PERIODICA POLYTECHNICA SER. ELECTR. ENG. VOL. 41, NO. 3, PP. 201-230 (1997)

THE RAFAEL MULTI-TARGET HETEROGENEOUS SIGNAL-FLOW GRAPH COMPILER

Gábor PALLER and Klára CSÉFALVAY

Department of Electromagnetic Theory
Technical University of Budapest
H-1521 Budapest, Hungary,
e-mail: paller@evt.bme.hu
csefalvay@evt.bme.hu

Received: June 23, 1995

Abstract

This paper describes a signal-flow graph compiler which produces distributed code for heterogeneous target systems. The compiler is devoted for mainly Digital Signal Processing problems. The code generator features reprogrammable operation library, the static scheduler supports fully heterogeneous systems and the input graph may contain run-time decisions in a limited way. The system has been implemented on IBM PC compatibles under MS-Windows so it does not require expansive host computer.

Keywords: compile-time scheduling, parallel processing, heterogeneous architectures.

1. Introduction

Writing programs for the modern Digital Signal Processors (DSPs) introduce difficult tasks for the software engineers because a painful trade-off exists between the computing power and the productivity/task complexity. Unfortunately the existing and well-known higher level programming environments (for example the 'C' language) perform very poorly on the DSP platforms because being general languages they cannot exploit the special capabilities of the DSPs (circular buffers, parallel instructions and so on) or avoiding pipeline effects. This can cause extremely high performance loss (can be as much as 1000% compared to the assembly realization). Several developments were made to improve C compilers on DSP platforms (LEARY and WADDINGTON, 1990) but generally they use system or DSP dependent language extensions and their performance is still not really convincing. So the developers have to choose — writing the DSP code in assembly for achieving higher performance thus lower hardware cost or using a high-level environment which will speed up the development but decrease the efficiency of the DSP so that more expensive DSPs must be chosen. It can even happen that the problem cannot be solved on high level.

The other problem is the embarrassing abundance of DSP architectures and languages. One often faces the problem of porting existing results

onto other DSP platforms. If the code is written in assembly, this will be a long and tiresome process. Some 'common languages' are needed but not having efficiently realizable high level platform this solution does not seem to be promising. Nowadays the solution is sought toward optimized software libraries (like the SPOX) which try to combine the power of assembly routines with the efficiency of C. The SPOX does accelerate the developing process but it is a fixed set of routines and if we extend it (for example we need an arithmetic routine or new algorithm that the SPOX cannot offer) we still have to write it in assembly losing the portability.

Nowadays the parallel DSP is in the focus of attention, first of all because real-world DSP problems often require immense computing power. A number of existing DSPs can be used for parallel realizations, some of them has been designed especially for parallel computing for example Texas Instrument's TMS320C40, TMS320C80 and Analog Devices ADSP21060. The task scheduling is an important part of the multiprocessor implementation of DSP algorithms. This equally means partitioning the tasks among multiple DSPs and scheduling the tasks on each DSP. Generally parallel programs are scheduled 'by hand' in the existing parallel development systems which is a difficult task and in the case of more complex tasks it cannot be done effectively. The other approach used frequently in the existing DSP operating systems uses the well proven real-time operating systems scheme (sometimes time-sliced scheduling is added). This scheme is based on separate tasks and a task scheduler program which changes the tasks when it is necessary. This task scheduler requires processing time.

Speciality of the DSP algorithm is that it does not require much run-time decisions. Very handy description form of these algorithms is the *signal-flow graph (SFG).* Signal-flow graph is a graphical description of an algorithm in which computations are represented by graph nodes and dependencies among the computations by graph branches. If we can cluster enough nodes together that their dependency graph and execution time do not depend on the input values, we can schedule in *compile time* thus eliminating the processor load of the dynamic scheduler.

Thus the DSP code generation problem is the following: we need a system which is flexible enough to be adapted to several existing DSP platforms, avoids the power loss of the high-level languages, solves the partitioning and scheduling problems and in addition it is easy-to-use for the DSP algorithm developer who is generally not a programmer. A proposition for this problem will be presented in this document describing Rafael, an intelligent code generator based on signal-flow graphs.

Rafael was designed as a small, flexible system which can run even on very small computers (it is implemented under Microsoft Windows on IBM PC compatible computers). It is a SFG compiler integrated into a simple

framework which allows DSP algorithms to be described in SFG form and the compiler translates this description into program for a heterogeneous multiprocessor hardware. The compiler distributes the SFG on the multiprocessor system, schedules the operations on each processor, creates the communication scheme among the processors and generates executable assembly source program for each processor. Rafael features a programmable DSP database and code generator library so it can be adapted easily to any processor. Small resources of the host computer do not allow us to compete with the comprehensive features of existing SFG compilers hosted on workstations but we hope to prove that Rafael can compete successfully on several domains with those systems.

2. Existing Data-Flow Compilers

A number of block-diagram based design systems have been introduced in the literature. We mention here the commercially available DSPlay (Burr-Brown) and SPW (Signal Processing Workstation) (Comdisco) systems. DSPlay is PC-based, it can simulate the input block-diagram and can generate code for AT&T DSP32. The Comdisco system started as a simple simulator but actually it is able to produce highly optimized code for almost all the DSP types and can even generate circuit description. Since June 1994 the partitioning on multiprocessor DSP system must have been done by hand. The Cathedral system (DE MAN et al., 1986; LANNEER, 1993) devoted to circuit synthesis features SFG partitioning-scheduling but it uses the Silage functional language (GENIN et al., 1990) as its input. The Ptolemy system (BUCK et al., 1991; BUCK, 1993; BUCK et al., 1994) is the most comprehensive existing simulation/code generation system. Ptolemy supports the coexistence of different computation models (called *domains* by their terminology) and offers clearly defined object-oriented interface for defining a new domain. Existing domains include static dataflow (LEE – MESSERSCHMITT, 1987), dynamic dataflow (BUCK, 1993), discrete event, message queue and communicating process (BUCK et al., 1994) models. Ptolemy makes almost no assumption about the internal structure of the computation models it supports, it is the biggest strongness and weakness of this system. It is a strongness as it allows modelling the whole system including its software, hardware and communication parts in one framework. It is weakness as Ptolemy allows mixing computation models that do not coexist well, it does not force a good design style. Nevertheless, Ptolemy has huge impact on the field and its importance grows continuously as existing computation models and tools are integrated with it.

Many ideas of the structure of Rafael were borrowed from the now historical Gabriel system. Gabriel was phased out in favor of the much bigger Ptolemy system but we found that some solutions introduced in Gabriel fit well to our much less powerful target platform. Gabriel (LEE et al,. 1989) was the first system capable of generating executable code at Berkeley in which the synchronous dataflow paradigm was implemented. Its predecessor, BLOSIM (MESSERSCHMITT, 1984) was only a simulator.

The operations (or *actors* by the terminology of the Berkeley team) are called *stars*. A cluster of stars forming an interconnected SFG is called *galaxy*. The final SFG can be hierarchical composed of a number of galaxies, a set of interconnected galaxies is called *universe*. Gabriel has two levels of user interface. The graphical dataflow organization is used where appropriate: when describing the algorithm in dataflow format. The stars have textual definition. This mixed description form helps to avoid the common problem of the graphical description systems which use graphical terms where they are not handy.

One of the most striking features of Gabriel is its *programmable star library* which influenced a lot the database of our Rafael system. A Gabriel star is described by a Lisp structure. The star library entry has a *header* and a *function body*. The header structure stores information about the inputs and outputs of the operation, a short textual description for human readers and the parameters and their default values. An entry in the header points to the *star function* which gets executed whenever the star is invoked. This star function can actually execute the operation assigned with the star in simulation mode or can generate a code for the actual target processor in code generation mode. It is important to note that the code generator star library is written in Lisp so a code generator function can be quite intelligent when it decides on the text to be generated depending on the parameters. size of the inputs, etc. Beside the star function, a Gabriel star can have initialization/termination functions that are called once before the first invocation and after the last invocation of a star. Processors are described in a similar way creating Lisp lists that contain the target system characteristics: number of processors, processor memory, special hardware units connected to processors, communication channel characteristics between the processors and communication code generator routines. The Gabriel system is strictly homogeneous: there can be only one star library in the memory.

The Gabriel system has the following interesting features:

- It handles multiple sample rates which result naturally from its input format, the synchronous dataflow graph.

- It has a second user level, the star library programming level in Lisp which allows the user to create new stars easily and to add intelligent optimization/code generation features to the existing star library.

The main weaknesses:

- It does not address the question of data dependent constructs, if-then-else, case, etc.
- It does not support heterogeneous systems.
- Its scheduler cannot be considered efficient.

Another system that influenced greatly our work is SynDEx (SOREL, 1994). SynDEx is a code generator environment designed, to be interfaced with the synchronous language compilers, SIGNAL (LE GUERNIC et al., 1991), LUSTRE (HALBWACHS et al., 1991), ESTEREL (BOUSSINOT – SIMONE, 1991). It has a graphical and textual user interface that allows users to construct the algorithm block diagram entirely in SynDEx. It is designed, however, rather to receive the algorithm graph from a synchronous language compiler. Actually SynDEx is interfaced in such a way with SIGNAL (BOURNAI, 1994) and work is under way to create a common format for the SIGNAL, LUSTRE, ESTEREL languages so that they can send the result of compilation to SynDEx or other code generators. The algorithm model of SynDEx is the *conditioned signal-flow graph*. It means that each node has a clock it is associated to which results in a *condition input* for each node (*Fig. 1*).

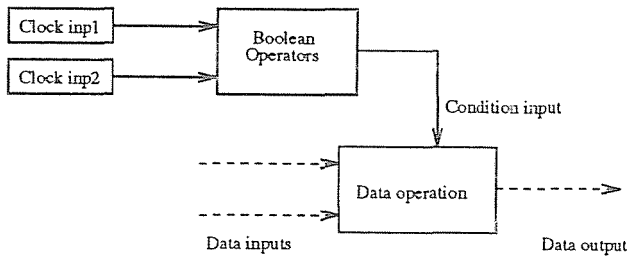

Fig. 1. Conditioned signal-flow graph

A node is fired if all its input variables (including the control variable) have been produced by predecessor nodes and its control variable is *true*. The scheduler considers the condition input dependency as any other dependency: it is equivalent with supposing that each condition is true and each node can be executed. This way the original conditioned signal-flow graph is transformed to a synchronous signal-flow graph and static scheduling can

be used. The original conditioned signal-flow graph is thus partitioned into a condition calculating part (which is unconditioned) and a data processing part (which can be conditioned). Is is the responsibility of the SIGNAL compiler (or the input graph designer) that a proper condition signal be assigned to each node.

The biggest problem about the SynDEx system is caused by the way it handles the conditions. The actual implementation does not use the condition tree (AMAGBEGNON et al., 1994), constructed laboriously by the SIGNAL compiler, the hierarchy of clocks disappears, all the clocks become 'level 1' clocks (inserted just under the root clock). The code generator does not group operations scheduled one after the other with the same conditions into one `if ... endif`. Other drawbacks are that SynDEx does not support heterogeneous architectures and it can generate only C code.

3. Major Design Considerations of the Rafael System

The Rafael structure was designed according to the four main goals introduced at the beginning of this chapter. The support of heterogeneous systems needed a flexible operation library or — even better — programmable code generator module. Considering the code generator programmer's convenience, compiled languages can be quickly eliminated because it would need the recompiling and relinking of the code generator modules each time the database is modified. A system constructed in this way would be much more prone to system crashes as compiled languages allow great liberty in manipulating the system resources. We decided that reprogrammable parts of the code generator be implemented in an *interactive, interpreted language*. As we intended to provide the possibility of important intelligence in these modules (as they determine the quality of the code generated) we wanted to choose a more powerful language. Considering the possible candidates we chose Lisp because of the following advantages:

- It is a very powerful language that allows run-time program creation and it is equipped with efficient database handling capabilities.
- Lisp interpreters are available in relatively small memory requirement versions which fit well to the small computer (PC) we planned the system to run on.
- Excellent quality public domain versions have been written and distributed for several platforms in source code.
- It is a common language in CAD systems.

We must consider, however, the slow execution speed of Lisp whichis an even more serious obstacle on a small PC system. Although in the sense

of ease of programming it would have been more advantageous to realize the system entirely in Lisp, this solution would have resulted in unacceptable run time on the target system.

4. The Structure of the Rafael System

For the reasons mentioned in the previous section reason we choose a hybrid structure depicted in *Fig. 2.*

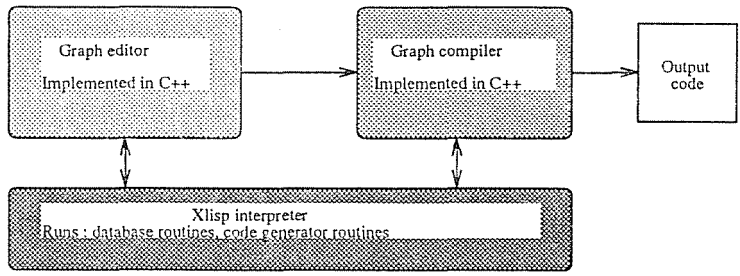

Fig. 2. Structure of the Rafael software

Each part of the software where user modifications are not supposed was implemented in C++. This gives us a relatively powerful language with acceptable execution speed. Programmability is provided at Lisp level where an interface has been defined for the database and code generator programmer. By means of this interface the user can extend the database and the code generator library. The compiler core calls these routines from C++ level and uses their return value appropriately.

This solution needed separate tasks and interprocess communication between the tasks. The minimal 'operating system' that is sufficiently popular and needs small resources was the Microsoft Windows. At that time Linux (a small Unix version for PCs) was not in the state that we could have considered it as an alternative against Windows. By my personal opinion Windows is a poorly designed, inefficient 'operating system', today we would choose some other platform.

Thus, Rafael was implemented under MS-Windows, parts of this software (*Fig. 2*) run as separate Windows tasks and they are connected through the interprocess communication channels of Windows. The popular Xlisp was chosen as Lisp interpreter for Rafael because it is close to Common Lisp and it is available in C source. Xlisp was ported to Windows platform and the necessary interprocess routines were inserted that allows this Lisp interpreter to run as a server task.

The three Rafael software components have the following tasks.

Graph editor The name is a bit exaggerating as the Rafael framework is far from a comfortable working environment. It features a multi-screen text editor for creating/modifying graphs in textual format, initializes the Xlisp server and launches the Rafael compiler on the actually edited graph.

Graph compiler It is the SFG compiler. The program analyses graph description, makes the scheduling and generates the output text. It can run standalone as well, not only from the framework.

Lisp interpreter The operation database and its associated code generator routines are realized in Lisp. The client programs launch the server and send requests to it through interprocess links. Requests are actually Lisp commands which are executed by the server and the result of the Lisp command evaluation is returned to the caller C^{++} program.

As we can see the Rafael software architecture is very similar to that of Gabriel hence the similarity of the names. Rafael is different from Gabriel at the following points:

- Rafael's whole structure is adapted to the small host systems it runs on. Not the whole compiler was implemented in Lisp, only a part of it.
- As we will see, Rafael's whole design including the database, the scheduler it uses is adapted to heterogeneous systems. Gabriel was *multi-target* as it supported multiple start libraries. Rafael is *truly heterogeneous* as multiple target processors can coexist in the same operation library.
- Rafael supports a limited form of run-time decisions as its importance has been underlined many times both in the literature and in the practical engineering work. It will be detailed in section 6.
- Rafael features more advanced and efficient scheduler algorithms.

5. Rafael Nodes and Connections

The Rafael software model defines *nodes* that represent certain operations and *connections* between them. Nodes can be of the following types.

Operations Operations cover functions attached to a certain node. An operation is a parametrizable function. The number of inputs, outputs, the execution time and the operation of the function itself can depend on constant parameters.

Probes Probes cover functions whose task is to acquire input data from the environment of the dataflow system and send output data to the environment of the dataflow system. Probes are treated as simple

operations (with non-zero execution time, if necessary), the only difference is that they are explicitly forced to certain processors by the user. It derives from the fact that in a given hardware system the input and output hardware are assigned to prescribed processors.

Delays Delays are special operators in the sense that they consist of two parts: a delay input (where new data is put into the delay) and delay output (where new data is retrieved from the delay). Rafael always treats delay parts as two distinct operations. It is guaranteed, however, that output of a delay be scheduled always before the input of the same delay.

Each node input/output can have a type. Type is a character string which is checked for matching when node inputs/outputs are connected. Rafael allows dynamic type names resolved in compile-time that match to every static type name and solves the type name ambiguities. In Rafael dynamic type names start with the 'TYPE' string, for example 'TYPE23' is a dynamic type string. An adder that can add any type of data can have 'TYPE23' type of each input/output node. When any of the inputs/outputs is connected to an output/input with static type, the dynamic type is replaced by the static type by the checker. For example if the output of the hypothetical adder above is connected to an input node with 'TIME' type, 'TYPE23' is replaced by 'TIME' for all the adder inputs/uotputs and type checking continues on the inputs. *Fig. 1* illustrates the process.

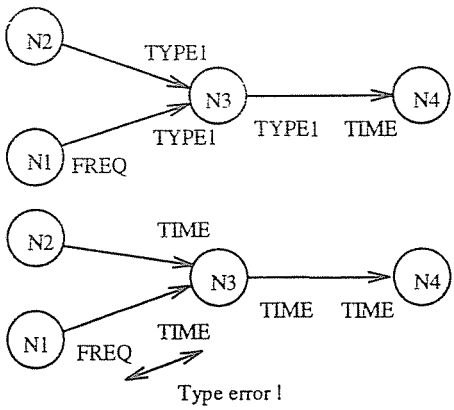

Fig. 3. Propagating type names in Rafael

Depending on the operation library, 'tokens' can have arbitrary size. The actual Rafael operation library supports one-dimensional vector tokens.

6. Rafael Software Model

Rafael accepts a restricted version of synchronous dataflow graphs (LEE – MESSERSCHMITT, 1987) for scheduling. This restriction means that if a node output produces or input consumes more than one token, it can be connected only to an input or output that consumes or produces one token. See *Fig. 4* for example. This simplified scheme allows Rafael to support practically relevant upsampling/downsampling operations without getting to a problematic loop scheduling problem (BHATTACHARYYA – LEE, 1994).

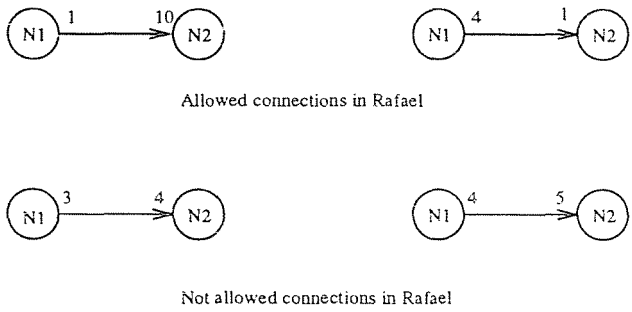

Fig. 4. Rafael's restricted synchronous dataflow graph

Rafael has two software models. The first one is a classical synchronous dataflow model which does not allow run-time decisions. This model has been proved to be too restrictive but this is the most effective one. It allows all kinds of supported operations in the dataflow graph but no conditional structures are permitted, we will call it *static model* in the future. The static scheduler will be invoked for this graph and a single-block schedule will be generated. This model is the restricted version of the second one that allows run-time decisions.

Based on the conditioned dataflow model of synchronous languages a *conditioned block dataflow model* was implemented in Rafael, we will call it *dynamic model.* Inserting `if ... endif` constructs around each operation and considering all conditions *true* it is an evident but not too efficient solution for the run-time decision problem. Instead Rafael forces the SFG designer to *group* parts of the graph to a *block.* A block contains a graph portion for which the following holds true:

1. Inside a block the graph portion is a synchronous dataflow graph without run-time decisions.
2. All the operations in this block depend on the *same condition.*

Outside the blocks only probes and blocks are allowed. This is called *root* level. Operations are embedded into blocks, this is the *block* level.

This simple scheduling scheme used in Rafael solves the scheduling problem in two passes.

1. First it prepares static schedule for each block independently. Variables are propagated through the root level block connections and static scheduler is invoked for the block.
2. Dynamic root-level scheduling. Blocks are considered as operations which run on all the processors at the same time. A list scheduler traverses the block connections and builds the order of the block considering only dependency relations. During the execution a block may or may not be executed depending on its condition input variable (if any).

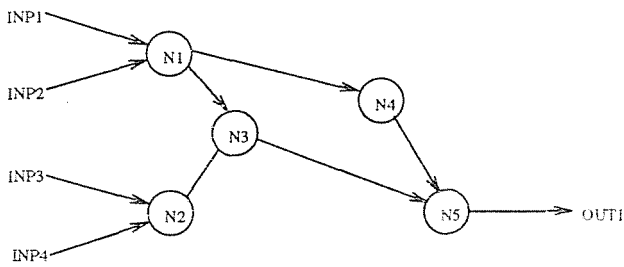

Fig. 5. Example static model graph

Fig. 7 demonstrates this method on the example dynamic model graph in *Fig. 6.*

Advantages of the conditioned block schedule are the following:

- We can provide conditional structures while preserving static scheduling.
- The user of the system is forced to group nodes with the same condition together, the performance loss resulting from the repeated conditional statements is thus avoided.
- The static scheduling algorithm estimates the reality much better than in the SynDEx case. As a block contains only synchronous dataflow, the static scheduling is always exact, not only in the worst case as in SynDEx.
- SIGNAL compiler makes readily the operation grouping itself.

We have to mention the following disadvantages:

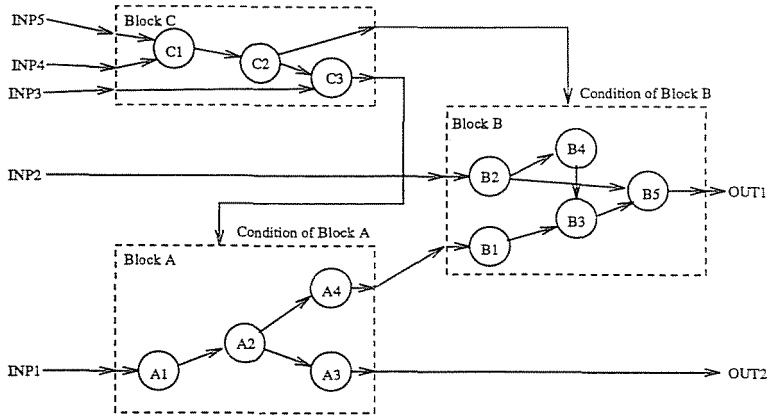

Fig. 6. Example of dynamic model graph

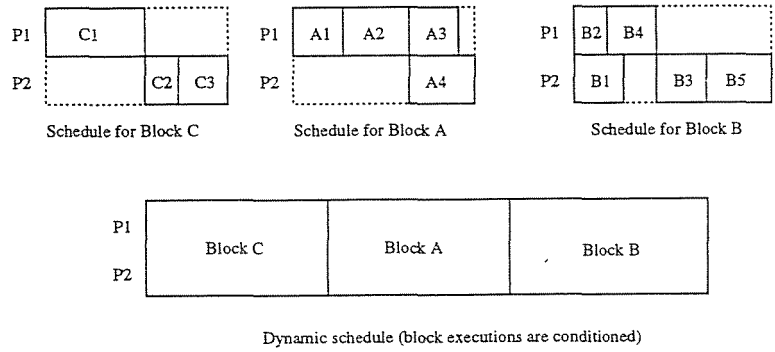

Fig. 7. Example of dynamic model scheduling

- If the blocks contain insufficient operations, static schedules of blocks can be too sparse. In this case even true dynamic scheduling could provide a better solution.
- It is very easy to construct an incorrect graph. Consider the graph in *Fig. 8.* In this example Block B depends on Block A and in the root-level dynamic scheduling it is scheduled after Block A. It cannot be guaranteed, however, that Block A was really executed because it depends on a run-time decision. If the condition of Block A is not true, Block B will get its input from obsolete temporary variables producing a bad result. As Rafael makes no effort to check the cal-

culation of condition variables, these situations cannot be signaled by the compiler.

- Other effect of the fact that Rafael does not analyse the condition calculation is that all the condition variables must be recalculated in each iteration. We can recall that SIGNAL compiler laboriously optimizes the condition tree so that its output program can be the 'laziest' which means that `if ... endif` structures belonging to a clock expression on the lower level of the clock tree will be appropriately nested into `if ... endifs` of upper level clocks. The scheme presented above will flatten the clock tree putting all clock expressions to level 1.

In spite of the disadvantages we consider that the Rafael conditioned block model avoids successfully the dynamic scheduling and in the case of large static blocks and few decisions (which is often true at a DSP algorithm) it is sufficiently efficient.

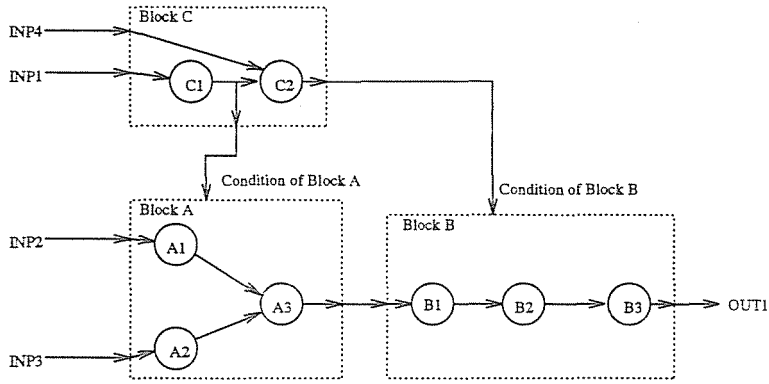

Fig. 8. Example of possibly erroneous graph

7. Rafael Hardware Model

Rafael supposes an arbitrary number of interconnected, heterogeneous processors as target system. The communication hardware connecting these processors can be heterogeneous as well. The static scheduling algorithm prescribes, however, that execution times of operations on all the processors of the target system and communication times on all the channels in the target system should be known in advance. These calculation/communication times can depend on certain parameters, in the case of calculations these

parameters are defined by the operation type, in the case of communication it depends on the amount of data units passed between the processors.

Rafael uses a simplified communication model, critiques say it is over-simplified. Rafael considers the communication structure totally interconnected but allows different communication costs for both directions of each channel. The actual Rafael implementation does not have router algorithm so if the target architecture is not totally interconnected, virtual communication layer must be provided by operation library programmer.

The basic Rafael communication notion is the *channel.* Channels are resources that are shared by processor pairs willing to communicate. A channel is assigned to each processor pair and that channel is occupied for the length of the communication between that processor pair. Other processor pairs having the same channel number have to wait with their request until the channel is free. Channels represent hardware resources used for communication (bus, network, communication links, etc.). The processor pair–channel number assignment is fixed in the hardware database.

Each communication activity can have three properties which are returned by the hardware database functions to the compiler core.

Activity time It is the time during which the communication activity occupies the processor it is scheduled on. If the communication hardware needs constant interaction with the processor (buffered serial line hardware, for example) the activity time is the same as the time required for the communication activity. In the case of DMA it is the DMA initialization time.

Survive time This is the time which is needed to finish the communication after the activity itself finishes. For example a DMA is initialized during the activity time then it accomplishes the task. During the survive time the variable which is sent cannot be reused and no new communication activities can be accomplished on that channel. On the receiving side all the calculations which need the received variable are delayed until the end of the survive time.

Synchronous flag This flag controls the scheduling of communication activities. If this flag is false for a certain communication activity, the scheduler can put the send activity before the receive activity of the same communication pair. No 'crosses' are allowed, however (see *Fig. 9*). If the synchronous flag is true, the send and receive activities are scheduled strictly at the same time.

Fig. 9. Allowed and not allowed communication schemes

8. Graph Description Language

The actual Rafael implementation does not contain a graph editor, the user must construct the input algorithm graph himself or herself. A simple graph description language is used for this purpose which will be described briefly in this section.

According to the two software models in Rafael. there are two variations of the graph description language. In the first variation (synchronous dataflow) only probes, nodes, delays and connections are allowed. Let us see an example graph:

```
PROBE I 1 1 A_TYPE 1 1
PROBE I 2 1 A_TYPE 1 1
PROBE 0 7 1
NODE 4 ADD (4)
NODE 5 ADD (4)
NODE 6 ADD (4)
NODE 8 MUL (4)
NODE 3 CONST ((1 2 3 4))
DELAY 9 4 1
CONNECTION 1_1 4_1
CONNECTION 2_1 4_2
CONNECTION 2_1 5_1
CONNECTION 3_1 5_2
CONNECTION 4_1 6_1
CONNECTION 5_1 6_2
CONNECTION 6_1 8_1
CONNECTION 3_1 9_1
CONNECTION 9_1 8_2
CONNECTION 8_1 7_1
```

PROBE <I/O> <nodenum> <type> <upsample> <downsample>
<I/O> is the input/output probe type, <nodenum> is the number of the node, <type> is its type name. For convenience of the compiler, Rafael stores the relative sample rate of the node in rational form. <upsample> is the nominator, <downsample> is the denominator of the relative sample rate (see section 11).

NODE <nodenum> <operation> <parameters>
<nodenum> is the node number, <operation> is the function attached to the node, <parameters> is the parameter list which depends on the function. In the case of the example ADD operator determines the size of the vectors to be added.

DELAY <nodenum> <delay size> <delay length>
<nodenum> is the number of the node, <delay size> is the size of one token it stores, <delay legth> is the number of delay stages data fed into the delay goes through. Delays explicitly have TYPE inputs/output types.

CONNECTION <onode>_<onum> <inode>_<inum>
Defines a connection between the output numbered <onum> of the node having <onode> node number and an input described by similar parameters.

The conditioned block dataflow model allows block definitions beside the elements above. In this model only probes, block definitions and connection definitions are permitted at root level.

```
BLOCK MADD2  I1->6_1:TYPE1 I2->5_2:  TYPE1
             I3->5_1:TYPE1 01->6_1:  TYPE1
NODE 5 MUL (4)
NODE 6 ADD (4)
CONNECTION 5_1 6_2
ENDBLOCK MADD2


BLOCK MUL2  C:BOOL I1->6_1:TYPE1 I2->5_2:  TYPE1
            I3->5_1:TYPE1 01->6_1:  TYPE1
NODE 5 MUL (4)
NODE 6 MUL (4)
CONNECTION 5_1 6_2
ENDBLOCK MUL2


PROBE I 1 1 A_TYPE 1 1
PROBE I 2 1 A_TYPE 1 1
```

```
PROBE I 3 1 A_TYPE 1 1
PROBE I 10 1 BOOL 1 1
PROBE 0 7 1

NODE 4 MADD2
NODE 5 MUL2
CONNECTION 10_1 5_C
CONNECTION 1_1 4_1
CONNECTION 2_1 4_2
CONNECTION 3_1 4_3
CONNECTION 1_1 5_1
CONNECTION 2_1 5_2
CONNECTION 4_1 5_3
CONNECTION 5_1 7_1
```

The only new element is the BLOCK ... ENDBLOCK definition pair. Blocks group their internal nodes into one virtual operator that can be placed by a NODE definition. A internal node in a block is identified by its block name and node number, two blocks can have internal nodes with the same node number as internal nodes are invisible outside of a block. The block header contains the following elements:

- I <inputnum> - ><inp nodenum>_<inp inputnum>:<typename> Connects <inputnum> input of the virtual operator represented by the block to <inp inputnum> input of <inp nodenum> internal node. Type of the block's input is set to <typename>. Data fed into that input of the block will be propagated to the internal node's input.
- O <onum> - ><onodenum>_<out outputnum>:<typename> Connects <onum> output of the virtual operator represented by the block to <out outputnum> output of <onodenum> internal node. Type of the block's output is set to <typename>. Data produced by that output of the internal node will be propagated through the output of the virtual node.
- C :<typename> Indicates that the block has condition input and the type of the condition input is <typename>. Condition input can be referenced as 'C' in the CONNECTION definition.

9. The Database

Rafael provides a programmable operation and hardware database stored in Lisp. The database is accessed by the compiler core through Lisp functions. The interface of these Lisp functions is documented so that the database programmer can interface to the compiler core.

The database consists of two parts: operation database and hardware database. Operation database stores the actual function set for all the supported hardware devices while hardware database provides Lisp functions that can calculate every characteristic of the target hardware system which is necessary for scheduling and code generation.

The database is handled and maintained through the XLisp interpreter and stored in Lisp lists. Because XLisp runs under Windows, all its memory is virtualized so we can store the whole database in the memory of XLisp. The simplifies greatly the implementation of the database management because we simply use the built-in list manipulating functions of LISP.

The Operation Database

The operation database has two parts: operator headers and compilation strategy functions. The operator headers are stored in lists which are bound to the operator name. This list stores the following information:

- The name of the compilation strategy routine.
- The description of the input(s) (type, size).
- The description of the output(s) (type, size, storage class, sample rate factor).
- The execution time in system clock beats.
- Parameters. The parameters and their meaning are defined by the creator of the operator library. For example the parameters for the FIR operator can be the length of the filter and the filter coefficients. The actual values of the parameters are supplied when the user places an operator, it is passed in the SFG script.
- Constructor and destructor routines. The compiler creates a constructor function for each operator which requests it. The constructors are invoked before the operator is executed first time. Similarly, before the SFG execution terminates, destructor functions are called for the operators which need it.

  The data structure above is described in a list like the following:

```
(  ( strategy list)
   ( inputs )
   ( outputs )
   ( time function )
   ( parameters ) )
   ( constructor strategy list )
   ( destructor strategy list )
)
```

The strategy list contains the names of the compilation strategy functions for each hardware device. It has the following format:

```
(  (device1 function1) (device2 function2)
   ... (deviceN functionN) )
```

The compilation strategy function is called each time during the code generation pass when the schedule contains a reference to that function and its program text must be generated. This LISP function gets the label lists of the input and output branch descriptors (effectively labels of data areas where the compiler allocated space for the temporary variables), the parameter list (which contains data like coefficient vector of a filter, etc.) and returns the program text to the compiler which writes it into the output file. The strategy function can decide on the subroutine chosen or the form of the generated program text depending on the input and output connections and the actual parameters. The subroutine bodies can be stored in an ordinary object library, in this case Rafael will place only references into the code which can be resolved by the linker which belongs to the DSP's development system. This subroutine library can be created and maintained by the assembler and library manager tools of the DSP development software package. Another design style is to inline all the operation bodies which result in slightly faster code but larger code size.

The excellent symbol handling capability of the LISP which makes this language so appropriate for the artifical intelligence applications can be exploited in this system and we can build significant intelligence into the strategy functions.

The input list stores the description of the operator's input. Its format is the following:

```
(  ( type1 size1 ) (type2 size2) ...
   (typeN sizeN) )
```

where type is the freely chosen signal type (for example time for time domain signals) and size is the size of the input vector accepted by this node. This size can also be a symbol from the parameter list (for example the size of an FFT input can be `N` where `N` is a parameter supplied by the SFG designer) or even a lambda function of the parameters. The type name can be either static or dynamic. Dynamic type names have the form of 'TYPEn' where `n` is an integer number. Dynamic type names are resolved when they are connected to a statical one.

The output list is similar, but beside type and size it also contains the storage class specifier and the upsample and downsample factors. Its format is the following:

```
(  ( type1 size1 st1 us1 ds1)
   (type2 size2 st2 us2 ds2) ...
   (typeN sizeN stN usN dsN) )
```

The storage class specifier shows whether the compiler has to allocate space for the output variable or the space is reserved by the operator. The us and ds values describe the change in sampling frequency caused by the operator. The us denotes the multiplication, ds is the division of the sampling frequency. For example the pair 2 1 means interpolation by 2.

The time function list stores Lisp functions which get the bound parameter list and return the execution time of the operator on a given hardware. The list has the following format:

```
(  ( device1 lambda1 ) ( device2 lambda2 ) ...
   ( deviceN lambdaN ) )
```

where `lambda1 ... lambdaN` are lambda expressions (no-header Lisp functions) which compute the execution time for the given device.

The parameter list contains operator-dependent data. For example in the case of an IIR filter it contains the size of the nominator and denominator coefficient vectors and the vectors themselves. In the operator header the list is stored in unbound form (without parameter values), the editor evaluates this list when placing an operator. The IIR parameter list would look like the following in unbound form:

```
(N COEF1 M COEF2) )
```

and in bound form (after the operator has been placed)

```
(3 (0.34 - 0.2 2.12) 4 (0.23 0.77 0.192 2.94) )
```

This bound form is stored in the SFG description file and is passed to the execution time computing and strategy functions when necessary.

The constructor and destructor strategy lists have the same format as the strategy function. An operator may have constructor and/or destructor functions — pieces of code which are executed before the operator's first run and after the operator's last run. If the operator does not need such functions, NIL is stored instead of the name.

The following small code piece shows the implementation of the ADD database entry for the TMS320C30 and DSP96002.

```
(setq add ' ((
; c30add is C30 strategy function
                (c30 c30add)
; dsp96kadd is 96K strategy function
        (dsp96k dsp96kadd)
        )
; Has two inputs, each of size n
; (n is the operation parameter)
                ((type1 n) (type1 n) )
; Has one output, size n, automatic storage,
; interpolating factor:  1
                ( (type1 n a 1 1) )
; Time functions for C30 ...
                ( (c30 (+ (* 2 n) 10) )
; and 96K
                (dsp96k (+ (* 2 n) 5) )
                )
; Has only one parameter (n)
                ( n )
; No constructor for C30 and 96K
                ( (c30 nil) (dsp96k nil) )
; No destructor for C30 and 96K
                ( (c30 nil) (dsp96k nil) )
        )
)
```

Target Hardware Database

The target hardware database provides the following information to the compiler core:

- Processor numbers and processor types in the target system.
- Activity, survive times and synchronization flag for any communication activity.
- Communication cost estimation for any communication path in the target system (for the scheduler).
- Channel-processor pair assignment for any processor pair.

A set of Lisp functions must be written for each target system. It is a relatively inconvenient solution but allows greater flexibility.

10. Rafael Memory Management

Rafael allocates memory for temporary variables in compile time. When the generated program runs on the target system, every variable is already assigned a memory address. Rafael implements a simple 'first fit' dynamic memory allocation scheme when compiling the graph.

When a node is scheduled, Rafael allocates its output variables (the input variables must have already been allocated). The scheduler keeps track of the actual state of memory map by the means of chunk lists which describe, actually what size of blocks are occupied at what address in the memory of the target processor. When allocating a variable the memory manager simply walks this chain and finds the memory block with the lowest address which is big enough to accommodate the variable to be allocated.

When an output variable is created, its 'scope' is established. A variable goes out of scope if all the operations that consume this variable hase already been executed. In this case the memory chunk assigned to the variable is freed and the place the variable occupied can be reused. As the scheduler cannot know when allocating the variable, on which processor(s) that variable will be consumed. every instance (variable sent to other processors) of that variable stays 'alive' on every processor until all operations that consume that variable terminate.

A variable can be local or global. Local variables are used internally by blocks. A variable is local if it is created in a block not at root level and it is consumed only by the operations of that block (so it is not connected to a block output). Every other variable is global. Blocks have their own address maps that start at relative address 0. At the end of the scheduling when we know, how much memory is required fir the global variables, local variable addresses are relocated so that these variables be allocated starting at the end of the memory allocated for global variables. Local variables of blocks thus overlay each other (*Fig. 10*).

11. Compiler Passes

Rafael compiler works in 5 passes.

Reading Graph Description File

The compiler reads in the SFG file and parses it syntactically. Then it analyses the connection definitions and signals connection errors (connecting to

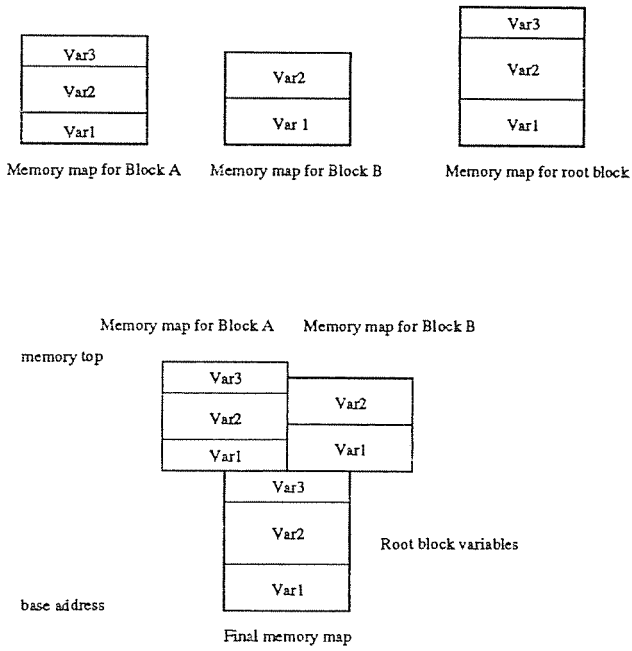

Fig. 10. Block memory overlaying in Rafael (supposing 1 processor)

nonexisting node, nonexisting input, etc.). During this phase the compiler rebuilds the tree in the memory of the computer, ready for analysis.

Type Checking

The compiler resolves the dynamic type names and checks if there are type errors (see section 4 for further explanation). The type checker is a recursive routine that propagates the static type names from node to node substituting dynamic type names with static ones and signaling errors if type name violation is found. The type checking starts at descendants of probes as they are the only nodes that surely do not have dynamic types.

IPF Checking

IPF stands for interpolation factor and is used to support Rafael's multirate features (section 6). IPF is the rate of the node's execution in the multirate

model. IPF is represented by two distinct numbers, the nominator and the denominator so IPF:1.4 means 1/4 execution rate.

Rafael uses a recursive subroutine similar to the typechecker to propagate IPFs along the graph and looks for the minimal IPF factor. Propagating IPF means that the IPF at the input of the operation is multiplied by the sample frequency multiplication factor stored in the database at the output description yielding output IPF then it is passed to all the nodes connected to the outputs. The actual implementation of Rafael prescribes that the output sample on all the outputs should be the same. During the IPF propagation the minimal IPF in the graph is recorded. As IPF is calculated by division or multiplication by integer factor, all IPFs in the graph must be integer multiple of the minimal IPF. So the factor

$$c_{loop} = \frac{IPF_{node}}{IPF_{min}}$$

is the loop count that determines, how many times an operation with IPF IPF_{node} must be repeated if the minimal IPF is IPF_{min}. Note that operation changing IPF are always executed on the higher input sample rate and output sample rates (*Fig. 11*).

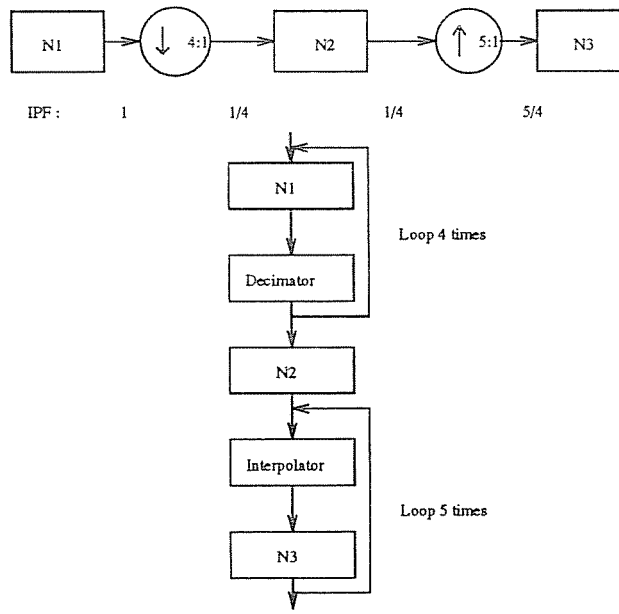

Fig. 11. IPFs in an example graph and looped schedule

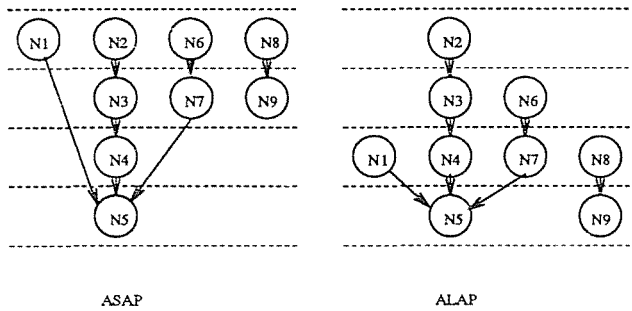

Fig. 12. ASAP and ALAP schedules

Scheduling

The formally correct, typechecked graph with IPF values for all the nodes calculated is then passed to the scheduler algorithm. The actual version of Rafael contains only the RHLS scheduler but work is under way to implement the much more efficient Springplay scheduler (PALLER – WOLINSKI, 1995) in the software.

RHLS is an ALAP-based list scheduler which was made suitable for heterogeneous environment. In the first step we create ASAP and ALAP schedules in order to get the ALAP levels. We present briefly ASAP and ALAP schedules below.

ASAP algorithm was presented first in HU's classical publication (HU, 1961). ASAP scheduler starts operations as soon as all the predecessor nodes terminate the computation that is

$$E(n_i) = \max(E(\mathrm{pred}(n_i))) + t_{n_i}^{e,asap}, \tag{1}$$

where $t_{n_i}^{e,asap}$ is the execution time of node i and $E(n_i)$ is the earliest time when n_i can be executed. Node with no predecessors have $E = 0$. This simple version is only for homogeneous architectures. The original version supposes unlimited resources and schedules nodes just at their E.

ALAP schedule is based on very similar principles. Nodes are scheduled as late as possible without increasing the length of the schedule.

$$L(n_i) = \min(L(\mathrm{succ}(n_i))) - t_{n_i}^{e,asap}. \tag{2}$$

$L(n_i)$ is the latest time when n_i can be executed in the case of minimal length schedule. L values of nodes with no successors are initialized to the maximal E value over the entire graph. *Fig. 12* depicts the ASAP and ALAP schedules of an example graph.

RHLS assume the we can always schedule the nodes on the fastest processor possible so minimum execution time is supposed when building the ASAP-ALAP schedules.

$$t_{n_i}^{e,asap} = \min(\bar{t}_n^e)\,,$$

where t_n^e is the execution time vector that is composed of execution time of node n on each processor. Then we define *urgency* of the operation n like the following:

$$u_{n_i} = L_{n_i} - t_v\,, \tag{3}$$

where t_v is the *virtual time* and it will be detailed later.

The base of the scheduling heuristic is to assign the nodes on the critical path to the fastest processor available. The more urgent it is to execute a node (as its delaying would set back the execution of the whole graph) the faster processor it deserves. The most urgent nodes are those which have the lowest ALAP time.

We pick hence the node to be scheduled based on the u_{n_i} urgency value defined above (lowest urgency value means more urgent node) and we need the best processor to execute it. The best processor selection is very simple: we try the node on each processor considering the communication costs and we pick the one on which the node achieves the earliest completion time. Before trying a node on a processor, necessary communication activities are scheduled tentatively so that we know how much time must be calculated for fetching the input variables produced on other processors.

The heuristic algorithm works like the following:

```
Create the ready node list from nodes that have no predecessors;
while the ready list is not empty do
     for all nodes do
          if u(i) < minimum so far
            Candidate = node i;
     end for
     Try the candidate on each processor considering communication
          cost;
     Choose the processor on which the task achieves the earliest
          ending time;
     Schedule candidate node and the necessary communication
          activities on candidate processor;
     Update u(i)s and tv;
     Add nodes that become ready to the ready list;
end while
```

As the real t_{n_i} node starting times will generally not be equal to the ideal ASAP or ALAP starting times the scheduler maintains *real processor*

times and t_v *virtual time*. The virtual time is used to track the time in the ALAP schedule graph while the real time is the scheduling time on the processors. The t_v variable shows where we are in the ALAP schedule graph, it is set to the lowest ALAP time among the ready nodes. The last step is the updating of urgency and virtual time variables.

The version implemented in Rafael differs from the algorithm presented above considers node repetition resulted by multiple sample rate loops (see IPF checking section). The schedulers consider effective node execution time as $c_{loop} \cdot t_n^{e,asap}$ and try to group nodes with the same IPF together.

Code Generation

The scheduling done, Rafael generates the output text for each processor. The code generator walks the activity list on each processor then asks the Lisp code generator database functions to produce output text for them which is then sent to the output file. Separate output files are generated for each processor. The model of output text will be discussed in detail in the next section.

Code Generation Model

Rafael has a parametrizable code generation that allows each section of the text generated to be redefined. The code generator invokes Lisp functions that receive the parameters of the text section and the device for which the code will be generated then it is the responsibility of these Lisp functions to produce the appropriate text. These code pieces are called *code generator service functions* and they complement the operation strategy routines. Every text section that Rafael writes to the output text file can be redefined by modifying either the operation strategy functions (in the case of operation texts) or the code generation service functions (headers, communication routine codes, etc.).

Rafael generates three text sections for each processor (that may be empty as well). For programmable processor-like devices that Rafael was designed for, the database programmer may wish to realize these three sections as subroutines. These sections are the following:

1. Constructor section. Called only once from the user program before the first iteration of the dataflow computation.

2. Operation section. Called once for each iteration. Calling the operation section entry label will actually execute the program generated from the SFG.
3. Destructor section. Called once after the last iteration of the operation section.

Each section has a start and end header that probably contain section head label in the start header and 'return' instruction in the end header. The sections contain the text generated by the operation constructor, strategy and destructor functions.

If the compiled SFG was written in block conditioned model, each section has a separated part for each block. In the constructor and destructor sections it is rather a formality as Rafael guarantees no specific order among the operators when it generates constructor and destructor sections. In the operation section each block has a start and end header. The current operation library realizes blocks as subroutines so the start header defines a block entry point label and the end header contains a 'return' statement. The block subroutine contains the operation body texts in the schedule order. Having block subroutines generated, Rafael emits the text for the root block that contains probe calls and block invocations. Block invocations in the current operation library result in subroutine 'calls' to block subroutines.

12. Conclusions

Rafael cannot compete in complexity with the most advanced systems partly because of the limited capabilities of the host computer we chose, partly because of the significantly less human resources we could devote to the project. The final product, the compiler itself has been implemented but many support programs that would make its usage convenient have not even been planned. For this reason the actual Rafael system is not so 'user-friendly'. As all the resources were concentrated on the compiler development, important parts of the system have not achieved the necessary level yet. The most important among them is the operation database that contains only about a dozen operations only for the TMS320C30 and DSP96002 DSPs. A brave user of Rafael must face the immediate task of filling up the database which requires Lisp programming. Lisp is considered a difficult language among the users although the simple functions needed by the compiler core should be easy to implement for a bit more experienced programmer.

Two distinct influences can be discovered in the Rafael design. The first one is Lee's synchronous dataflow approach and the Gabriel system

which gave us the first notions, how Rafael should look like. We quickly faced, however, the need of run-time decisions and the difficulties it causes in a system based on synchronous dataflow. The second influence that we embedded into Rafael was the way the synchronous language compilers work and SynDEx transformes their output to distributed code. Critique of the SynDEx approach was given and a model that was easy to implement to an existing synchronous dataflow system was developed and realized. Limits of this model were pointed out but we consider that in many practical cases, notably in the DSP case they are acceptable. Further researches are conducted to find a better way for handling dynamic structures in a dataflow system.

So Rafael project achieved its aims at the following points:

- A flexible multi-target SDF compiler has been realized on PC platform.
- Effective scheduling algorithms have been developed for the heterogeneous case.

Rafael still has a long way to go at the following fields:

- More user-friendly environment (graph editor, database editor tools, etc.).
- Complete database for various DSP processors.
- Better communication model.

References

AMAGBEGNON, T. – BESNARD, L. – LE GUERNIC, P.: Arborescent Canonical Form of Boolean Expressions, *INRIA Research Report.* No. 2290. June, 1994.

BHATTACHARYYA, S. S. – LEE, E. A.: Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms, *IEEE Transactions on Signal Processing,* Vol. 42, No. 5, pp. 1190–1201, May 1994.

BURNAI, P. – LAVERENNE, C. – LE GUERNIC, P. – MAFFEÏS, O. – SOREL, Y. (1994): Interface SIGNAL-SynDEx, INRIA RESEARCH REPORT,No. 2206.

BOUSSINOT, F. – SIMONE, R. (1991): The ESTEREL Language, *Priceedings of IEEE,* Vol. 79, No. 9. pp. 1293–1303.

BUCK, J. T. – HA, S. – LEE, E. A. – MESSERSCHMITT, D. (1991): Multirate Signal Processing in Ptolemy, *Proc. IEEE ICASSP-91* Toronto, Canada, April 1991.

BUCK, J. T. (1993): Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token FDlow Model, Ph.D. dissertation, University of California at Berkeley.

BUCK, J. T. – HA, S. – LEE, E. A. – MESSERSCHMITT, D. G. (1994). A Framework for Simulating and Prototyping Heterogeneous Systems, *International Journal of Computer Simulation,* special issue on Simulation Software Development, January, 1994.

GENIN, D. – HILFINGER, P. – RABAEY, J. – SCHEERS, C. – DE MAN, H. (1990). DSP Specification Using the Silage Language, *ICASSP-90,* pp. 1057–1060, Albuquwrque, April, 1990.

HALBWACHS, N. – CASPI, P. – RAYMOND, P. – PILAUD, D. (1991): The Synchronous Data Flow Programming Language LUSTRE, *Proceedings of IEEE*, Vol. 79, No. 9. pp. 1305–1319, September 1991.

LANNEAR, D. (1993): Design Models and Data-Path Mapping for Signal Processing Architectures, Ph.D. dissertation, Katholike Universiteit Leuven, March 1993. LEARY, K. W. – WADDINGTON, W. (1990): DSP/C: A Standard High Level Language for DSP and Numeric processing, *IEEE ICASSP-90*, pp. 1065–1068, Albuquerqu, New Mexico, April 1990.

LEE, E. A. – MESSERSCHMITT, D. G. (1987): Szazic scheduling of Synchronous Data Flow Programs for Digital Signal Processing, *IEEE Transactions on Computers*, pp. 25–35, Vol. C-36, No. 1, January 1987.

LEE, E. A. – HO, W-H. – GOEI, E. E. – BIER, J. C. – BHATTACHARRYYA, S. (1989): Gabriel: A Design Environment for DSP, *IEEE Trans. on Acoustics, Speech and Signal Processing*, Vol. 37, No. 11, November 1989.

LE GUERNIC, P. – GAUTIER, T. – LE BORGNE, M. – LE MAIRE, C. (1991): Programming Real-Time Applications with SIGNAL, *Proceedings of IEEE*, Vol. 79, No. 9, pp. 1321–1336, September 1991.

HU, T. C. (1961): Parallel Sequencing and Assembly Line problems, *Oper. Res.*, Vol. 9, pp. 841–848, November 1961.

DE MAN, H. – RABAEY, J. – SIX, P. – CLAESEN, L. (1986): Cathedral-II, A Silicon Compiler for Digital Signal Processing, *IEEE Design @ test*, pp. 13–24, December 1986.

MESSERSCHMITT, D. G. (1984): A tool for structured functional simulation, *IEEE J. Selected Areas of Communication*, Vol. SAC- 2, Jan. 1984.

PALLER, G. – WOLINSKI, C. (1995): A New Class of Compile-Time Scheduling Algorithm for Heterogeneous Target Architectures, *IFAC/IFIP Workshopon Real Time Programming*, Fort Lauderdale, November 1995.

SOREL, Y. (1994): Massively Parallel Computing Systems with Real Time Constraints – The Algorithm Architecture Adequation Methodology, *Proc. Massively Parallel Computing Systems*, Ischia, May 1994.