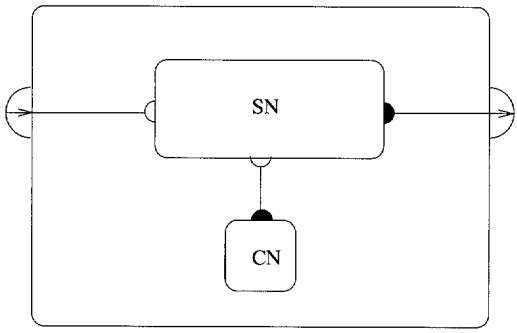

Figure 16.4: The DG node after regularization consisting of a single segment-node 'SN' and a control generator node 'CN'.

The control is generated by a control generator node which is specified as an AST. In principle, a control node is specified for each control variable propagated through the regularized graph segment. The output port of the control node is of type port-domain and connected by an (hierarchical) edge to the proper control input ports of the nodes of the graph segment. The functionality of the nodes of the segment depend on their index position. By adding control. this index dependent behavior is resolved by external control input to the nodes of the segment. Suppose that the index space is divided in two half-spaces defined by the inequality $aI + b \geq 0$. The control node generates control values that depend on the evaluation of the index inequality. For instance, it generates zeros for the nodes that lie in the half-space, and ones otherwise. The template of the algorithm forming the body of the control AST is shown in program 16.2.1. The output port is represented in the algorithm by the variable C. The algorithm consists of a nested loop program that assigns control values to the variable (port) $C(I)$ of the control node. The value depends on the condition statement. The tool *nlp2ast* can be used to convert this control function into an AST type.

Program 16.2.1 [Template Control Generator]

Let C be the hierarchical output port of type (p_c, D_c), with D_c a port-domain and p_c a port. The type of p_c is the same as the type of the ports of the nodes of the segment to which it is to be connected.

For each index $I \in D_c$, the control program assigns each output port $C(I)$ a control value depending on the conditional statement.

```
[NodeType name:  'CN'];
[PortType output:  'C' type:(p_c, D_c) ]
code = {
    for I ∈ D_c do
        if a^T I + b ≥ 0,
            C(I) = 0;
        else
            C(I) = 1;
        end
    end
}
```

□

16.3 AST splitting

The AST produced by the regularization tool defines control to switch between functions (or ports).

However, when a controlled function is actually specified as a structure, as we have shown in the previous chapter, this control does not allow pipelining. Each function state of the control AST that refers to a structure type corresponds with an A-S relationship. By applying the structuring transformation such a function state is split into an F_w and an F_r function state, with F_r being the next function state of F_w. In F_w tokens are supplied to the structure. In F_r the output tokens of the structure are read. As a result, the control AST synchronizes on all tokens input to and output from the structure before going to its next function state.

To introduce pipelining, the control AST is split into an AST that contains the F_w function states, and an AST type that contains the F_r function states.

Figure 16.5(a) shows a control AST that selects between two node types N_1 and N_2. The nodes are structure types. Figure 16.5(a) is obtained after applying the structuring transformation. After splitting the control AST figure 16.5(b) is obtained.

It is easy to verify that the sequencing constraint defined by the original state transition from F_w to F_r is satisfied by the token passing between the control ASTs and the nodes N_1 and N_2. Note that an additional AST is needed to duplicate the control tokens. By inserting additional control buffers in the structure, the pipelining capability of structures can be exploited.

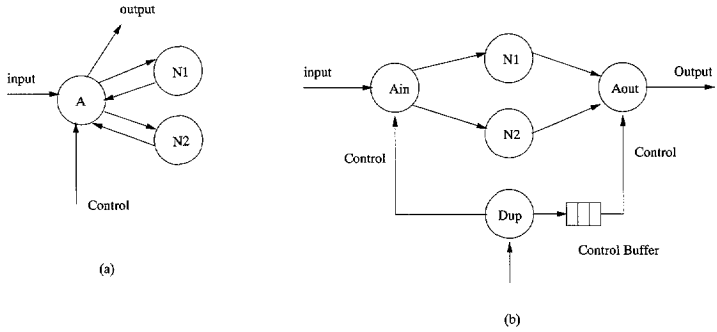

Figure 16.5: (a) Input-output control disabling pipelining. (b) input and output control which allows pipelining.

16.4 Projection

The second transformation we describe is linear projection. Linear projection is applied on a dependence graph captured by a node at a certain level of hierarchy inside the design graph.

Without loss of generality, we assume the DG to consist of a single strictly regular segment. Let I be the index vector on the segment.

Before projecting the segment, it has to have been space-time transformed. Let T be an $n \times n$ integer matrix. Space-time transformation is a partitioning of the index domain into an $(n-1)$-dimensional spatial domain and a 1-dimensional temporal domain.

$$T = \begin{pmatrix} P^T \\ \lambda^T \end{pmatrix}$$

The $(n-1) \times n$ matrix P^T is the subspace onto which the graph is to be projected. The $1 \times n$ vector λ is the schedule vector. The vector u satisfying $P^T u = 0$ is the projection vector. We assume that λ and u satisfy the causality constraint.

Thus the index I on the domain is decomposed into:

$$\begin{pmatrix} t \\ p \end{pmatrix} = TI$$

where t stands for the time index and p of size $n-1$ is the index vector on the projected domain.

The projection tool produces a data-flow graph or structure consisting of AST nodes, with each AST node computing the nodes of the DG that lie on a straight line parallel to the projection vector u [23]. These AST nodes are thus reused.
Dependence graph nodes have by definition functional behavior. This means that nodes consume a single token at all the input ports and produce a single token at all the output ports. The structure that results after projection may consume and produce streams of tokens at its input and output ports.
To keep the input-output behavior invariant, *port-adaptors* are introduced for each incoming and outgoing edge-domain of the projected segment. These port-adaptors order the incoming and outgoing token streams of the structure. The ordering of the tokens is characterized by the space-time transformation matrix T. The input port-adaptor is an AST node that inputs a single token at its input ports and outputs streams of tokens at its output ports. The output port-adaptor inputs a stream of tokens at its input ports and outputs a single token at its output ports.

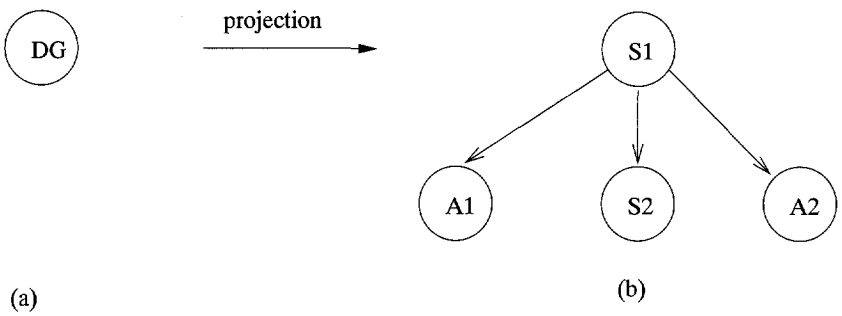

Figure 16.6: The design graph before (a) and after projection (b).

Figure 16.6 illustrates the projection transformation in terms of design graphs. The input to the projection tool is a design object (node type) containing a DG consisting of a single segment-node as shown in figure 16.7(a). The output is a structure type S_1 that instantiates the port-adapters A_1 and A_2 and the structure S_2 that result after projection of the segment. The structure corresponding to the design graph output by the tool is shown in figure 16.7(b).

Port adaptors are specified for each incoming and outgoing edge-domain to the segment that we project. Let ED be an incoming edge-domain of the graph segment with input port-domain $PD_{in} =< p_{in}, D_{in} >$.
The port-adaptor has two ports a_{in} and a_{out}. The input port a_{in} is an hierarchical port of type PD_{in};the output port $a_{out} = (p_{in}, D_{out})$ is an hierarchical port of which the domain D_{out} is defined by the projection of the domain D_{in}.
The input port of the port-adaptor is connected to ED. Its output port is connected to the ports of the nodes of the structure.

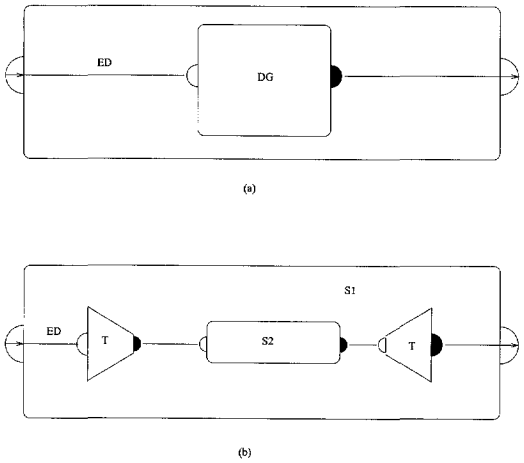

Figure 16.7: The graph segment before projection (a) and the structure encapsulated with port-adaptors after projection (b).

Program 16.4.1 [Template Input port-adaptor for Projection]

Let l_1, l_2, ..l_n and u_1, u_2, ..., u_n be the expressions of loop bounds.

```
[NodeType name:  'IPA'];
[PortType input:  a_in type:(p_in, D_in) ]
[PortType output:  a_out type:(p_in, D_out) ]
code = {
for t = l_1 to u_1,
    for p_1 = l_2 to u_2,
        ...
            for p_{n-1} = l_n to u_n,
                a_out(p_1, .., p_{n-1}) = a_in(t, p_1, ..., p_{n-1})
            end
        ...
    end
end
}
```

□

Program 16.4.1 shows the template of the algorithm of an input port-adaptor. The algorithm scans the space-time transformed domain of its input port. The loop iterator of the outer loop of this program corresponds with the time or sequencing index. So the

tokens are read from the input port and put at the output ports in the order defined by the schedule vector λ.

The scan program can be obtained by the tool 'dg2sap'. We obtain the desired timing behavior by converting this algorithm into an AST, which we can derive with the tool 'nlp2ast'.

16.5 Partitioning

To obtain a structure with a fixed-size and to make the size of the structure independent of the size of the algorithm, we partition the dependence graph into subgraphs of fixed-size [90][50][91] [73].

The input for partitioning is a segment of the DG. The tool produces two node types: (1) the tile graph TG and (2) the tile node TN. The tile nodes are instances of the tile graph. Partitioning introduces thus hierarchy.

Let I be the index on the segment to be partitioned.
Let I_q be the index of the tile graph and let I_r be the index of the tile node.

We consider only a special kind of partition called *tiling* and consider only tiles whose shapes are parallellepipeds.
Let S be an $n \times n$ matrix and $b_s \in \mathbf{Z}^n$.
Partitioning (tiling) is an affine index transformation:

$$I = SI_q + b_s$$

From this it follows that the indices I_q and I_r are given by:

$$I_q = \lfloor S^{-1}(I - b_s) \rfloor$$
$$I_r = I - SI_q$$

In addition, the partitioning transformation produces

- an input port-adaptor, to distribute data to the tiles
- an output port-adaptor, to collect the output data of the tiles.

The adaptors perform a port type conversion. The type of the ports are converted to ports of types that match with the types of the ports of the tiles.

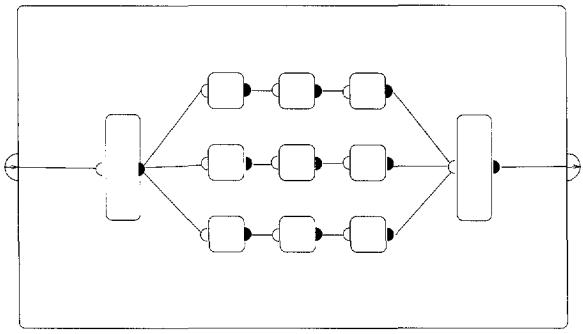

Figure 16.8: The tile graph consisting of nine tile nodes, an input and an output port-adaptor.

Port-adaptors for partitioning are introduced for each incoming and outgoing edge of the segment that is to partitioned.

Let the incoming (hierarchical) edge be $ED =< PD_{in}, PD_{out}, F >$, with PD_{in} an input port-domain, PD_{out} an output port-domain, and F an indexing function.

PD_{in} becomes the type of the input port of the port-adaptor and is connected to ED. The edge is thus not affected. Let p_{tn} be the hierarchical port of the tile node. Then the type of the output port p_{out} is defined by the port-domain $< p_{tn}, D_{tn} >$, with D_{tn} the domain resulting from partitioning of PD_{in}.

The output port is an hierarchical port of which the types of the subports match with the types of the ports of the tiles. The edges of the tile graph are hierarchical edges and stand thus for a set of edges. They carry thus a set of individual tokens. It is convenient to regard sets of individual tokens as hierarchical tokens at the tile graph level.

The input port-adaptor reads the tokens from the (hierarchical) input port and places the tokens onto the output port. Program 16.5.1 shows the template for the input port-adaptor of which the body is defined by an algorithm. The algorithm scans the domain PD_{in} and reads at each iteration I the input port $p_{in}(I)$ and assigns the value of the tokens to the output port whose index is obtained by the partitioning transformation.

We have specified the port-adaptors as an AST node consisting of a single function state. This causes all tokens to become synchronized. This can be avoided by refining the AST to a structure of AST nodes controlling the individual tokens of the hierarchical token.

Program 16.5.1 [Template Input port-adaptor for Partitioning]

[NodeType name: 'IPA'];
[PortType input: p_{in} type: PD_{in}]
[PortType output: p_{out} type: (p_{tn}, D_{tn})]
code = {
    for $I \in PD_{in}$ do
        $I_q = \lfloor S^{-1} I \rfloor$
        $I_r = I - SI_q$
        $p_{out}(I_q)(I_r)$ = $p_{in}(I)$;
    end
}

□

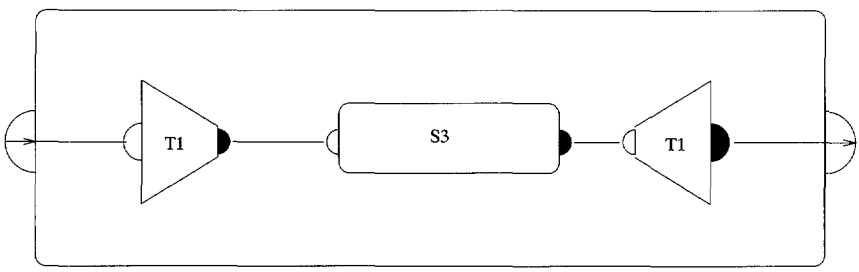

Figure 16.9: The structure resulting after projection of the tile with input and output port-adaptor introduced by the projection tool.

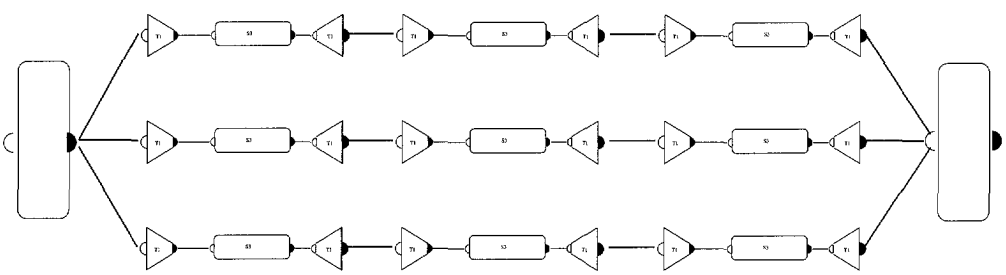

Figure 16.10: The structure resulting after partitioning of the DG and projection of the tiles.

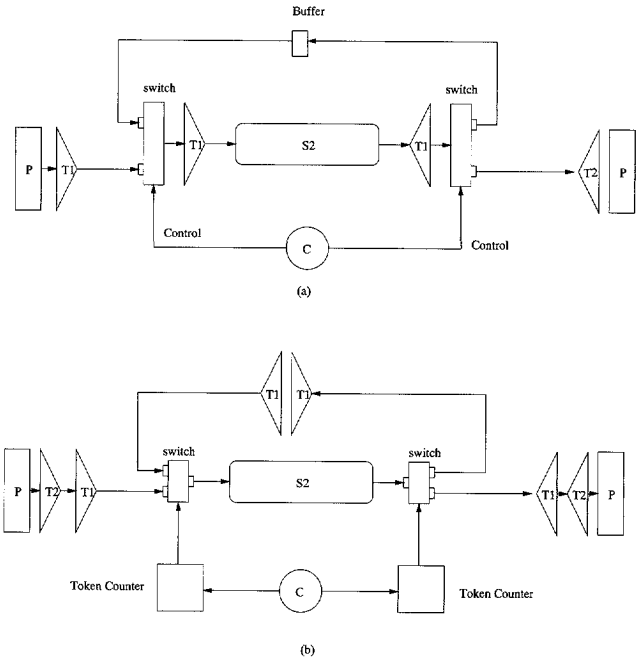

Figure 16.11: (a) The DG after partitioning and projection with control specified on (hierarchical) tokens at the tile graph level. (b) the optimized structure with control on tokens at the DG level.

16.6 Combinations of Transformations and Optimizations

In this section, we combine the three transformations discussed in the previous sections. Without loss of generality, we illustrate this for a particular combination of transformations.

We start with the partitioned DG shown in figure 16.8.

Secondly, we project the tile. The projected tile result is shown in figure 16.9. It is a structure with input and output port-adaptors that order the incoming and outgoing tokens.

Thirdly, we project the tile graph shown in figure 16.8 without bothering about the refinement of the tile nodes. In order to project it, we specify control by the regularization tool. This control is needed to select between the local edges and the external edges connected to the port-adaptors introduced by the partitioning transformation. Projection of the regularized tile graph results in the structure shown in figure 16.11(a).

The ports of the AST switches, introduced by the regularization, are of type domain. Let D be the domain of one of the ports. At this level of hierarchy the edges are in fact sets of edges. Each control token corresponds to an hierarchical data token. The function states of the switch AST fires only when all $||D||$ subtokens of the hierarchical token are present at its input (sub-) ports. After firing, a new control token is read. It has thus a function node behavior.

In this structure there is thus considerable synchronization overhead because the switch ASTs are introduced at the tile graph level. To reduce this overhead, we can split the switch AST up into a structure of AST nodes, on a node-domain defined by D, switching between the individual tokens of the hierarchical token.

Another, more attractive, option is to replace the switch ASTs by ASTs before the output port-adaptor and after the input port-adaptor. See figure 16.11(b). These AST nodes thus switch on the individual tokens in the token streams. Control tokens are generated for each individual data token by a token counter node. The body of the token counter node is specified by an algorithm that scans the domain D and outputs a control token for each point of D. It counts a sequence of $||D||$ tokens.

The input and output adaptors in the self-loop of the structure (figure 16.11 (b)) can be combined. In general, an input port-adaptor connected to an output port-adaptor can be combined into a single AST node with a state variable. The size of this local variable can be optimized by life-time analysis [49]. When the parameters of the input and output adaptor are the same, the AST can be replaced by FIFOs.
Figure 16.11(b) shows a concatenation of port-adaptors between the original input and output port and the structure. They perform the input and output data stream processing. These operations are typically carried out by a host.

16.7 Conclusion

We have shown how transformation tools can be applied at a certain level of hierarchy. The regularization tool defines control at a certain level of hierarchy without effecting the nodes at lower levels of hierarchy. However, in general, the higher the level at which control is introduced, the larger is the size of the set of tokens represented by a hierarchical token and the larger the synchronization overhead. This overhead can be reduced by transforming the control for an hierarchical token to control of the individual tokens represented by the hierarchical token; in other words to specify to control at a lower level of hierarchy. Or in other words, the use of distributed control is a good solution for these problems.

The projection and partitioning tools provide also the port-adaptors by which the interface is kept invariant. This allows to combine the transformations. Finally, this chapter has shown that the HiFi's set of conversion tools are in a way complementary to the set of transformation tools operating on the dependence graphs.

Chapter 17

Conclusion

In this dissertation, we have addressed the problem of mapping algorithms on architectures, in particular parallel processing architectures. These architectures are needed in digital processing applications where huge computational power is required. We have presented a number of design tools that are part of the HiFi design system. The tools can be used to model algorithms and architectures. The HiFi model is based on the AST model. Architectures are represented by data-flow networks and are modeled by structures of AST nodes. Algorithms are represented by the dependence graph model, which has been superimposed on the AST model.

The first problem that a designer has to solve is to find a suitable algorithm and to express it in the DG model. In other words, the algorithm has to be expressed in an applicative specification which makes the parallelism in the algorithm explicit.

We chose to start from a procedural specification of the algorithm instead of an applicative specification. This means that the algorithm is written in a procedural programming language such as C or MATLAB. This offers several advantages for the designer: (1) the imperative programming style is well-known (2) many algorithms are available in this form (3) he does not have to learn a new programming language (4) he avoids direct writing in an applicative language, which is often a tedious and error-prone process. However, it has to be noted that not each sequential program is suited. The sequential program must implicitly possess the parallelism. To obtain a suitable program for parallelization, a lot of algorithmic engineering may be required.

We have implemented the tool *HiPars* by which the designer can derive automatically an applicative specification from a procedural specification of an algorithm. We call the class of programs that *HiPars* accepts *nested loop programs*, as control is typically specified by one or more nested loop stages. The nested loop programs may be parameterized, but they must have static control and their expressions must be affine ones.

The tool HiPars uses the PIP routine for the data dependence analysis. This routine finds closed index expressions for the data dependencies that result from the read and write accesses to variables in an algorithm, by solving a parameterized linear programming problem.

There are other techniques for solving the data dependency problem, but these techniques have more or less the same restrictions as PIP, and they can also not deal with data dependent dependencies and dependencies resulting from nonlinear index expressions. As we have shown, PIP can be used to handle quasi-linear index expressions and step sizes greater than one in loop statements.

The data dependence analysis is detailed down to the level of iterations of the algorithm. Thus, the analysis goes further than dependence tests which are often used in parallel compilers.

The output of *HiPars* is a single assignment program (SAP), which is functionally equivalent to the input nested loop program. The SAP program is an executable program, which can be tested by running it in the MATLAB environment. In the dissertation, we have derived the SAP program for the SVD program. It shows that a SAP program can have a complicated control structure consisting of nested conditional statements and possibly additional integer division expressions. The control structure is often difficult to interpret. We have shown how this control structure can be simplified by rewriting the integer divisions as modulo operations.

HiPars can be used as a stand-alone tool without the HiFi system. The tool is available via Internet (http://dutentb.et.tudelft.nl/research/hifi.html).

The SAP forms an intermediate format from which it is a small step to the DG model. The DG is piecewise regular and consists of a number of segments which are composed of port-, node- and edge-domains. The domains are characterized by linearly bounded lattices. The DG forms a memory and control-free algorithm specification. The edges between the ports of the nodes of the DG are specified by affine index functions. The index function produces the index of the output port given the index of the input port. It is possible to use relations instead of functions to specify the edges. The advantage of using functions is that their mathematical properties are well understood, and can be exploited to define and implement the transformation tools.

Apart from historical reasons, the network or dependence graph approach has several advantages.

- It gives a complete, transparent description of the data dependencies.
- It lends itself easily to geometrical interpretation. For instance, irregularities are easily to detect by visualizing a structure as a graph.
- It introduces hierarchy in a natural way.

The model has thus been developed from an architectural point of view and the DG is regarded as special kind of network (structure). The model is closed, which means that also the architecture that results after a series of transformations is expressed in the model. The alternative for the DG model is to use an applicative language such as SILAGE and ALPHA. One might argue that the use of a language is a more formal approach than the DG model, which is a more pragmatic approach. Perhaps it is desirable to propose a common language/format. This is possible because the languages have much in common, such as, for instance, the fact that they all use linearly bounded lattices to describe regularity. However, due to the different backgrounds, it is not easy to enforce the use of a common language among different research groups.

The fact that the DG model is based on the AST model means that an explicit timing specification is part of it. We made some refinements in the specifications of AST nodes. Firstly, the possible next function states are specified for each function state explicitly, which makes it possible to draw state transition graphs. A second refinement we made was to split the state variables into control and data variables. We have shown how the sequential behavior of nested loop programs can be modeled by AST nodes. Note that the class of programs that can be modeled by an AST is larger than the restricted class of programs that HiPars accepts. Further the AST plays an important role in the control specification. The regularization tool, for instance, combines several functions into a single AST node that is controlled by external control tokens. We have also shown that the control token generators are modeled by AST nodes.

The design methodology can be summarized as follows. First, a completely parallel specification is derived in the form of a dependence graph. Then, the DG is folded back so that parts of the DG are expressed again as functions or ASTs. In a way, the tools applied in the transformation trajectory do the inverse operation of *HiPars*. The clustering and projection tools clearly illustrates this. The result of the transformations is an irregular network of concurrently operating nodes (processors) which are connected to each other by edges. It shows the importance of an integrated model that allows both regular and irregular networks to be described.

The network model lends itself also in a natural way to hierarchical design. There are two ways to refine functions (1) temporally and (2) structurally. The structures and AST nodes are special as they may be replaced by the functions from which they are derived. Based on the functional behavior of the nodes, we defined the design graph as consisting of nodes (objects) and hierarchy relationships between them. Based on functional equivalence, nodes in the design graph may be substituted. Typically, a design is entered as a design graph consisting of function nodes. In this form also the SVD algorithm has been entered into the system. As an example, we have transformed the design graph of the SVD algorithm into a design graph in which the outermost loop is modeled by an AST node and the inner loop stages by a structure node, which is, in this case, a very logical decomposition because the inner loops contain most of the available parallelism.

We have also built the tool *HiCompose* that composes an abstract or hierarchical graph

of a DG. This graph can be interpreted as a network consisting of hierarchical nodes. This representation shows resemblance to drawings of reduced dependence graphs as can be found in the literature.

Hierarchy also affected the way the transformation tools were set up. The objective is to apply transformations on one level of hierarchy. For this reason, the transformation tools should be functionally invariant transformations. The transformed graph should have the same functional input-output behavior. We have realized this by introducing input and output port adapters so that nodes at a higher hierarchical level in the design graph are not affected. The use of hierarchy may lead to inefficient designs, as it may introduce unnecessary synchronization. However, the power of hierarchy as a design concept is that it gives overview and helps to build prototypes. Many questions remain to be solved on the efficient hierarchical partitioning of the design.

We have built the HiFi system on top of the NELSIS CAD Frame. The use of this CAD frame offers several advantages. In the first place, it offers a design repository consisting of design objects in which the design is stored. Secondly, it offers data management such as versioning and concurrency control, which are certainly needed when several designers are working on the same project. Further, the CAD frame features a graphical user interface. The tools are arranged in flow graphs via which the design tools can be invoked by mouse clicking. We have made separate flow graphs for the set of conversion tools, transformation tools and design support tools. The NELSIS CAD Frame is also used to capture the hierarchy relationships in a design graph.

The usage of the HiFi design system is as follows. The HiFi system has been set up as an open and interactive system where the designer has to make the design decisions. For instance, he has to decide which part of the algorithm is to be realized as structure and which part as AST. To make these decisions, the designer has to have measures of the quality of the resulting architecture. The DG model can be analyzed to obtain these cost and performance measures. The measures of cost can be divided into those of time and space. Measures of space are the number of processors (AST) and edges. Measures of time are computation and communication time. For this purpose, the designer is supported by a set of tools such as monitor and query tools by which the measures can be determined. These tools have not been described in the dissertation. An important performance measure that can be determined by these tools is the theoretical speed up, which is defined as the ratio between the maximum and minimum computation time. The maximum computation time is the time to evaluate the algorithm sequentially. The minimum time corresponds with an 'as soon as possible' schedule and is determined by a longest path analysis of the DG. The actual speed up depends on the scheduling of the operations and lies somewhere in between the minimum and maximum computation time.

Another reason why the actual speed up is lower than expected is because of overhead, for instance, because of the port-adaptors that are needed to input and output data to a processor array. To a major extent, this overhead is caused by the time wasted due to communication and synchronization. Here lies one of the main topics for future research.

How do we find an optimal design strategy to obtain an architecture with minimal overhead? Or more generally, how can a designer easily explore the design space and come up with an effective or optimal design. With an optimal solution we mean a solution that just meets the design constraints defined when specifying the problem. This is a very complicated optimization problem as there are many design parameters involved. In addition, the design constraints typically vary per design problem. Sometimes we need to find an optimal design in terms of processing power, another time, we need a design that has maximal throughput. As the problem is so complicated, the solution should be based on highly automated optimization techniques. The question whether the complete design trajectory can be fully automated remains open. We have chosen the HiFi system to be a semi-automatic system. Future extensions to the HiFi system, will be needed to support the designer in the iterative design process. Thus tools should provide the designer with the necessary design information on which he can base his decisions, and which keep him on track through the design space leading to the optimal design, in terms of some predefined design constraints. However, this will still require a lot of research in the future.

Bibliography

[1] Nana2 - novel parallel algorithms and new real-time vlsi architerural methodologies part ii. Technical report, October 1991.

[2] R. Roy A. Paulraj and T. Kailath. A subspace rotation approach to signal parameter estimation. In *Proc. IEEE*, volume 74, pages 1044–1045, 1986.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proc. ACM SIGPLAN '91*, pages 39–50, 1991.

[4] J. Annevelink. A hierarchical design system for vlsi implementation of signal processing algorithms. In *Proceedings of IEEE/ProRISC Workshop on Circuits, Systems and Signal Processing*, 1985.

[5] J. Annevelink. *HIFI: A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays.* PhD thesis, Delft University of Technology, The Netherlands, 1988.

[6] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Comm. ACM*, 21:613–641, 1978.

[7] D.G. Baltus and J. Allen. Efficient exploration of nonuniform space-time transformations for optimal systolic array synthesis. pages 428–441, 1993.

[8] Donald G. Baltus. *Efficient Exploration Of Affine Space-Time Transformations For Optimal Systolic Array Synthesis.* PhD thesis, Massachusetts Institute of Technology, February 1994.

[9] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[10] W. Barnier and J.B. Chan. *Discrete Mathematics with Applications.* West Publising Compagny, 1989.

[11] R.P. Brent, F.T. Luk, and C. van Loan. Computation of the singular value decomposition using mesh-connected processors. *J. VLSI Computer Systems*, 1:242–270, 1984.

[12] L. Brickman. *Mathematical Introduction to Linear Programming and Game Theory.* Springer-Verlag, 1989.

[13] J. Bu. *Systematic Design of Regular VLSI Processor Arrays.* PhD thesis, Delft University of Technology, Delft, The Netherlands, May 1990.

[14] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, pages 155–182, April 1994.

[15] INC CAD FRAMEWORK INITIATIVE. Tool encapsulation specification, version 1.0.0-112592. Technical report, CAD FRAMEWORK INITIATIVE INC, 1992.

[16] Z. Chamski. *Environment logicel de programmation d'un accélérateur de calcul parallèle.* PhD thesis, L'Université de Rennes, France, 1994.

[17] Z. Chamski. Fast and efficient generation of loop bounds. In *Parallel Computing: Trends and Apllications*, pages 265–272. North-Holland, 1994.

[18] Z. Chamski. Mathpip: A mathematica interface for pip. user's guide and reference manual. Technical Report 94-6-1, Univerisity of Manchester, 1994.

[19] Z. Chamski. Enumeration of dense non-convex iteration sets. In *Proc. IEEE of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, 1995.

[20] Mentor Graphics Corporation. Dsp architect dfl user's and reference manual. software version 8.5_4. Technical report, 1993.

[21] L. Dekker. Structured parallel computation. In *Pro. of Twenty-Fourth Annual Hawaii Int. Conference on System Sciences*, volume 1, Januari, 1991.

[22] E.F. Deprettere, editor. *SVD and Signal Processing: Algorithms, Applications and Architectures.* North-Holland, 1988.

[23] E.F Deprettere, P. Held, and P. Wielage. Model and methods for regular array design. *Int. J. of High Speed Electronics and systems*, 4(2):Special issue on Massively Parallel Computing–Part II, 1993.

[24] P. Le Guernic et al. Signal-a data flow oriented language for signal processing. *IEEE Trans. Acoust., Speech, Signal Processing*, ASSP-34(2):362–374, 1986.

[25] P. Feautrier. Parametric integer programming. *Recherche Opérationelle; Operations Research*, 22(3):243–268, 1988.

[26] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[27] F. Fernandez and P. Quinton. Extension of chernikova's algorithm for solving general mixed linear programming problems. Internal Report 437, IRISA, Campus Universitaire de Beaulieu, 35042-Rennes Cédex, France, October 1988.

[28] K. Vince Fernando. Linear convergence of the row cyclic jacobi and kogbetliantz methods. *Numerische Mathematik*, 56:73–91, 1989.

[29] C. Geschke. Postscript language, tutorial and cookbook. Technical report, Adobe Systems, 1990.

[30] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM Journal on Numerical Analysis*, 2, 1965.

[31] G.H. Golub and C.F. Van Loan. *Matrix Computations, (2nd ed.)*. John Hopkins University Press, 1989.

[32] P.R. Halmos. *Naive set theory*. Springer, New York, 1970.

[33] G. Hekstra and R. de Zwart. Token based circuits. Technical report, Dept. Electrical Engineering, Delft University of Technology, 1993.

[34] G.J. Hekstra. *vlsi* **cordic** *redesign* : phase i. Technical report, Dept. Electrical Engineering, Delft University of Technology/STW DEL00.2331, 1991.

[35] P.C. Held. Hipars' reference guide. Technical report, Dept. Electrical Engineering, Delft University of Technology, 1993.

[36] P.C. Held. *HiPars*: a tool for automatic conversion of nested loop programs into single assignment programs. Technical report, Dept. Electrical Engineering, Delft University of Technology, 1994.

[37] P.C. Held and E.F. Deprettere. Hifi: From parallel algorithm to fixed-size vlsi processor array. In Francky Catthoor and Lars Svensson, editors, *Application-Driven Architecture Synthesis*, pages 71–92. Kluwer Academic Publishers, Dordrecht, 1993.

[38] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[39] K. Jainandunsing. *Parallel Algorithms for Solving Systems of Linear Equations and Their Mapping on Systolic Arrays*. PhD thesis, Delft University of Technology, Delft, The Netherlands, January 1989.

[40] G. Kahn. The semantics of a simple language for parallel processing. In *Proc. of the IFIP Congress 74*. North-Holland, 1974.

[41] A.C.J. Kienhuis. Parallelizing nested loop programs containing div, floor, ceil, mod and step functions. Master's thesis, Delft University of Technology, Department of Electrical Engineering, 1994.

[42] H.T. Kung. The structure of parallel algorithms. *Adv. Comput.*, 19:65–111, 1980.

[43] H.T. Kung. Why systolic architectures? *Computer*, pages 37–45, Jan. 1982.

[44] H.T. Kung and C.E. Leiserson. Systolic arrays for vlsi. In *Sparse Matrix Proceedings*, pages 245–282. Philadelphia:Society of Industrial and Applied Mathematicians, 1980.

[45] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[46] F. Lorenzelli and K. Yao. An integral matrix-based technique for systematic systolic design. Technical report, University of California at Los Angeles, 1995.

[47] F.T. Luk. A triangular processsor array for computing singular values. *Linear Algebra and its Applications*, 77:259–273, 1986.

[48] F.T. Luk. On the equivalence and convergence of parallel jacobi svd algorithms. *Proc. SPIE Int.Soc.Opt.Eng*, 862:152–159, 1987.

[49] F.Balasa F.Franssen F.Catthoor H.De Man. Transformation of nested loops with modulo indexing to affine recurrences. In C.Lengauer P.Quinton Y.Robert L.Thiele, editor, *Special issue of* Parallel Processing Letters *on* Parallelization techniques for uniform algorithms. World Scientific Pub., 1994.

[50] J.A. Martens. Partitioning of parametrized dataflow graphs, concepts and implementation. Master's thesis, Delft University of Technology, Department of Electrical Engineering, 1993.

[51] V. Maslov. Lazy array data-flow dependence analysis. In *Proceedings of the 21st annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 311–325, January 1994.

[52] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[53] J.K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley Publishing, Redwood City, CA, 1994.

[54] A. Farina P. Kapteijn, E.F. Deprettere. Implementation of the recursive qr algorithm on a 2×2 cordic testboard: a case study for radar applications. In *Proc. 25th European Microwave Conference*, pages 500–505, 1995.

[55] C. C. Paige. Computing the generalized singular value decomposition. *SIAM J. Sci. Stat. Comput.*, 7:1126–1146, 1986.

[56] C. C. Paige. On the quadratic convergence of kogbetliantz's algorithm for computing the singular value decomposition. *Linear Algebra and its Applications*, 77:301–313, 1986.

[57] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[58] P. Quinton. The systematic design of systolic arrays. *Tech. Rep. 193*, 1983.

[59] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Int. Symp. on Comput. Arch.*, pages 208–214, June 1984.

[60] C. Ussery R. Lipsett, C. Schaefer. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[61] S. Rajopadhye, L. Mui, and S. Kiaei. Piecewise linear schedules for recurrence equations. In *VLSI Signal Processing V*, pages 375–384, Napa Valley, 1992.

[62] J.F. Collard P. Feautrier T. Risset. Construction of do loops from systems of afffine constraints. Internal Report 93-15, ENS Lyon, May 1993.

[63] T. Risset. *Parallélisation automatique du modèle systolique à la compilation des nids de boucles.* PhD thesis, L'Ecole Normale Superieure de Lyon, France, 1994.

[64] R. Roy. *ESPRIT.* PhD thesis, Stanford University, CA, 1987.

[65] H. Samson. *Formal Verification and Transformation of video and image specifications.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 1995.

[66] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley, 1986.

[67] D.S. Scott. Parallel block jacobi eigenvalue algorithms using systolic arrays. *Linear Algebra and its Applications*, 77:345–355, 1986.

[68] D.N. Smith. *Concepts of Object-Oriented programming.* Mc Graw-Hill,Inc, 1991.

[69] The Stepstone Corporation, 75 Glen Road; Sandy Hook, CT 06482. *Objective-C Compiler v4.0. User Reference Manual*, 1989.

[70] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Technical University of Eindhoven, The Netherlands, July 1991.

[71] G. Strang. *Linear Algebra and its Applications.* Academic Press, 1980.

[72] G. Strang. *Introduction to Applied Mathematics.* Wellesley-Cambridge Press, 1986.

[73] J. Teich. *A Compiler for Application-Specific Processor Arrays.* PhD thesis, University Saarbruecken, 1993.

[74] Inc. The MathWorks. *PC MATLAB User's Guide.* 1987.

[75] Adaptive Filter Theory. *S. Haykin.* Prentice-Hall International Editions, 1991.

[76] L. Thiele. On the design of piecewise regular processor arrays. In *Proc. IEEE Symp. on Circuits and Systems*, pages 2239–2242, Portland, 1989.

[77] L. Thiele. On the design of piecewise regular processor arrays. In *Proc. IEEE ISCAS*, 1989.

[78] L. Thiele and U. Arzt. On the synthesis of massively parallel architectures. *Int. J. of High Speed Electronics and Systems*, 4(2):99–131, 1993.

[79] T.H. Tzen and L.M. Ni. Dependence uniformization: A loop parallelization technique. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 4(5):547–558, 1993.

[80] A.J. 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.

[81] A.J. van der Hoeven, A.A.J. de Lange, E.F. Deprettere, and P. Dewilde. A new model for the high level description and simulation of vlsi networks. In *Proc. 26th DAC Conf.*, June 1989.

[82] A.J. van der Veen and E. F. Deprettere. Parallel vlsi matrix pencil algorithm for high resolution direction finding. *IEEE Trans. Signal Processing*, 39(2):383–394, 1991.

[83] P. van der Wolf. *CAD Frameworks: Principles and Architecture.* Kluwer Academic Publishers, Boston/Dordrecht/London, September 1994. ISBN 0-7923-9501-8.

[84] H.W. van Dijk and Ed. F. Deprettere. Transformational reasoning on time-adaptive jacobi type algorithms. In *Proceeding of the 3rd International Workshop on SVD and Signal processing*, pages 277–286, 1994.

[85] H.W. van Dijk and E.F. Deprettere. Transformational reasoning on time-adaptive jacobi type algorithms. In M. Moonen and B. De Moor, editors, *SVD and Signal Processing III*, pages 277–286. North-Holland, 1995.

[86] M.F.X.B. van Swaaij. *Data Flow Geometry: Exploiting Regularity in System-level Synthesis.* PhD thesis, Katholieke Univeriteit Leuven, Belgium, December 1992.

[87] H. Le Verge. *Un environment de transformations de programmes pour la synthèse d'architectures régulières.* PhD thesis, l'Université de Rennes, 1993.

[88] J. Warnock. Postscript language, reference manual. Technical report, Adobe Systems, 1990.

[89] P. Wielage. PhD thesis, Delft University of Technology, to appear.

[90] P. Wielage, P.C. Held, and E.F. Deprettere. On the design of fixed-size processor arrays for piecewise regular algorithms. In *Proceedings of IEEE/ProRISC Workshop on Circuits, Systems and Signal Processing*, pages 267–273, march 1994.

[91] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Int. J. of High Speed Electronics and Systems*, 2(4):452–471, 1991.

[92] Kung S. Y. *VLSI Array Processors*. Prentice Hall, 1988.

Samenvatting

Functioneel ontwerp van data-flow netwerken

Digitale signaalbewerking wordt steeds vaker gebruikt in allerlei toepassingsgebieden zoals mobiele telefonie, radar, video, audio etc. In de meeste gevallen betekent het gebruik van een signaalbewerkingsalgoritme dat zeer veel operaties moeten plaatsvinden in een korte tijd. Een veel belovende techniek om de benodigde rekenkracht te verkrijgen, is het laten uitvoeren van de signaalbewerkingsalgoritmen op speciaal daarvoor ontwikkelde computer architecturen, waarin de bewerkingen in serie en parallel worden uitgevoerd. Het afbeelden van een algoritme op zo'n architectuur is in het algemeen een complexe zaak. Voor een bepaalde klasse van algoritmen zijn formele ontwerp methoden en technieken ontwikkeld. Kenmerkend voor de methode die in dit proefschrift gevolgd wordt, is de afhankelijkheidsgraaf. Via deze graaf kan precies bepaald worden welke operaties van een algoritme in serie, dan wel parallel kunnen worden uitgevoerd. In dit proefschrift zullen een aantal programma's (tools) worden beschreven die deze ontwerpmethode ondersteunen. Deze 'tools' zijn geïntegreerd in het ontwerpsysteem 'HiFi'. De resulterende architecturen kunnen worden omschreven als data flow netwerken. Deze netwerken bestaan uit een aantal gelijktijdig opererende processen die elk een bepaald programma uitvoeren en onderling kunnen communiceren. De communicatie tussen de processen is asynchroon.

De gevolgde methode vereist dat het algoritme in het model wordt uitgedrukt. Dit houdt in dat het algoritme moet worden beschreven in een applicatieve specificatie, die het parallellisme in het algoritme expliciet maakt.

We hebben ervoor gekozen om van een procedurele specificatie van het algoritme uit te gaan in plaats van een applicatieve specificatie. Het algoritme wordt dus geschreven in een procedurele programmeertaal. Dit biedt een aantal voordelen aan de ontwerper: (1) de procedurele programmeerstijl is algemeen bekend, (2) veel algoritmen zijn in deze vorm beschikbaar, (3) het schrijven in een applicatieve programmeertaal wordt vermeden, omdat het vaak een moeizame en foutgevoelige taak is. Natuurlijk moet het sequentiële programma geschikt zijn, d.w.z. het sequentiële programma moet impliciet parallellisme bezitten. Het verkrijgen van een programma dat geschikt is voor parallellisatie kan veel algoritmisch ontwerp en onderzoek vergen.

In hoofdstuk 2 geven we als voorbeeld een sequentieel programma voor het vinden van de singuliere waarde van een matrix (SVD).

Voor het converteren van een procedurele specificatie naar een applicatieve specificatie is het programma *HiPars* ontwikkeld. In de hoofdstukken 3-7, beschrijven we achtereenvolgens de klasse van algoritmen die *HiPars* kan analyseren, de methode waarmee het onderhavige data afhankelijkheids probleem wordt opgelost en tenslotte de uitvoer. De uitvoer is een 'single assignment' program' (SAP), dat precies de data afhankelijkheden tussen de individuele operaties van het programma beschrijft. Een kenmerkende eigenschap van een SAP programma is dat elke variable precies één enkele keer een waarde krijgt.

In hoofdstuk 8 wordt de relatie beschreven tussen index domeinen en de besturingsstructuur van het SAP programma.
In hoofdstuk 9 wordt het afhankelijkheidsgraaf model gepresenteerd. De graaf kan bestaan uit een aantal regelmatige delen. Deze delen van de graaf worden beschreven door middel van index domeinen. De afhankelijkheidsgraaf beschrijft het maximaal beschikbare parallellisme van het gekozen algoritme. De graaf vormt tevens het algebraïsche object waarop transformaties kunnen worden toegepast. Als voorbeeld worden enkele transformaties uitgevoerd op de afhankelijkheidsgraaf die is afgeleid van het SVD algoritme.

In hoofdstuk 10, wordt een programma beschreven waarmee we zogenaamde hiërarchische graven kunnen afleiden van afhankelijkheidsgraven. Door toepassing van het concept hiërarchie wordt een overzicht verkregen van doorgaans complexe afhankelijkheidsgraven. Het aantal componenten waaruit de hiërarchische graaf bestaat, is direct gerelateerd aan het aantal regelmatige delen waaruit de onderliggende graaf is opgebouwd. Elke component kan op zich weer bestaan uit een graaf.

Het is vaak niet realistisch om de afhankelijkheidsgraaf direct op hardware af te beelden vanwege het grote aantal processoren en verbindingen die daardoor zouden ontstaan. Om een architectuur te verkrijgen van realistische afmetingen is het veelal noodzakelijk meerdere operaties samen te voegen, te clusteren, in een enkele processor. Het gedrag van deze processor wordt gespecificeerd door middel van een procedureel programma. Voor dit doel hebben we het programma *dg2sap* geschreven dat wordt beschreven in hoofdstuk 11.

Elk proces in het uiteindelijke netwerk is in principe onafhankelijk en heeft een bepaald dynamisch gedrag.
In hoofdstuk 12 wordt het dynamisch gedrag van processen gemodelleerd.

Ten slotte wordt in de hoofdstukken 13 - 16 ingegaan op de systeem aspecten van het ontwerp systeem. Aan de orde komt de wijze waarop de ontwerpen in een data base worden opgeslagen en hoe de ontwerper stap voor stap en interactief een architectuur kan afleiden door het aanroepen van de tools. Ook wordt aandacht besteed aan de implementatie van het systeem, dat gebaseerd is op moderne object-georiënteerde methoden en technieken.

Peter Held

Acknowledgements

I would like to thank all the people who helped, supported, and encouraged me during the last five years. I have found the help and support of colleagues, friends, and family indispensable. But first of all, I would like to thank my promotor Prof. dr. P.M. Dewilde who gave me the unique opportunity to work as a researcher at the Network Theory section. I have always been proud to be part of his section. He kept me on track at a number of occasions. Then I have to thank Dr. E.F Deprettere, my project leader. I remember the many fruitful brainstorming sessions I had with him. When there was a problem, he always managed to find a good solution, and he was at his best when there was a great time pressure.

Then I would like to thank my direct colleagues in the project. My thoughts go to my friend and colleague Mark Bloemendaal, who did a great job in managing the HiFi group in the first years of the project. Also I have to thank my colleague Paul Wielage, who was on the project whole the time, and with whom I had many useful discussions.
I was also very fortunate to have several students around who contributed to the project: Arnoud Martens, who worked on the user interface, and Bart Kienhuis, who did a great job in further developing the tool HiPars.
Further, I would like to thank the colleagues of the NELSIS CAD Frame work, Pieter van der Wolf, Alfred van der Hoeven, and Olav ten Bosch.
Finally, I would like to thank all the people of the Network Theory section of whom I want to mention explicitly Patrick Groeneveld, Gerben Hekstra, Alle-Jan van der Veen and secretary Corrie Boers. At this place, I also want to thank all the colleagues of the NANA partners; it was a great pleasure to cooperate with them.

Also I have to thank Andries, Koos and my parents for the support I have received during the years, and above all, I'm grateful for the love I received from Ilse.

Curriculum Vitae

Peter Held was born in 1964 in 's-Gravenhage, The Netherlands. He attended the Groen van Prinsterercollege in that city, and received the VWO diploma. In 1982, he became a student at Delft University of Technology in the Faculty of Electrical Engineering.
He joined the Network Theory Section in 1985, where he wrote a Master's thesis on data-flow simulation for parallel computer networks. He received the Master's degree in Electrical Engineering in 1987. During the next three years he worked for Philips Consumer Electronics in Eindhoven. There he was involved as a software engineer in the development of an electronic still picture camera, and later in the development of a semi-professional camcorder. In 1991, he joined again the Network Theory Section as a researcher, and became a member of the HiFi research group, which developed a design system for massively parallel systems. The HiFi project was headed by Prof. dr. P.M. Dewilde and Dr. E.F. Deprettere and was part of two European Basic Research Action programs (BRA 3280 - NANA, and BRA 6632 - NANA2). The main part of this dissertation is about the HiFi system. During the years, he extended the design methodology and concepts on which the system has been built and worked on the implementation of the HiFi system by adding several design tools. His current interests lie in the field of software engineering and he is currently working for ICT, a software company in The Netherlands.