6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Clean-CORBA Interface Supporting Pipeline Skeleton*

Zoltán Hernyák[a], Zoltán Horváth[b], Viktória Zsók[b]

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

[b]Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest
e-mail: {hz, zsv}@inf.elte.hu

Abstract

Modern functional programming deals with parallel and distributed computation. The lazy functional programming language Clean was extended for cluster computations using CORBA as middleware.

The earlier Clean-CORBA interface supports developing and using skeletons for distributed and cluster computing in a limited way only. The interface user encounters three major problems. Clean skeletons are parameterized by types while the IDL compiler does not generate stubs for polymorphic functions. The other major problem is that there is a need for such a component of the middleware system which starts the skeleton according to a strategy given as a parameter. The third problem is that the actual Clean-CORBA does not support asynchronous communication.

The present paper proposes solutions for the above problems. By introducing a pattern language for generic description, the Clean-CORBA skeletons are extended with the description of the formal parameters: type parameters and strategy parameters. The user of the skeletons provides the description of the actual parameters corresponding to the formal parameters. Additionally to the Clean and the IDL compilers, we propose a code generator, which interprets the formal and actual parameters, generates IDL descriptions and instances of the server objects according to the actual parameters, invokes the IDL and the Clean compilers for generating stubs and objects. The second problem is solved by running such objects at each computing node, which starts the components of the skeleton according to the

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742 and by Bolyai Scholarship of Hungarian Academy of Sciences.

actual strategy. The implementation of the channels supports the asynchronous communication.

Categories and Subject Descriptors: D.1 Programming Techniques: D.1.1 Applicative (Functional) Programming, D.1.3. Concurrent Programming.

Key Words and Phrases: Skeleton, distributed functional programming, Clean, CORBA, design patterns, middleware.

1. Introduction

One of the major research directions of the parallel functional programming [1] is the use of skeletons as higher order functions for computation patterns. The functional skeletons can be triply parameterized by types, by functions and by evaluation strategies [2].

A solution based on multiparadigm programming was proposed in order to support the skeletal distributed programming with functional components written in Clean [3]. The lazy functional programming language Clean was extended for cluster computations using CORBA as middleware. Clean components are CORBA clients connected by communication channels implemented as CORBA server objects. However the user of the Clean-CORBA interface encounters some impediments in the development of skeletons for distributed and cluster computing. The IDL to Clean compiler has no support for parameters of polymorphic types. The IDL type `any` allows data types to be examined and extracted using the type code interface, but it is not a complete solution for the problem. The CORBA stub generation for polymorphic functions are not supported.

An alternative solution is proposed in the next section, which introduces control language elements for the application of the Clean-CORBA skeletons. The client and server objects are described as patterns with type parameters. Additionally to the Clean and the IDL to C compilers we propose a code generator. The code generator interprets the parameters of the skeleton, instantiates IDL descriptions, client and server codes according to the actual parameters, invokes the IDL to C and the Clean compilers for generating stubs and objects.

The other major problem is that there is a need for such a component of the middleware system which starts the application of the skeleton according to the strategy parameter. The instantiated client and channel objects are distributed over the computation nodes of the cluster by a generated starting script.

In the earlier Clean CORBA interface receive operations lead to busy waiting when the channel was empty. The new implementation of channel objects is based on the C-CORBA interface. Asynchronous communication is enabled due to the multithread-safe server architecture. Whenever a Clean client code effectuates a retrieve operation on an empty channel, the client is automatically suspended without busy waiting.

2. The pipeline skeleton

This paper deals with Clean skeletons which are parameterized by types, functions and strategies. An important skeleton, the pipeline skeleton is used as case study. The proposed solutions for the deficiencies identified in the introduction are presented through the pipeline problem. The implementation is tested on the cluster. The programming environment is built up of several layers.

The starting layer is the abstract description of the problem, the pipeline skeleton. The abstract pipeline contains annotations in order to separate the description of the computation from the details of the concrete topology and granularity of the computation nodes.

```
distributed module pipe_one

DistrStart = pipesk 'on' data 'applying' functionlist 'with' strategy
   where
      data                = [i \\ i <- [1.0 .. 1600.0]]
      functionlist        = [f0] ++ [f \\ i <- [1..10]] ++ [fn]
      strategy            = par
```

Several working phases are identified from the starting point until the program produces the final result (figure 1).

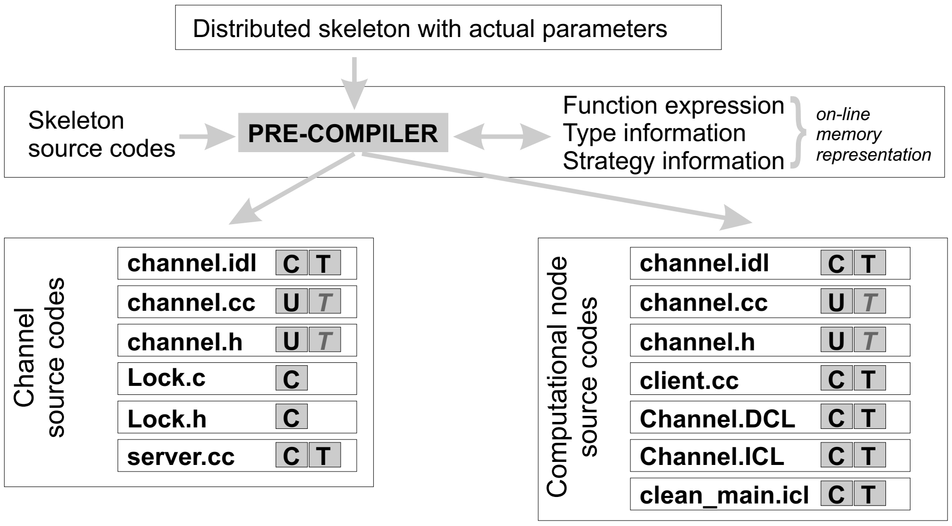

Figure 1: Working phases

The meanings of the letters on the figure 1 are the following:

- C denotes a source code generated by our pre-compiler.

- U denotes a source code generated by a third-party utility (in this case by the IDL to C compiler).
- T Ű denotes a source code which depends on the actual parameters of the distributed skeleton. The type information is included by the pre-compiler.
- *T* Ű also denotes a code which depends on the actual parameters of the distributed skeleton, but it depends in an indirect way (this code is generated by a 3rd party utility based on a file which directly depends on the type information).

The distributed source codes are generated from the skeleton source codes, which are stored in a directory as separate files. The pre-compiler uses these files, modifies them according to the actual information, and generates the source code for all the CORBA channels and the Clean computational nodes. See the figure 1 for the set of generated files. These source files containing the skeleton source code usually have the same names as the generated files – except the `clean_main.icl` files.

Detailed description of the working phases is given below.

1. The distributed skeleton with actual parameters is passed through a preprocessor phase at the first step. The pre-compiler analyses the higher order description of the computation. This description is the pipeline skeleton (`pipesk`) indeed, which is parameterized with actual parameters. The pre-compiler separates and identifies all the information necessary to generate the parts of the distributed application. It collects the component functions, the type information regarding the input data and functions. The type information is needed for establishing the type of the communication channels. The topology of functions which is determined by the strategy is defining the relationships and the communication channels between the client programs evaluating and computing the component functions.

2. There are three generic skeleton components written for the pipeline computation. They contain the channel operations of the client programs, which are the primitives of channels: `send`, `sendStream`, `collectStream`. The primitives are parameterized by types and by functions.

   The pre-compiler uses and completes the components – selecting the suitable one from the skeleton source codes. The first component produces elementwise the data needed for the pipeline.

   This component is stored in the skeleton source code directory as `clean_first.icl`.

```
module clean_main
import Channel, StdEnv, StdDebug

Start w
   #! w          = Corba_INIT w
```

```
                        // CORBA initialization
   #! (chn_out, w) = Channel_FIND <CHANNEL_FIRST> w
                        // the output channel
   #! w            = sendStream <FUNCTION_NAME> d chn_out w
                        // data sending to the first channel
   #! w            = Channel_FINISHED chn_out w
                        // sending the extremal data
   = w

// sending the data recursively
sendStream f0 [] chn w      = w
sendStream f0 [lh:lt] chn w = sendStream f0 lt chn w2
   where
      w2 = Channel_STORE chn (f0 lh) w
```

The second component elementwise processes the data. This is stored in the skeleton source code directory as `clean_immed.icl`.

```
module clean_main
import Channel
import StdEnv

Start w
   #! w            = Corba_INIT w   // CORBA initialization
   #! (chn_inp, w) = Channel_FIND <CHANNEL_INP> w   // the input channel
   #! (chn_out, w) = Channel_FIND <CHANNEL_OUT> w   // the output channel
   #! w            = ewp <FUNCTION_NAME> chn_inp chn_out w
                        // data processing
   #! w            = Channel_FINISHED chn_out w
                        // sending the extremal data
   = w

ewp f channel_inp channel_out w
   #! (ok, data, w2) = Channel_RETRIEVE channel_inp w
   | ok == False     = w2
   | otherwise
      #! sentw = Channel_STORE channel_out (f data) w2
      = ewp f channel_inp channel_out sentw
```

The third component elementwise consumes the data from the last channel of the pipeline using the `collectStream` primitive. This is stored in the skeleton source code directory as `clean_last.icl`.

```
module clean_main
import Channel,StdEnv

Start w
   #! w            = Corba_INIT w   // CORBA initialization
   #! (chnf, w)    = Channel_FIND <CHANNEL_LAST> w   // the input channel
   #! (result, w)  = collectStream <FUNCTION_NAME> chnf [] w
                                   // data processing
   = (result, w)
```

```
collectStream fn channel list w
   #! (ok, data, w2) = Channel_RETRIEVE channel w
   | ok == False     = (reverse list ,w2)
   | otherwise
      #! (list3, w3) = collectStream fn channel list w2
      = ([(fn data):list3], w3)
```

The pre-compiler completes the skeleton components of the library according to the collected information and generates the clients. Each client is one computation node. They are inserted into separate subdirectories in the same order as it was in the function list of the higher level skeleton. The formal parameters are replaced by the actual parameters.

3. At the next step the channel interfaces are generated to complete the source codes of the computational nodes.

   The generic IDL file (`Channel.idl`) has to be instantiated to describe for every channel the CORBA interfaces.

```
interface Channel {
char Store   ( in  chn_data data, in unsigned long timeOutSec );
char Retrieve( out chn_data data, in unsigned long timeOutSec );
char Finished( in unsigned long timeOutSec );};
```

   This IDL skeleton is instantiated by the pre-compiler using the type information collected at the first phase. The pre-compiler inserts the types, including its `typedef` into the skeleton.

```
typedef double chn_data;
```

4. The IDL2C utility creates the stubs (`Channel.cc`, `Channel.h`) of the instantiated channel. It generates both the server stub and the client stub in C. These stubs are already instantiated according to the actual type parameter included into the `Channel.idl`.

5. The `Client.cc` file contains the functions to find and use a channel. This file has to be instantiated according to the actual type of the channel. So the pre-compiler generates one `Client.cc` file for every base type used by the channels.

6. The library contains the `Channel.icl` and `Channel.dcl` codes too, which are Clean-C wrapper templates. This wrapper is the Clean-side image of the `Client.cc` file. The pre-compiler generates these files instantiating them to allow the computational nodes to call the client stub functions from Clean.

7. The Clean client code (`clean_main.icl`) which evaluates and computes one component function calls the instantiated client stub functions through the

Clean-C wrapper. This main module is generated by selecting and instantiating one of the library modules (at this time selecting one of the three generic skeleton components described at the 2. point).

There are several layers of code reading from and writing to a remote channel inside the skeleton library. Using these layers properly one can create new primitive operations and insert it into the library. Let see the structure of the CORBA server (figure 2), which reads the data element from the input channel, modifies (takes a computational step on) the data, and sends it to the next channel (the broken lines mean the communication through the network).

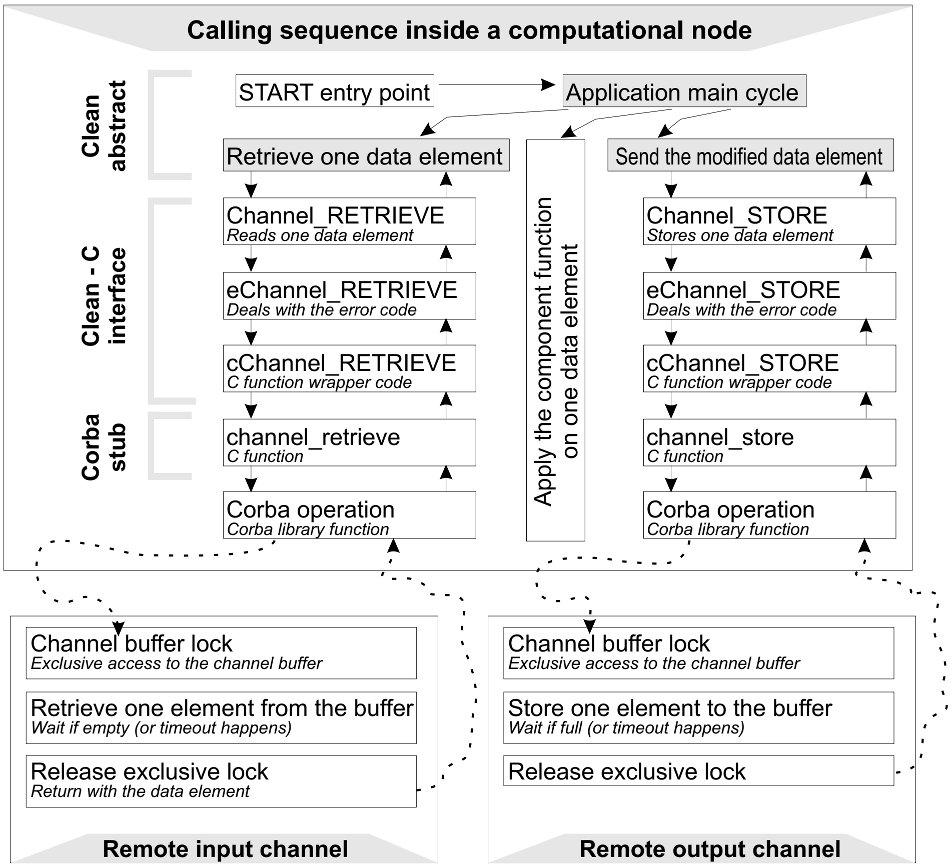

Figure 2: The structure of the CORBA server object

8. The same steps are made for a channel. The implementation of the channels supports the asynchronous communication and the busy waiting problem

is solved too. Whenever a Clean client code effectuates a retrieve operation on an empty channel, the client is automatically suspended. There are two threads which are connected to the channel: the retrieve and the store threads. The server is multithread-safe.

Instantiating the `Channel.idl` the Corba IDL2C can generate the same `Channel.cc` and `Channel.h` stubs. Afterwards the `Server.cc` is generated, which handles the channel operations using the thread synchronization methods stored in the `Lock.c` file. This locking mechanism does not depend on any type information. The generic files have the same name in the skeleton library.

9. Makefiles are also generated for compiling the programs. In addition to these there is a script file written for the distribution of the client codes over the cluster. This identifies the computers, maps the files to them and generates the starting scripts.

3. Measurements

Several measurements were done. Here we present two charts. The first one (figure 3) shows the decreasing computation time when more (up to ten) computers were added to the list of the computation nodes.

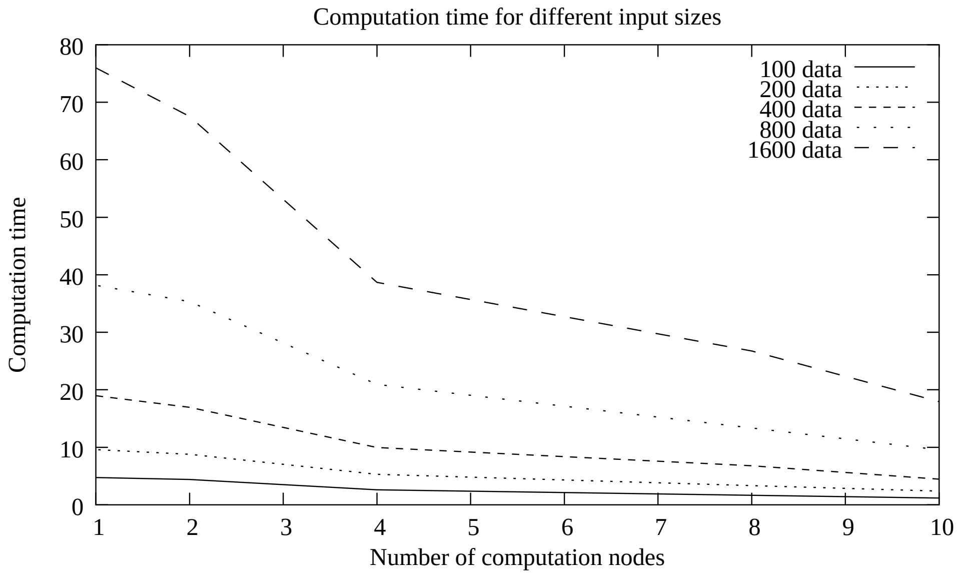

Figure 3: Computation time depending on the number of computation nodes

The second chart (figure 4) is illustrating the speedup obtained in case of different number of computation nodes. The pipeline skeleton had as function parameter a list of weighted functions. We can observe a significant speedup when the number of computation nodes are increased.

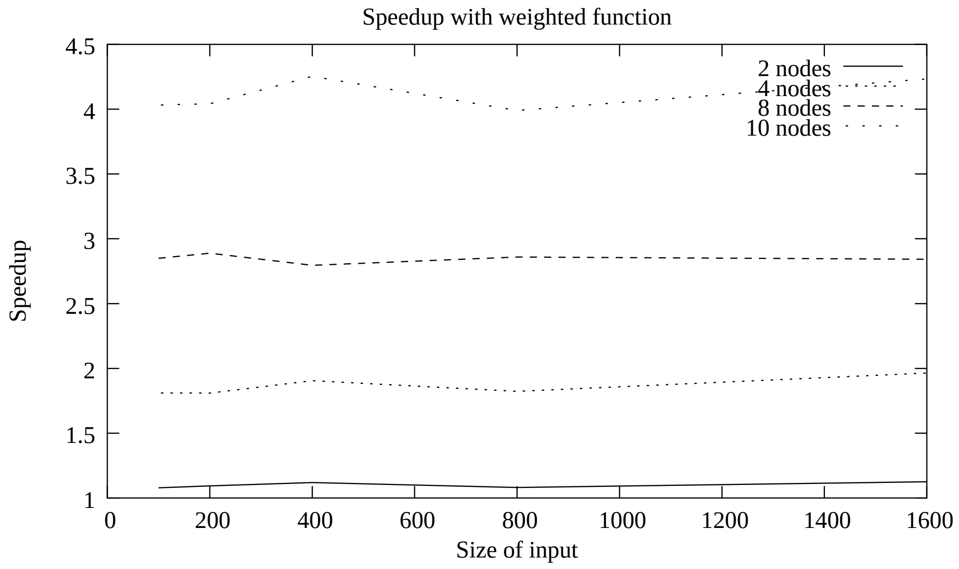

Figure 4: Speedup

4. Conclusions and future work

The pipeline skeleton is parameterized by type and by function. Our precompiler generates stubs for polymorphic functions. The middleware system starts in distributed way the computational threads evaluating the component functions. The asynchronous communication is enabled by our middleware due to the C-CORBA channel interfaces. The distributed evaluation of functions and the communication between clients needs high-level process description and control mechanism [7]. In the future we would like to specify a control language for the specification of the abstract skeletons. A strategy description is needed for controlled behaviour of the process-network. The high-level language elements will coordinate the component functions and the process-network in the distributed environment. These elements will hide all the technical details of the distributed environment from the user.

5. Acknowledgements

Here we thank to Péter Turi for the implementation of the starting scripts. We also thank to Csaba Seres for helping at the technical support of the cluster.

References

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

[2] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, *Mathematical and Computer Modelling*, No. 38, 2003, pp. 865-875.

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

[4] Ugron B., Hajdara Sz.: Synthesis of the synchronization of pipeline systems, In: *Proceedings of the 6th International Conference on Applied Informatics*, Eger, Hungary, January 27-31, 2004, to appear.

[5] 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.

[6] Mowbray, T. J., Malveau, R. C.: Corba Design Patterns, Wiley Computer Publishing, 1997.

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

Postal addresses

Zoltán Hernyák
Department of Information Technology
Eszterházy Károly College
H-3300, Eszterházy tér 1., Eger
Hungary

Zoltán Horváth
Department of Programming Languages and Compilers
Eötvös Loránd University
H-1117 Pázmány Péter sétány 1/C., Budapest
Hungary

Viktória Zsók
Department of Programming Languages and Compilers
Eötvös Loránd University
H-1117 Pázmány Péter sétány 1/C., Budapest
Hungary