Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 85–95

www.elsevier.com/locate/entcs

Generic Executable Semantics for D-Clean [1]

Viktória Zsók[2]

Department of Programming Languages and Compilers, Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary

Pieter Koopman[3] Rinus Plasmeijer[4]

Nijmegen Institute for Computing and Information Sciences
Radboud University Nijmegen
Nijmegen, The Netherlands

Abstract

D-Clean primitives are first class citizens which allows the coordination of a dynamical work distributions over a cluster. The computations are distributed automatically over the Grid by the middleware system. The programmer controls the computation nodes in the generated boxes and the communication on the generated channels. In order to obtain highly abstract description about how the coordination primitives work, a generic model of the executable semantics is needed.
This paper provides a more general version of the simulation of the real parallel computation in the D-Clean extension of the Clean language. First, the executable semantics definition for each D-Clean primitive is given in an abstract way. Second, we describe a graphical system that generates the computation scheme visualizing the maximum amount of parallelism. Finally, we state properties of the executable description of the distributed system designed for D-Clean and D-Box.

Keywords: D-Clean, executable semantics, distributed functional programming, skeletons.

1 Introduction

Distributed Clean [8,19], or D-Clean, is a distributed extension for Clean [16] over cluster. This parallel system was designed to coordinate functional programming tasks on a cluster using a high level coordination language parameterized by functional programming computation nodes. However, the designed coordination model can work with computations implemented in other programming languages too.

[1] Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003

[2] Email:zsv@inf.elte.hu

[3] Email:pieter@ru.cs.nl

[4] Email:rinus@ru.cs.nl

doi:10.1016/j.entcs.2011.11.040

D-Clean is the top layer of the distributed system for which it was designed. It has coordinating role, and it is used to define parallel computation schemes, skeletons.

Using D-Clean the programmer indicates by a high level coordination language how a distributed computation can be placed into generated computational nodes corresponding to D-Box *computation boxes* connected via *communication channels* (buffered tools for communications).

The D-Box language is the intermediate level language, designed based on Petri nets [3]. The task of a box is to calculate the function wrapped into a node of the distributed computation graph. The boxes are compiled into Clean programs, and they communicate via channels according to well established protocols.

Channel and box generation is part of the distributed system, however here especially the boxes are ignored to focus on the definition of the semantics of the primitives of the higher level language D-Clean.

The functional computation nodes are coordinated by a well defined, relatively small number of specific D-Clean primitives, and they are applied to dataflows. The dataflows carried by the channels are typed according to the rules of the input-output protocols of the boxes generated for the evaluation of the computations.

The semantics of the two languages is described informally in [8]. In this paper first we give a more general and executable description of the semantics using Clean as description language abstracting from the real distributed environment and leaving out the details of the box and channel generation.

In a number of D-Clean examples high speed-up for parallelism was obtained (see in [20] the D-Clean variant of the problem described in [9]). Here we will emphasize mainly the generic and executable description of the operational semantics of the D-Clean language primitives. The executable semantics allows us to depict the potential amount of parallelism graphically as well. The expected parallelism (analogously to the real distributed system) will be drawn by the graphical visualization of the boxes and channels generated in parallel for a well defined distributed computation.

The actual amount of parallelism, and hence the speed-up, depends on many factors like the order of channel creation, the amount of work on one channel, the speed of the data retrieving and data storage in the channels, the complexity of the computation described in the distributed graph. Nevertheless, the information on the potential amount of parallelism is a valuable tool in the design of D-Clean programs with good speed-ups. Therefore, the visualization of the distributed computation using the executable semantics provides a useful support when programming in the real distributed system.

The paper is organized as follows: first we give the D-Clean primitives and their executable semantics, then we depict graphically D-Clean programs, and finally we formulate properties of the executable semantics of the visualized examples.

2 D-Clean primitives and their executable semantics

In the original distributed system a D-Clean coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence

of input and output channels. The coordination primitives return the result dataflow on the specified 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.

Here in the executable semantics version we define a D-Clean primitive using the `DExpr` algebraic type transforming a D-Clean expression into a `DFun` executable expression (the string information is needed only at the visualization).

```
// the type of a DClean expression, one function or a composition of functions
:: DExpr a b = F String (a → b)
             | D (DFun (Ch a) (Ch b))
```

The primitive can be either a Clean function (or a composition of functions), or D-Clean expression parameterized by Clean functions placed into concurrently evaluated and executed boxes. Semantically, a primitive is a state transition function, transforming the input state into an output state.

```
// every DClean expression is a state transition function
:: DFun a b     := a State → (b, State)
```

Here we make some abstractions of the real distributed world ignoring the details of box generation and specifying only the established communication channels.

Therefore, the description and the sanity checks of D-Clean programs are made more easily before programming in the real distributed system.

The communication channels are defined explicitly simply by numbering them. Two kind of channels can be used: single channels for one threaded communication, and multiple channels for the simulation of data transfer on several, parallel computational threads.

```
// single channel, used with simplified numbering
:: Ch a         := (Int,a)
// multiple channel, defined as a list of single channels
:: MCh a        := [Ch a]
```

2.1 *The `DStart` and `DStop` primitives*

The task of the `DStart` primitive is to start building up the distributed computation by generating the communication channels for the input dataflow of the distributed graph. It has no input channels, only output channels.

The `DStart` primitive will take its input function together with the input dataflow and starts the computation. The results are sent to the output channels.

Each D-Clean program contains at least one `DStart` primitive. The functional description using Clean functions is as follows:

```
DStart :: a (DFun a b) State → (b, State)
DStart a expr state = expr a state
```

The other coordination primitive which must be included in any D-Clean program is the `DStop` primitive. The task of this primitive is to terminate all the communication channels and save the result of the computation process. It has as many input channels as the function expression requires, but it has no output channels.

Each D-Clean program contains at least one DStop primitive, and it is the final element of a D-Clean program, i.e the last element of the process network. The function for the semantics of the primitive is given as follows:

```
DStop :: (a, State) → (a, State)
DStop (result, state) = (result, takeWorld state)
where
    takeWorld (n,t) = (n,reverse t)
```

In order to assure that in any executable D-Clean program both primitives are included, we constructed a wrapper called DExec. Therefore, DExec combines immediately the DStart and the DStop mandatory primitives into one encapsulation. It lifts the "normal" world of Clean into the distributed world of D-Clean, and it is parameterized by an input dataflow and the D-Clean expression `expr` to be tested.

```
DExec :: a (DFun a b) → (a, (b, State))
DExec a expr  = (a, DStop (DStart a expr (0,[])))
```

DExec together with the >>= combinator enables to write our D-Clean program simulation in monadic way (the >>= combinator renames the bind operation).

```
(>>=) infixl 1
(>>=) f g := f `bind` g
```

In the following we use mkFunBox for generating boxes corresponding to computation nodes:

```
mkFunBox :: String (a → b) →  DFun (Ch a) (Ch b) | toString b
```

As mentioned earlier, the boxes created by mkFunBox are running concurrently in the original distributed system. Here mkFunBox specifies that the primitive will be placed in the computational node of the generated box. The boxes are communicating via channels using well established protocols.

In the following the details of every D-Clean primitive will be given together with the executable signature.

2.2 The DApply primitive and its variants

The DApply primitive class enables to apply its parameter function(s) to the dataflow(s) of the distributed computation. Several versions are defined since they represent the core primitives of the D-Clean distributed system.

The simplest one is DApply used in a single threaded computation, i.e. with one function parameter applied to a dataflow of a single input channel.

```
DApply  :: String (a → b)  → DFun (Ch a)  (Ch b)  | toString b
DApply name fun = mkFunBox name fun
```

However, in most of the cases the DApply primitive is used when multiple threads are needed in the computation, illustrated in our executable semantics by the MCh type of the channels.

The multiple function application can be done in two variants. The first variant, DApplyN, is the most general one. It may apply different function expressions on different dataflows of separated computation threads.

The function sequence to be applied is given in a list of expressions, specified as `[DExpr a b]`. The primitive operates analogously to the `zip` function, i.e. it takes one function expression from the sequence of functions and applies it to its corresponding pair dataflow from the `inflows` list of inputs. Obviously the length of the two sequences should be equal in order to have proper function and dataflow matching. As the multiple channel input datatype suggests, the input dataflows are flowing on separated computational threads.

```
DApplyN :: [DExpr a b]  → DFun (MCh a) (MCh b) | toString b
DApplyN funs = handleDExpr funs
where
   handleDExpr :: [DExpr a b] (MCh a) State → (MCh b, State)| toString b
   handleDExpr             [] []      state = ([], state)
   handleDExpr             [] inflows state = abort "run-time error"
   handleDExpr [F name fun:funs] [i:is]  state
   # (result, state)    =   mkFunBox name fun i state
   # (results, state)   =   handleDExpr funs is state
   = ([result:results], state)
   handleDExpr [D dfun:funs] [i:is] state
   # (result,state)     =   dfun i state
   # (results,state)    =   handleDExpr funs is state
   = ([result:results], state)
```

If a thread is only taking the dataflow from the input channel transferring it to the output channel, without performing an operation on it, then the `id` function should be mentioned in the list of functions on the corresponding position of the computation thread.

The second variant, `DApply1` applies the same function parameter n times on different computation threads each one with its own dataflow, i.e. each channel operates on different dataflows.

```
DApply1 :: (DExpr a b)  → DFun (MCh a) (MCh b) | toString b
DApply1 dexpr
        = λinflows → DApplyN (repeatn (length inflows) dexpr) inflows
```

From the definition it can be observed, that it uses the most general variant `DApplyN` repeatedly applying the same `dexpr` on the input dataflows. However, this repetition is a simulation of concurrent function application on separate computation threads.

2.3 The `DApply` special cases

Derived from distributed application experiences, several special `DApply` primitive cases where handy, therefore we defined them as separated primitives with their own names. In the executable semantics version we could define them using one of the `DApply` variants.

The `DMap` utility primitive together with `DMap2` are basic distributed `map` operations, while `DReduce`, `DProduce` and `DFilter` are `DApply` utility primitives with some restrictions.

2.3.1 The `DMap` utility primitives

`DMap` is the distributed version of the well known standard `map` library function. The D-Clean variant is a computational node which applies the parameter expression to

every element of the incoming dataflow. The parameter function of a DMap must be an elementwise processable function.

It can be observed, that DMap is a special case of DApply, where a fun parameter is applied on a single thread (i.e. it has single channel input).

```
DMap :: (a → b) → DFun (Ch [a]) (Ch [b])  | toString b
DMap fun = DApply "DMap" (map fun)
```

For multiple channel input, the DMap2 primitive is defined. Due to its multiplicity property, it can be expressed easily using the DApply1 primitive with concurrent application of the fun parameter on n different threads with own dataflows on the channel input.

```
DMap2 :: (a → b) → DFun (MCh [a]) (MCh [b]) | toString b
DMap2 fun = DApply1 (F "Map2" (map fun))
```

2.3.2 The DApply utility primitives with restrictions

The following three utility primitives are special cases of DApply with some type restrictions for input dataflows.

DReduce is one of the utility primitives. A valid expression for DReduce has to reduce the dimension of the input dataflows, i.e. the input is a list of dataflows transformed into one dataflow as for output.

```
DReduce :: ([b] → a) → DFun (MCh [b]) (MCh a) | toString a
```

The opposite of the DReduce is the DProduce utility primitive, in which the DFun parameter expression has to increase the dimension of the output dataflows, it simply transforms the one input dataflow into a list of dataflows.

```
DProduce :: (a → [b]) → DFun (MCh a) (MCh [b]) | toString b
```

DFilter is a special case of DApply in a different way compared to the above primitives. It is filtering the input dataflow according to a filter (boolean) expression analogously to the filter Clean function.

```
DFilter :: (a → Bool) → DFun (MCh [a]) (MCh [a]) | toString a
```

2.4 The DDivide primitive

The DDivide primitive has a DFun expression as parameter, which splits the input dataflow into several parts according to a divider parameter function. After splitting, it broadcasts the input dataflows on different computation threads with their own computational tasks. The main type feature of a division is given in the type of the channel parameters: the input dataflow is transferred on a single channel, while the output will be transferred on a multiple channel. It can be observed, that all the output dataflows have the same type.

The DDivideD version determines dynamically the number of computation threads it needs to sparkle, and it transfers the dataflows to them.

```
DDivideD :: (a → [b]) → DFun (Ch a)  (MCh b) | toString b
```

`DDivideS` is a static divider. This primitive is called a static divider since the number of threads (n) is known at compilation time, and it is given as parameter to the dataflow divider function.

```
DDivideS :: (Int [a] → [[a]]) Int → DFun (Ch [a]) (MCh [a]) | toString a
```

2.5 The `DMerge` primitive

The counterpart of the `DDivide` is the `DMerge` primitive, which collects the input dataflows from input channels of a multiple channel, and builds up the only one output dataflow for the single channel of the output thread. All the input channels must have the same type.

```
DMerge  :: ([b] → a) → DFun (MCh b) (Ch a)   | toString a
```

The above two primitives are extremely used when distribution or collection of dataflows are needed in the distributed computations.

2.6 Primitive compositions

In the original D-Clean system we have designed composed language primitives as well, and they are defined in the executable semantics model too. A typical example is the `DLinear` primitive, for easier composition in pipeline of the primitives:

```
DLinear :: (Pipe a b) → DFun (Ch [a]) (Ch [b]) | toString b & toString a
```

The primitive is used especially in data driven skeletons, where the amount of data and the number of composed functions are determining the type of the problem implementation.

3 Vizualization

The visualization of the execution of the D-Clean programs in distributed system are very important when analyzing the results of the executions and when debugging the complex distributed programs.

The executable semantics provides a tool for understanding how the distributed computation is done, leaving out the middleware details: the generation of boxes for the computational nodes and channels for the communications. The distributed programs can be drawn and visualized graphically for testing the semantics of its execution before applying it in the real distributed system. For each language primitive we provided such visualization, which enables more flexible distributed program creation.

Let us consider the well known farm skeleton example. Here we illustrate the problem using our visualization method by computing in parallel simple tasks.

After a `DMap` operation is done on the input dataflow, the master computation node applies a static division. On each sparkled threads individual operations are done by worker nodes, after which the subresults are merged by the master node again.

```
Start w = dumpGraph (DExec myval myflow) w
where
myval = [1..10]
myflow i
   = DMap inc (0,i)
 >>= DDivideD (divide 3)
 >>= DApplyN [F "map inc" (map inc),
     D (λi → DMap inc i >>= DMap inc),
     F "id" id]
 >>= DMerge flatten
```

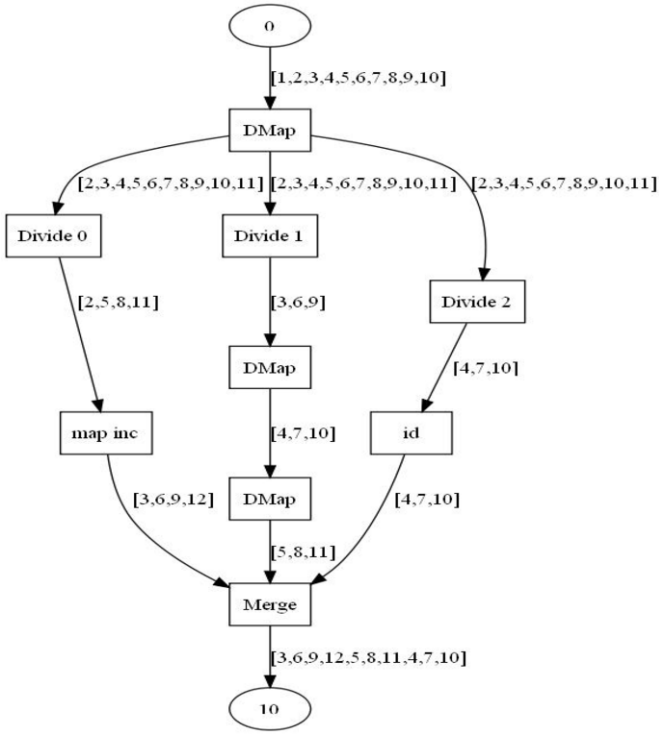

Fig. 1. Farm example visualization

In its visualization (see Figure 1.) we can observe how the input dataflow is divided into 3 parts between the workers, which perform their own individual tasks. At the end, the master collects the subresults into a final dataflow. More skeleton examples can be found in [19].

4 Some properties and transformations

Since channels behave in some sense similar to lists, we can apply a number of list-like transformations. These transformations can be used to change the operational behaviour of D-Clean programs as well as to introduce new channels.

For list we have equivalencies like for instance `map f o map g` ≡ `map (f o g)` (see [4] for more list properties). The denotational semantics of both language constructs are equal, which implies that for all functions `f` and `g` and for all argument lists these expressions produce the same result. Nevertheless, there is an operational differ-

ence since `map f o map g` produces an intermediate list which is omitted in `map (f o g)`. Hence, the later expression is expected to be operationally more efficient.

Similarly to this list transformation we can change the number of functions operating on channels by the following transformation:

```
DApply1 (F "f" f) >>= DApply1 (F "g" g) ≡ DApply1 (F "fg" (g o f))
```

This effect can also be achieved when we apply several different functions to the channels using the transformation:

```
DApplyN [F "f1" f1, .., F "fn" fn]
>>= DApplyN [F "g1" g1, .., F "gn" gn]
≡ DApplyN [F "fg1" (g1 o f1), .., F "fgn" (gn o fn)]
```

In a similar way we can transform combinations of `DApplyN` and `DApply` into a single application of `DApplyN`.

```
DApplyN [F "f1" f1, .., F "fn" fn]
>>= DApply1 (F "g" g)
≡ DApplyN [F "fg1" (g o f1), .., F "fgn" (gn o fn)]
```

and conversely we can have:

```
DApply1 (F "f" f)
>>= DApplyN [F "g1" g1, .., F "gn" gn]
≡ DApplyN [F "fg1" (g1 o f), .., F "fgn" (gn o f)]
```

The last transformation we show here covers the lifting of an ordinary list processing function in Clean to a channel processing construct over `n` channels in D-Clean.

```
map f list
```

≡

```
    DDivideD (divide n) (0, list)
>>= DApply1 (F "f" f)
>>= DMerge id
```

The transformations applied to a code should always preserve the semantics defined in section 2.

5 Related work

Several related works can be enumerated with different common aspects.

Earlier the parallel and distributed computations in Clean where reported in two PhD thesis: [11] and [18]. The first one tests Clean programs on parallel supercomputers, while the second is based on concurrent evaluation of parts of a function inside one computer using parallel function evaluation annotations. Our approach in the D-Clean system emphasizes the evaluation and coordination of separated Clean programs running on different computers of a cluster.

The executable semantics of `iTasks` introduced in Clean is present in the `iTasks` and `iData` property-testings using the model-based test tool G∀st of [12]), while fully automatic testing with functions as specifications are in [13]. `iTasks` are programmed

web-based coordinated by a centralized server application. The D-Clean programs are placed into computational boxes operating using Petri nets rules.

Several other functional languages have different type of parallel computations (e.g. Eden [1] and [2], or JoCaML [6]). The comparison of parallel functional languages of Haskell dialects are done in [14] and tested on clusters in [15]. They are based on different inherent process definitions, while our approach uses the separately distributed, individually executable programs.

Two research books are important for the literature of skeletons: [7] and [17]. The first one illustrates several functional programming skeleton applications, while the second one is more a theoretical approach of the topic. The main feature of the D-Clean system is the distributed skeleton application on clusters, not present in the above two books.

The parallel dataflow problems and a comparison for their implementation can be found in [10]. Our dataflow programming is more complex. The distributed environment uses two coordination languages for dataflow manipulations on two different layers of the system using middleware services.

The graphical functional dataflow language designed to visualize algorithms and their execution is present for example in NiMo (Nets In Motion) [5]. Our visualization for executable semantics is more focussing on explaining the details of the meaning of the D-Clean language primitives.

6 Conclusions

In this paper we had given an executable semantics for the D-Clean in more general way. The D-Clean extension of Clean is a coordination language used for distributed programming on clusters. The functional description of the primitives enables to understand what is computed and how the computation is done in the real distributed system of the D-Clean.

The graphical visualization can be used to depict the amount of work made in the parallel execution, where the computations are made in the nodes (boxes) and the communications on channels. Finally, we have formulated properties of the executable definition of the semantics for D-Clean primitives.

References

[1] Berthold, J: Explicit and Implicit Parallel Functional Programming Concepts and Implementation, PhD Thesis, Philipps Universität Marburg, 2008.

[2] 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.

[3] 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.

[4] Bird, R.: Introduction to functional programming using Haskell (second edition). Prentice Hall, 1998. ISBN 0-13-484346-0.

[5] Clerici, S., Zoltan, C., Prestigiacomo, G.: NiMoToons: a Totally Graphic Workbench for Program Tuning and Experimentation Electronic Notes in Theoretical Computer Science (ENTCS), Volume 258 Issue 1, December 2009, pp. 93-107.

[6] 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, , 2003, Springer, LNCS 2638, pp. 129-158.

[7] Hammond, K., Michaelson, G. (Eds.): *Research Directions in Parallel Functional Programming*, Springer Verlag, 1999.

[8] Horváth Z., Hernyák Z., and Zsók V.: Coordination language for distributed clean. *Acta Cybernetica*, 17(2):247–271, 2005.

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

[10] Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages, *ACM Computing Surveys* 36 (1), ACM Press, March 2004, pp. 1-34.

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

[12] Koopman, P. and Plasmeijer, R.: Automatic testing of higher order functions. In N. Kobayashi (ed.) *Proceedings of the 4th Asian Symposium on Programming Languages and Systems* APLAS'06, *LNCS* vol. 4279, pp. 148–164, Sydney, Australia, 8-10, Nov. 2006. Springer-Verlag.

[13] Koopman, P. and Plasmeijer, R.: Fully automatic testing with functions as specifications. In *Central European Functional Programming School*, *LNCS* vol. 4164, pp. 35–61, Budapest, Hungary, 4-16, July 2006. Springer-Verlag.

[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, *Higher-Order and Symbolic Computation* 16 (3), Kluwer Academic Publisher, September 2003, pp. 203-251.

[15] 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, Intellect, 2001, pp. 39-52.

[16] Plasmeijer, R. and van Eekelen, M.: *Concurrent Clean language report (version 2.0)*, Dec. 2001. `http://www.cs.ru.nl/ clean/`.

[17] Rabhi, F.A., Gorlatch, S. (Eds.): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, 2002.

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

[19] Zsók V., Hernyák Z., and Horváth, Z.: Designing distributed computational skeletons in D-Clean and D-Box. In *Central European Functional Programming School*, *LNCS* vol. 4164, pp. 223–256, 2006.

[20] Zsók V., Hernyák Z., and Horváth, Z.: Improving the Distributed Elementwise Processing Implementation in D-Clean. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools* SPLST 2007, Dobogókö, Hungary, June 14-16, 2007, pp. 256-264, Eötvös University Press, 2007.