Coordination Language for Distributed Clean[*]

Zoltán Horváth[†], Zoltán Hernyák[‡], and Viktória Zsók[†]

Abstract

The distributed evaluation of functional programs and the communication between computational nodes require high-level process description and coordination mechanism. This paper presents the D-Clean high-level functional language, which supports the distributed computation of Clean functions over a cluster. The lazy functional programming language Clean is extended by new language elements in order to achieve parallel features. The distributed computations of functions are expressed in the form of process-networks. D-Clean introduces language primitives to control the dataflow in a distributed process-network.

A process scheme defines a partial computation graph, where the nodes are functions to be evaluated and the edges are communication channels. The computational nodes are implemented as statically typed Clean programs. The schemes are parameterized by functions, types and data for defining process networks.

D-Clean is compiled to an intermediate level language called D-Box. The D-Clean generic constructs are instantiated into D-Box expressions. D-Box is designed for the description of the computational nodes. D-Box expressions hide implementation details and enable direct control over the process-network. The asynchronous communication is based on language-independent middleware services.

The present paper provides the syntax and the informal semantics of both coordination languages. To illustrate the definition of a distributed functional computational pattern using the D-Clean language a farm skeleton running example is presented.

Keywords: D.1 Programming Techniques: D.1.1 Applicative (Functional) Programming, D.1.3 Concurrent Programming. Distributed functional programming, coordination language, Clean, functional skeleton.

[*]Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr.T037742 and by the Bolyai Research Scholarship.

[†]Department of Programming Languages and Compilers, Eötvös Loránd University, Hungary. e-mail: `hz@inf.elte.hu, zsv@inf.elte.hu`

[‡]Department of Information Technology, Eszterházy Károly College, e-mail: `aroan@ektf.hu`

Introduction

Nowadays it is prevailing to develop and to test parallel functional applications on PC clusters [13]. The distributed evaluation of functional programs and the communication between computational nodes require high-level process description and coordination mechanism [1, 15]. Therefore it became important to provide distributed environments making possible the development of applications with client-programs written in functional programming languages. The proposed D-Clean language is an extension of the functional programming language Clean and supports the distributed computation of Clean functions. The computation of functions is expressed in the form of distributed process-networks. D-Clean primitives control the dataflow in the process-networks.

Skeletons are computation patterns, algorithmic schemes that captures common computation mechanism. Skeletons [12, 17] can be defined and parameterized by functions, types and data. They are widely used in parallel computations. In functional programming skeletons can be combined with evaluation strategies in order to obtain optimal parallel behaviour [11]. D-Clean supports the composition of the coordination primitives to build compound coordination structures. The coordination structures are used to define distributed functional computational skeletons, process schemes. D-Clean schemes are parameterized by types and by functions. Before instantiation the actual values of the type parameters have to be inferred from the type description of the embedded Clean expressions[1]. In the case of the widely used skeletons (like farm, divide and conquer, pipe and reduce [4, 5, 8]) it is easier to deal with the type inference problem than in general.

D-Clean is compiled to an intermediate level language called D-Box, similar to the idea of [2]. D-Box is designed for the description of the computational nodes implemented as Clean programs, which use middleware services for asynchronous communication [9, 20]. D-Box expressions hide implementation details and enable direct control over the process-network.

The D-Clean control language has a higher level coordination role, while D-Box has a lower abstraction level. The syntax and informal semantics of both coordination languages are described. We also present a mapping from D-Clean expressions to D-Box expressions.

The D-Box language is a description language for the source codes of computational nodes. In this language input and output protocols can be defined. A transformation of D-Box definitions into Clean language programs is described. A graphical developer environment was built to support a direct use of the D-Box language.

In section 1 the main concepts of the D-Clean language are presented. As an example a farm computation is defined. Section 2 presents the context-free syntax of the D-Clean language. The semantics of the D-Clean language is described in section 3 in an informal way. In section 4 the D-Box expressions corresponding to the farm example are included. Section 5 defines the context-free syntax of the

[1]Similar to the C++ template parameter deduction [19].

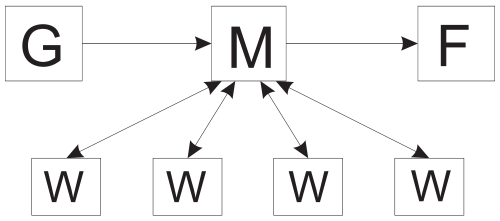

Figure 1: Farm scheme

D-Box language. The semantics of the D-Box language is described in section 6 also in an informal way and in section 7 the mapping from D-Clean to the D-Box language is given. Section 8 shows an example how to compile D-Box definitions into Clean language programs using channels. Section 9 presents measurements about the object code to be generated. A second example is presented in section 9 to demonstrate the expressiveness and the ease-of-use of D-Clean and the D-Box graphical developer environment. In section 10 the related works are discussed and in the section 11 the conclusion closes the paper.

1 An example written in D-Clean

The present paper uses the well-known farm skeleton as running example (see Figure 1). The farm skeleton divides the input data into n parts. The n parts of the original input are sent to n different worker processes. Each worker applies the same function on the partial data and calculates the n sub-results. The final result can be combined when every worker completes the tasks.

In our example the farm scheme is used as a simple distributed computation, where the computation node 'G' generates a finite list of integer numbers to be sorted. The master node 'M' receives the sortable data, splits into sublists and sends them to its workers (a worker is denoted by 'W'). The workers sort the sublists and send them back to the farm master node ('M'). The master node receives the sublists and merges them on-the-fly using the comparer function `lessThan`. The finally produced sorted list can be forwarded to the last computational node (denoted by 'F') for any further processing.

A D-Clean program consists of a start expression, in which a collection of user-defined D-Clean process schemes can be applied. A process scheme itself is written in D-Clean too. The start expression is given as the `DistrStart` function definition.

D-Clean coordination structures are mappings between communication channels and are designed as generic templates parameterized by types and by functions. The value of type parameters are determined by type inference. The templates are instantiated by the D-Clean pre-compiler at compile time.

The matching of types between the base types of channels and the types of embedded Clean expressions is a static semantic requirement. A D-Clean expression may be a compound expression or a direct use of coordination primitives. Process scheme definitions are named D-Clean expressions with formal parameters. A process scheme library can be built using the coordination primitives and the already defined schemes[2].

In our running example a user-defined scheme FARM can be applied in the start expression in the following way:

```
DistrStart =
   (DStop finish) (FARM comb solve divide N) (DStart generator)
      where
         generator = [1,4,2,3,8,6]
         finish    = (WriteResultDat "sorted.dat")
         comb      = (mergeSort lessThan)
         solve     = qsort
         divide    = (divide N)
         N         = 2
```

The above start expression is a *composition* of three coordination structures. It applies two D-Clean language coordination primitives (`DStop` and `DStart`) and the user-defined scheme called FARM (see below the D-Clean definition of the FARM scheme).

By **Ch a** we denote the type of a channel carrying elements of type **a**. A D-Clean expression is a mapping from a sequence of input channels to a sequence of output channels. The sequence of the types of the input and output channels of a coordination structure is given in the `<Ch a, Ch b, ...>` form. In our example the inferred signatures of the instantiated control structures are as follows:

```
DStart generator ::  <Ch Int>
FARM :: <Ch Int> -> <Ch Int>
DStop finish ::  <Ch Int>
```

`DStart` generates the data to be sorted (corresponds to the box 'G' in Fig. 1). An expression called `generator` is used for generating the input data. The `generator` is actually a Clean function with the output type `[Int]`. The generated data are sent via a channel to the FARM coordination structure, which computes the sorted list and forwards it to the last component of the computation. Finally `DStop` applies the `WriteResult "sorted.dat"` function on the sorted list received from the farm in order to save the result (it corresponds to the box 'F' in Figure 1). The pseudo codes of `DStart` and `DStop` primitives are presented in the appendix.

The definition of the process scheme FARM uses four parameter functions: `comb`, `solve`, `divide` and `n`. The process scheme is the composition of the coordination primitives `DMerge`, `DApply` and `DDivideS`:

```
SCHEME FARM comb solve divide n =
   (DMerge comb) (DApply solve) (DDivideS divide n)
```

[2]We present a solution to the problem of the recursively called instances of the coordination structures in a language extension of the D-Clean [10].

The process scheme FARM composes the three actions taken by the farm computational pattern: first the incoming data list is divided into n parts using the parameter function `divide` as the parameter of `DDivideS`. After that the `solve` function is applied by `DApply` on every sub-list. Finally their sub-results are collected and merged by `DMerge` into one list applying the parameter function `comb`.

The instance of the process scheme FARM has the signature `<Ch Int> -> <Ch Int>`, while the instances of the components have the following signatures[3]:

```
DDivideS divide n :: <Ch Int> -> <Ch Int, Ch Int, ..., Ch Int>
DApply solve :: <Ch Int, ..., Ch Int > -> <Ch Int, ..., Ch Int>
DMerge comb :: <Ch Int, Ch Int, ..., Ch Int> -> <Ch Int>
```

The set of the worker boxes corresponds to the boxes generated from the `DApply solve` expression.

The functions `mergeSort lessThan`, `qsort`, `divide`, `N` are the actual parameters of the FARM process scheme in our example. The role of the master node is shared between two computation nodes, implementing the tasks of `DDivideS` and `DMerge` respectively. The input is divided into `N` pieces by the user defined `divide` function, where `N=2` is a constant value. The sublists are sent to the worker nodes using a special `splitk` output protocol (see section 3). The workers solve the main task - the sorting of the sublists using the `qsort` standard Clean function. After sorting the sublists, the workers send their results to the collector node, which receives the input sublists and merges them using the `mergeSort lessThan` function.

2 The syntax definition of the D-Clean language

We introduce the following extensions to the standard BNF syntax:

- **{notion}+** means that the notion occurs at least once,
- **{notion}*** means that the notion occurs zero, one or more times,
- **{notion}-list** means one or more occurrences of the notion separated by commas,
- **terminals** are closed between apostrophes.

```
⟨DISTART_RULE⟩ :=="DistrStart" "=" ⟨DEXPR⟩
⟨DEXPR⟩        :==⟨DPRIMITIVE⟩ | ⟨SCHEME_NAME⟩ { ⟨act_param⟩ }*
                   | ⟨DEXPR⟩ ⟨DEXPR⟩
⟨DPRIMITIVE⟩   :==⟨DStart_USE⟩ | ⟨DStop_USE⟩ | ⟨DMap_USE⟩ |
                   ⟨DDivideS_USE⟩ | ⟨DMerge_USE⟩
⟨SCHEME_DEF⟩   :=="SCHEME" ⟨SCHEME_NAME⟩ { ⟨formal_param⟩}*
                    "=" { ⟨DEXPR⟩ }+
⟨SCHEME_NAME⟩  :=={⟨UpCaseLetter⟩ }+
```

[3]The description of the type inference system in general is out of the scope of this paper.

Every D-Clean program contains exactly one start expression given as the right-hand side of the `DistrStart` definition. A scheme is a compound D-Clean coordination structure parameterized by types and by functions. The actual parameters of schemes are Clean expressions [16]. The identity function is the simplest Clean expression which can be embedded into a D-Clean expression.

The type of the arguments of the Clean expressions determines the type of the communication channels, which is restricted by the limitations of the used middleware interface. Due to these limitations we say that $\mathcal{T}$ is a transmissible type, if $\mathcal{T}$ is `Int`, `Real`, `Bool`, `Char` or a record built from these basic Clean types. At this time functions cannot be transferred through channels.

⟨act_param⟩	:==	⟨fun_expr⟩
⟨fun_expr⟩	:==	⟨clean_expr⟩ | "(|" ⟨DEXPR⟩ "|)" | ⟨fun_expr⟩ ⟨fun_expr⟩
⟨formal_param⟩	:==	⟨identifier⟩

The direct use of coordination primitives is a parameterized form of basic dataflow structures.

⟨DStart_USE⟩	:==	"DStart" ⟨act_param⟩
⟨DStop_USE⟩	:==	"DStop" ⟨act_param⟩
⟨DDivideS_USE⟩	:==	"DDivideS" ⟨act_param⟩ ⟨number⟩
⟨DMerge_USE⟩	:==	"DMerge" ⟨act_param⟩
⟨DMap_USE⟩	:==	⟨Simple_DMap_DEF⟩ | ⟨Multi_DMap_DEF⟩
⟨Simple_DMap_USE⟩	:==	⟨DApplyVariations⟩ ⟨act_param⟩
⟨Multi_DMap_USE⟩	:==	⟨DApplyVariations⟩ "[" { ⟨act_param⟩ }-list "]"
⟨DApplyVariations⟩	:==	"DApply" | "DMap" | "DReduce" | "DProduce" | "DFilter"
⟨UpCaseLetter⟩	:==	"A" | "B" | "C" | "D" | . . . | "Z"

3 Informal semantics of the D-Clean language

This section presents the newly introduced coordination primitives in an informal way. The figures of the section illustrate the working mechanism. F denotes the function expression embedded into the coordination primitive.

The coordination structures use channels for receiving the input data required for the arguments of their function expressions. The results (components of a k tuple in general case) of the function expression are sent to the output channels. Every channel is capable of carrying data elements of a specified base type from one computational node to another one. We use the unary algebraic type constructor **Ch a** to construct channel types, where the base type **a** is a transmissible type.

A coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence of input channels. The coordination primitives return a sequence of output channels. The signature of the coordination primitive, i.e. the types of the input and output channels are inferred according to the type of the embedded Clean expressions. In the following the `aCh` denotes a channel type, while `aCh`* denotes a finite sequence of channel types.

DStart fun_expr :: `aCh`*

The task of `DStart` primitive is to start the distributed computation by producing the input data for the dataflow graph. It has no input channels, only output channels. The results of the `fun_expr` are sent to the output channels. Each D-Clean program contains at least one `DStart` primitive (see Figure 2).

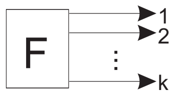

Figure 2: DStart node

DStop fun_expr :: `aCh`* –> <>

When a function expression embedded into a `DStop` primitive has k arguments, then the computation node evaluating the expression needs k input channels. Each input channel carries one argument for the function expression.

The task of this primitive is to receive and save the result of the computation. It has as many input channels as the function expression requires, but it has no output channels. `DStop` closes the computational process. Each D-Clean program contains at least one `DStop` primitive (see Figure 3).

`DStop` is the last element of the D-Clean composition, the last element of the control flow. In some cases when the control flow contains forks, the network has multiple `DStop` elements.

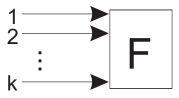

Figure 3: DStop node

DApply fun_expr :: `aCh`* –> `aCh`*

This variant of `DApply` applies the same function expression n times (see Figure 4/a) on $n * k$ channels. When the function expression has k arguments of types: $t_1, t_2, ..., t_k$, the number of input channels is $n * k$. The types of the arguments periodically match the type of the channels:
$< Ch\ t_1, Ch\ t_2, ..., Ch\ t_k, Ch\ t_1, Ch\ t_2, ..., Ch\ t_k, ..., Ch\ t_1, Ch\ t_2, ..., Ch\ t_k >$.
If the expression produces a tuple with m elements of the type $(p_1, p_2, ..., p_m)$, then

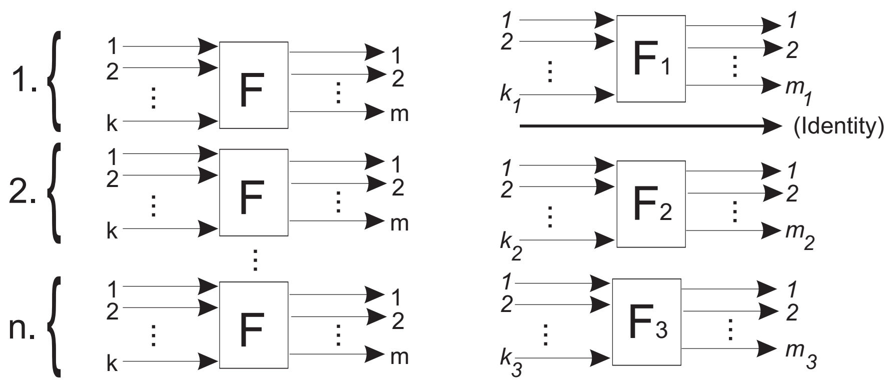

Figure 4: DApply variant a) and variant b).

the output channel sequence will contain $m * n$ elements, repeating the m type-sequences n times:
$< Ch\ p_1, ..., Ch\ p_m, Ch\ p_1, ..., Ch\ p_m, ..., Ch\ p_1, ..., Ch\ p_m >$.

DApply <fun_expr> :: `aCh`* –> `aCh`*

The second variant of `DApply` may apply different function expressions, which are given in the `<fun_expr>` sequence. The types and the number of the arguments of the function expressions can also be different. If the `<fun_expr>` sequence contains an identity function, then data received via the corresponding channel is directly forwarded to the next node.

The sequence of the input channels is constructed out of the channels required by the function expressions in the `<fun_expr>` sequence. The output sequence of channels is built up according to the results obtained by applying the function expressions. For example `DApply <`F_1`, id, `F_2`, `F_3`]>` yields the structure presented in Figure 4/b.

DFilter (a –> Bool) :: `aCh`* –> `aCh`*
DFilter <a –> Bool> :: `aCh`* –> `aCh`*

The `DFilter` primitive filters the elements of the input channels using a boolean function. It has two variants similarly to `DApply`. This is the D-Clean variant of the standard *filter* library function. This variant filters the incoming data elements before sending them to the outgoing channels.

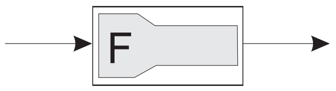

Figure 5: A DFilter node

DMap fun_expr :: `aCh`* –> `aCh`*
DMap <fun_exp> :: `aCh`* –> `aCh`*

`DMap` is a special case of `DApply` where the function expression must be an elementwise processable function [11]. It is the D-Clean variant of the standard *map* library function. It modifies the incoming data elements processing them one by one.

A valid parameter function expression for `DMap` can be a function expression either of type `a->b` or of type `[a]->[b]`. Suppose we have a list of n sublists as input data, then the `qsort::[!a]->[a]` sorting function[4] is a valid function expression as parameter for `DMap`. It takes every sublist element of the input list and applies the parameter function expression, i.e. the `qsort` function on it. The result will be the list of the n sorted sub-lists.

Figure 6: DMap nodes

DReduce fun_expr :: `aCh`* –> `aCh`*
DReduce <fun_expr> :: `aCh`* –> `aCh`*

`DReduce` is another special case of `DApply` with similar restrictions. A valid expression for `DReduce` has to decrease the dimension of the input channel type[5]. A valid expression has the type of form `[a]->b`. For example the `sum::[a]->a` function - which computes the sum of the elements of the input list - is a valid expression for `DReduce`.

DProduce fun_expr :: `aCh`* –> `aCh`*
DProduce <fun_expr> :: `aCh`* –> `aCh`*

`DProduce` is another special case of `DApply`. The expression has to increase the dimension of the channel type[6]. A valid expression must be of the form `a->[b]`. For example the `divisors::Int->[Int]` function - which generates all the divisors for an integer number - is a valid expression for a `DProduce`.

DDivideS fun_expr n :: `aCh`* –> `aCh`*

`DDivideS` is a static divider (see Figure 7). The expression splits the input data list into n parts and broadcasts them to n computational nodes. This primitive is called static divider since the value of n must be known at pre-compile time.

[4]! denotes strict evaluation of the argument.
[5]For example: list of lists → list.
[6]For example: list → list of lists.

The base type of the sublists has to be the same type as of the original list. Therefore the types of the output channels are the same as of the input ones. Consequently there will be n output channels.

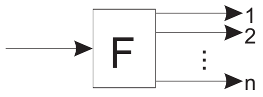

Figure 7: DDivideS node

DMerge fun_expr :: `aCh`* –> `aCh`*

`DMerge` collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type (see Figure 8).

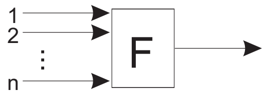

Figure 8: DMerge node

DLinear <fun_expr> :: `aCh`* –> `aCh`*

`DLinear` is a special coordination primitive. It simplifies the definition of the pipeline computation graph, where the nodes are connected to each other in a linear way (see Figure 9).

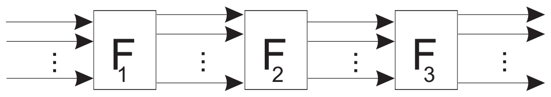

Figure 9: DLinear nodes

`DLinear <`$expr_1, expr_2, \ldots, expr_k$`>` is equivalent to the following composition of `DMap` primitives: `(DMap` $expr_k$`) ...(DMap` $expr_2$`) (DMap` $expr_1$`).`

4 Examples on D-Box language

The D-Box language is used to generate the Clean code for a computational node. D-Clean expressions are mapped to D-Box definitions, the details of the mapping are given in section 7. Every box definition describes a computational node which contains an embedded expression, the input protocol and the output protocol (see Figure 10).

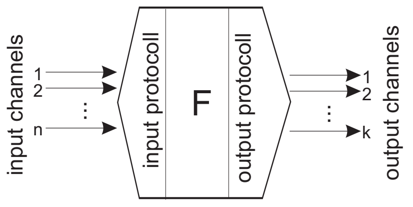

Figure 10: A computational node

```
BOX <BOXID>
  { { <INPUT_DEF> }, { <EXPRESSION_DEF> }, { <OUTPUT_DEF> } }
```

A computational node may use more than one input channel. At this level a channel identification mechanism is used. One input channel is described by its type and by the unique id of the channel. Notation $[\mathcal{T}]$ is used in the type description of a channel, which is used to transfer a single list of elements of the base type $\mathcal{T}$. Whenever a list of lists is sent via a channel, type $[[\mathcal{T}]]$ is associated to it.

The input protocol also determines the synchronization mode of the input channels. There are three modes: `memory`, `join1` and `joink` (see section 5). The input is completely defined when the list of the input channels (`<INPUT_CHANNEL_LIST>`) and the input protocol (`INPUT_PROC_MODE`) are given.

The number and/or the base types of the input channels can be different from the types of arguments of the expression (`<ARGUMENT_TYPE_LIST>`). The matching of channel types to argument types is completed at code generation time according to the actual protocol. The same holds for the `<RESULT_TYPE_LIST>` too.

The output protocol definition has the same structure as the input definition. A complete D-Box definition has the following parts:

```
BOX <BOXID>
{  { (<INPUT_CHANNEL_LIST>), INPUT_PROC_MODE },
   { (<ARGUMENT_TYPE_LIST>), <EXPRESSION>, (<RESULT_TYPE_LIST>) },
   { (<OUTPUT_CHANNEL_LIST>), OUTPUT_PROC_MODE } }
```

The running example presented in section 1 is mapped into D-Box expressions. A detailed description of the generated D-Boxes is given in the following.

The `BoxID_00` definition describes a computational node, which generates the data. Because it requires no input, there are no input channels, and the input protocol is the *memory* protocol. It produces a list of integers, which values are sent to the channel with the id #1. The *split1* protocol means a one-to-one mapping of the components of the results to the output channels.

```
BOX BoxID_00  // for DStart generator
{    { ( null ), memory }, // INP CHNS and PROT
     { ( null ), generator, ( [Int] ) }, // EXPR
     { ( ( [Int], 1 ) ), split1 } // OUTP CHNS and PROT }
```

The `BoxID_01` definition describes the first task of the farm master node (the node marked with 'M' in the farm scheme in Figure 1). It receives integer elements from channel #1. The *join1* protocol reads this input channel and passes the data elements to the expression as arguments. The expression applies the `divide` function on it with the constant parameter `N`. The result is a list of N sublists. In the running example $N = 2$. These two sub-lists are sent to channels #2 and #3 by the *splitk* output protocol.

```
BOX BoxID_01  // for DDivideS divide N
{     { ( ( [Int], 1 ) ),  join1 }, // INP CHNS and PROT
      { ( [Int] ), divide N, ( [[Int]] ) }, // EXPR
      { ( ( [Int], 2 ), ( [Int], 3 ) ), splitk } // OUTPUT }
```

The `BoxID_02` definition implements the first farm worker node. It receives the input list from channel #2, then sorts the list using the `qsort` function. The sorted list is sent to channel #4.

The *split1* protocol sends the elements of the result directly to a channel.

```
BOX BoxID_02   // for DMap qsort
{      { ( ( [Int], 2 ) ), join1  },
       { ( [!Int] ), qsort, ( [Int] ) },
       { ( ( [Int], 4 ) ), split1 } }
```

The `BoxID_03` describes the second farm worker node. The only differences are the id-s of the input and output channels: #3 and #5 respectively.

The `BoxID_04` definition presents the D-Box code of the second job of the farm master node. The data received from the two farm worker nodes on channels #4 and #5 are merged. After reconstructing the list of lists it applies the `mergeSort lessThan` function composition. In this particular case the expression merges two sorted lists and sends to channel #6.

The *joink* protocol merges the different input channels and constructs a list of lists. The elements of sub-lists are received on different channels.

```
BOX BoxID_04 // for DMerge (mergeSort lessThan)
{    { ( [Int], 4 ), ( [Int], 5 ) ), joink },
     { ( [[Int]] ), mergeSort lessThan, ( [Int] ) },
     { ( ( [Int], 6 ), split1 }}
```

The last D-Box definition describes the final box. It receives the input data from channel #6 and saves it to a file. The box has no output channel, so the *memory* protocol is used.

```
BOX BoxID_05 // for DStop (WriteResultDat "sorted.dat")
{    { ( ( [Int], 6 ), join1 },
     { ( [Int] ), WriteResultDat "sorted.dat",  (null ) },
     { ( null ), memory } }
```

5 Syntax definition of the D-Box language

The D-Box language has a lower abstraction level for describing the distributed computation. Each D-Box definition defines one computational node. The definition consists of three parts: the input protocol, the embedded expression and the output protocol. Both protocols contain the descriptions of the channels and the processing mode.

```
⟨BOXDEF⟩        :== ”BOX” ⟨BoxID⟩
                    ”{” ⟨InpProt⟩ ”,” ⟨ExpressionDef⟩ ”,” ⟨OutProt⟩ ”}”
```

The expression part contains the specification of the types of the arguments, the Clean expression itself and the types of the components of the result.

```
⟨ExpressionDef⟩ :== ”{” ”(” ({⟨TypeDef⟩}-list | ”null”) ”)” ”,” ⟨Expression⟩ ”,”
                     ”(” ({⟨TypeDef⟩}-list | ”null”) ”)” ”}”
```

The expression can be a pure Clean expression, or a composition of Clean functions and embedded D-Clean expressions. An embedded D-Clean expression must be lifted out [10]. It generates a sub-graph, which has an entry and an exit box. On this level (D-Box level) the box id-s of these boxes must be given as arguments of a *BOXES* expression.

```
⟨Expression⟩    :== ⟨BoxesExpr⟩ | ⟨CleanFv⟩ | ⟨Expression⟩ ⟨Expression⟩
⟨BoxesExpr⟩     :== ”BOXES” ”(” ⟨BoxID⟩ ”,” ⟨BoxID⟩ ”)”
⟨BoxID⟩         :== ⟨string⟩
```

The input section contains the description of the channels (types and id-s) and the concept of synchronizing and mapping the incoming data elements.

```
⟨InpProt⟩       :== ”{” ({⟨IChannelDef⟩}-list|”null”) ”,” ⟨InpProtMain⟩ ”}”
⟨InpProtMain⟩   :== ”join1” | ”joink” | ”memory”
⟨IChannelDef⟩   :== ”(” ⟨TypeDef⟩ ”,” ⟨IChannelID⟩ ”)”
⟨IChannelID⟩    :== ⟨Number⟩
```

Similarly the output section defines the output channels and the output protocol.

```
⟨OutProt⟩       :== ”{” {⟨OChannelDef⟩}-list ”,” ⟨OutProtMain⟩ ”}”
⟨OutProtMain⟩   :== ”split1” | ”splitk” | ”memory”
⟨OChannelDef⟩   :== ”(” ⟨TypeDef⟩ ”,” ⟨OChannelID⟩ ”)”
⟨OChannelID⟩    :== ⟨Number⟩
```

Transmissible types, and list or list of list of transmissible types are allowed.

```
⟨TypeDef⟩       :== ⟨TypeName⟩ | ”[” ⟨TypeName⟩ ”]” | ”[[” ⟨TypeName⟩ ”]]”
```

6 Informal semantics of the D-Box language

A box defines a computational node which is a Clean language program. It receives input data from input channels, then executes the computation and sends the results to output channels. The channels in our environment are remote objects providing operations to retrieve data elements from and to store elements. A computational node may perform the following actions:

1. connects to the input channels and gathers their identifiers into a list,
2. reads the input data elements from all the input channels,
3. processes the input data elements using the embedded expression of the box,
4. calculates the result of the computation,
5. connects to the output channels and gathers their identifiers into a list,
6. sends the result to the channels.

Some of these actions may overlap in time as a consequence of the lazy evaluation strategy of the host functional language.

The protocols define the processing mode of the input and the output channels, including the mapping of the input channels to the arguments of the expression. First we enumerate the protocol names, then we define their meanings.

The input protocols are the following: `memory`, `join1`, `joink`, while the output protocols are: `memory`, `split1`, `splitk`. In the following we give a detailed description of the above mentioned actions and protocols.

If a box has no input channels, then actions 1 and 2 are not performed. The input protocol is the `memory` protocol.

When the input arguments of the expression are carried by different input channels, a one-to-one mapping between arguments and channels is required. This input protocol is `join1`.

Let us consider an example. The first channel (list_a) is processed lazily, but the second one is processed strictly (list_b). The difference is given by the type definition of the embedded expression. The `!` annotation indicates the strict evaluation mode, since the default evaluation is the lazy one. Observe that the lazy or strict processing of the channels depends on the type definition of the expression. The processing mode is automatically implemented by the protocol.

```
expr:: [a] [!b] -> [c]
list_a = list of all the elements from input channel 1
list_b = list of all the elements from input channel 2
result  = expr list_a list_b
send result to the output channel
```

Let us observe how a strictness annotation controls the semantics of the protocol. The first parameter of the `expr` is a list of `a` evaluated lazily. Reading and evaluating of the next element of `list_a` will always be postponed until it is really needed by the evaluation of the expression. The type of the second argument is annotated for a strict evaluation. The Clean reduction system is forced to read and evaluate the whole `list_b` before the evaluation of the expression starts.

An argument of the expression may have the type of list of lists (sub-lists). When elements belonging to different sub-lists are carried by different input channels, then the list of lists is built by the input protocol `joink`.

Similar rules are valid for the output protocols. When the result of the expression is a tuple $(r_1, r_2, \ldots, r_k)$, we need k output channels, one for each result component. In this case the output protocol is the `split1` protocol. The result elements are sent to the output channels one by one. When a computation is terminated (all the input elements are processed), then a terminal signal is sent to all the output channels.

When the expression produces list of sub-lists, then the `splitk` protocol may be used. In this case we need as many output channels as many sub-lists the result list contains. The `splitk` protocol sends the sub-lists to different channels. When the box needs no output channel (for example a DStop-box), the output protocol is `memory`.

7 Mapping from D-Clean to D-Box

First we define several functions. The `elementTypeOf` function gives the base type of the given list. Type inference is done at compile time. It is required to determine all the types of the necessary channels before the code generation starts.

The `lengthOf` function determines how many elements are in a finite list. In case a list is a list of lists, then `lengthOf` determines the number of the sublists. The `nextChannelID` function generates the next free channel id number which was never used before and they are positive integer numbers. The `nextBoxID_nn` function generates a unique box id. It is required to be in form "BoxID_nn" and must be unique.

Here we give the structure of a TUP, the base type used in the description of the coordination structures at D-Box level:

TUP = (TypeDef, Id)

where `TypeDef` defines the type of the channel and the `Id` is a unique identification number of the communication channel.

In addition we define an `OUTPROT_LIST` expression using the following algorithm. The algorithm processes the type of the components of the result of a given expression and generates the output protocol list for the box. The output protocol list is constructed according to the result types of the expression and generating id-s for the channels in parallel.

```
OUTPROT_LIST :: [TypeDef] -> [TUP]
OUTPROT_LIST []     = []
OUTPROT_LIST [x:xs] = [(x, nextChannelID) : OUTPROT_LIST xs]
```

Function `INPROT_LIST` processes a list of types (the list of the types of the arguments of an expression) and a TUP list in parallel and generates the input protocol list for the box. The types of the arguments has to match the types of the channels.

```
INPROT_LIST :: [TypeDef] [TUP] -> [TUP]
INPROT_LIST [] [] = []
INPROT_LIST [x:xs] [(t, c) : ts]
   | match(x,t)    = [(t, c) : INPROT_LIST xs ts]
   | otherwise     = abort "Error!"
```

For each D-Clean coordination primitives we give the description of the generated boxes including the protocols used in the following.

The input protocol of a `DStart` structure is `null`. The output channel list is defined by processing the output types of the expression. This primitive uses the `split1` protocol. The result of the mapping is a box definition and a TUP list of output channels.

$\mathcal{D}$([**DStart expr**] `= [ (B, RESULT_OUTPUT_TUP_LIST) ]`, where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
B = BOX NextBoxID_nn
{   { (null), memory },
    { inputTypeOf expr, expr, outputTypeOf expr }
    { RESULT_OUTPUT_TUP_LIST, split1 } }
```

The output protocol of a `DStop` primitive is always `null`. The input channel list is determined by the previous control structure in the computational graph. The result of the mapping is a box definition and an empty output TUP list.

$\mathcal{D}$ [**DStop expr TUP_list**] `= [ (B, RESULT_OUTPUT_TUP_LIST) ]`, where

```
RESULT_OUTPUT_TUP_LIST = []
BOX NextBoxID_nn
{   { INPROT_LIST (inputTypeOf expr) TUP_List, join1 },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { (null), memory } }
```

The `DApply` primitive is mapped to n box definitions (see Section 3). One box uses k channels as its own input channels from the TUP list. The mapping is done when all the channels are bound. Each box will contain the same expression. The input protocol is always `join1` and the output protocol is `split1`. The result output TUP list is the merged list of all the output channels of all the boxes. The result also contains a set of boxes.

$\mathcal{D}$ [**DApply expr TUP_list**] `= F TUP_list [] expr`, where

```
F [] Y expr = Y
F TUP_List Y expr =
  F (drop k TUP_List) (Y ++ [(B, OUTPUT_TUP_LIST)]) expr
 where
  OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
  k = lengthOf (inputTypeOf expr)
  B = BOX NextBoxID_nn
  { { INPROT_LIST (inputTypeOf expr) (take k TUP_list), join1 },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { OUTPUT_TUP_LIST, split1 } }
```

The second variant of `DApply` is mapped to more box expressions (see Section 3), but each box contains different expressions. Each box uses different numbers of channels as their input channels according to the actual number of the arguments. If the expression is the identity expression, there is no need to define a real box. The result output TUP list is the merged list of all the output channels of all the boxes. The result also contains a set of boxes.

$\mathcal{D}$ [**DApply expr_list TUP_list**] = `F TUP_LIST expr_list []`, where

```
F TUP_list [] Y = Y
F TUP_List [expr:xs] Y
     | expr == id
         = F (drop k TUP_List) xs (Y ++ [(B, OUTPUT_TUP_LIST)])
     | otherwise
         = F (drop k TUP_list) xs [Y : (B, OUTPUT_TUP_LIST)]
  where
     OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
     k = lengthOf (inputTypeOf expr)
     B = BOX NextBoxID_nn
     { { INPROT_LIST (inputTypeOf expr) (take k TUP_list), join1 },
       { inputTypeOf expr, expr, outputTypeOf expr },
       { OUTPUT_TUP_LIST, split1 } }
```

The following keywords are special cases of the `DApply` coordination primitive and their semantics can be given in an analogous way:
$\mathcal{D}$ [**DMap expr**] = $\mathcal{D}$[`DApply expr`]
$\mathcal{D}$ [**DMap expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DReduce expr**] = $\mathcal{D}$ [`DApply expr`]
$\mathcal{D}$ [**DReduce expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DProduce expr_list**] = $\mathcal{D}$ [`DApply expr`]
$\mathcal{D}$ [**DProduce expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DFilter expr**] = $\mathcal{D}$ [`DApply (filter expr)`]
$\mathcal{D}$ [**DFilter** $[expr_1, expr_2, \ldots, expr_k]$] =
$\mathcal{D}$ [`DApply` $[filter\ expr_1, filter\ expr_2, \ldots, filter\ expr_k]$]
`DLinear` keyword is a special case of the `DMap` primitive, so the semantics of it can be given as the following composition:
$\mathcal{D}$ [**DLinear** $[expr_1, expr_2, \ldots, expr_n]$] =
$\mathcal{D}$ [DMap $expr_n$ ... DMap $expr_2$ DMap $expr_1$]

`DDivideS` is mapped to a box definition where the output protocol is always `splitk`. The output TUP list contains N elements. The divider expression splits the input list into N sub-lists.
$\mathcal{D}$ [**DDivideS expr N TUP_list**]= `[ (B, RESULT_OUTPUT_TUP_LIST) ]`, where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT (outputTypeOf expr) N
OUTPROT _ 0 = []
OUTPROT l k = OUTPROT_LIST l ++ OUTPROT l (k-1)
B = BOX NextBoxID_nn
    {  { INPROT_LIST (inputTypeOf expr), TUP_list, join1 },
       { inputTypeOf expr, expr, outputTypeOf expr },
       { RESULT_OUTPUT_TUP_LIST, splitk } }
```

`DMerge` is mapped to a box definition where the input protocol is always `joink` and the output protocol is `split1`. The result of the mapping produces an output TUP list.

$\mathcal{D}$ [**DMerge expr TUP_list**] = `[ (B, RESULT_OUTPUT_TUP_LIST) ]`, where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
B = BOX NextBoxID_nn
{   { INPROT (inputTypeOf expr), TUP_List, joink },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { RESULT_OUTPUT_TUP_LIST, split1 } }
where
    INPROT _ [] = []
    INPROT l t  = INPROT_LIST l t ++ INPROT l (drop k t)
    k = lengthOf (InputTypeOf expr)
```

The problem of the embedded D-Clean expressions is discussed in [10].

8 Mapping from D-Box to Clean

The conversion between a D-Box code and a Clean program is straightforward. We can use several functions defined in the middleware interface library. The library contains skeletons for the pre-compiler, who instantiates them according to the actual input types. First we initialize the middleware. Each computational node communicates with at least one other computational node. Notation `#` introduces a let expression, while `#!` forces an immediate evaluation of the let expression. `#!` induces a sequential evaluation of the expressions. Function `CHANNEL_FIND` is implemented according to the actual middleware.

The code generated from the following D-Box definition is presented:

```
BOX BoxID_04  // for DMerge (mergeSort lessThan)
{     { ( [Int], 4 ), ( [Int], 5 ) ), joink },
      { ( [[Int]] ), mergeSort lessThan, ( [Int] ) },
      { ( ( [Int], 6 ), split1 } }
```

After the middleware is initialized, the input channel list is built up by using their ChannelID-s.

```
Start w =
   #! w              = MIDDLEWARE_INIT w
   # (inp_chan_1,w)  = CHANNEL_FIND 4 w
   # (inp_chan_2,w)  = CHANNEL_FIND 5 w
   # input_channels  = [inp_chan_1, inp_chan_2]
```

ChannelID-s are determined from the input protocol section of the box. Afterwards, the input process is started in order to gather all the incoming data elements from the input channels.

```
   # (input_data,w)  = Joink input_channels w
```

Then we process the data:

```
   # result          = mergeSort lessThan input_data
```

Before sending the result, connections to the output channels are required. The ChannelID-s included into the box output protocol section are used.

```
# (outp_chan_1,w) = CHANNEL_FIND 6 w
# output_channels = [outp_chan_1]
```

The resulted data are sent to the output channels:

```
# w               = Split1 output_channels result w
```

The complete generated code of our D-Box example is given here:

```
Start w =
   #! w              = MIDDLEWARE_INIT w
   # (inp_chan_1,w)  = CHANNEL_FIND 4 w
   # (inp_chan_2,w)  = CHANNEL_FIND 5 w
   # input_channels  = [inp_chan_1, inp_chan_2]
   # (input_data,w)  = Joink input_channels w
   # result          = mergeSort lessThan input_data
   # (outp_chan_1,w) = CHANNEL_FIND 6 w
   # output_channels = [outp_chan_1]
   # w               = Split1 output_channels result w
   = w
```

9 Measurements

A sequential and several parallel implementations of the running example are measured and compared. We sorted integer lists of 250, 500, 1000, 2000, 4000 elements and we used 2 and 4 PC-s for the parallel implementations. The first diagram (see Figure 11) shows the computation time in seconds (Y axis) and the length of the lists (X axis). We used a weighted comparer function `lessThan` to slow down the computation simulating a complicated comparison of two data elements.

The diagram of the Figure 12 shows the speed-up. The diagrams show that

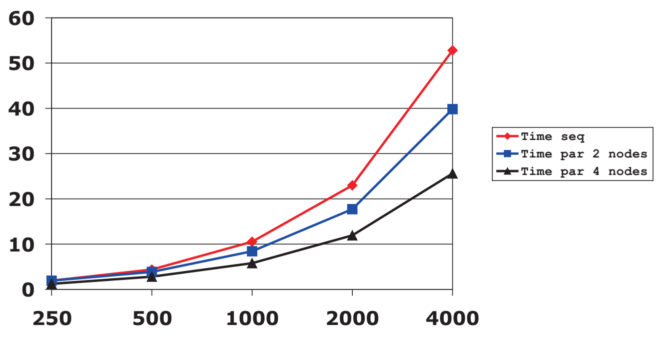

Figure 11: The computation time of the FARM skeleton

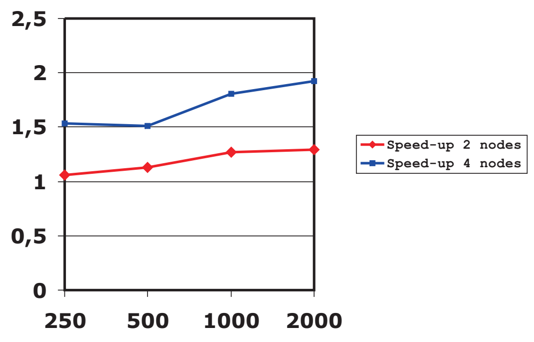

Figure 12: The speed-up of the FARM skeleton

the distributed implementation is efficient using the mapping from D-Clean to D-Box. A slight increasing of the speed-up can be observed. In case the workers are computing a weighted function, the overheads obtained during the communications are negligible.

Similar measurements can be found in earlier papers [9, 20].

Henceforth a second example is presented to *demonstrate the expressiveness* and ease-of-use of D-Clean. A matrix m is given with k columns and k rows. A sequence of vectors $vs =< v_1, \dots v_n >$ is generated, the size of each vector is k. The sequence of products $< v_1 * m, \dots, v_t, \dots, v_n * m >$ has to be calculated elementwise. We compute the sequence sequentially, but the k elements of each product $(v_t * m)(1..k)$ is calculated in a parallel way. Every vector v_t is replicated in k copies $(v_{t_1}, \dots v_{t_k})$ first and the scalar products of the columns and a copy of the vector, $(\sum_{j=1}^{k}(v_{t_i}(j) * m(j)(i)))$ is computed in parallel for every $i \in [1..k]$.

We present the structure of a D-Clean solution and the diagram of the corresponding process network. We omit the details of the D-Box definitions, which may be obtained by compiling D-Clean to D-Box or by generating D-Box source text by the D-Box graphical developer tool [6] (see Figure 13).

```
DistrStart =
    (DStop saver) (DMerge vectorize)
    (DApply [multiply column_1, multiply column_2, ..., multiply column_k])
    (DDivideS repeater k) (DStart vector_generator)
  where
    k = 4
    saver = saveToFile "result.dat"
    vectorize = id
    column_1 = getColumn matrix 1
    ...
    column_k = getColumn matrix k
    repeater k inp = take k (repeat inp)  // k copies
vector_generator:: [[Int]] // generates the vectors
getColumn:: [[Int]] Int -> [Int] // gets the ith column
multiply::[Int] [Int] -> Int
```

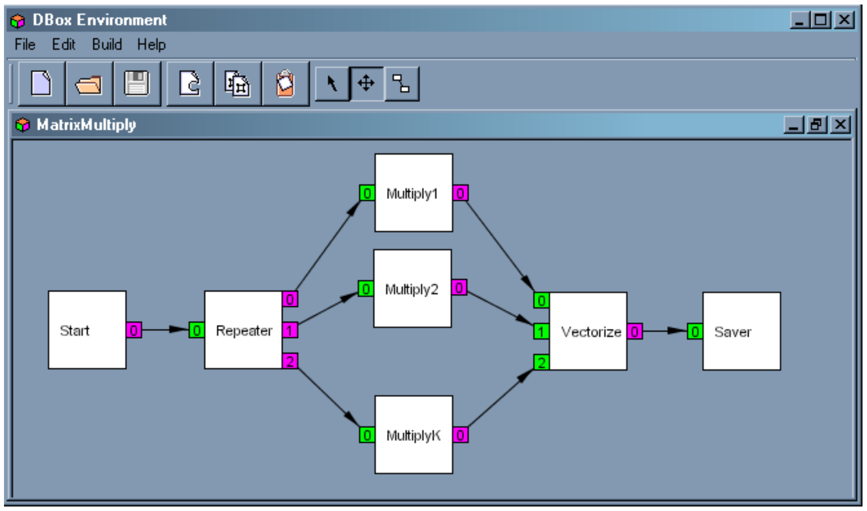

Figure 13: Matrix-multiplier example

10 Related works

- PMLS and GpH are implicit parallel extensions of ML and Haskell respectively [14], on the other hand D-Clean uses explicit coordination structures.

  Opposed to skeleton based languages, D-Clean is designed to implement skeletons of distributed functional computations in the language itself.

- Eden [15, 13] extends Haskell to explicitly define parallel computation. Eden program consits of processes and uses communication channels, and the programmer has explicit control over communication topology. The execution is based on GHC implementation of concurrency, the run-time system controls sending and receiving messages, process placements and data distribution. On the other hand the middleware supporting the implementation of DClean and DBox languages is not language specific, components developed using other languages can be integrated into easily distributed applications.

- Nimo [3] is a visual functional dataflow language, supporting process networks. Nimo allows totally graphic programming only, while DClean and DBox programs can be expressed in textual code form too. Nodes in Nimo are restricted for a fixed set of primitive operations of Haskell prelude, while in DClean nodes Clean expressions are allowed to achieve full power of functional programming at node level. Nimo does not support distributed computing, only concurrent execution is supported.

- JoCaml is an extension of Objective Caml with primitives for network-transparent distributed and mobile programming [7] based on the join-calculus model instead of a pure data flow approach.

Advanced discussion and survey of the dataflow languages can be found in [18]. Data oriented skeletons (like the farm skeleton) can be implemented using primitives which are quite similar to the primitives of dataflow languages.

11 Conclusion and future works

The distributed functional skeletal programming requires higher order coordination structures in order to coordinate the computation of several functional clients in an abstract way. We extended Clean with powerful coordination language elements as tools for description of distributed computation patterns. We proposed a higher level and an intermediate level coordination language, the D-Clean and the D-Box languages. The implementation is based on a multi-paradigm environment, using an object-oriented middleware, which supports the interconnection of client and server programs written in different programming languages.

The high-level coordination language D-Clean is appropriate for definition of functional skeletons at a very high abstraction level. These skeletons can be parameterized by types and by functions. As future work we plan to extend the language with the possibility of parameterization of schemes by distribution strategy and to enable the description of dynamic configurations. We also plan a description of a more formal type system and type constraints. Dynamic creation of the computational nodes, communication channels and dynamic start of the functional components are highly needed at the development of the applications using recursive expressions.

12 Acknowledgements

The authors would like to thank the anonymous referees for their insightful comments, which has helped immensely in improving the quality of the paper.

References

[1] Berthold, J., Klusik, U., Loogen, R., Priebe, S., Weskamp, N.: High-level Process Control in Eden, In: Kosch, H., Böszörményi L., Hellwagner, H. (Eds.): *Parallel Processing, 9th International Euro-Par Conference, Euro-Par 2003*, Proceedings, Klagenfurt, Austria, August 26-29, 2003, Springer Verlag, LNCS Vol. 2790, pp. 732-741.

[2] Best, E., Hopkins, R. P.: B(PN)^2 - a Basic Petri Net Programming Notation, In: Bode, A., Reeve, M., Wolf, G. (Eds.): *Parallel Architectures and Languages Europe, 5th International PARLE Conference, PARLE'93*, Proceedings, Munich, Germany, June 14-17, 1993, Springer Verlag, LNCS Vol. 694, pp. 379-390.

[3] Clerici, S., Zoltan, C.: A Graphic Functional-Dataflow Language, In: Loidl, H.W. (Ed.): *Proceedings of the Fifth Symposium on Trends in Functional Programming*, Ludwig-Maximilians University, 25-26 November, Munich, Germany, 2004, pp. 345-359.

[4] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G. (Eds.): *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.

[5] Danelutto, M., Di Cosmo, R., Leroy, X., Pelagatti, S.: Parallel Functional Programming with Skeletons: the OCAMLP3L experiment, In: *Proceedings of the ACM, Sigplan Workshop on ML*, Baltimore, USA, September 1998, pp. 31-39.

[6] Dezső B.: *DBox Developer Environment*, Project work documentation, Department of Programming Languages and Compilers, University Eötvös L., Budapest, Hungary, 2005.

[7] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: A Language for Concurrent Distributed and Mobile Programming, In: Johan Jeuring, Simon Peyton Jones (Eds): *Advanced Functional Programming, 4th International School, AFP 2002, Oxford*, Revised Lectures, Springer, LNCS 2638, pp. 129-158, 2003.

[8] Hammond, K., Portillo, A. J. R.: Haskel: Algorithmic Skeletons in Haskell, In: Koopman, P.; Clack, C. (Eds.): *Implementation of Functional Languages, 11th International Workshop IFL'99*, Lochem, The Netherlands, September 7-10, 1999, Selected Papers, Springer Verlag, LNCS Vol. 1868, 2000, pp. 181-198.

[9] Hernyák Z., Horváth Z., Zsók V.: Clean-CORBA Interface Supporting Skeletons, To appear in: *Proceedings of 6th International Conference on Applied Informatics*, Eger, Hungary, January 27-31, 2004.

[10] Hernyák Z., Horváth Z., Zsók V.: Design of Language Elements for Dynamic Distributed Computation of Clean Expressions on Clusters, In: Loidl, H.W. (Eds.) *Proceedings of the Fifth Symposium on Trends in Functional Programming*, Ludwig-Maximilians University, 25-26 November, Munich, Germany, 2004, pp. 257-270.

[11] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, In: *Mathematical and Computer Modelling* 38, pp. 865-875, Pergamon, 2003.

[12] Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

[13] Loidl, H.W., Klusik, U., Hammond, K., Loogen, R., Trinder, P.W.: GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster In: Gilmore, S. (Ed.): *Trends in Functional Programming*, Vol. 2, pp. 39-52, Intellect, 2001.

[14] Loidl, H-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G.J., Peña, R., Priebe, S., Rebón Portillo, Á.J., Trinder, P.W.: Comparing Parallel Functional Languages: Programming and Performance, In: *Higher-Order and Symbolic Computation* 16 (3), pp. 203-251, Kluwer Academic Publisher, September 2003.

[15] Peña, R, Rubio, F., Segura, C.: Deriving Non-Hierarchical Process Topologies, In: Hammond, K., Curtis, S.: *Trends in Functional Programming*, Vol 3. pp. 51-62, Intellect 2002.

[16] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001, http://www.cs.kun.nl/~clean/Manuals/manuals.html.

[17] Serrarens, P.R.: *Communication Issues in Distributed Functional Computing*, Ph.D. Thesis, Catholic University of Nijmegen, January 2001.

[18] Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar: *Advances in dataflow programming languages*, ACM Comput. Surv., ACM Press, New York, USA, 36(1), 2004, pp. 1-34.

[19] Zólyomi I., Porkoláb Z., Kozsik T.: An extension to the subtype relationship in C++ implemented with template metaprogramming, *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, Erfurt, Germany, Springer-Verlag New York, 2003, pp. 209-227.

[20] Zsók V., Horváth Z., Varga Z.: Functional Programs on Clusters, In: Striegnitz, Jörg; Davis, Kei (Eds.): *Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'03)*, Interner Bericht FZJ-ZAM-IB-2003-09, July 2003, pp. 93-100.

13 Appendix

The pseudo code of `DStart expr`:

```
Start w
   # (result1, result2, ..., w) = expr w
   # (channel1, w) = Channel_FIND output_channel_id_1 w
   ...
   # channels = [channel1, channel2,...]
   # w        = split1 channels result_1 result_2 ... w
   = w
```

The pseudo code of `DStop expr`:

```
Start w
   # (channel1, w) = Channel_FIND input_channel_id_1 w
   ...
   # channels = [channel1, channel2,...]
   # (data, w) = join1 channels w
   = expr data w
```

The pseudo code of `split1` and `splitk`:

```
split1 ::[ChannelID]  [a] [b] [c] [d] ... *World -> *World
split1 channels data0 data1 data2 ... w
     # w = SendStream channels!!0  data0 w
     # w = SendStream channels!!1  data1 w
     ...
     = w
splitk :: [ChannelID] [[a]] *World -> *World
splitk [] [] w = w
splitk [ch:channels] [d:data] w
   = SendStream ch d w2
     where
       w2 = splitk channels data w
```

The pseudo code of `join1`:

```
join1 ::[ChannelID] *World -> ([a],[b],[c],...,*World)
join1 channels w
      # (a,w) = ReceiveStream channels!!0 w
      # (b,w) = ReceiveStream channels!!1 w
      ...
      = (a,b,c,...,w)
```

The pseudo code of `joink`:

```
joink:: [ChannelID] *World -> ([[a]], *World)
joink [] w = ([],w)
joink [ch:cs] w  = ( [data : remaining], w4 )
  where
    (data, w3) = ReceiveStream ch w
    (remaining, w4) = joink cs w3
```