Chapter 1

Design of Language Elements for Dynamic Distributed Computation of Clean Expressions on Clusters

Zoltán Hernyák¹, Zoltán Horváth², Viktória Zsók²

Abstract: Our aim is to express computations in the form of distributed process-networks, to develop and test parallel functional applications on PC clusters with Clean components using a middleware for the distributed communication and synchronization.

The core version of the D-Clean and the D-Box languages supporting the distributed computation of the Clean client programs were presented earlier. In this paper extensions to these languages are added in order to support dynamic behaviour, which is highly needed in order to make possible to write recursive and also embedded expressions.

The paper aims to define the requirements of such a distributed programming environment which supports dynamic loading and starting of functional components and to present API functions for the communication between the components.

Keywords: D.1 Programming Techniques: D.1.1 Applicative (Functional) Programming, D.1.3 Concurrent Programming.

¹Department of Information Technology, Eszterházy Károly College, e-mail: aroan@ektf.hu
²Department of Programming Languages and Compilers, Eötvös Loránd University, Hungary, e-mail: hz@inf.elte.hu, zsv@inf.elte.hu.

Supported by the Hungarian National Science Research Grant (OTKA) Grant Nr. T037742, by the Bolyai Research Scholarship and by the IKTA JGrid 2003-2004 Project.
Key Words and Phrases: distributed functional programming, coordination language, Clean, functional skeleton, dataflow.

1.1 INTRODUCTION

In a previously developed environment Clean functional client-programs could be interconnected via CORBA [2, 3]. We have implemented skeletons[6][8][19] in the functional language Clean using CORBA server objects referenced by the parameterized clients. This made possible the distribution of the processes containing functional components and an asynchronous communication.

It turned out that distributed evaluation of functions and the communication between clients needs high-level process description and control mechanism.

For this reason two control languages D-Clean and D-Box and their semantics were introduced [1], making easier and more elegant the parallel functional skeletal programming in a dataflow style. A DistStart expression of D-Clean program is mapped to a dataflow program, to a directed graph where data flows between functional program components [17]. A node of the process network is a coordination structure with an embedded Clean expression. The distributed computation patterns are identified and described by compositions of coordination elements. In the current paper we present additional new high-level language elements for the dynamic coordination of the evaluation of component functions in a distributed environment. We have summarized all the introduced coordination structures in the appendix A.

The figure 1.1 shows the different layers and components of the D-Clean environment. The D-Clean level hides the details of process network construction, ports and channels. D-Clean introduce a fixed set of coordinations elements, to build up networks in a structured way.

The D-Clean expressions are compiled to D-Box definitions. D-Box definitions provide a platform and middleware independent description of computation nodes, channels and protocols. The D-Box level has a fixed semantics used by the D-Box pre-compiler. This semantics contains a set of functions, which must be implemented by the actual application environment.

The D-Box language has a lower abstraction level. It supports the description of all the D-Clean structures, but in this language other (not usual) connections can be described as well. It is possible to generate D-Box definitions not only from a D-Clean expression but also by using a graphical programming environment [15].

The coordination structures are using channels for receiving the input data needed for the arguments of their function expressions, respectively for sending the output data of the results. Every channel is capable of carrying data elements of a specified type from one computational node to another one. The communication channels are represented by tuples. The base type TUP is used in the type description of the coordination structures. The structure of the type is $TUP = \langle \text{TypeDef}, \text{ChannelID} \rangle$, where TypeDef defines the type of the channel (this corresponds to the type of the elements transported by the channel) and ChannelID is a unique identification number of the communication channel.
The signature of the primitives using TUP types also suggests the semantics of the different D-Clean language primitives. Every primitive has as parameters a function expression (or a list of function expressions) and a TUP-list. The coordination primitives return lists of TUPs. The result type of an expression is a tuple of \( k \) components according to the signature of the \( k \) output channels. Each input channel carries one argument for the function expression.

The following D-Clean example demonstrates how to write a distributed sorting using the farm scheme.

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

SCHEME FARM comb solve divider n =
  DMerge comb DApply solve DDivideS divider n
```

### 1.2 DYNAMIC BEHAVIOUR

In the previous work the D-Clean language supported only static networks. This means that the D-Clean pre-compiler analyzes the D-Clean start expression, and builds up a connection graph with computation nodes as the nodes of the graph, and channels as the edges. This connection graph is a static construction with static channels between them, and static number of nodes.

We extend the language with new elements to express dynamic network behavior.

The D-Clean keyword DDivideS is a static divider which divides the input data list into \( N \) parts and splits them in to \( N \) directions to different computation nodes. The value of \( N \) must be known at pre-compiling time. More flexibility is needed for many functional skeletons (e.g. divide and conquer, torus, farm, etc.). A new DDivideN keyword is added to support these. DDivideN evaluates the value of \( N \) at run-time, starts output channels depending on the value of \( N \).
We permit a computational expression to be a composition of Clean and D-Clean expressions. The D-Clean part of the expression is called embedded expression.

For example the expression 'if(lengthOf data > 10000) (FARM DMerge merge-sort DApply qsort DDivideS circular 10) (qsort)' contains an embedded D-Clean expression. This means if more than 10000 data elements arrive, use a farm to sort them - otherwise sort them locally. The farm will be launched at run-time only, and only when it is needed.

1.3 APPLICATION ENVIRONMENT

Dynamic behavior means that some computational nodes and channels are loaded and started at run-time when they are needed. The nodes must discover run-time the dynamically started components of the project (other nodes and channels).

As was presented in the introduction, the only restriction for the current Application Environment is to implement the functions required by the D-Box semantics. The following concepts and operations should be supported:

- **Computation Node** (shortly node): pre-compiling one D-Box definition produces a Clean program which implements the expression to be calculated. This is referred as computation node. It receives input data elements from other programs, makes some computation on it, and sends the results to other programs through channels. Each node holds an unique BoxID (string) identifying its box definition.

- **Channel**: A channel is a queue with a bounded storage capacity. One node can send data elements through a channel to another node. Each channel has its own data type (ChannelType). Elements of this type can be transported through that channel. At run-time the same kind of channels can be instantiated many times. The different instances are identified by unique run-time generated ChannelID-s.

- **Sub-graph**: a set of computational nodes, which are connected to each other. The sub-graph has an input (entry) node. An external node can send data to the sub-graph through this node. The sub-graph makes the computation on these data elements, then the output (exit) node of the sub-graph sends the result to its output channels. The external node can read these channels to retrieve the result.

  A sub-graph can be referenced by a SubgraphID. The same kind of sub-graphs can be instantiated several times at run-time. The different instances can be identified by their unique ThreadID-s.

- **ProjectID**: A D-Clean expression is a project from the point of view of the D-Clean compiler. Compiling it produces a lot of D-Box definitions with the same ProjectID. Compiling the D-Box definitions we get channels and computational nodes. They hold the original project identifier to separate them.
from components generated from other projects. The AE must know which code belongs to which project separating them from each-other.

The proposed application environment must support the following operations:

- **start a channel**: This operation starts a new channel instance. The request is parameterized by ProjectID and ChannelType, the result of this request must be an unique ChannelID.

- **start a sub-graph**: It starts all the computational nodes of the given sub-graph. This operation is parameterized by the ProjectID and the SubGraphID. The result of this operation must be an unique ThreadID.

- **find a channel**: find an already started channel. The request is parameterized by ProjectID and ChannelID.

- **query of the channels by box-id**: query the channel id-s of a computational node (input and output channels). This operation is parameterized by the ProjectID, ThreadID, BoxID, and the kind of channels (input or output). The result of this request must be a list of channel id-s.

- **query of the channels by thread-id**: query the channel id-s of a whole sub-graph (which usually consists of more than one node). This request is parameterized by ProjectID, ThreadID and the kind of channels (input or output). The result of this request must be a list of channel id-s.

Our environment is shown in figure 1.2.

A **Local Communicator** (LocalComm) defines the local access point of the application environment. This provides API functions for the computation nodes, requests and queries can be sent to it. The Local Communicator usually forwards the API request to the other elements of the environment which implements the functionality. There are several reasons to use the Local Communicator. First to provide a local, transparent interface to the other environment elements (servers). The location of the Local Communicator doesn’t need to be discovered by the computational nodes and channels because it is on the local computer on a fixed port. Second: a Local Communicator can be used to download and start a node or channel code on the supervised hardware elements. Third: a Local Communicator knows what kind of codes (platform restrictions) can be started on the supervised platform - and can inform the AppStarter on its possibilities.

A **Code Library** CodeLib server holds the binary codes of the computational nodes and channels - and acts as a kind of file server.

A **Registration Center** (RegCenter) is another server, where the nodes and channels may register themselves for being discoverable for other nodes. This server acts as a kind of name service, but provides the creation of new channel id-s and thread id-s.

An **Application Starter** (AppStarter) server is a scheduler which can instruct the supervised nodes to start a new channel or node instance.
When somebody calls the AppStarter to start a new channel, it queries the CodeLib servers to find where are the binary codes of the channel, then the Local Communicators to find the best place for the new channel. The CodeLib servers and LocalComm-s are registered in the RegistCenter servers. The AppStarter prefers the location of the channel reader node as the location of the newly started channel. If it is impossible, or that location is heavy loaded, selects another location. Starting a node, or a set of nodes (sub-graph) uses a very similar way.

The Application Environment consists of several parts. These parts work together. This environment is abbreviated as AE in the remaining part of this document.

1.4 THE EXTENSIONS OF D-BOX LANGUAGE FOR SUPPORTING DYNAMIC BEHAVIOR

The syntax of the D-Box language is defined in Appendix. The main point is that the D-Box language is a platform and middleware independent description language to define a computational node. It describes the expression - which makes the computation, and the input and output channels to transport the data to and from.

A general form of a new D-Box definition, including the new term SubGraphID is:

BOX "BoxID"
The **SubGraphID** classifies the box definitions, defines to which subgraph belongs the node. The **input channel definitions** contains the descriptions of the input channels (types, identifiers). The **input protocol** specifies how to read and collect the incoming data elements from the given channels. Similar specification is given at the output channels and protocol.

**auto**: For supporting of starting dynamic channels a new keyword **auto** is added to replace the role of the static channel id. This is an example of using auto-starting channels, a node for sorting the incoming **int** list. The code of this node uses auto-started input and output channels, so multiple instances of this code can be started at run-time.

```plaintext
BOX Box_01 // QSORT box
3,
{ { [Int], auto }, join1 }, // use an auto-started "Int" input channel
{ { [Int] }, qsort, ( [Int] ) }, // qsort the input data elements
{ { [Int], auto }, split1 } // use an auto-started "Int" output channel
```

The **join1** protocol maps one input channel to one input argument of the expression. The **split1** protocol splits the output arguments, one argument to one output channel.

The draft of the generated code (according to the D-Box -> Clean semantics) is the following:

```plaintext
Start w
    // starts an "Int"-type channel dynamically
    # (chan_1,w) = StartChannel "Int"
    // reads the input data from that channel
    # (input_data,w) = Join1 chan_1 w
    // apply the expression
    # result = qsort input_data
    // start a new Int channel
    # (outp_chan,w) = StartChannel "Int"
    // sends the result to the new channel
    # w = Split1 outp_chan result
...
```

The **StartChannel** is an element of the interface library. Its role is to start a new channel on the supervised cluster which is capable of transporting the data elements with the specified type. The result of this function must be a channel identifier which can be used later to identify this channel.

**startGraph**: a set of boxes defines a sub-graph. The simplest sub-graph consists of one box only. Let us suppose the sub-graph #3 (it is the sub-graph id of the **Box_01** node in the previous example) consists of only one qsort box. In the following example this box starts an instance of this qsort box run-time, dynamically, implied by the keyword **startGraph**.
BOX Box_02 // QSORT starter {
  ,
  { ([Int],auto), join1 },
  { ([Int]), multiplyWith 2, ([Int]) },
  { (startGraph 3 1 ([Int])), split1 }
}

Start w

... // start all the nodes of sub-graph #3
# (threadID,w) = startSubGraph 3 1 w
// query the input channels of the given thread
# (out_chans,w) = getChannels threadID inpCHANNELS w
// sends the result to the new channel
# w = Split1 outp_chans result
...

The startSubGraph is an interface function too. In this example it means that all the nodes in the #3 sub-graph must be started one single time. This function gives back its unique threadID. Then the input channels of this new thread must be discovered using the getChannels interface function, which retrieves the necessary channel identifiers.

**connBox**: if a node wants to read the output data from the dynamically started qsort box, it can use the **connBox** to find and query its output channels:

This example supposes that the the dynamically started #3 sub-graph got the threadID #13. In a real situation the threadID will be known only at run-time only, so instead of using the constant #13 - an expression must be used which determines the threadID at run-time with the help of the AE. Since each sub-graph gets a new threadID, and is registered in the RegCenter with its additional parameters (which box has started this sub-graph, for what reason was it started) - this threadID can be retrieved run-time. In this example we omit this steps for sake of simplicity.

BOX Box_03 // QSORT reader {
  ,
  { connBox 13 "Box_01" ([Int]), join1 },
  { ([Int]), multiplyWith -1, ([Int]) },
  { ([Int],auto), split1 }
}

Start w

... # (inp_chans,w) = getChannels 13 outCHANNELS w
// reads the sorted data element
# (inp_data_list,w) = Join1 inp_chans w
...

**autoConnBox**: This special keyword is used by the DMerge boxes which can collect the results of threads started by a DDivideN box.

To use the dynamic behavior on the D-Clean language, we add an additional keyword: DDivideN.

This keyword defines a dynamic divider node. The fun_expr is the divider expression. It receives the input data list from the input channels, and divides the inout list into N parts. The number of parts (N) defines how many different
At each output direction the same kind of sub-graph (set of computation nodes) interprets and processes the outgoing data elements. The value of $N$ does not need to be known at pre-compiling time, but an $AE$ is needed to instantiate and start the sub-graph $N$ times (see figure 1.3).

1.5 EXAMPLE

In the following example we multiply the same $K \times M$ matrix with several vectors of length $K$. In the solution we use $M$ threads, each one has the task to multiply with one column and generate one element of the result vector.

The syntax and the short semantics of DMerge, DApply, DDivideS and other keywords are presented in the Appendix.

```
DistrStart = DStop saver DMerge vectorize DApply [multiply column_1, multiply column_2, ..., multiply column_K] DDivideS repeater K DStart vector_generator\nwhere
  K = 4
  saver = saveToFile "result.dat"
  vectorize = id
  column_1 = getColumn matrix 1
  column_2 = getColumn matrix 2
  ...
  column_K = getColumn matrix K
  repeater = [inp_0,inp_0,inp_0,...,inp_0] // K times
vector_generator:: [[Int]] // generates the vectors
ggetColumn:: [[Int]] Int -> [Int] // gets the Nth column
multiply::[Int] [Int] -> Int
```

This computation node generates several vectors, and sends them to a statically allocated channel with a static (#1) channel id.

```
BOX BoxID_1000 // DStart
| 1,
```
This box reads the input channel #1, and repeats one incoming vector 4 times, and sends the 4 vectors to 4 directions to channels #11, #12, #13, #14.

**Box BoxID_1001 // DivideS**

```plaintext
1,
{ ( [Int],1 ), join1 }, // reads the static channel #1
{ ( [Int] ), repeater, ( { [Int] } ) }, // repeats the vector 4 times
{ ( [Int],[11],[Int],[12],[Int],[13],[Int],[14] ), split1 } // sending
```

This box reads one vector from the channel #11, gets the 1st column from the constant matrix, multiply them, and sends the produced int number to channel #21. Similar boxes are needed to read the #12, #13, #14 channels, to multiply the vectors by the different columns, and send the output to channel #22, #23, #24.

**Box BoxID_1002 // DApply multiply column_1**

```plaintext
1,
{ ( [Int],11 ), join1 }, // reads the #11 channel
{ ( [Int] ), multiply (getColumn matrix 1), ( Int ) }, // multiply
{ ( Int,21), split1 } // sends the result to #21
```

This box merges the result of the four vector multiplication, and creates a vector using these int numbers. To do this, reads the four output channels. Since the `join1` merges the incoming data elements from the channels, the expression to creating the vector is the `identity` function.

**Box BoxID_1006 // DMerge vectorize**

```plaintext
1,
{ ( [Int],[21],[Int],[22],[Int],[23],[Int],[24] ), join1 }, // 4 input channels
{ ( [Int] ), id, ( [Int] ) }, // restore the vector form
{ ( [Int],[31] ), split1 } // send the vector to channel #31
```

This box reads the output of the merging box and saves the vectors one by one to a disk file.

**Box BoxID_1007 // DStop saveToFile "result.dat"**

```plaintext
1,
{ ( [Int],[31] ), join1 }, // reads channel #31
{ ( [Int],World ), saveToFile "result.dat", ( null ) }, // save
{ ( null ), memory } // no output channel
```

On figure 1.4 this project can be seen using the graphical D-Box development environment[15].

### 1.6 EMBEDDED D-CLEAN EXPRESSIONS

Every D-Clean keyword have at least one parameter: a `fun_expr`. This expression is usually a standalone, pure Clean expression, but may contain embedded D-Clean expressions as well.
FIGURE 1.4. Matrix-multiplyer example

The following example determines the maximum of a large number of data elements. The expression is a composition of two expressions. An embedded D-Clean expression (‘FARM merge max divide’) pre-calculates the maximum using the FARM scheme. The final maximum is selected by the \texttt{max} function of the small amount of sub-maximums.

\[ \ldots \text{DApplay (max FARM merge max divide)} \ldots \]

The complex expression can be converted into...

\[ \ldots \text{DApplay (max (STUB sub\_graph\_id))} \]

The embedded D-Clean expression must be compiled separately into a sub-graph, and its identifier is the \texttt{sub\_graph\_id}.

A sub-graph stub function is the local part of a sub-graph, which hides the implementation details in the generated source code. A stub is generated and included into a node source code if it contains an embedded D-Clean expression. Actually, the embedded D-Clean expression is replaced by a stub function by the D-Clean to D-Box pre-compiler. The D-Box pre-compiler generates the source code of the stub function according to the actual input and output channels of the replaced sub-graph.

Using stub functions we have the typical consumer-producer problem. It receives its results from, and sends the arguments back to the sub-graph. The data sent to the sub-graph flows on the channels from one node to the next until it comes to the output channels of the last node. Every channel has a limited capacity. By a chain reaction every node may be stopped.
In the right way the stub starts a separate thread to send the data to the sub-graph, and the main thread reads the outgoing data from the sub-graph. It avoids the channels to reach the ‘full’ state.

The draft of the stub code looks like as the following:

```
STUB x y w
   // start the ‘nn’ sub-graph 1 times
   # (threadID,w) = startGraph nn 1 w
   // and gets the input channels of that
   # (inpChannels,w) = getChannels threadID inpCHANNELS w
   // start a new thread to send the arguments
   # w = startNewThread (Split1 inpChannels x y ) w
   // gets the output channels of the given thread
   # (outpChannels,w) = getChannels threadID outCHANNELS w
   # (result,w) = join1 outpChannels w
   = result
```

Since the sub-graph has only one entry node - the number of its input channels defines the number of the arguments of the stub function. Similarly, the output channels of the exit node of the sub-graph defines the type of the result of the stub. So, when the sub-graph id is known, the code of the stub function can be generated.

### 1.7 CURRENT STATE AND FUTURE WORK

Currently a basic application environment is developed in Microsoft.NET using the ICE as the middleware. A D-Box compiler and the skeletons is in testing phase to work with this AE.

A prototype of the GUI of D-Box is implemented. The D-Clean → D-Box compiler is a future work.

### 1.8 RELATED WORKS:

- PMLS and GpH are implicit parallel extensions of ML and Haskell respectively [16], on the other hand D-Clean uses explicit control structures.

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

- Eden [12][11] extends Haskell to explicitly define parallel computation. Eden program consists 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, 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 easily into distributed applications.

- Nimo [18] 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 are allowed Clean expressions allowing full power of functional programming at node level. Nimo does not support distributed computing, concurrent execution is supported only.

- JoCaml is an extension of Objective Caml with primitives for network-transparent distributed and mobile programming [20] based on the join-calculus model instead of a pure data flow approach.

1.9 CONCLUSION

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 expressions defining dynamic process networks.

The extension of the D-Box and D-Clean language adds the dynamic behavior to the coordination languages. Using these extensions more powerful functional skeletons and more complex applications can be defined in an easy way.

The implementation is based on a multi-level, layered, multi-paradigm environment. We can use any standard 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 functions, by types and by data.

REFERENCES


Appendix A - D-Clean language reference

- **DStart fun expr :: [TUP]** starts the distributed computation.
- **DStop fun expr [TUP] :: ()** receives and saves the result of the computation.
- **DApply fun expr :: [TUP] -> [TUP]** applies the same function expression \(n\) times on \(n \times k\) channels.
- **DApply [fun expr] :: [TUP] -> [TUP]** applies different function expressions on the input channel list.
- **DFilter fun expr :: [TUP]** filters the elements of the input channels using a boolean function.
- **DFilter [fun expr] :: [TUP]** filters the elements of the input channels using boolean functions.
- **DMap fun expr :: [TUP]** applies an elementwise processable function of type \(a \rightarrow b\) on channels.
- **DMap [fun expr] :: [TUP]** applies a list of elementwise processable function of type \([a] \rightarrow [b]\).
- **DReduce fun expr :: [TUP]** applies a function of type \([a] \rightarrow b\).
- **DReduce [fun expr] :: [TUP]** applies a list of functions of type \(a \rightarrow [b]\).
- **DProduce fun expr :: [TUP]** applies a function of type \(a \rightarrow [b]\).
- **DProduce [fun expr] :: [TUP]** applies a list of functions of type \(a \rightarrow [b]\).
- **DDivideS fun expr n :: [TUP]** splits the input data list into \(n\) parts and broadcasts them to \(n\) computational nodes.
- **DDivideN fun expr :: [TUP]** is the dynamic version of **DDivideS** where the number of threads \(N\) is calculated at run-time.
- **DMerge fun expr :: [TUP]** collects the input sublists from channels and builds up the output input data list.
- **DLinear [fun expr] :: [TUP]** simplifies the definition of the pipeline computation graph where the nodes are connected in a linear way.
- **DLinear [expr1, expr2, ..., exprk] [TUP]** is equivalent with the following composition **DMap expr1** ... **DMap exprk**, **DMap expr1**, ..., **DMap exprk** [TUP].
- **DDivideN fun expr :: [TUP]** is the dynamic version of **DDivideS** where the number of threads \(N\) is calculated at run-time.

Appendix B - D-Clean syntax reference

```plaintext
langle DISTART_RULE
  := "DStart" := (DEXPR)
  (DEXPR)
  := (DCONTROL) | (SCHEME_NAME) { (act_param) }*
(DCONTROL)
  := (DStartUSE) | (DStopUSE) | (DMapUSE) | (DDivideSUSE) | (DDivideNUSE) | (DMergeUSE) | (DLinearUSE)
(SCHEME_DEF)
  := "SCHEME" (SCHEME_NAME) { (formal_param) }*
  := "=" (DEXPR) |
(SCHEME_NAME)
  := { (UpCaseLetter) }*
(act_param)
  := (fun_expr)
  := (clean_expr) | ("" (DEXPR) "") | (fun_expr) (fun_expr)
(formal_param)
  := (identifier)
(DStartUSE)
  := "DStart" (act_param)
(DStopUSE)
  := "DStop" (act_param)
(DDivideSUSE)
  := "DDivideS" (act_param) (number)
(DDivideNUSE)
  := "DDivideN" (fun_expr)
(DMergeUSE)
  := "DMerge" (act_param)
(DMapUSE)
  := (SimpleDMap_DEF) | (MultiDMap_DEF)
```

15
SimpleDMapUSE ::= (DApply Variations) (act param)
(MultiDMapUSE ::= (DApply Variations) "[" (act param) "]"
(DApplyVariations) ::= "DApply" | "DMap" | "DReduce" | "DProduce" | "DFilter"
(DLinearUSE ::= "DLinear" "[" (act param) "]"

UpCaseLetter ::= "A" | "B" | "C" | "D" | ... | "Z"

Appendix C - D-Box syntax reference

BOXDEF ::= "BOX" (BoxID) "[" (SubGraphID) ""
          (InpProt) ""]" (ExpressionDef) ""]" (OutProt) "]"

ExpressionDef ::= "(" (TypeDef) ")" | "null"

Expression ::= (StubExpr) | (CleanFv) | (Expression) (Expression)

StubExpr ::= "STUB" (SubGraphID) ""

BoxID ::= (string)

InpProt ::= "[" (InpChannelDefs) "null" ""]" (InpProtMain) ""]"

InpProtMain ::= "join1" | "joink" | "memory"

InpChannelDefs ::= (Iway1) | (Iway2)

Iway1 ::= (IChannelDef) | (connBoxDef) | (Iway1), (Iway1)

Iway2 ::= (autoConnBoxDef)

(IChannelID) ::= (Number) | "auto"

OutProt ::= "[" (OutChannelDefs) "null" ""]" (OutProtMain) ""]"

OutChannelDefs ::= (Oway1) | (Oway2)

Oway1 ::= (OChannelDef) | (connBoxDef) | (Oway1), (Oway1)

Oway2 ::= (startGraphDef)

(startGraphDef) ::= "startGraph" (subGraphID) (count_expr)

(count_expr) ::= (Number) | (fun_expr)

(connBoxDef) ::= "connBox" (ThreadId) (BoxID) "[" (TypeDef) "]"

(count_expr) ::= (Number) | (fun_expr)

(subGraphID) ::= (Number) | (fun_expr)

(OChannelDef) ::= "[" (TypeDef) "]" (OChannelID) ""]"

(OChannelID) ::= (Number) | "auto"

(OutProtMain) ::= "split1" | "splitk" | "memory"

(TypeDef) ::= (TypeName) | "[" (TypeName) "]" | "[[" (TypeName) "]]"

(TypeName) ::= (UpCaseLetter) | (LocCaseLetter) | (UpCaseLetter) | (Digit)

(ThreadID) ::= (Number) | (fun_expr)