*Annales Univ. Sci. Budapest., Sect. Comp. 37 (2012) 19–46*

# RAPID PROTOTYPING FOR DISTRIBUTED D-CLEAN USING C++ TEMPLATES

**Viktória Zsók and Zoltán Porkoláb**

(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received January 2, 2012; revised February 23, 2012; accepted March 6, 2012)

**Abstract.** Earlier we have designed two coordination languages, D-Clean and D-Box for high-level process description and communication coordination of functional programs distributed over a cluster. D-Clean is the high level coordination language for functional distributed computations. The language coordinates the pure functional computational nodes required by language primitives, and it controls the dataflow in a distributed process-network. In order to achieve parallel features, D-Clean extends the lazy functional programming language Clean with new language primitives. Every D-Clean construct generates a D-Box expression. D-Box is an intermediate level language and describes in details the computational nodes hiding the low level implementation details and enabling direct control over the process-network. Practical experiences of the two language usage showed the difficulties of distributed program development, especially in testing and debugging. This paper aims to provide software comprehension application for a better way of understanding and utilizing the D-Clean language. Here we provide a new modeling approach of the coordination language elements and a new view of the D-Clean distributed system behaviour using C++ templates. The strong type system of C++ templates guarantees the correctness of the model. Using templates we can achieve impressive efficiency by avoiding run-time overhead.

*Key words and phrases:* Distributed computation, skeletons, templates.
*2010 Mathematics Subject Classification:* 68N18, 68N19.
*1998 CR Categories and Descriptors:* D.3.3.
The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).
https://doi.org/10.71352/ac.37.019

## 1. Introduction

The D-Clean distributed environment is a multiparadigm programming tool for distributed programs based on the Clean pure functional language. The system uses two coordination languages to provide language primitives for the development of applications with client-programs written in functional programming languages. The syntax and informal semantics of them are described in [18, 19], while the mappings from D-Clean expressions to D-Box expressions are in [7].

The D-Clean language consists of a minimal set of language primitives with several advantages. Typical parallel programming algorithms can be implemented elegantly using subproblems described in functional programming style. The users can test their parallel and functional programming knowledge in a multiparadigm environment. The set of the D-Clean primitives is minimal and easy to use. However, debugging distributed programs can be a very difficult task in a cluster environment. We have implemented in [20] a Clean executable version of the sophisticated D-Clean primitives to provide an easier comprehension tool for D-Clean program developers.

This paper aims to present a new modeling approach of the distributed and parallel functional programming examples tested now in C++ in templated executable environment with distributed language elements, boxes, channels and messages. The D-Clean programs are presented in a tutorial style starting from simple skeleton examples to more complex ones, while the new C++ templates are explained in details.

Our main motivation is to verify the semantics of the primitives by program execution. The template examples are handy to understand and to compare the semantics of the distributed system with the executable version one. We provide a framework as software comprehension tool for better understanding the role of the language primitives.

The distributed evaluation of functional programs on a cluster and the communication between computational nodes require high-level coordination mechanism for the description of processes. Therefore, the D-Clean coordination language has a higher level control role, while D-Box has a lower abstraction level. The coordination of functions is expressed in the form of distributed process-networks. D-Clean primitives control the dataflow on the channels of the process-network.

A process network 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 process networks are defined by skeletons, algorithmic schemes parameterized by functions, types and data.

D-Clean is compiled into an intermediate level language D-Box, based on Petri nets [2]. D-Box is designed for the description of the computational nodes implemented as pure Clean programs. It uses middleware services for asynchronous communication. D-Box expressions hide implementation details and enable direct control over the computation nodes. The language defines input and output protocols for the communications via channels.

In functional programming, skeletons are higher order functions [4]. In D-Clean a skeleton is a high level, abstract definition of the distributed computation, and it is parameterized by functions, types and data. A scheme is actually identified and described by compositions of coordination primitives. The coordination primitives have the role of manipulating and controlling the pure functional components written in Clean. The middleware services (for the technical details see the second part of [19]) enable the distributed communication between the computational nodes.

Here in this paper we define in templated style the algorithmic skeletons of parallel programming. The tested D-Clean programs are using the executable definitions of the complex primitives of the distributed system (see the basic list of primitives in Appendix A). The template description of the D-Clean distributed model enables expressing the semantics of the D-Clean programs using another programming paradigm than functional.

This templated model simulates in one sequential program the distribution of tasks of the original D-Clean system. This new way of description demonstrates the expressiveness and ease of use of the two coordination languages. It leaves out the details of the distribution and communication in the original environment. The programmer can do the sanity check of the distributed program, can verify and test what to expect when the application is running in the real distributed system.

The strong type system of C++ templates guarantees the correctness of the model. Using templates we can avoid run-time overhead achieving impressive efficiency. The provided framework enables for users familiarized with object-oriented programming to test skeletons in specific, templated way.

The paper is organized as follows: we present first the D-Clean language via examples, we explain the main components of the D-Clean using C++ template classes in the model section, we enumerate related works, and finally we conclude in the last section outlining further research plans too.

## 2. The D-Clean language primitives

D-Clean is a Clean-like language for distributed computations on clusters, and it consists of a relatively small number of coordination language primi-

tives. Here we define in monadic style the algorithmic skeletons of parallel programming. The tested D-Clean programs are using the executable Clean definitions for the complex primitives of the distributed system (see the basic list of primitives in the appendix).

In functional programming skeletons are higher order functions [4]. We call a D-Clean skeleton scheme. A scheme is a high level, abstract definition of the distributed computation, and it is parameterized by functions, type and data. Schemes are actually identified and described by compositions of coordination primitives. The coordination primitives have the role of manipulating and controlling the pure functional components written in Clean, which however express the pure computational aspects. The middleware services (for the technical details see the second part of [19]) enable the distributed communication between the computational nodes.

A coordination primitive uses channels for receiving the input data required for the arguments of their function expressions. The results of the function expressions 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. In the executable version of the D-Clean system, we made an abstraction of channels by numbering them and visualizing them by arrows.

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 transfer their results to 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. For a more detailed D-Clean language description see [7].

The executable semantics version of the D-Clean system is easier to understand, and it is used for testing the program before running it on a cluster. This system exploits the fact that the D-Clean coordination primitives can be composed in a way similar to functions in a functional programming style, using a very easy signature, the primitives were given in executable way in a new system defined in [20]. This section presents the D-Clean examples and their visualization using this latter system.

The D-Clean language primitives have the same semantics in the two execution ways. However, due to abstractions from the real distributed environment, some of the distributed D-Clean properties are difficult to express in the system version designed especially for the testing semantics in executable way.

This section presents the D-Clean implementation of some useful skeletons. The figures in this section are generated by the execution of the D-Clean programs in the executable semantics' system, developed for testing and graphical visualizing.

The examples of this section usually have a generated input dataflow (for simplicity, here usually a list of integers is generated). The generator can easily be extended to any other data structure.

### 2.1. The `DExec` encapsulation

The task of the `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 `DStart` primitive will take the input given by the `generator` function and it then starts the computation. The results of the `generator` are sent to the output channels. Each D-Clean program contains at least one `DStart` primitive.

Another coordination primitive which must be included in any D-Clean program is the `DStop` primitive. If 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 the `DStop` 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.

The executable system encapsulates these two mandatory primitives in the `DExec` wrapper and enables writing directly in it the scheme to be tested.

In the following, we present some of the earlier defined schemes in [19] together with their code-snippets developed here for the system version of the executable semantics, and we generate their graphical visualizations too.

The D-Clean primitives (listed also in Appendix A, except the `DStart` and `DStop` due to the above encapsulation) are composed using the `>>=` combinator as the monadic `bind` operation.

### 2.2. Apply operations

Our first simple example illustrates how to use the distributed version of the well known standard `map` library function. The D-Clean variant of the `map` function is a simple computational scheme, and it applies the `DMap` primitive designed for the purpose of elementwise processing of the incoming data elements.

The parameter function of the `DMap` primitive can be any Clean elementwise function (see more details in [8]). Here, the `DMap` example computes the squares of the elements of a list of integers. `DExec` is written in monadic style, the parameters are lifted, the expression to be calculated is `myflow`, the dataflow to which is applied is `myval`, the node numbering necessary for the visualization is starting from 0.

```
generator :: [Int]
generator = [1, 2, 5, 7, 12, 14, 17, 25, 30, 45]

square :: Int → Int
square x = x^2

Start w = dumpGraph (DExec myval myflow) w
where
    myval = generator
    myflow i
      = DMap square (0,i)
```

The execution generates the following figure:

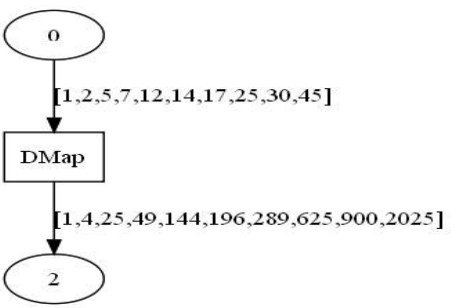


*Figure* 1. The `DMap` primitive

Since the coordination primitives can be composed, D-Clean skeletons can be written as compositions of D-Clean language primitives. Suppose we define a D-Clean scheme which is composed of three `DMap` elements given as in the following code:

```
generator :: [Int]
generator = [1, 2, 3, 5, 1, 4, 3, 6, 3, 5]

f1 :: Int → Int
f1 x = ((^) 2) x
f2 :: Int → Int
f2 x = x*x
f3 :: Int → Int
f3 x = ((+) 1) x

Start w = dumpGraph (DExec myval myflow_map3) w
where
    myval = generator
    myflow_map3 i
```

```
=   DMap f1 (0, i)
>>= DMap f2
>>= DMap f3
```

The computation scheme is visualized as in the Figure 2.

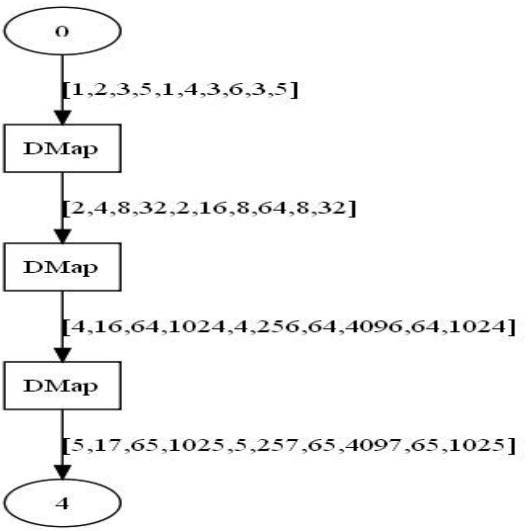


*Figure* 2. The `Map3` skeleton

In the following we will ignore the `Start` expression writing in the code-snippet only the applied primitive composition (given in monadic way). The input dataflow is considered the list given by the `generator`, while the D-Clean scheme is given in the `myflow` expression.

Similar schemes can be written in a more general way, using the `DApply` coordination primitive.

```
generator :: [[Int]]
generator = [[11, 223, 445, 21, 5], [5, 88, 7, 6, 3]]

mfold :: [[Int]]→[Int]
mfold data = map total data

total :: [Int] → Int
total x = foldr (+) 0 x
```

For example, by using the above `total` function, which computes the sum of the elements of a sublist, the `DApply` primitive can be used as follows.

```
myval = generator
myflow_apply i
  = DApply "mfold" mfold (0, i)
```

`DApply` applies the same `mfold` function to the two sub-lists on different computational threads. Contrary to the other primitives, in the case of `DApply` we use a string parameter to visualize explicitly the function performed by primitive. It can also be observed that `DMap` is a special case of `DApply`.

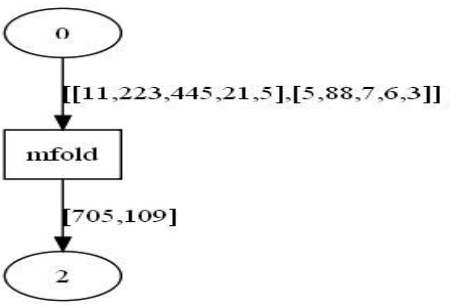


*Figure* 3. The `DApply` primitive

The second variant, `DApplyN` applies different function expressions, which are given in a function sequence, to different input dataflows. If the function sequence contains an identity function, then the data received via the corresponding channel is forwarded directly to the output channel and afterwards to the next node. `DApply1` applies $n$ times the same function to $n$ different threads.

```
generator :: [Int]
generator = [1, 2, 3, 6, 4, 5, 7, 6, 8]

f1 :: [Int] → [Int]
f1 x = map ((^) 2) x
f2 :: [Int] → [Int]
f2 x = map f x
f3 :: [Int] → [Int]
f3 x = map ((+) 1) x
```

Analogously to the `Map3` scheme, a composition of `DApply` primitives can be written.

```
myval = generator
myflow_a3 i
  = (DApply "f1" f1) (0, i)
  >>= (DApply "f2" f2)
  >>= (DApply "f3" f3)
```

Figure 4 illustrates the above example similarly to Figure 2.

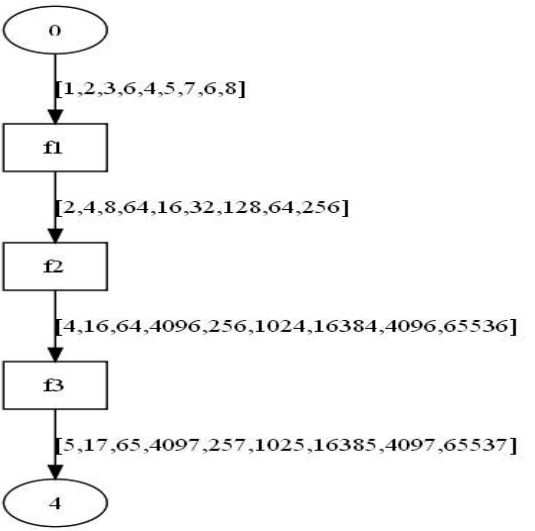


*Figure* 4. The `DApply3` skeleton

Some computational skeletons have similar functionality, i.e compute the same result, but using different coordination primitives and function parameters. Since the `DMap` coordination primitive is a special case of the `DApply` primitive, the `Map3` can be also considered a special case of the `Apply3` scheme. Choosing carefully the function parameters, the same computational skeleton can be expressed by different compositions of basic primitives.

### 2.3. Farm skeleton versions

The farm skeleton divides the original task into subtasks, computes the subresults and builds up the final results from the subresults (see Figure 5).

The skeleton uses two more D-Clean primitives: `DDivideD` for dividing the input into $n$ parts and `DMerge` for merging inputs. `DDivideD` splits the input dataflow into $n$ parts and broadcasts them to $n$ computational nodes (the value of $n$ must be known at compile time). `DMerge` collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type.

```
generator :: [Int]
generator = [1, 9, 4, 6, 2, 8, 5, 3, 10, 7]

combine_lists :: [[Int]] → [Int]
combine_lists [] = []
```

```
combine_lists [x:xs] = merge x (combine_lists xs)

divide :: Int [Int] → [[Int]]
divide n xs = [split n (drop i xs) \\ i←[0..n-1]]
where
    split n [] = []
    split n [x:xs] = [x : split n (drop (n-1) xs)]

qsort :: [Int] → [Int]
qsort [] = []
qsort [a:xs] = qsort [x \\ x ← xs | x < a] ++ [a] ++
               qsort [x \\ x ← xs | x > a]
```

The farm computational skeleton uses the following Clean function arguments for the dividing and combining phases.

```
myval = generator
myflow_farm i = DDivideD (divide 3) (0, i)
  >>= DApply1 (F "qsort" qsort)
  >>= DMerge combine_lists
```

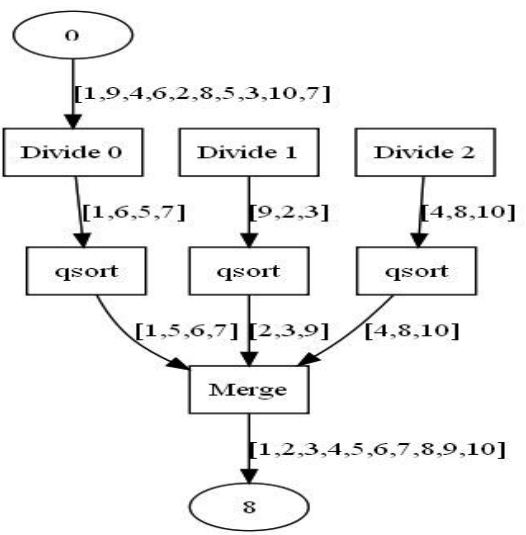


*Figure* 5. Farm skeleton

The `DDivide` boxes also contain the threads numbers obtained after division. The `divide` function divides the input into `n` parts, `DApply1` applies the same `qsort` function to the divided parts and `DMerge` will merge the subresults into a final one using the `combine_list` function. `DApply1` as the other apply operations

uses the `F` type constructor for Clean functions, while the `D` type constructor stands for D-Clean expressions (see type definitions in the appendix A). Special vizualization is applied for the `DDivide` computation node. Instead of depicting one box for the primitive itself, we depict as many boxes as many computation threads will appear after division. In this proper way, a box appears for every subthread sparkled after division.

Another version of the farm skeleton may use the `DReduce` primitive instead of `DApply` (see Figure 6). `DReduce` is a special case of `DApply` with some restrictions. A valid expression for `DReduce` has to reduce the dimension of the input channel. The modified farm skeleton can be used with the same `divide` function and the `sum::[a]→a` function as below.

```
myflow_farm_reduce i
  = DDivideD (divide 3) (0, i)
  >>= DReduce sum
  >>= DMerge sum
```

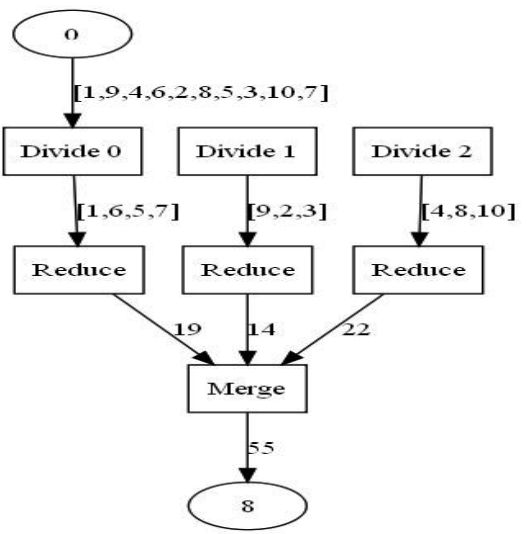


*Figure* 6. Farm reduce skeleton

The opposite of the `DReduce` is the `DProduce` primitive, which is another special case of `DApply`. The expression has to increase the dimension of the channel type.

### 2.3.1. Farm and pipeline skeleton composition

The following example uses the same farm computation skeleton, but here the subtasks are pipelined functions. The square root values of the elements

given by the `generate` function are computed using Newton iterations. The approximate square root of the value $a$ is calculated according to the following formula:

$$x_0 = \frac{a}{2}$$
$$x_{i+1} = \frac{1}{2} * \left(\frac{a}{x_i} + x_i\right)$$

The generated real numbers are first converted into a record containing the proper value and the first iteration (the half of the value).

```
:: Pair = { d :: Real
          , a :: Real
          }

generator :: [Real]
generator = [1.0,9.0,4.0,6.0,2.0,8.0,5.0,3.0,10.0,7.0]

t :: Real → Pair
t x = {d = x/2.0, a = x}

transform :: [Real] → [Pair]
transform x = map t x

divide :: Int [Pair] → [[Pair]]
divide n xs = [split n (drop i xs) \\ i←[0..n-1]]
where
    split n [] = []
    split n [x:xs] = [x : split n (drop (n-1) xs)]

step :: Pair → Pair
step x = {d = 0.5*((x.a/x.d)+x.d), a = x.a}

f :: [Pair] → [Pair]
f x = map step x

combine_lists :: [[Pair]] → [Pair]
combine_lists x = flatten x
```

The `Farm` scheme is used for distributing the values on three different computation threads. The input is a list of `Pair` elements obtained by the `transform` function. `DDivideD` splits the input dataflow into three parts. After division, on each branch a pipeline composition of `g` functions is applied. Finally, `DMerge` collects the subresults into a final list.

```
myflow i
  = DDivideD (divide 3) (0,i)
  >>= DApplyN [F "f" (f o (f o f)), F "f" (f o (f o f)),
               F "f" (f o (f o f))]
  >>= DMerge combine_lists
```

The generated visualization enables seeing the parallel operations and the subresults on the threads as in Figure 7.

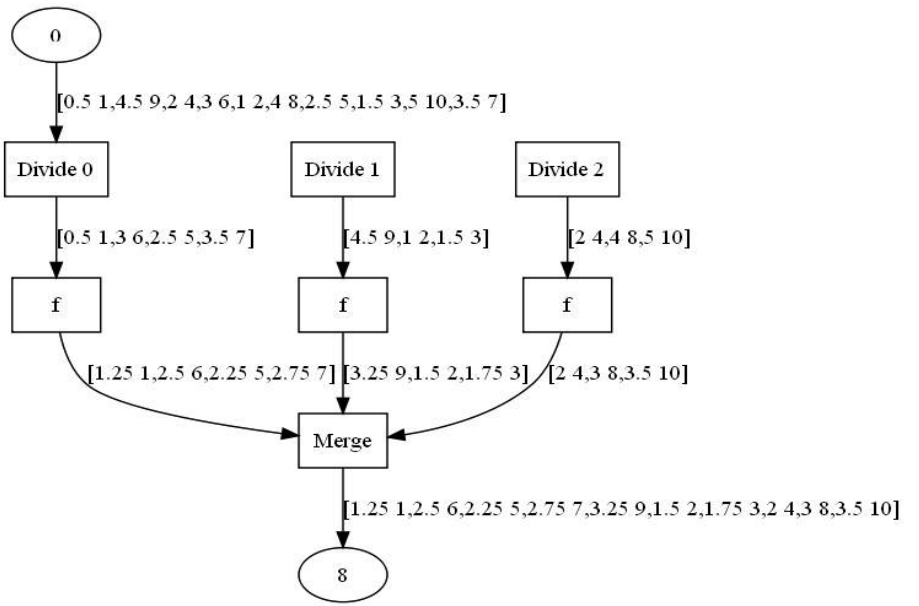


*Figure* 7. Farm and pipe combination

## 3. The model of D-Clean distributed system using C++

*Templates* are key elements of the C++ programming language [16]. They enable data structures and algorithms to be parameterized by types, thus capturing commonalities of abstractions at compilation time [17]. In C++, in order to use a template with some specific type, an instantiation is required. This process can be initiated either implicitly by the compiler when a template with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and afterwards the code is compiled. As the result, the generated code will be free from run-time overhead.

The abstraction provided by templates is frequently needed when using general algorithms in data structures, or defining the data types like a vector or a stack of elements of the same type. This method of code reuse is often called parametric polymorphism to emphasise that here the variability is supported by compile-time template parameter(s). Generic programming [13] is the programming paradigm for writing highly reusable components (containers and algorithms) using parametric polymorphism. The most notable example of generic programming is the Standard Template Library, part of the standard C++ library [10].

The Standard Template Library defines template classes for the most common data structures, called *containers*, and generic *algorithms* working on containers. Algorithms are parameterized by *iterators* – special references to container elements.

Many times variations on algorithms are expressed by further template parameters – *functors*. Functors are objects from function classes. In C++ function classes are the generalisation of the concept of function or function pointer. These classes define an `operator()` (the parenthesis operator), and therefore functor objects can act like functions, i.e. they are callable. At the same time, they are real C++ objects, so they may (and usually) have constructors, they can be stored in variables and can be passed as function parameters.

C++ templates are type-safe. The template instantiation happens as part of the compilation and code generation process and any inconsistent usage of templates cause compile-time errors. This feature makes the template an ideal building block for constructing complex software architecture.

Due to the above advantages we decided to model the D-Clean distributed system with C++ templates.

### 3.1. The model architecture

The executable model architecture consists of two major components: the D-Clean foundation template library and the generated client code running on top of the library. The foundation library contains the base architectural components: the channels, the messages and the D-Clean elements. This is the fixed part of the architecture: re-used but not modified for the various D-Clean application models. On the other side, the client code is specific for the given D-Clean application we want to model, and it is generated from the actual D-Clean application. This is the code part where we assemble the various elements of the foundation library to make them a working model.

We designed the foundation library according to the generic programming paradigm but we have also utilized object-oriented techniques. All of the foundation library elements are implemented as template classes. Template param-

eters of these classes represent the types carried by the D-Clean messages, the cardinality of the channels connecting D-Clean coordination primitives and functors representing the D-Clean skeletons. At the same time, an object-oriented hierarchy provides some basic executable behaviour for model execution and debugging purposes. In the following we explain this architecture in details.

### 3.2. DMessage

The `DMessage` class represents the unit of information propagated between D-Clean coordination primitives. The message is a finite sequence of unspecified elements from the parameter type `T`. We implemented the `DMessage` class via the `std::vector` due to its simplicity and efficiency. The class is conceptually the following:

```
template <typename T>
class DMessage : public std::vector<T>
{
  // ...
};
```

As `T` could be any other well-defined type, a message could be a sequence of integers, a sequence of complex structures (including classes) or even a sequence of different finite sequences. The following examples are valid messages:

```
struct X { /* ... */ };

DMessage<int> dmi;
DMessage<X>   dmx;

DMessage<std::list<dmx> > dmlx;
```

representing a sequence of integers, a sequence of elements with type X, and a sequence of lists with elements of type X, respectively.

### 3.3. DChannelS

The `DChannelS` class represents a multi-dimensional communication channel between two coordination primitives. These channels have both statically bound dimension (the S notation at the end of their name yields this) and statically bound message type given as template parameters. In other words, an instance of the `DChannelS` class can carry only a given type of messages and have a fixed dimension. Conceptually, a `DChannelS` is the following:

```
template <typename T, int N>
class DChannelS
{
  // ...
};
```

where `T` is the type parameter of the `DMessage` objects transmitted via the channel, and the `N` integer parameter is the static dimension. Thus, `N`, the dimension of a `DChannelS` object represents N channels in a DClean expression.

The following examples are 2, 3 and 5 dimensional channels able to carry messages defined in the earlier example:

```
DChannelS< int, 2>        dchi;  // transfers two DMessage<int>
DChannelS<   X, 3>        dchx;  // transfers three DMessage<X>
DChannelS<std::list<X>,5> dchlx; // .. five DMessage<std::list<X>>
```

The channel class is implemented as an N-dimensional array of `std::deque`s containing `DMessage` objects with the appropriate type. These parameters are accessible during run-time as they are part of the class public interface:

```
template <typename T, int N>
class DChannelS
{
public:
  typedef T Type;
  static const int Size = N;
  int dim() const { return N; }
  // ...
};
```

The main interface to write and read messages to and from a channel are the `pop` and `push` functions:

```
DMessage<T> pop(int i);
void        push(int i, const DMessage<T>& m);
```

These methods are applied to the `i`-th subchannel. Messages are written and read in a sequential manner. The `write()` function places a new message to the i-th subchannel. The `read()` function returns the subsequent message from the i-th subchannel and also removes it from the channel.

Every channel object has a name attribute. This is set by the constructor and accessible during run-time for debug purposes.

### 3.4. DBox

The DBox container stores pointers to DElem class, the common abstract base class of every D-Clean coordination primitives. DBox is a generic template container, hosting various coordination primitives. When the model is built up, the coordination primitives should be registered to a DBox object. The main feature of a DBox is the tick() method, which represents an execution step in the model simulation. When tick() is called, it iterates over the container and calls the tick() method of every coordination primitive object registered to DBox.

```
class DBox
{
public:
  void tick(bool verbose = false);
  std::vector<DElem*>& elems() { return _elems; }
};
```

### 3.5. DElem

The D-Clean coordination primitives are implemented in distinct template classes derived from a common base class: DElem. Each primitive has a unique name for debug purposes and a virtual tick() method. Each call of the tick() method executes a simulation step in the coordination primitive. To follow these steps, the optional verbose parameter can be set to print extra debug information.

```
class DElem
{
public:
  virtual void tick(bool verbose = false) = 0;
  std::string name() const { return _name; }
};
```

The DElem class is *abstract* class, and the intention is to instantiate only derived classes. Derived classes define an overriding method of tick() to implement coordination primitive specific actions for one execution step in the model simulation.

Every coordination primitive is represented as class templates derived from the (non-templated) DElem class. The first template parameter of coordination primitives is the type of the messages. The other template parameters are the dimensions of input channels and of output channels, respectively. These

parameters are compile time `int` constants. The last template parameter is the *functor* executed by the coordination primitive.

The D-element classes are the following: `DDivide`, `DApply1`, `DApplyN` and `DMerge`.

### 3.6. DDivide

The class `DDivide` represents a coordination primitive responsible to distribute information from the incoming channel to the outgoing one. The type parameters are the type of the message (`T`), the dimensions of the incoming and outgoing channels (`Ni` and `No`, respectively) and the functor executed as the skeleton (`F`).

The main feature of the `DDivide` (as well as all of the other derived classes inherited from `DElem`) is the overriding version of the `tick` method. This method will be responsible to call the functor `F` and also to write relevant debug information in case the `verbose` parameter was set.

Therefore the conceptual structure of `DDivide` is the following:

```
template <typename T, int Ni, int No, typename F>
class DDivide : public DElem
{
public:
  DDivide(const char *n_, DChannelS<T,Ni>& in_,
                          DChannelS<T,No>& out_, F func_);
  virtual void tick(bool verbose = false);
  // ...
};
```

Creating C++ objects of template classes directly using the constructor is a pain, as we have to explicitly specify each template parameter of the class. This is especially uncomfortable due to the presence of functor parameters. Therefore, we provide free functions as factories to create D-element objects using template parameter deduction from actual arguments.

```
template <typename T, int Ni, int No, typename F>
DDivide<T,Ni,No,F>* make_DDivide(const char *name_,
                                 DChannelS<T,Ni>& in_,
                                 DChannelS<T,No>& out_,
                                 F func_)
{
  return new DDivide<T,Ni,No,F>(name_, in_, out_, func_);
}
```

The following function call creates a new `DDivide` object reading from a 2-dimensional input channel `ich` and writing to a 5-dimensional output channel `och`, redistributing incoming messages. The return value is a `DDivide` pointer which has been safely upcasted to `DElem` pointer.

```
DElem *dePtr = make_DDivide("div", ich, och, RedistF<int,2,5>());
```

### 3.7. DApplyN

The most generic coordination primitive is `DApplyN` implementing a skeleton reading and writing arbitrary channels while executing a functor parameter.

```
template <typename T, int Ni, int No, typename F>
class DApplyN : public DElem
{
public:
  DApplyN(const char *n_, DChannelS<T,Ni>& in_,
                          DChannelS<T,No>& out_, F func_);
  virtual void tick(bool verbose = false);
};
```

A usual situation is when the input and the output channels have different dimensions. Therefore, `DApplyN` usually changes the channel dimension. This class has a special version called `DMerge` to map multiple dimension input channel into a one dimensional output channel.

### 3.8. DApply1

In many cases, a skeleton executes the same function on every dimension of the input channel independently of the other dimensions. Such coordination primitives can be implemented using `DApplyN` only with unnecessary complexity. For such special cases we provide `DApply1`, a primitive for one-dimensional transformation.

```
template <typename T, int N, typename F>
class DApply1 : public DElem
{
public:
  DApply1( const char *n_, DChannelS<T,N>& in_,
                           DChannelS<T,N>& out_, F func_);
  virtual void tick(bool verbose = false);
};
```

While the parameter list of `DApply1` is exactly the same as the one of `DApplyN`, the functor is executed in a loop for every subchannel individually. We suppose that the input and output channels have the same dimensions, otherwise we emit compile-time errors.

```
template <typename T, int N, typename F>
void DApply1<T,N,F>::tick(bool verbose_)
{
  // debug if verbose_
  for (size_t i = 0; i < N; ++i)
  {
     DMessage<T> m = _func(_input.pop(i));
     _output.push(i,m);
  }
  // debug if verbose_
}
```

## 4. Working example

In this section we show a complex scenario to describe the behaviour of the executable model.

The scenario is the following. One integer number, K, is given in the input. The first coordination primitive, a `DDivide`, will distribute this single number into a 16-dimensional output channel, repeating K 16 times. The next primitive is a random number generator implemented by a `DApply1` on each single dimension of the input channel generating a random number between 0 and K-1. The generated number is placed to the corresponding output.

Thus, 16 random numbers will travel in a parallel way to the next primitive, which is a `DApplyN` node, converting the 16-dimensional input channel into a 4-dimensional output channel, just creating a sequence of the input numbers. Here, another `DApply1` will sort each of the dimensions. The last two nodes are `DMerge` nodes, merging a par of input sequences into a single ordered sequence. At the very end, we will get a single ordered sequence of 16 numbers.

At the beginning we create a `DBox` object (`db`), where all the foundation library elements will be registered. We create the `DChannelS` objects (`input`, `ch1`, ..., `ch5`, `output`) with the corresponding message type and dimensions as template parameters.

```
DBox db;
DChannelS<int,1>  input("input");
```

```
DChannelS<int,16> ch1("ch1");
DChannelS<int,16> ch2("ch2");
DChannelS<int,4>  ch3("ch3");
DChannelS<int,4>  ch4("ch4");
DChannelS<int,2>  ch5("ch5");
DChannelS<int,1>  output("output");
```

Then we create the coordination primitives connecting the correct channels. We register the objects to the `DBox` object. To create the objects we use the factory methods introduced in 3.6. We apply the `push_back` method of the standard C++ `std::vector` class to insert the new elements to `DBox`.

```
DMessage<int> m1;
m1.push_back(10);
input.push(0,m1);
db.elems().push_back(make_DDivide("divide", input, ch1,
                                  divF<int,1,16>()));
db.elems().push_back(make_DApply1("rand", ch1, ch2, randF));
db.elems().push_back(make_DApplyN("zip", ch2, ch3,
                                  zipF<int,16,4>()));
db.elems().push_back(make_DApply1("sort", ch3, ch4, sortF));
db.elems().push_back(make_DApplyN("merge4", ch4, ch5,
                                  mergeF<int,4,2>()));
db.elems().push_back(make_DApplyN("merge2", ch5, output,
                                  mergeF<int,2,1>()));
```

We place the only input to the input channel and start the simulation calling the `tick()` function.

```
DMessage<int> m1;
m1.push_back(10);
input.push(0,m1);
db.tick(true);
```

Now, the simulation iterates on all registered objects calling their virtual `tick()` functions. The output of the example can be found in Appendix B. A simplified Clean scenario and its skeleton cane be found in Appendix C.

## 5. Related work

Nowadays parallel functional applications are largely researched and tested on Grid systems. The D-Clean language is an extension of the functional programming language Clean with a relatively small sets of primitives to support

the distributed computation of Clean functions over clusters. Early parallelizations of Clean for parallel super-computers are in [11, 15]. However, it became important to provide tools for the new type of distributed environments too, making it possible to program parallel skeletons on Grid systems. Our distributed process network approach is related to the Eden dialect [1] of Haskell (for comparisons of parallel Haskell dialects see [12]).

*Skeletons* are computation patterns, algorithmic schemes that capture common computation mechanism [4]. Skeletons can be defined and parameterized by functions, types and data. They are widely used in parallel computations, see [14]. In functional programming, skeletons can be combined with evaluation strategies in order to obtain optimal parallel behaviour, e.g. [8].

D-Clean is based on functional composition of the coordination primitives. This enables us to build compound coordination structures, and they are applied on dataflows. Using the coordination primitives process schemes can be defined easily. The D-Clean schemes are distributed functional computational skeletons 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. In the case of the widely used skeletons (like farm, divide and conquer, pipe and reduce), it is easier to deal with the type inference problem than in general. Measurements demonstrated high speed-ups for more complex problems. Several parallel functional languages ([5, 6]) are dealing with skeletons applied to dataflows [9, 3].

## 6. Conclusion and future work

In this paper we have presented the D-Clean model implemented using C++ templates. This implementation of the D-Clean distributed system model proved to be an efficient software comprehension tool for testing parallel programs. Complex D-Clean skeletons can be modeled and analyzed in a deterministic, repeatable, single threaded environment. The strongly typed C++ templates guarantees the correctness of the generated code and provides convincing runtime efficiency.

We plan to further strengthen the C++ description of the distributed environment, to polish the generation of the templates directly from the D-Clean definitions, which saves the programming efforts when testing distributed applications. Fully automated generation provides a maximum compatibility between the D-Clean system and the generated C++ model. We also plan to develop larger applications for comparing the two different executable semantics.

# Appendix

## A. The D-Clean language reference

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

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

// single channel
:: Ch a         :== (Int, a)

// multiple channel
:: MCh a        :== [Ch a]

// wrapper
DExec :: a (DFun a b) → (a, (b, State))

// D-Clean primitives, toString restriction used to allow trace to be made.
DApply      :: String (a → b)  → DFun (Ch a)  (Ch b)      | toString b
DDivideD    :: (a → [b])       → DFun (Ch a)  (MCh b)     | toString b
DApplyN     :: [DExpr a b]     → DFun (MCh a) (MCh b)     | toString b
DMerge      :: ([b] → a)       → DFun (MCh b) (Ch a)      | toString a

// D-Clean utility functions

DMap        :: (a → b)         → DFun (Ch [a]) (Ch [b])   | toString b
DApply1     :: (DExpr a b)     → DFun (MCh a)   (MCh b)   | toString b
DProduce    :: (a → [b])       → DFun (MCh a)   (MCh [b]) | toString b
DReduce     :: ([b] → a)       → DFun (MCh [b]) (MCh a)   | toString a
DMap2       :: (a → b)         → DFun (MCh [a]) (MCh [b]) | toString b
DFilter     :: (a → Bool)      → DFun (MCh [a]) (MCh [a]) | toString a
```

## B. The sorting example - output

```
++ next step: divide ++
** Pre ****
input[1], size=1
    { {10,} }
ch1[16], size=0
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
** Post ****
input[1], size=0
    { }
ch1[16], size=1
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
++ next step: rand ++
** Pre ****
ch1[16], size=1
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
    { {10,} }
ch2[16], size=0
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
** Post ****
ch1[16], size=0
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
ch2[16], size=1
    { {3,} }
    { {6,} }
```

```
    { {7,} }
    { {5,} }
    { {3,} }
    { {5,} }
    { {6,} }
    { {2,} }
    { {9,} }
    { {1,} }
    { {2,} }
    { {7,} }
    { {0,} }
    { {9,} }
    { {3,} }
    { {6,} }
++ next step: zip ++
** Pre ****
ch2[16], size=1
    { {3,} }
    { {6,} }
    { {7,} }
    { {5,} }
    { {3,} }
    { {5,} }
    { {6,} }
    { {2,} }
    { {9,} }
    { {1,} }
    { {2,} }
    { {7,} }
    { {0,} }
    { {9,} }
    { {3,} }
    { {6,} }
ch3[4], size=0
    { }
    { }
    { }
    { }
** Post ****
ch2[16], size=0
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
    { }
ch3[4], size=1
    { {3,6,7,5,} }
    { {3,5,6,2,} }
    { {9,1,2,7,} }
    { {0,9,3,6,} }
++ next step: sort ++
** Pre ****
ch3[4], size=1
    { {3,6,7,5,} }
    { {3,5,6,2,} }
    { {9,1,2,7,} }
    { {0,9,3,6,} }
ch4[4], size=0
    { }
    { }
    { }
    { }
** Post ****
ch3[4], size=0
    { }
    { }
    { }
    { }
ch4[4], size=1
    { {3,5,6,7,} }
    { {2,3,5,6,} }
    { {1,2,7,9,} }
    { {0,3,6,9,} }
++ next step: merge4 ++
** Pre ****
ch4[4], size=1
    { {3,5,6,7,} }
    { {2,3,5,6,} }
    { {1,2,7,9,} }
    { {0,3,6,9,} }
ch5[2], size=0
    { }
    { }
** Post ****
ch4[4], size=0
    { }
    { }
    { }
    { }
ch5[2], size=1
    { {2,3,3,5,5,6,6,7,} }
    { {0,1,2,3,6,7,9,9,} }
++ next step: merge2 ++
** Pre ****
ch5[2], size=1
```

```
    { {2,3,3,5,5,6,6,7,} }
    { {0,1,2,3,6,7,9,9,} }
output[1], size=0
    { }
** Post ****
ch5[2], size=0
 { }
 { }
output[1], size=1
 { {0,1,2,2,3,3,3,5,5,6,6,6,7,7,9,9,} }
```

## C. Example in D-Clean

```
Start w = dumpGraph (DExec myval myflow) w
where
  myval = [1, 5, 2, 4, 8, 9, 3, 6, 12, 7, 14, 10, 15, 0, 11, 13, 12]
  myflow i = DDivideD (divide 4) (0,i)
   >>= DApplyN [F "sort" sort, F "sort" sort,  F "sort" sort,
                F "sort" sort]
   >>= DMerge (sort o flatten)
```

The code generates the following graph:

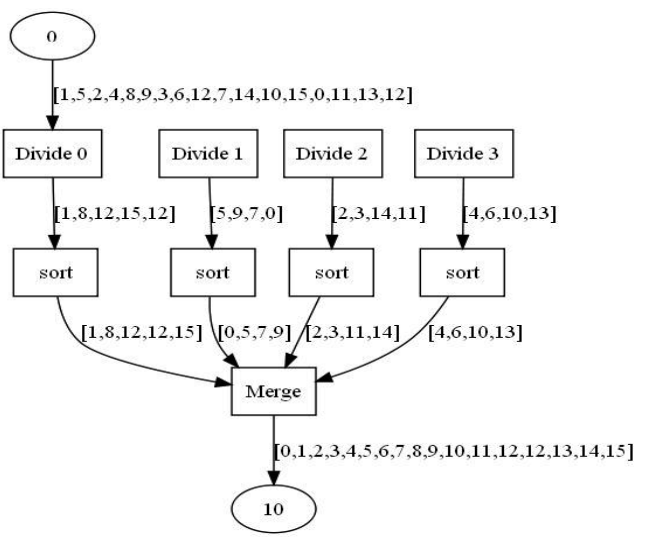


*Figure* 8. Example

## References


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

[2] **Best, E. and R.P. Hopkins,** 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., C. Zoltan and G. Prestigiacomo,** NiMoToons: a totally graphic workbench for program tuning and experimentation, *ENTCS*, Volume 258 Issue **1**, December 2009, pp. 93–107.

[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., R. Di Cosmo, X. Leroy and S. Pelagatti,** 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] **Fournet, C., F. Le Fessant, L. Maranget and A. Schmitt,** JoCaml: A language for concurrent distributed and mobile programming, in: Jeuring, J., Peyton Jones, S. (Eds): *AFP 2002*, Oxford, Revised Lectures, Springer, LNCS 2638, pp. 129–158, 2003.

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

[8] **Horváth Z., V. Zsók, P. Serrarens and R. Plasmeijer,** Parallel elementwise processable functions in concurrent clean, in: *Mathematical and Computer Modelling* **38**, pp. 865–875, Elsevier, Pergamon, 2003.

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

[10] **Josuttis, N.,** *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN: 0-201-37926-0, 1999.

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

[12] **Loidl, H-W., F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, R. Peña, S. Priebe,**

**Á.J. Rebón Portillo and P.W. Trinder,** Comparing parallel functional languages: Programming and performance, in: *Higher-Order and Symbolic Computation* 16 (**3**), pp. 203–251, Kluwer Academic Publisher, September 2003.

[13] **Musser, D. R. and A.A. Stepanov,** Algorithm-oriented generic libraries, *Software-practice and experience*, 27(**7**) July 1994, pp. 623–642.

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

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

[16] **Stroustrup, B.,** *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.

[17] **Vandevoorde, D. and N.M. Josuttis,** *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[18] **Zsók V., P. Koopman and R. Plasmeijer,** An executable semantics for D-Clean, in: Hage, J. (ed): *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages*, IFL 2010, Alphen aan den Rijn, September 1–3, 2010, Technical Report UU-CS-2010-020 August, 2010, Utrecht University, The Netherlands, pp. 173–179.

[19] **Zsók V., Z. Hernyák and Z. Horváth,** 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., P. Koopman and R. Plasmeijer,** Generic executable semantics for D-Clean, in: Porkoláb Z. et al (eds), *Proceedings of the Third Workshop on Generative Technologies*, WGT 2011, ETAPS 2011, Saarbrücken, Germany, March 27, 2011, *ENTCS*, Vol. 279, Issue **3**, Elsevier, December 2011, pp. 85–95.

**V. Zsók and Z. Porkoláb**
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
zsv@elte.hu
gsd@elte.hu