Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures.

Bart Kienhuis
UC Berkeley
Cory Hall 524, Berkeley
California, 94720 USA
kienhuis@eecs.berkeley.edu

Edwin Rijpkema
Leiden University
P.O. Box 9512
Leiden, The Netherlands
rijpkema@liacs.nl

Ed Deprettere
Leiden University
P.O. Box 9512
Leiden, The Netherlands
edd@liacs.nl

ABSTRACT

This paper presents the Compaan tool that automatically transforms a nested loop program written in Matlab into a process network specification. The process network model of computation fits better with the new emerging kind of embedded architectures that use coprocessors. Process networks can describe both fine-grained and coarse-grained parallelism, making the mapping of the applications easier.

Keywords

Process Networks, Matlab, Mapping, Embedded Architectures.

1. INTRODUCTION

A new kind of embedded architectures is emerging that is composed of a microprocessor, some memory, and a number of dedicated coprocessors that are linked together via some kind of programmable interconnect (See Figure 1). These architectures are devised to be used in real-time, high-performance signal processing applications. Examples of these new architectures are the *Prophid* architecture [12], The *Jacobium* architecture [15], and the *Pleiades* architecture [1], to be used in respectively, video consumer appliances, adaptive radar processing, and mobile communication devices. These architectures have in common that they exploit parallelism using instruction level parallelism offered by the microprocessor and coarse-grained parallelism offered by the coprocessors. Given a set of applications, the hardware/software codesign problem is to determine what needs to execute on the microprocessor and what on the coprocessors and furthermore, what should each coprocessor contain, while being programmable enough to support the set of applications.

The applications that need to execute on the architectures are typically specified using an imperative model of computation, most commonly C or Matlab. In Figure 1, for example, we show an algorithm written in Matlab. Although the imperative model of computation is well suited to specify applications, it does not reveal parallelism due to its inherent sequential nature. Compilers exist that are able to extract instruction level parallelism from the original specifications at a very fine level of granularity. They are, however, unable to exploit coarse-grained parallelism offered by the coprocessors of the architectures. This makes the mapping of the applications onto the architecture difficult.

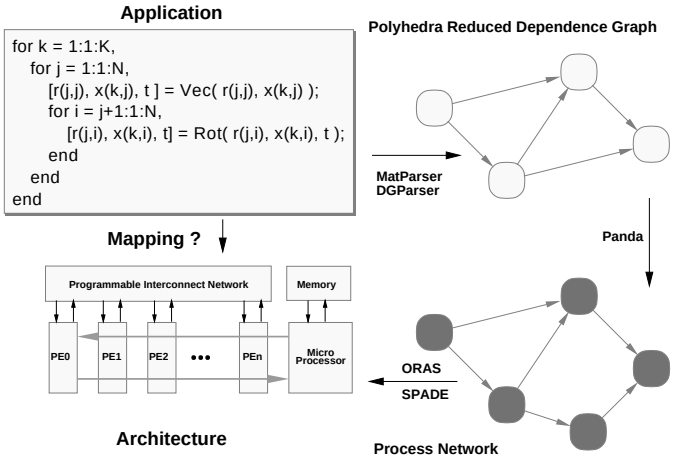

Figure 1: Mapping the application onto an architecture is difficult because the Model of Computation of the application does not match the way the architecture operates.

Instead, a better specification format would be to use an inherently parallel model of computation like *Process Networks* [7; 11]. This describes an application as a network of concurrently executing processes. It describes parallelism naturally from the very fine-grained to the very coarse-grained, it does not pre-impose any particular schedule, and it describes each process in a process network using an imperative language. The mapping then becomes putting the processes either on a microprocessor or on a coprocessor as shown by tools like ORAS [10], or SPADE [13]. Using these tools, a Y-chart [8] can be constructed, allowing the quality assessment of mappings on architectures.

This paper describes the *Compaan* tool that automatically transforms a Matlab application into a process network description, as shown in Figure 1. It converts a Matlab application into a *polyhedral reduced dependence graph*, that is subsequently converted into a *process network* description. The Compaan tool is confined to operate on affine nested loop programs (NLP) [6], but the applications of interest are often described this way.

The Compaan tool describes applications as a process network, which is at a much more coarse-grained level description than a Control Data Flow Graph (CDFG). Moreover, it does a data-dependency analysis on the array domain that goes far beyond the conventional data-

Errata w.r.t. the published version: In the result section, Figure 9. Edge `f` *communicates 500 tokens instead of 15. Again result section; the sequence C,D, and E has to be C, D, and B.*

To be presented at the 8th International Workshop on Hardware/Software Codesign May 3-5, 2000, San Diego, USA

dependence analysis performed on CFDGs. Finally, Compaan syntheses the processes in a way, that each process is a possible implementation model for a coprocessor [8] or a piece of code that executes on the microprocessor.

The outline of the paper is as follows. Section 2 describes the way we decompose the transformation task that Compaan performs into smaller tasks. Section 3 deals with the polyhedral reduced dependence graph (PRDG) that is the model from which Compaan generates a process network. Section 4 explains how processes are structured in so called SBF objects. Section 5 and Section 6 describe the tools inside Compaan in more detail. Section 7 describes how we make the process networks available. Section 8 gives some results and Section 9 gives conclusions.

2. THE COMPAAN TOOL

We developed the Compilation of Matlab to Process Networks (Compaan) tool, which transforms a nested loop program written in Matlab into a process network specification. The tool does this transformation in a number of steps, shown in Figure 2, leveraging a lot of techniques available in the Systolic Array community [16]. In Figure 2, a box represents a result and an ellipsoid represents an action or tool.

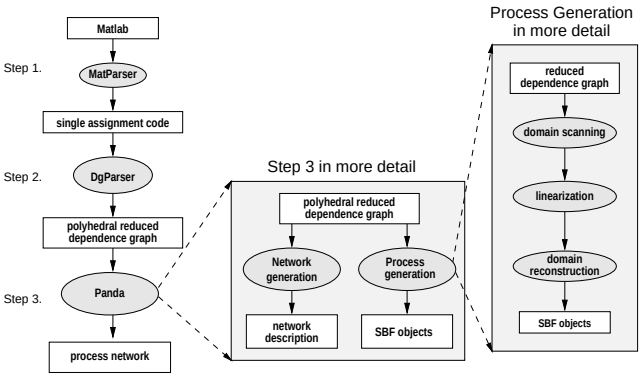

Figure 2: Compaan consists of three tools that transform a Matlab specification into a process network specification.

Compaan starts the transformation by converting a Matlab specification into a *single-assignment code* (SAC) specification. This describes all parallelism available in the original Matlab specification. Next, it derives the polyhedral reduced dependence graph (PRDG) specification from the SAC. From this PRDG, the network description and the individual processes are derived. The three steps done in Compaan are realized by separate tools, respectively, *MatParser*, *DgParser*, and *Panda*.

The last mentioned tool, Panda, uses the PRDG description to generate the network description and the contents of the processes that make up the process network description. The processes are structured in a particular way based on the SBF model, which is explained in Section 4. The SBF model is equivalent to Process Networks [7], with the exception that processes in the SBF model are more structured. The generation of the processes is further decomposed into *domain scanning*, *domain reconstruction*, and *linearization*.

In Section 5 and Section 6, the three tools are discussed in more detail. In the next two sections, we discuss what a PRDG is as well as what the SBF model is.

3. POLYHEDRAL REDUCED DEPENDENCE GRAPH

A *polyhedral reduced dependence graph* is a compact representation of a dependence graph (DG) using parameterized polyhedra, making a DG description more amenable to further mathematical manipulation. A polyhedral reduced dependence graph (PRDG) is a directed graph $G = (V, E)$, where V is a set of *node domains* and where E is a set of *edge domains*. In Figure 3, an PRDG is shown consisting of 5 node domains and 12 edge domains. It is the PRDG representation of the algorithm given in Figure 1.

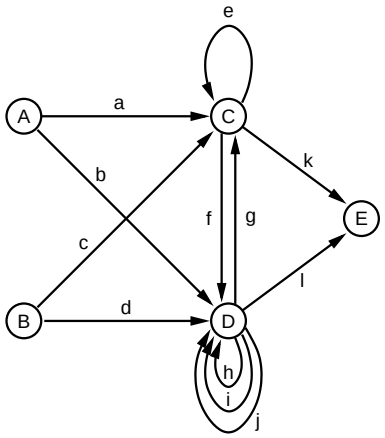

Figure 3: An example of a polyhedral reduced dependence graph.

3.1 Node domain

A node domain is a collection of polytopes [17], a function, and a set of port domains. An *iteration domain* is defined by a polytope in which each point contained corresponds to a *node* in the original DG. With every point inside this iteration domain, the same function is associated. A function has a number of *input ports* and *output ports*. An input port corresponds with an argument of the function; an output port corresponds with a value the function returns. The points of a node domain of which one input port reads data from, or the points of a node domain to which an output port writes data to, form respectively the *input port domain* (IPD) and the *output port domain* (OPD).

3.2 Edge domain

An edge domain is the ordered pair (v_i, v_j) of node domains together with the ordered pair (p_i, p_j) of port domains where p_i is the OPD of v_i and p_j the IPD of v_j. This ordered pair corresponds with a data dependency in a DG, which is expressed using an affine mapping M.

3.3 Example

To illustrate the notion of node and port domains, we show in Figure 4 a node that represents node C in Figure 3. The figure shows the node domain (a), its iteration domain with the iterators i and j (b), its port domains (c)-(f), and its view as it appears in the PRDG (g). Thus, the four port domains (c)-(f) partition the node domain (a) of node C.

In (c) and (d), we show IPDs and in (e) and (f), we show OPDs. In (c) we identify two IPD functions, $ipd_1(i, j)$ and $ipd_2(i, j)$. In (e) we identify two OPD functions, $opd_1(i, j)$ and $opd_2(i, j)$. The figure shows one dependency between opd_1 of port domain (e) and

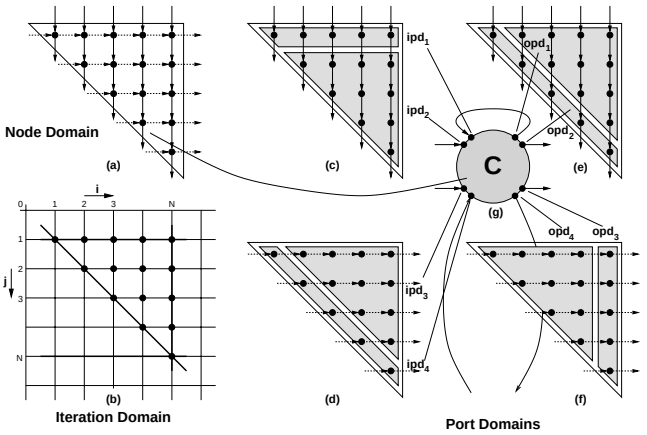

Figure 4: A node domain and its corresponding port domains.

ipd_1 of port domain (c). The dependencies in the PRDG are expressed in terms of an affine function $M(i, j)$ between the different IPDs and OPDs.
The PRDG is basis for the construction of the network description in the SBF model, which is explained next.

4. THE SBF MODEL

The SBF Model [8] describes an application as a network of SBF objects that are interconnected by channels. A *channel* is an unbounded FIFO queue that can contain an infinite sequence of tokens, i.e. a *stream*. SBF objects can write to a channel unconditionally, but can only read from the channel when the queue is non-empty, like a regular process network. An SBF object, however, describes a process in terms of a controller, a state, and a set of functions, as illustrated in Figure 5.

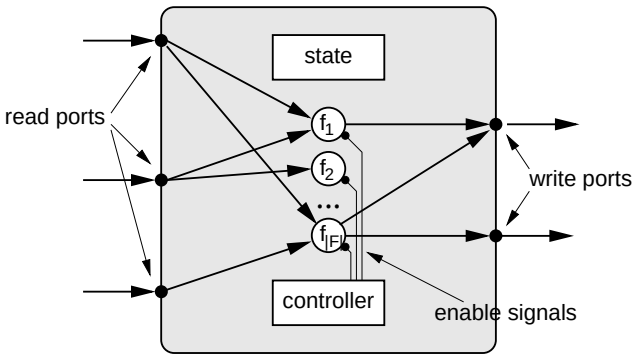

Figure 5: An SBF object.

The set of functions $F = \{f_1, f_2, \cdots, f_{|F|}\}$ defines the *function repertoire*. The state of the object consists of *control state* c and *data state* d. The controller operates on the control state and defines two functions, i.e. the *transition function* ω, and the *binding function* μ which are defined as follows:

$$\begin{aligned} \omega &: C \rightarrow C \\ \mu &: C \rightarrow F, \end{aligned} \tag{1}$$

where C is the space of all possible values of c. The transition function ω determines the new state s' from the current state s. The binding function μ determines what function has to be enabled for the current state s and exactly one function is associated with a state. Enabling of a function is called a *firing*. When a process executes, a sequence of firings will occur as given in equation 2.

$$f_{init} \xrightarrow{\mu(\omega(c))} f_a \xrightarrow{\mu(\omega(c))} f_b \xrightarrow{\mu(\omega(c))} \ldots f_x \xrightarrow{\mu(\omega(c))} \ldots \tag{2}$$

When a function fires, it consumes data from the read ports, from the state, or from both, and it produces data on the write ports, on the state, or on both. Each function knows where to get its input data from and where to send its output data. This leads to so-called *function variants*, which are functions with the same functionality but bound differently to read and write ports, and state.

```
%% Single Assignment Code Generated by MatParser
for k = 1 : 1 : K,
   for j = 1 : 1 : N,

      if k-2>= 0,
        [ in_0 ] = ipd( r_1( k-1, j ) );
      else %% if -k+1 >= 0
        [ in_0 ] = ipd( r( j, j ) );
      end
      if j-2>= 0,
         [ in_1 ] = ipd( x_1( k, j-1, j ) );
      else %% if -j+1 >= 0
         [ in_1 ] = ipd( x( k, j ) );
      end

      [ out_0, out_1, out_2 ] = Vec( in_0, in_1 );

      [ r_1( k, j) ]  = opd( out_0 );
      [ x_1( k, j) ]  = opd( out_1 );
      [ t_1( k, j) ]  = opd( out_2 );
   end
end
```

Figure 6: Single Assignment Code

5. MATPARSER & DGPARSER

In the path from Matlab to the PRDG, Compaan uses the tools *MatParser* [9; 6] and *DgParser* [6]. MatParser is an *array dataflow analysis* compiler that finds all parallelism available in NLPs written in Matlab using a very aggressive data - dependency analysis technique based on integer linear programming [4]. We focus on Matlab since many signal-processing algorithms are written in this language. Just by writing another language front-end, MatParser can also operate on NLPs written in other languages, for example C.
MatParser finds whether two variables are dependent on each other, and moreover, at which iteration. It partitions the iteration space defined by the for-next loops, and gives the dependence vector between partitions. For the simple program given in Figure 1, MatParser solves about a hundred parametric integer program problems to find all data-dependencies.
In Figure 6, part of the output of MatParser is shown for the algorithm given in Figure 1. It shows how the iteration space spanned by the for-next iterators k and j is partitioned using if/else statements. Consequently, for different partitions, different data-dependencies may apply. In case of input argument in_0 of function `Vect`, a value previously defined by function `Vect` should be used (i.e., $r_1(k-1, j)$, defining the data-dependency vector $M()$ or a value from the original r matrix (i.e., $r(j, j)$).
DgParser converts the SAC description into the PRDG description, which is a straightforward conversion. Accordingly, the shape of the node domain is given by the way the for-next loops are defined and the partitioning of the node-domain corresponds with the if/else

conditions. In addition, the terms `ipd` and `opd` used in Figure 6 relate to the IPD and OPD defined in section 3.

6. PANDA

Once DgParser has established a PRDG model of an algorithm, the Panda tool can generate a network description and the individual processes. The network description is straightforward, as it follows the topology of the PRDG. Each node in the PRDG is mapped onto a single SBF object and each edge represents an unbounded FIFO. In case of Figure 3, nodes A, B, C, D, and E define an SBF Object and the edges a until l define an unbounded FIFO.

As shown in Figure 2, the Panda tool divides the generation of an SBF object into three different steps; domain scanning, domain reconstruction, and linearization, which we now discuss in more detail.

6.1 Domain Scanning

Panda needs to derive a transition function ω for each SBF object, a process we call *domain scanning*. For now, Panda constructs ω such that it follows the lexicographical order imposed by the original nested-loop program. Nevertheless, another ordering could have been selected. This may, however, lead to out-of-order problems.

6.2 Domain Reconstruction

MatParser generates a SAC description in which only the IPDs are explicitly specified. This means that the input arguments in_0 and in_1 in Program 6, are surrounded by if/else statements, while the output values out_0, out_1, and out_2 are not. A consequence of this is that output values can be generated that are never used by some input domain. Hence, Panda needs to reconstruct the OPD.

Making the output port domains explicit is illustrated in Figure 7. It shows two communicating node domains ND_p and ND_c. The tokens produced by port domain P_p of node domain ND_p are to be consumed by port domain P_c of node domain ND_c, as described by the data-dependency with mapping M. Port domain P_p is an OPD and port domain P_c is an IPD. To make P_p explicit, Panda applies $M()$ that is derived by MatParser, to IPD P_p, which is a operation on $\mathbb{Z}$-polyhedra [14].

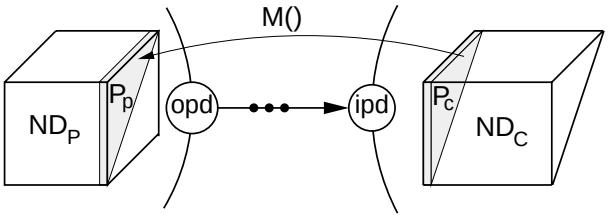

Figure 7: Making the output port domain explicit.

6.3 Linearization

The channels between processes are FIFO buffers and the processes operate using blocking reads. Therefore, the order in which a consuming process reads token from a channel should be the same as the order in which tokens are written onto the channel by the producing process. Now, the way tokens are written on and read from channels is determined by the ω of each process, and can unfortunately easily be chosen in such a way that an out-of-order consumption pattern results. That is, tokens need to be read too early, to allow the process to make progress.

Panda solves the out-of-order problem by storing tokens temporarily in the state of an SBF object, thus operating as a piece of random access memory. This requires that Panda is able to find the proper read and write address for this piece of memory, a process that is called *linearization*.

The linearization method in Panda relies on methods to count the number of integral points contained by a polytope using so-called *Ehrhart Polynomials* [2]. Using such polynomial, and the ω of both the producing and consuming processes, Panda is able to statically derive the read and write address solving the out-of-order processing. Ideally, it should do this under some constraint like throughput or trying to keep the amount of memory needed inside the state of SBF objects to a minimum, as well as the memory required in the FIFO buffers between processes. For the situation shown in Figure 7, the solution with the least memory is the one with the traversal of P_p and P_c the same, requiring no additional state and a very small FIFO.

7. PROCESS NETWORKS

The resulting process networks need to be made accessible in some kind, such that it can be simulated. We generate the process network description for two PN-simulators.

One simulator is *SBFsim*, which is a very fast, very simple simulator in C++ based on threads [8]. In this case, the SBF Objects are generated as C++ classes.

The other simulator is the Ptolemy II framework [3]. In this case, we make a process network available in the PN-domain. Compaan generates the network description in MoML, which is a modeling markup language based on XML [5] used in Ptolemy II for specifying interconnections of parameterized components. The process generation step in this case, generates the Ptolemy II actors in the PN-domain. A MoML description can be executed as an application using a command-line interface or as a visual rendition in the Ptolemy II block diagram editor *Vergil*, as shown in Figure 8. This view of the screen shows the same network as given in Figure 3.

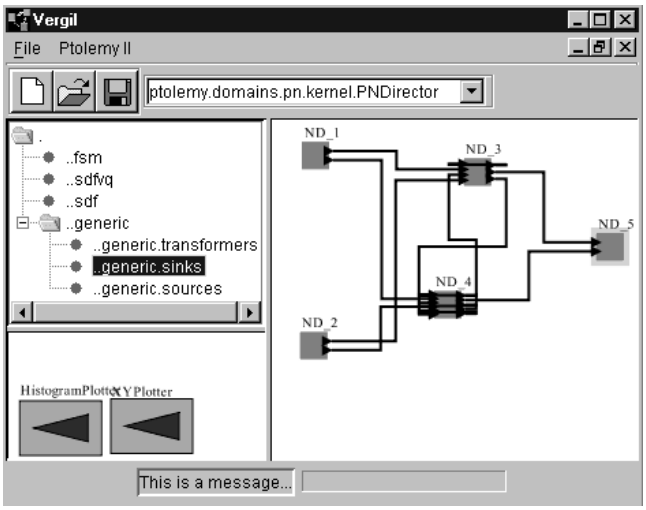

Figure 8: The derived PN network in the Ptolmey II framework.

The Ptolemy II framework enables us to combine the derived process network descriptions with predefined actors like sources to read Matrices and sinks to read and visualize Matrices. It also let us combine process networks with other domains, enabling the description and simulation of more complex systems.

8. RESULTS

We have executed the PN network shown in Figure 8, with the parameter values N=6 and K=100. This gives us the number of times a particular SBF object fired and how many tokens were transported over FIFO buffers between nodes as shown in Figure 9, which describes the same network as given in Figure 3. Thus SBF object A and E fired 21 times, SBF object B and C fired 600 times, and SBF object D fired 1500 times. Furthermore, we see that for example edge b transported 15 tokens, while edge g transported 500 tokens and edge e transported 594 tokens. In Figure 9, the SBF objects that fire more frequently are colored darker and the edges have a different width depending on their communication load.

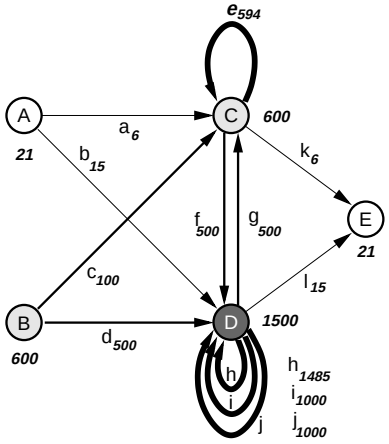

Figure 9: The firing rates found for SBF objects and the communication load found for the FIFO buffers when executing the PN network in Ptolemy II.

From the Figure, we see that some SBF objects fired many times (i.e, node B, C, and D), while others do so sporadically (i.e., A and E). Based on this insight, we can suggest a partition for the architecture shown in Figure 1; the frequently fired SBF objects become candidates for coprocessors whereas the incidental fired SBF objects are put on the microprocessor. This could mean that SBF object C, D and B become coprocessors, while SBF object A and E are mapped onto the microprocessor. Consequently, edges $\{a, b, i, k\}$ map onto the low-bandwidth communication structure that connects the coprocessors with the microprocessor. Edges $\{c, d, f, g\}$ map onto the programmable interconnect network, which is the high - bandwidth communication structure. Edge e and edges $\{i, j, h\}$ map onto internal communication structures inside the coprocessor for node C and D, respectively. This very high-bandwidth communication is thus kept local to the coprocessors. Suggesting such a partition on the basis of the original Matlab program is unlikely.
To further determine the quality of this partition, especially in context of time and limited resources, we can relay on tools like ORAS [10] or SPADE [13]. Because Compaan obtains the network of SBF objects automatically, it could be used in combination with a design space exploration tool.

9. CONCLUSIONS

In this paper, we have described the Compaan tool that can automatically derive a process network description in the SBF model from a nested loop program written in Matlab. Such a network description reveals the parallelism present in the original sequential program. This network description makes the mapping onto the new emerging architectures easier as the granularity and model of computation better fit. A lot of effort is in the synthesis of the SBF objects. An SBF object can now serve as a possible implementation model for a coprocessor, or equally, be put onto a microprocessor. The PRDG model gives us a good mathematical framework to structure SBF objects. We hope, we can exploit this PRDG model to get, for example, SBF objects that use limited state memory inside and require small sized FIFO buffers between processes as shown in Section 6. All elements of the Compaan tool are implemented in Java. With respect to the Panda tool, we are still working on further improvement of the linearization problem. Nevertheless, we have shown for some Matlab programs, that we can automatically compile it, using the trajectory illustrated in Figure 2. For more information about the Compaan work, see `http://www.gigascale.org/compaan`.
This work was supported in part by the MARCO/DARPA Gigascale Silicon Research Center. Their support is gratefully acknowledged.

10. REFERENCES

[1] A. Abnous and J. Rabaey. Ultra-low-power domain-specific multimedia processors. In *VLSI Signal Processing, IX*, pages 461–470, 1996.

[2] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.

[3] E.A. Lee *et al.* Heterogeneous concurrent modeling and design in java. Technical report, University of California, Dept EECS, Nov. 1998.

[4] P. Feautrier. Parametric integer programming. *Recherche Opérationelle; Operations Research*, 22(3):243–268, 1988.

[5] C. F. Goldfard and P. Prescod. *The XML Handbook*. Prentice Hall, June 1998.

[6] P. Held. *Functional Design of Data-Flow Networks*. PhD thesis, Dept. EE, Delft University of Technology, May 1996.

[7] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[8] B. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, Jan. 1999.

[9] B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical Report UCB/ERL M00/9, University of California, Berkeley, CA-94720, USA, Feb. 2000.

[10] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. The construction of a retargetable simulator for an architecture template. In *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, Mar. 15–18 1998.

[11] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[12] J. A. Leijten, J. L. van Meerbergen, A. H. Timmer, and J. A. Jess. Prohid, a data-driven multi-processor architecture for high-performance dsp. In *Proc. ED&TC*, Mar. 17-20 1997.

[13] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings of the 1999 IEEE Workshop in Signal Processing Systems*, Taipei, Taiwan, 1999.

[14] P. Quinton, S. Rajopadhye, and T. Risset. On Manipulating $\mathbb{Z}$-polyhedra. Technical report, Institut de Recherche en Informatique et Systèmes Aléatoires, 1996.

[15] E. Rijpkema, E. F. Deprettere, and G. Hekstra. A strategy for determining a jacobi specific dataflow processor. In *Proceedings ASAP'97 conference*, July 1997.

[16] S.Y. Kung. *VLSI Array Processors*. Prentice Hall Information and System Sciences Series, 1988.

[17] L. Thiele. Resource constrained scheduling of uniform algorithms. In *Conference on Application Specific Array Processors*, volume 20, pages 29–40, October 1993.