McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Dataflow language embedded in CML

Marija Čubrić

ACAPS Technical Memo 83
February 28, 1994

Research supported in part by NSERC and FCAR.

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

In this paper, we show how one can embed Kahn-McQueen style dataflow in CML. This allows one to combine the dataflow paradigm, in particular the ability to have feedback loops, with the higher-order features that CML provides. The context for this research was the development of a language for digital signal processing applications. This application demands that we take finite buffer sizes seriously. We accordingly develop techniques for guaranteeing that only finite buffers are needed. From the point of view of ML programmers the main interest in the present work is to show how one can build a new concurrent abstraction on top of the CML primitives. In particular one needs a nonobvious notion of dataflow process in order to make feedback possible while still preserving the possibility of abstraction of dataflow processes.

Contents

1 Introduction

This paper is about the development of a dataflow language intended for DSP applications. The imagery of stream processing fits very well with the programming needs of DSP. Indeed there is a major effort led by E. Lee at Berkeley [3] to develop a program development environment, called Ptolemy, for DSP practitioners. The underlying programming model of the Ptolemy system is the dataflow model. Our research into bounded-buffer dataflow has been driven by the exigencies of DSP programming. However, implementing dataflow on top of CML allows one to exploit the advantages offered by the ML programming paradigm.

Most of the recent theoretical work in *dataflow*, has been concentrated on topics such as full abstraction (Jonsson, Kok, Rabinovich and Traktenbrot, Russell), generalized *Kahn's principle* for indeterminate networks (Abramsky, Broy, Keller and Panangaden, Misra, Park, Stark, Rabinovich and Trakhtenbrot) or studies of expressiveness (Panangaden and Shanbhogue, Panangaden and Stark). The main theoretical concern in our present work is guaranteeing that a given dataflow network can execute without requiring infinite buffers and calculating the minimum buffer sizes needed [11].

Early work in the study of operational semantics was done by Jack Dennis and his group at MIT [2]. This effort focused on "firing rules" and ways of making sure that dataflow networks were *well-behaved* i.e. that they can execute an infinite computation using only finite buffer space. They propose syntactic restrictions, known as *well-construction* rules which make possible to build well-behaved networks in a systematic way. One of the restrictions that they made is that each process in a network is *homogeneous* i.e it can consume and produce at most one token per channel at each firing. This restriction does not make it possible to model systems where processes communicate with different speed [6]. We restrict the set of dataflow primitives, in such a way to make it easier to determine (when possible) whether network can execute with finite buffer space. For a restricted class of *regular* processes, (those ones which consume and produce a fixed, finite number of tokens at each firing), we can predict buffer sizes needed at compile time.

What makes our DFG (Data Flow Graphs) different from the other dataflow languages (like: Lucid, Sisal, Lustre, Signal) is that it uses another *meta language* (ML) at the process level. This means, that every node in the network graph can execute (componentwise) some ML code. Another difference is that the above languages permit only homogeneous processes i.e. it is not possible to produce a stream which has different frequency of data arrival then the corresponding input streams. Thus we have the power of so-called *multi-rate* dataflow networks.

DFG is implemented on top of CML [9, 8], which is a concurrent programming language built as an extension of Standard ML [1, 7]. Amongst other, CML provides primitives for: thread creation, creation of typed buffered channels and asynchronous communication. In DFG the activity of each node in the network is either a predefined basic process or (user specified) ML code. The behavior of the entire system is described by the CML code generated by the `Compile` function, which translates DFG process specification into CML process. After compilation the resulting CML code can be executed using the `RunCML.doit` function.

The rest of the paper is organized as follows: section 2 defines the language DFG ; section 3 provides more details about CML implementation of DFG , especially how *feedback* is implemented; section 4 describes possible applications of DFG . The appendix contains signatures used for DFG implementation.

2 Language Definition

Following the intuition behind the dataflow model (originally due to Kahn and McQueen [5]) we describe a system in DFG as a network (graph) consisting of processes (nodes) and channels (arcs). Each process can execute some sequential ML program, which can be either a simple arithmetic operation or a complex code. Thus, there is a different granularity at the process level. Channels are unidirectional, unbounded FIFO buffers. Communication is asynchronous via blocking read and asynchronous write. Processes are composed by identifying channels with the same name. There are two different types of entities in the language: tokens and processes.

2.1 Tokens

A token is a unit of communication. A token can carry any of the following values: boolean, real or integer [1] Tokens can be used to build: *blocks* (finite list of tokens), *streams* (possibly infinite sequences of tokens [2]) and *functions*, which have one of the following types:

$$\texttt{pwFun} = \texttt{token list} \rightarrow \texttt{token} \quad \text{or} \quad \texttt{bwFun} = \texttt{block list} \rightarrow \texttt{block}$$

The idea is that `pwFun` functions serve as a program content for homogeneous processes, while those of the type `bwFun` can be used for defining regular processes.

2.2 Processes

Each process can be specified either as basic process or regular process or composition of finitely many other processes. Processes are specified as values of the type `procSpec` (see Appendix).

We will use the "arrow" notation to denote the *type* of the process. We say that some process has the type $m \rightarrow n$ if m and n are respectively the number of its input and output channels.

Regular processes can be specified in one of the following ways:

[1] Conceptually, it is possible for tokens to carry higher-order values, for example functions.

[2] For `stream` forming operations see the Appendix

$\mathtt{PW}\{\mathtt{input} = \mathtt{m}, \mathtt{output} = [\mathtt{f}_1, \mathtt{f}_2, ..., \mathtt{f}_\mathtt{n}]\}$ has the type $m \to n$. It consumes one token from each of its input channels, and applies the "pointwise" function f_i (for every output channel i) on the list of consumed tokens.

$\mathtt{BW}\{\mathtt{input} = [\mathtt{l}_1, \mathtt{l}_2, ..., \mathtt{l}_\mathtt{m}], \mathtt{output} = [(\mathtt{r}_1, \mathtt{f}_1), (\mathtt{r}_2, \mathtt{f}_2), ..., (\mathtt{r}_\mathtt{n}, \mathtt{f}_\mathtt{n})]\}$ is a generalization of the process $\mathtt{PW}\{\_,\_\}$. It consumes a block of the size r_i from each input channel i, and applies "blockwise" function f_j, for every output channel j, on the list of consumed blocks.

We expose the type of regular processes in their definitions, to be able to use it as a parameter in a process specification (see example 2).

We choose the following set of basic processes:

`SW` is a non-regular process which has the type $2 \to 2$. First input channel is a data channel, and the second one is control (boolean) channel. Its output channels are labeled with T and F. This process consumes one token from each of its input channels, and depending on the boolean value received from its control channel it transmits the data token to corresponding output channel.

`SEL` is dual of `SW` process. It has three input channels, first one labeled with T, second with F and third one serving as a control channel. It consumes one token from its control channel, and depending on the boolean value received it transfers a token from corresponding input channel to the output.

`COPY(n)` has one input channel and n output channels. It copies the content of its input channel to all of its output channels.

`GEN(s)` is a generator process of the type $0 \to 1$. It generates the stream s on its output channel.

`SINK` has the type $1 \to 0$. It consumes tokens from its input channel and after it does nothing.

The above set of basic nodes is not intended to be a minimal such set. For example, `SINK` process can be easily implemented using $\mathtt{PW}\{\_,\_\}$ specification. The choice of the above processes is merely due to our intention to separate regular from dynamic behaviors (such as `SW` and `SEL`), which will make it easier to control buffer usage.

Finally, a process can be specified as an interconnection of other processes i.e as a directed marked graph, where each node of the graph corresponds to a process and each arc to a channel in the network.

We use the following auxiliary types to define the interconnection process specification:

`type graph` = `(procSpec*int list*int list) list`,   `type marking` = `(int*block) list`

So, we represent graph as a list of triples, where the first component denote process name, second - list of its input buffers and third - list of its output buffers. Marking is specified as a list of buffer-block pairs, to mean that the buffer is initially marked with the specified block of tokens.

`IC{graph = g, marking = m}` is a composition of processes defined as nodes of the graph `g`, with the initial marking `m`. The type of this process is $i(g) \rightarrow o(g)$, where $i(g)$ and $o(g)$ are respectively number of input and output arcs of the graph g.

Example 1 The following is the DFG specification of a network that computes Fibbonacci's sequence of numbers.

```
local
      val c = attach([Bool(true)], tokenStream (Bool(false)))
      val add = PW{input=2,output=[fn[Int(x),Int(y)]=>Int(x+y) | ...]}
      val rest = IC{graph = [(SW, [1,2],[4,3]),
                             (GEN(c), [], [2]),
                             (SINK, [4], [])],
                    marking=[]}
in
      val fib = IC{graph = [(add,[1,2],[3]),
                            (COPY(3),[3],[2,4,5]),
                            (rest,[4],[1])],
                   marking = [(3,[Int(1),Int(1)])]}
end
```

3 Implementation details

DFG is implemented in CML [9]. It uses the following CML features: thread creation, creation of typed channels and asynchronous communication (`buffer.sml` library). Process specification syntax is implemented through `procSpec` datatype (see Appendix).

The operational semantics of the DFG is given through the function `Compile` which translate DFG process specification into the following CML type:

`process` = `buffer list` $\rightarrow$ `buffer list` $\rightarrow$ `unit`, where buffer is a buffered token channel (CML.buffer_chan). This peculiar type definition allows us to implement loops in the network. This is possible due to the fact that **both** input and output arcs of each node are **exposed** for communication. Another benefit of this type definition is that we can specify *parameterized* processes (see example 2).

Here, we explain how the compilation is done for "blockwise" process specification and for "interconnection" specification. The rest of the Compile function is done in a similar fashion.

- A BW{_,_} process specification is compiled into a process (actually to a process type value) which first consumes the specified number of tokens from each of its input buffers (using the function blockAccept, then applies the specified bwfun function on that input, and finally sends the result of that calculation to its output buffers (using the function blockSend).
- An interconnection process specification is compiled into a value of process type which is a functional value whose local variables are exactly internal buffers of the network. For example, the result of the compilation of fib process specification "ressembles" the following CML code:

```
fn []  => fn [y] =>
      let
         val [z1,z2,z3,z4] = map buffer() [1,2,3,4]
      in
         bufferSend(z3,Int(1));
         bufferSend(z3,Int(1));
         CML.spawn(fn()=> (Compile(add) [z1] [z2]));
         CML.spawn(fn()=> (Compile(COPY(3)) [z3] [z2,z4,y]));
         CML.spawn(fn()=> (Compile(rest) [z4] [z1]));
         ()
      end
```

Beside the Compile function, we have implemented the following:

Input: Exposes an output buffer of a generator process for possible communication.

Output: Allows observing the output of a given process specification on a particular output channel of the network.

Trace: Allows observing the execution of the network: every node creation, buffer creation, [3] send and receive events are recorded.

[3] For this purpose we slightly modified CML library file buffer.sml.

isRegular: Determines whether a given network consists only of regular actors.

weightChecking: Tests whether given regular network is well-behaved. [4]

After compilation, and necessary type adjustments (using functions `Output` or `Trace`) resulting CML code can be executed using the `RunCML.doit` function.

4 Applications

Language DFG is intended to be used as a process specification language for *digital signal processing* applications. DSP is a natural application area for the dataflow model, since this model corresponds exactly to the intuition of the system designers: streams of input values are processed by network elements producing new streams of output values. Also, many, but not all, DSP processes have regular behavior, with very little data-dependent decision making. Hence, a large number of digital signal processing applications, fits into the class of regular networks [3, 6]. For this class of networks, well-behavior is decidable [4], thus it is possible to predict buffer sizes needed at compile time. The algorithm for checking well-behavior for DFG networks has currently been implemented.

Example 2 One of the common DSP elements is Finite Impulse Response filter [3]. This filter has one input line and one output line. It takes "sliding windows" of its input stream and applies a certain function on each of these windows. The function being applied and the size of the window are arguments of this module. Here is an example how FIR can be specified in DFG using parameterized process specifications.

```
local
    val f = fn(n) => BW{input=[1,(n-1)], output=[(n,fn[x,y] => y@x | _ ...)]}

    val g = fn(n) => BW{input=[n], output=[((n-1),fn [h::t] => t | _ => ...)]}

    val c = fn(n) =>
        attach (tokenList(Bool(true),n-1), tokenStream (Bool(false)))
in
    val fir = fn(n,h) =>
        IC{marking = [],
           graph = [(GEN(c(n)), [],[1]),
                    (COPY(2),[1],[2,3]),
                    (SW,[0,2],[5,4]),
```

[4]This part is currently being implemented.

```
                     (SEL,[5,6,3],[7]),
                     (f n,[4,7],[8]),
                     (COPY(2),[8],[9,10]),
                     (g n,[9],[6]),
                     (BW{input = [n], output = [(1,h)]},[10],[11])]}
end
```

For example `fir(10,(fn(l) => (List.fold add l Int(0)))` will produce sums of sliding windows of size 10.

DFG can be used as a concrete syntax for dataflow networks with finite buffers. Similarly as in [10] we can define CCS-style DFG calculus, as a formal system consisting of basic processes, regular processes (all of which are described in the previous section), and the following *process-forming operations*: sequential composition, parallel composition. feedback and hiding. Each of this operation can be defined using `IC{_,_}` process specification (for details, see Appendix). This makes it possible to prove equations on DFG process specifications, which are valid under *buffer bisimilarity* [10].

5 Conclusion

In this paper we show how one can build Kahn's style dataflow networks inside the CML . We achieved this in two steps. First, we define the `procSpec` datatype for the dataflow process specification. This is the syntax for the language DFG . Next, we implemented the function `Compile` which translates a given process specification into the value of `process` type, which is defined using CML parameterized buffered channel type (semantics of DFG). One might argue that the syntax of DFG is "cumbersome", but our intention here was not to build a real language, but rather to add to CML a useful set of types and functions to be used for dataflow implementation purposes. More precisely, we think that DFG can be useful as a specification language for DSP . There are several reasons for this: first, we had choose a set of basic process which are common in every DSP application (regular processes, switch and select). Henceforth, we are able to control and to minimize the buffer utilization for a restricted class of processes, but still very useful class for DSP domain. Finally, in contrast to other dataflow language, we have allowed an entire high level language (ML - in our case) to be used at the node level. Thus, the level of parallelism that one want to have in its dataflow implementation can be directly controlled.

References

[1] AT&T Bell Laboratories. *Standard ML of New Jersey - User's Guide*, February 1993. version 0.93.

[2] Jack B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 362–376, Paris, France, April 9–11, 1974. Springer-Verlag.

[3] Department of Electrical Engineering and Computer Science, University of California at Berkeley. *The Almagest: Manual for Ptolemy*, 1992. Version 0.4.

[4] G. Gao, R. Govindarajan, and Prakash Panangaden. Well-behaved dataflow for DSP computation. In *Proceedings of ICASSP-92, the 1992 International Conference on Acoustics, Speech, and Signal Processing*, volume I, pages 561–564, San Francisco, California, March 23–26, 1992. IEEE Signal Processing Society.

[5] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 471–475, 1974.

[6] Edward Ashford Lee. Static scheduling of data-flow programs for DSP. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 19, pages 501–526. Prentice-Hall, Englewood Cliffs, NJ, 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, held in conjunction with the 16th Annual International Symposium on Computer Architecture in Eilat, Israel, May 1989.

[7] R. Milner, M. Tofte, and R. Harper. The definition of Standard ML. Cambridge, Massachusetts, 1990.

[8] J. H. Reppy. *The Concurrent ML Manual.* AT&T Bell Laboratories, February 1993. Version 0.9.8.

[9] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, Ontario, June 26–28, 1991. ACM SIGPLAN. *SIGPLAN Notices,* 26(6), June 1991.

[10] E. W. Stark. Calculus of dataflow networks. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 125–136, Santa Cruz, California, June 22–25, 1992. IEEE Computer Society and ACM/SIGACT.

[11] Marija Čubrić and Prakash Panangaden. Minimal memory schedules for dataflow networks. In Eike Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory*, number 715 in Lecture Notes in Computer Science, pages 368–383, Hildesheim, Germany, August 23–26, 1993. Springer-Verlag.

A DFG signatures

```
signature TokenSig =
   sig
      structure Buffer : BUFFER_CHAN

      datatype token = Int of int | Bool of bool | Real of real
      type buffer = token Buffer.buffer_chan
      val tokenSend: buffer * token -> unit
      val tokenAccept: buffer -> token

      type block = token list
      val blockAccept : buffer * int -> block
      val blockSend : buffer * block -> unit

      datatype stream = stream of unit -> (token *  (unit -> stream))
      val attach : block * stream -> stream
      val tokenStream : token -> stream
      val blockStream : block -> stream
   end

signature ProcessSig =
   sig
      structure Token : TokenSig
      structure CML : CONCUR_ML

      type pwFun = token list -> token
      type bwFun = block list -> block

      datatype procSpec =
               PW of {input: int,
                      output: (pwFun list)}
             | BW of {input: int list,
                      output: (int * bwFun) list}
             | IC of {graph:(procSpec * int list * int list) list,
                      marking: (int * Token.block) list}
             | COPY of int
             | GEN of Token.stream
             | SW
             | SEL
             | SINK
```

```
    exception badArgs of string

    type process = buffer list -> buffer list -> unit
    val Compile : procSpec -> process
    val Input : procSpec -> Token.buffer
    val Output : (procSpec * Token.buffer list * int) -> unit
    val Trace : (procSpec * Token.buffer list) -> CML.thread_id

    val Type : procSpec -> (int * int)
    val Weight : procSpec -> (int list * int list) option
    val isRegular : procSpec -> bool
    val weightChecking : procSpec -> bool
  end


signature CalculusSig =
  sig
    structure Process : ProcessSig

    val feedBack: Process.procSpec -> Process.procSpec
    val parComp:  Process.procSpec * Process.procSpec -> Process.procSpec
    val seqComp:  Process.procSpec * Process.procSpec -> Process.procSpec
    val hideOut:  Process.procSpec -> Process.procSpec
  end
```

B DFG installation

```
Installation
==============

DFG package consists of the following files:

README           this file
dfg-load-all     specify the files that should be loaded
dfg-buffer.sml   defines asynchronous communication.
                 (slightly modified buffer.sml file from CML library)
dfg.sml          implementation of DFG, including token, procSpec, buffer
                 and process type implementation, together with functions:
                 Compile,Input,Output,Trace,Type,Weight,isRegular,isWB
dfg-buffer.sml   defines asynchronous communication.
                 (slightly modified buffer.sml file from CML library)
dfg.sml          implementation of DFG, including token, procSpec, buffer
                 and process type implementation, together with functions:
                 Compile,Input,Output,Trace,Type,Weight,isRegular,isWB
dfg-lib.sml      defines often used process specifications
dfg-calc.sml     defines process-forming operations:
                 FBACK,SCOMP,PCOMP,HIDE
ex-integral.sml Implementation of integral and RC circuit.
ex-fib.sml       Implementation of fibonacci's sequence.
ex-fir.sml       Implementation of FIR filter.
ex-fib1.sml      Implementation of fibonacci's sequence using DFG-calculus
ex-bad.sml       Example of a network which is not regular (Guernic's example)
ex-ftp.sml       Example of a regular network which is not well-behaved
ex-hamming.sml   Implementation of Hamming's sequence.
ex-rem-dupl.sml Eliminates adjacent duplicates from the input stream

To run DFG you need to have CML installed. To load DFG in CML simply type:
        use "load-all"
on DFG directory. After it you can load your file or work interactively
(which is not recommended, since CML is not made for interactive usage.)
If you want to work interactively, you should open the structure DFG
(Lib and Calculus - optional), and follow the steps defined below.

DFG File format suggestion:
=============================

local
```

```
        open DFG;
        open Lib;       (* optional *)
        open Calculus;  (* optional *)
in
        val x = ...         (* define values of procSpec type *)

        val inL = ...       (* inL are of the type  'buffer list'*)
                            (* it could be spec as Input(...) or sb(...) *)

        fun out() = Output(x,inL,n)
        fun trace() = Trace(x,inL,outL)
end
```

After loading your file with "use" command you can use either:
RunCML.doit(out,NONE) or RunCML.doit(trace,NONE) to run it in CML.

Where is it?
============
You can copy DFG from \home\acaps\u1\cubric\Pub\DFG