Chapter 1

Deriving Non-Hierarchical Process Topologies

Ricardo Peña[1], Fernando Rubio[1] and Clara Segura[1]

Abstract: Eden is a parallel functional language which extends Haskell with new expressions to de£ne and instantiate processes. These extensions allow the easy de£nition of parallel process topologies as higher order functions. Unfortunately, by only using process abstractions and instantiations it is not possible to implement non-hierarchical topologies, as processes can only communicate with its parent or its children. In this paper we show how to implement non-hierarchical topologies in Eden by using its *dynamic channels*. The topologies will be speci£ed by only using process abstractions and instantiations, so that they will really be hierarchical. Afterwards, they will be re£ned into really non-hierarchical topologies using the dynamic reply channels. The usefulness of the translation method will be shown by examples, highlighting the key points to be taken into account to achieve the desired behaviour.

1.1 INTRODUCTION

Processes in Eden are dynamically instantiated while executing recursive functions and/or process de£nitions. When a new process is created, their channels are connected to its parent process. The parent is responsible for feeding child's input channels with values and for receiving values from child's output channels. This implies that, in principle, only hierarchical process topologies can be created. But there are some useful topologies such as a pipeline or a ring that are inherently non-hierarchical. When trying to de£ne these topologies in Eden, hierarchical topologies are obtained, where some intermediate dummy threads just copy the values received from one child to an input channel of another child. In [KPS00] a *bypassing analysis* detecting only some of these situations was presented. But this

[1]Departamento Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain; e-mail: {ricardo,fernando,csegura}@sip.ucm.es

analysis cannot deal with complex problems like a ring topology. In this paper we show a methodology that allows the development of non-hierarchical topologies taking as a basis Eden hierarchical programs. This methodology will take advantage of the so called *dynamic channels* of Eden, and it will be able to manage any topology the programmer can have in mind.

The plan of the paper is as follows. In the next section, the Eden language is presented. Afterwards, in Section 1.3 a general methodology to derive non-hierarchical topologies is presented. Then, several increasingly complex non-hierarchical topologies are introduced, showing how to implement them in Eden by using the presented methodology. Finally, Section 1.5 presents some conclusions and future work.

1.2 EDEN

Basic Constructions. Eden [BLOP98] extends the functional language Haskell by syntactic constructs that explicitly de£ne processes. There exists a new expression `process x -> e` of a prede£ned type `Process a b` to de£ne a *process abstraction* having variable `x::a` as input and expression `e::b` as output. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Additionally, when the output (resp. input) expression is a tuple, i.e. `e::(t1,...,tn)` a separate concurrent thread is created for the evaluation of each tuple element (we will refer to each tuple element as a *channel*).

A *process instantiation* is achieved by using a prede£ned in£x operator whose type is `(#) :: (Transmissible a,Transmissible b) => Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, the instantiating process will be responsible for evaluating and sending `e2`, while a new process is created to evaluate the application `(e1 e2)`. We will refer to the latter as the *child* process, and to the owner of the instantiation expression as the *parent* process. The instantiation protocol deserves some attention to explain Eden's semantics:

- Closure `e1` together with all its dependent closures are *copied* unevaluated to a new processor and the child process is created there to evaluate it.
- Once created, the child process starts producing eagerly its output expression.
- Expression `e2` is eagerly evaluated in the parent process, and the result value is eagerly transmitted to the child process. If it is a tuple, an independent concurrent thread is created to evaluate each component.

This protocol implies that using process abstractions and process instantiations only allows the creation of hierarchical topologies, where the communications are held between a parent and a child. However, there are many useful non-hierarchical topologies where two processes not related as parent and child are directly connected. In those cases, Eden achieves a hierarchical topology in which the producer and the consumer are connected through one or more intermediate

parent processes. There is a thread in the intermediate process that just copies the values received from the producer to an output towards the consumer. It is desirable to eliminate these intermediate processes, that is, to directly connect producers to consumers. The examples in this paper show two kinds of situations where bypassing is desirable, but which cannot be automatically done by the compiler:

Bypassing between generations An input (resp. an output) channel of a process is connected through an intermediate process to an output (resp. an input) channel of a descendant (resp. ancestor) process. This is the case in the pipeline skeleton in Figure 1.1.

Bypassing between siblings An output channel of a process is connected through the parent to an input channel of a sibling process. This is the case in the ring skeleton of Figure 1.4.

Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way in which Eden processes synchronize. To be able to transmit a value, its type must belong to the class `Transmissible`, which includes a function defining how to reduce to normal form and how to send a value. See [KOP99] for more details about Eden's implementation.

Dynamic Channels. A process may generate a new *dynamic channel* and send a message containing its name to another process. The receiving process may then either use the received channel name to return some information to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Both possibilities exclude each other, to guarantee that two processes cannot send values through the same channel.

Eden introduces a new unary type constructor `ChanName` for the names of the dynamically created channels. Moreover, it also adds a new expression `new` (*ch_name*, *chan*) *e* which declares a new channel name *ch_name* as reference to the new input channel *chan*. The scope of both is the body expression *e*. The name should be sent to another process to establish the communication. A process receiving a channel name *ch_name*, and wanting to reply through it, uses an expression *ch_name* `!*` e_1 `par` e_2 . Before e_2 is evaluated, a new concurrent thread for the evaluation of e_1 is generated, whose normal form result is transmitted via the dynamic channel. The result of the overall expression is e_2, while the communication through the dynamic channel is a side effect.

In most situations —in particular in all the topologies presented in this paper— by using only process instantiations it is possible to create the same topologies that could be created by using dynamic channels, except for the fact that some channels will connect the intended processes through intermediate threads in other processes. By using dynamic channels, those will be *direct* connections. In this

sense, this feature can be seen as an optimization using a low-level construct provided by the language rather than as a radically new concept.

Runtime System. Eden's compiler has been developed by extending GHC [Pey96] (Glasgow Haskell Compiler), in order to reuse its ef£ciency and portability. Eden's RTS (Runtime System) is an implementation of the DREAM abstract machine [BKL+98] on top of a message passing library. In the current compiler, both PVM [GBDJ94] and MPI [Mes94] can be used. Thus, the compiler can be ported to any architecture where GHC and either PVM or MPI are available.

Eden provides no *placement annotations*. However, Eden's RTS supports two modes to map processes to processors, which can be chosen by the user for each execution. *Round-robin mode*: If several processes are instantiated from a particular processor p, they are mapped to consecutive processors starting with the one numbered one more than p. *Random mode*: Each processor maps instantiated processes to randomly chosen processors. Notice that the £rst of them allows the programmer to control somehow the mapping of processes. In this paper, we will always assume that the *round-robin mode* is being used.

In the RTS, a dynamic channel is a tuple (p,i,a) where p denotes the processor identity, i is a unique identi£er within such processor identifying the channel, and a is the physical address of the closure where the values received through the channel will be saved, and it is used to actually read the received values.

1.3 METHODOLOGY TO DERIVE NON-HIERARCHICAL TOPOLOGIES

In [KPS00], an analysis detecting some bypassing situations was presented. Unfortunately, the analysis is not yet implemented. Moreover, it can only deal with programs where the names of all the particular channels are explicit. In particular, this means that it cannot deal with programs where a list of processes is instantiated by a unique parent process.

In this section we present a methodology to obtain non-hierarchical topologies using as speci£cation Eden programs whose real implementation is hierarchical. By using this methodology, it will be possible to implement any topology, without needing any kind of bypassing analysis. In the next section, several real topologies will be shown, clarifying how the methodology really works.

Following the classi£cation introduced in [KPS00], we will show how to deal with each of the different kinds of bypassing that may be needed in a program.

1.3.1 Bypassing between Siblings

Unfortunately, there are many situations where different siblings need to communicate amongst them, but they can only perform communications with its parent. A ring, a torus or a grid are examples where direct connections between siblings are needed. As it is not possible to directly connect two siblings by only using process abstractions and instantiations, dynamic channels will be needed. For each communication between siblings, the receiver of the data should create a

new dynamic channel, and it should send its name to the sender of the data. Afterwards, the sender of the data should use the received name of the channel in order to actually send the messages. Thus, the solution requires implementing the reverse topology, so that the receivers of the messages can send the names of the dynamic channels to the producers of the messages. For each process abstraction, a new one using dynamic channels is created following these steps:

- For each output `out` of type `t`, if it is to be bypassed:
  - Remove that output
  - Introduce a new input `cn` of type `ChanName t`
  - Send through `cn` the value that used to be sent through `out`
- For each input `inc` of type `t`, if it is to be bypassed:
  - Remove that input
  - Introduce a new output `ocn` of type `ChanName t`
  - Create a new dynamic channel `(cn,c)`
  - Send `cn` through `ocn`
  - Read from `c` the value that was expected to be received from `inc`

In the parent process it is necessary to create the reverse topology of the original one. In the case of regular topologies it is enough to shift in the opposite way the outputs of the processes, while in the general case more modifications are needed. In addition to obtaining the reverse topology, when the parent needs to communicate with the children through a channel that has been converted into a dynamic one, the parent has to be adapted by following these steps: (1) Create as many dynamic channels as there are values to be received from the children; (2) where a value from a child was used, the corresponding value of the dynamic channel is used; (3) instead of sending values to the children, the names of the new dynamic channels are sent; and (4) values are sent to the children through the names of the dynamic channels received from them.

1.3.2 Bypassing between Generations

There are two types of this kind of bypassing, depending on the direction of the communications: It can be necessary to send values directly from a descendant to an ancestor or vice versa.

From a descendant to an ancestor An example is the pipeline that will be presented in the next section. In the general case, for each channel to be bypassed a new dynamic channel is needed. In addition, the process abstractions need to be modified to transmit appropriately the name of that channel. For each process abstraction, a new one is created, where for each output `out` of type `t`, if it is to be bypassed these steps are performed:

- Remove that output.
- Introduce a new input `cn` of type `ChanName t`.
- In case this process abstraction is the real sender, send the value through `cn`, otherwise send `cn` to the appropriate child through the child's `cn` channel.

In the parent process, for each input channel `inc` to be bypassed from a descendant, the following steps are done:

- Create a new dynamic channel `c`.
- Send `c` to the child through the extra input channel created before.
- Read from `c` the value that was expected to be received from `inc`.

From an ancestor to a descendant This kind of bypassing rarely appears in Eden applications, but we include it for completeness. As an ancestor needs to send values directly to a descendant, the descendant has to create a dynamic channel and send the name to the ancestor, by using the intermediate processes to forward such name. In the original hierarchical program, the intermediate processes should have an extra input parameter to forward values from the ancestor to the descendant. This parameter will disappear, and it will be replaced by an output parameter in order to send the name of the dynamic channel created by the actual receiver of the messages. Summarizing, for each process abstraction, a new one is created, where for each input `inc` of type `t`, if it is to be bypassed the following steps are performed:

- Remove that input.
- Introduce a new output `ocn` of type `ChanName t`.
- In case the process was the real receiver of the input values, it must: (1) Create a new dynamic channel `(cn,c)`; (2) read from `c` the values that used to be read from `inc`; (3) send `cn` through the output `ocn`. Otherwise it only sends through `ocn` the name `cn` received from the corresponding `ocn` of its son.

The ancestor needs also to be modified. For each output `out` to be bypassed:

- The channel name `cn` will be received from the corresponding child.
- The value that was to be sent through `out` has to be sent now through `cn`.

1.4 EXAMPLES OF NON-HIERARCHICAL TOPOLOGIES

In this section we present four increasingly complex examples of non-hierarchical topologies, showing how to implement them in Eden.

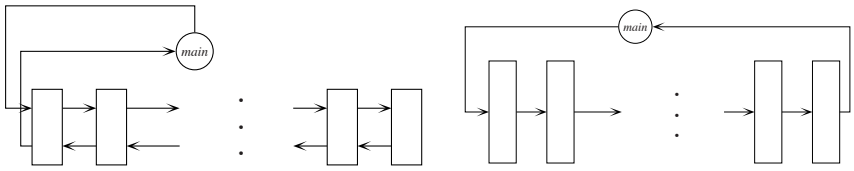

FIGURE 1.1. Topologies obtained with `pipe` (left), and with `pipeD` (right)

```
pipe :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipe [f] xs = process xs -> f xs
pipe fs xs  = (ppipe fs) # xs

ppipe :: Transmissible a => [[a]->[a]] -> Process [a] [a]
ppipe [f]    = process xs -> f xs
ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

FIGURE 1.2. Pipeline skeleton without dynamic channels

1.4.1 Pipeline Skeleton

Consider for instance a pipeline. This can be speci£ed in Eden as shown in Figure 1.2, where each process creates its successor process in the pipe. The topology obtained with this de£nition is not the desired one, but the one shown in Figure 1.1(left). Notice that the output of the last process is not sent to the main one. Instead, it is sent to the previous process, then the message is forwarded to the previous one, and so on (this is represented by dots in the £gure). As it was introduced in the methodology, to solve this problem, the main process of the pipe creates a dynamic channel, and the name of such channel is forwarded to the last process of the pipeline. By doing so, this last process will send messages directly to the main process, obtaining a real pipeline topology (see Figure 1.1(right)). Figure 1.3 shows the Eden program.

The similarities between both programs are remarkable. In fact, the second has been *derived* from the £rst one systematically. We were interested in performing a *bypassing between generations from a descendant to an ancestor*. Notice that to implement it manually by using dynamic channels we have followed the steps given by the methodology: (1) The process abstraction has been modi£ed by removing the *bypassed* output channel and by introducing a new input `cn` containing a dynamic channel. The £nal value is sent through `cn` when appropriate; and (2) the parent is modi£ed by creating a new dynamic channel, by sending it to the child through its new extra input parameter, and by reading the £nal value through the new dynamic channel.

1.4.2 Ring Skeleton

A ring is a well-known topology where each process receives values from its left neighbour and sends values to its right one, forming a ring. In addition to that, all the processes can communicate with the main one — See Figure 1.4. This topology is appropriate for uniform granularity algorithms in which the workers

```
pipeD :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipeD [f] xs = process xs -> f xs
pipeD fs xs = new (cn,c) let dummy = (ppipeD fs) # (xs,cn) in c

ppipeD :: Transmissible a => [[a] -> [a]] -> Process ([a], ChanName [a]) ()
ppipeD [f]    = process (xs,cn) -> cn !* (f xs) par ()
ppipeD (f:fs) = process (xs,cn) -> (ppipeD fs) # (f xs,cn)
```

FIGURE 1.3. Pipeline skeleton with dynamic channels

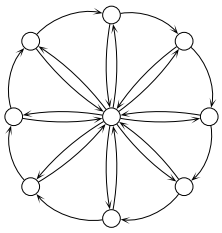

FIGURE 1.4. Ring topology

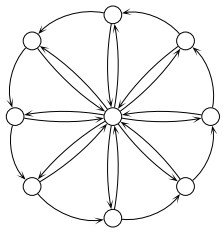

FIGURE 1.5. Ring topology reversed

at the nodes perform successive rounds. Before the £rst round, the main process sends the initial data to the workers. After that, at each round, every worker computes, receives messages from its left neighbour, and then send messages to its right neighbour. Eden's implementation uses lists instead of synchronization barriers to simulate rounds.

The `ring` function creates the desired topology by properly connecting the inputs and outputs of the different `pring` processes. As we want processes to receive values from its previous process, it is only necessary to shift the outputs of the list of processes before using them as inputs of the same list. Each `pring` receives an input from the parent, and one from its left sibling, and produces an output to the parent and another one to its right sibling. The £rst parameter is the worker function, which receives an initial datum of type `b` from the parent and a list `[a]` from the left neighbour, and it produces a result of type `[a]` for its neighbour and a £nal result of type `c` for its parent. Figure 1.6 shows the source code, where `mzip2` is a lazier version of `zip2`, needed to break the circular dependencies.

The problem with this approach is that, as said before, what is really created is a set of processes that are only connected to the main process. Therefore, the communications between neighbours are not direct, as they go through the main process, that becomes a bottleneck. This is a typical case of *bypassing between siblings*. Following our methodology, the solution is that, for each channel to be bypassed, a dynamic channel is created to establish the direct connection. The readers of the original channels should create the new dynamic channels and send the names to the writers. For doing that, it is necessary to send them through the parent process. As now the actual receivers of data need to send a value (the name of the channel) to the actual producers, the parent just need to establish the reverse connection topology of the original one.

In the ring case, the steps to be followed are these:

```
ring :: (Transmissible a,Transmissible b,Transmissible c) =>
        ((b,[a]) -> (c,[a])) -> [b] -> [c]
ring f input = outsToParent   where
    outs   =  [(pring f) # outA' | outA' <- outs']
    (outsToParent,outsA) = unzip outs
    outsA' = last outsA : init outsA
    outs'  = mzip2 input outsA'
pring ::(Transmissible a,Transmissible b,Transmissible c) =>
         ((b,[a]) -> (c,[a])) -> Process (b,[a]) (c,[a])
pring f = process (fromParent, inA) -> out
    where out = f (fromParent, inA)
mzip2 (x:xs) ~(y:ys) = (x,y) : mzip2 xs ys
mzip2 _ _            = []
```

FIGURE 1.6. Ring skeleton without dynamic channels

```
pring ::(Transmissible a,Transmissible b,Transmissible c) =>
         ((b,[a]) -> (c,[a])) -> Process (b,ChanName [a]) (c,ChanName [a])
pring f = process (fromParent,outChanA) -> out
    where out = new (inChanA, inA) let (toParent,outA) = f (fromParent,inA)
                                   in outChanA !* outA par (toParent,inChanA)
```

FIGURE 1.7. Ring skeleton with dynamic channels

- The output channel used to send values to the neighbour is replaced by an extra input parameter representing the dynamic channel to be used to communicate with its neighbour.
- Instead of using the former output channel, the same values are sent through the new dynamic channel received as input parameter.
- The input channel used to receive values from the neighbour is removed. Instead of it, a new dynamic channel is created locally, and its name is sent through a new output channel.
- Instead of using the former input channel, values are read from the newly created dynamic channel.

Thus, now each `pring` receives an input from the parent, and a channel name to be used to send values to its sibling, and produces an output to the parent and a channel name to be used to receive inputs from its sibling, as shown in Figure 1.7. Notice that the source code of the parent has not been modi£ed yet. As we need to obtain the reverse topology of the original one, if we do not modify it we will obtain the topology shown in Figure 1.5, where communications are in the opposite direction. To obtain the correct topology we only need to shift the outputs of the processes in the reverse way, by modifying `outsA'` de£nition:

```
outsA' = tail outsA ++ [head outsA]
```

1.4.3 Grid Skeleton

A grid is a two-dimensional topology where each process is connected to its four neighbours. The difference with a two-dimensional ring is that the £rst and last processes of each row and column are not neighbours. Moreover, nodes have

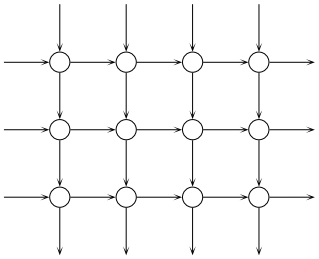

FIGURE 1.8. Grid topology

```
pgrid :: (Transmissible a,Transmissible b) =>
         (([a],[b])->([a],[b])) -> Process ([a],[b]) ([a],[b])
pgrid f = process (inA,inB) -> (outA,outB)
    where (outA,outB) = f (inA,inB)
grid :: (Transmissible a,Transmissible b) =>
        (([a],[b])->([a],[b])) -> ([[a]],[[b]]) -> ([[a]],[[b]])
grid f (insA,insB) = (outsA,outsB)   where
    nr = length insA  -- number of rows of the grid
    nc = length insB  -- number of columns of the grid
    outss           = [[(pgrid f) # outAB | outAB <- outs'] | outs' <- outss']
                      `using` (spine.concat)
    (outssA,outssB) = unzip (map unzip outss)
    outssA'         = mzipWith (:) insA (map init outssA)
    outssB'         = insB : init outssB
    outss'          = zipWith zip outssA' outssB'
    outsA           = map last outssA
    outsB           = last outssB
```

FIGURE 1.9. Grid skeleton without dynamic channels

not the two extra connections to send/receive values to/from the parent: Only the nodes on the £rst row or column have an input from the parent, and only the nodes on the last row or column have an output to the parent —See Figure 1.8. So, the main process is considered a neighbour of those nodes. At each round, every worker receives messages from its left and upper neighbours, computes, and then send messages to its right and lower neighbours. Eden's implementation (Figure 1.9) uses lists instead of synchronization barriers to simulate rounds. The `grid` function creates the desired topology by properly connecting the inputs and outputs of the different `pgrid` processes. Each `pgrid` receives two inputs from its siblings, and produces two outputs to its siblings.

In the Eden program, the number of rows *n* and columns *m* of the grid are inferred from the lengths of the input lists. The £rst parameter is the worker function, which receives a list `[a]` from the left neighbour and a list `[b]` from its upper neighbour. It produces results `[a]` and `[b]` for its neighbours.

Now, the process abstractions are modi£ed following exactly the same four steps as in the previous case. Thus, each `pgrid` receives two channel names to be used to send values to its siblings, and produces two channel names to be used to receive inputs from its siblings (see Figure 1.10). The `grid` function needs also to be modi£ed. As in the previous case, the reverse topology has to be created, shifting in the opposite way the outputs of the processes. But now,

```
pgrid :: (Transmissible a, Transmissible b) => (([a],[b])->([a],[b])) ->
         Process (ChanName [a], ChanName [b]) (ChanName [a], ChanName [b])
pgrid f = process (outChanA,outChanB) -> something
    where something = new (inChanA,inA) new (inChanB,inB)
                      let (outA,outB) = f (inA,inB) in
                      outChanA !* outA par
                      outChanB !* outB par
                      (inChanA,inChanB)
```

FIGURE 1.10. Each process of the grid skeleton using dynamic channels

```
grid :: (Transmissible a,Transmissible b) =>
        (([a],[b])->([a],[b])) -> ([[a]],[[b]]) -> ([[a]],[[b]])
grid f (insA,insB) = everything   where
    nr = length insA  -- number of rows of the grid
    nc = length insB  -- number of columns of the grid
    outss           = [[(pgrid f) # outAB | outAB <- outs'] | outs' <- outss']
                      `using` (spine.concat)
    (outssA,outssB) = unzip (map unzip outss)
    outssA'         = mzipWith (++) (map tail outssA) (map (:[]) outsAToParent)
    outssB'         = tail outssB ++ [outsBToParent]
    outss'          = zipWith zip outssA' outssB'

    -- The parent creates new channels to receive values from
    -- the last column and the last row children of the grid
    channelsA             = generateChannels nr
    channelsB             = generateChannels nc
    (outsAToParent,outsA) = unzip channelsA
    (outsBToParent,outsB) = unzip channelsB

    -- The parent sends values to the first column and first row of the grid,
    -- and waits until receiving values from the last column and last row
    everything = sendInitAs
    sendInitAs = sendValues (zip insB (head outssB)) sendInitBs
    sendInitBs = sendValues (zip insA (map head outssA)) (outsA,outsB)

-- Generates and returns a list of dynamic channels
generateChannels 0 = []
generateChannels n = new (cn,c) ((cn,c):generateChannels (n-1))

-- Sends a list of values through their dynamic channels, and continues with e
sendValues [] e           = e
sendValues ((v,ch):more) e = ch !* v par sendValues more e
```

FIGURE 1.11. Parent process of the grid skeleton using dynamic channels

as the `pgrid` processes only use dynamic channels, and the main process needs to communicate with them, it is necessary to adapt it to use dynamic channels. This is done with four steps: (1) creating as many dynamic channels as values are to be received from the children; (2) everywhere a value from a child was to be used, the corresponding value of the dynamic channel is used; (3) instead of sending values to the children, the names of the new dynamic channels are sent; and (4) values are sent to the children through the names of the dynamic channels received from the children.

Figure 1.11 shows the source code of `grid`. Note that the main structure is still the same as before, and that it has been straightforward to add the extra code.

1.5 CONCLUSIONS AND FUTURE WORK

Even though Eden process abstractions and instantiations only allow the development of hierarchical topologies, we have presented a methodology that allows

the obtaining of non-hierarchical ones. The hierarchical implementations are used as high-level speci£cations, that are re£ned into a lower lever implementation by using dynamic channels. By providing both levels in the same language we can easily transform the programs. Moreover, the lower level will only be used in the critical points, and it will only be used by following a clear methodology, avoiding the necessity of handling the typical gory details of the low-level features.

The usefulness of the method has been proved with three typical examples of non-hierarchical topologies, namely a pipeline, a ring and a grid. In [PRS01], satisfactory actual speedups has been obtained for a real application using a torus, outperforming by a factor of three the results obtained with the best implementation that uses a hierarchical topology.

As future work we plan to implement an automatic transformation to help obtaining non-hierarchical topologies. The programmer should only need to annotate appropriately the program with the bypassing information that will be used by the transformation tool.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for valuable comments on a draft version of this paper. This work was partly supported by the Spanish project TIC2000-0738 and the Spanish-British Acción Integrada HB 1999-0102.

REFERENCES

[BKL+98] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *Implementation of Functional Languages, IFL'97*, pages 250–269. LNCS 1467. Springer-Verlag, 1998.

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language De£nition and Operational Semantics. Technical Report, Berich 96-10. Revised version 1998, Philipps-Universität Marburg, Germany, 1998.

[GBDJ94] A. Geist, Ad. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[KOP99] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. Selected Papers*, pages 103–119. LNCS 1595, 1999.

[KPS00] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In *Trends in Functional Programming (Selected papers of the 1st Scottish Functional Programming Workshop, SFP'99)*, pages 2–10. Intellect, 2000.

[Mes94] Message Passing Interface Forum. MPI: A Message-passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4), 1994.

[Pey96] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *ESOP'96, LNCS 1058*, 1996.

[PRS01] R. Peña, F. Rubio, and C. Segura. Deriving Non-Hierarchical Process Topologies. In *Draft Proceedings of the 3rd Scottish Functional Programming Workshop, SFP'01)*, pages 157–168, 2001.