# High-level Parallel and Concurrent Programming in Eden*

S. Breitinger, R. Loogen
Philipps-Universität Marburg†

Y. Ortega-Mallén, R. Peña-Marí
Universidad Complutense de Madrid‡

**Abstract**

Parallel and concurrent programming is commonly handled at a low level of abstraction. Even concurrent declarative languages which are especially appropriate for symbolic computation provide only primitive constructs for that purpose. In contrast, the functional concurrent language Eden is an extension of the lazy functional language Haskell by *high level* constructs for the explicit specification of dynamically evolving process systems.

The paper contains a detailed investigation of the expressiveness of Eden. In particular, we study commonly used coordination constructs and show the modular design of concurrent systems by two case studies.

## 1 Background

During the last decade several extensions to functional programming languages have been proposed in order to facilitate the specification of explicitly concurrent applications, as e.g. reactive and distributed systems, in a functional framework. Among the most well known are Facile [7], Concurrent ML [13], Erlang [10], and Concurrent Haskell [12]. These concurrent functional languages provide primitive constructs for the dynamic creation of concurrent processes (spawn or fork) and the exchange of messages between such processes (send and receive). Thus, concurrency is handled at a low level of abstraction which violates the flavour of functional languages.

A delicate problem in concurrent functional languages is the appropriate treatment of choice and nondeterminism which are intrinsic in reactive systems. Whenever nondeterminism is introduced, referential transparency will be lost and consequently the equational reasoning techniques for functional languages cannot be used any more. In order to prevent the nondeterminism that arises from concurrency from pervading the whole language, a clean distinction between nondeterministic activities and deterministic computations is necessary. Concurrent Haskell uses e.g. IO monads to model processes and concurrent activities and thus encapsulates concurrency completely in monads. Even deterministic computations which are to be executed in parallel have to be defined as IO monads.

A completely different approach is that of exploiting *implicit parallelism*, where existing sequential languages are speeded up by automatic parallelization. In many cases annotations are introduced in order to optimize the parallel implementation with regard to the

*Supported by the DAAD (Deutscher Akademischer Austauschdienst) and the Spanish Ministry of Education and Science in the context of the German-Spanish Acción Integrada n.142B.

†Fachbereich Mathematik, Fachgebiet Informatik,Hans Meerwein Straße, Lahnberge, D-35032 Marburg, Germany, {breiting,loogen}@informatik.uni-marburg.de

‡Sec. Dept. de Informática y Automática, Facultad de C.C. Matemáticas, E-28040 Madrid, Spain, {yolanda,ricardop}@eucmax.sim.ucm.es

granularity of processes and the placement of processes on parallel processors. Typical examples are para-functional programming [8] and Concurrent Clean [14]. The annotations are semantically transparent, i.e. the semantics of a program with annotations is identical to the semantics of the program without annotations, a fact that shows the drawbacks of this approach with respect to language expressivity.

The *concurrent constraint programming* paradigm also introduces implicit parallelism and makes heavy use of shared information in the form of a global constraint store. The language Goffin [5] uses constraints for coordination of concurrent activities, but leaves the mapping to parallel processes quite implicit.

Our language Eden [4, 3] differs from the above-mentioned approaches to the integration of concurrency into functional languages in the following aspects:

- The concepts of processes, communication and synchronisation are more abstract and expressive, thus simplifying the task of concurrent programming.
- Nondeterminism is encapsulated into processes, but the separation between the functional world and the process world is less restrictive. The type inference system marks nondeterministic processes, the results of which cannot be used in the bodies of functions, in order to retain referential transparency.
- While most concurrent functional languages assume a shared memory, Eden is tailored for distributed systems.

Eden extends the lazy functional language Haskell [9] by syntactic constructs for *explicitly* defining processes. These processes communicate by exchanging values via communication channels modelled by lazy lists. Communication is asynchronous and *implicit*, i.e. exchange of messages is done automatically and need not be specified by the programmer using send and receive commands. Abstracting from interprocess communication with streams is also the underlying paradigm of the Caliban system [11] which can be used to program static process networks in a functional framework. In contrast to Caliban, Eden supports dynamic process networks and incorporates special concepts for the efficient treatment of general reactive systems, i.e. systems which maintain some interaction with the environment and which may be time-dependent.

*Paper outline.* Section 2 contains a short introduction into the key concepts of Eden. The expressiveness of Eden is investigated in Section 3 by studying general coordination constructs and by comparing Eden with Concurrent Haskell which we have chosen as a representative for other concurrent functional languages. In Section 4 it is shown that complex programs can be developed in a modular and easily comprehensible way. Finally we draw conclusions.

# 2 Eden's essentials

## Processes

A process that maps inputs $in_1, \ldots, in_m$ to outputs $exp_1, \ldots, exp_n$ is specified in a functional way by a *process abstraction* expression of the following form:

```
process (in_1,...,in_m)->(exp_1,...,exp_n)
where equation_1 ... equation_r
```

with type `Process` $(\tau'_1, \ldots, \tau'_m)$ $(\bar{\tau}_1, \ldots, \bar{\tau}_n)$, where `Process` is a predefined binary type constructor, and $\tau'_1, \ldots, \tau'_m$ and $\bar{\tau}_1, \ldots, \bar{\tau}_n$ are the types of the inports and outports respectively. The optional `where` part is used to define auxiliary functions and common subexpressions which occur within the process definition.

A process may have as input (respectively, output) a single channel, a tuple of channels, or a list of channels. If a channel has type `[a]` (i.e. it is a list), it will be treated as a stream, transmitting its contents element by element. The annotation `<a>` expresses that `a` is a channel. These annotations do not introduce new types, but are used by the compiler to infer types correctly. For instance, `[<a>]` means "list of channels", used to define process abstractions with an unknown number of channels.

**Example.** A process which merges two sorted input streams into a single sorted output stream is specified by the following process abstraction:

```
merger :: Process ([a],[a]) [a]
merger = process (s1,s2) -> smerge s1 s2
         where smerge [] l = l
               smerge l [] = l
               smerge (x:l) (y:t) = if x<=y then x:smerge l (y:t)
                                            else y:smerge (x:l) t
```

Process creation takes places at *process instantiation*, i.e. at the application of a process abstraction to a tuple of input expressions, yielding a tuple of output channels for the newly created process.

$$\boxed{(out_1, \ldots, out_n) = p \ \# \ (input\_exp_1, \ldots, input\_exp_m)}$$

**Example.** A dynamic sorting network can be created by instantiating the `merger` process abstraction given in the previous example:

```
sortNet :: Process [a] [a]
sortNet = process list -> sort list
          where sort [] = []
                sort [x] = [x]
                sort xs = merger # (sortNet # l1, sortNet # l2)
                          where (l1,l2) = unshuffle xs
                unshuffle [] = ([],[])
                unshuffle [x] = ([x],[])
                unshuffle (x:y:t) = (x:t1,y:t2) where (t1,t2) = unshuffle t
```

In Eden the duplication of running processes is neither desirable nor possible because they are dynamic entities with an internal state. The type of a process instantiation is simply the type of the results generated. There is no type that represents a *running* process. However, process abstractions are first class values which can be passed as arguments to functions or other process abstractions.

Basically, the whole computation of a process is driven by the evaluation of its outport expressions, for which there is always demand. This rule overrides normal lazy evaluation in favour of parallelism. There will be a separate concurrent thread of execution for each output channel. Thus, we find in Eden two levels of concurrency: the concurrent or parallel evaluation of processes and the concurrent evaluation of different threads within processes. Furthermore, each process immediately evaluates all *top level* process instantiations. This

may lead to speculative parallelism and again overrules lazy evaluation to some extent, but speeds up the generation of new processes and the distribution of the computation.

A user process with all its outputs closed will terminate immediately. On termination its input channels will be eliminated and the corresponding outports in the sender processes will be closed.

## Communication and synchronization

At the abstract level communication between processes is asynchronous and $1 : n$. Only fully evaluated objects are transferred between processes. The only exception are *streams* (i.e. list channels), whose transmission is done piecewise and does not have to be delayed until the whole list of values is computed. As soon as a value of the corresponding type has been transmitted in full, the channel is closed and the communication port is abolished. The transfer of information via communication channels is done automatically, without explicit send or receive commands.

Interprocess synchronization is performed exclusively by the exchange of information via channels. The evaluation of outports is independent of the consumption of the produced values, but a concurrent thread will be suspended if it needs some value from an inport whose channel buffer is empty.

## Reactive systems

The language constructs presented so far are sufficient for the programming of deterministic systems. Additional extra-functional concepts lead to a language with enhanced expressive power, so that reactive concurrent systems can be defined as well.

*Nondeterminism* is introduced in Eden by a predefined process abstraction `MERGE` which creates a nondeterministic *fair* merging process for a list of stream channels:

```
MERGE :: Process [<[a]>] [a]
```

In order not to destroy referential transparency at the functional level, we restrict the use of `MERGE` and non-deterministic processes in general[1]. The rule is that functions can never instantiate a non-deterministic process in order to compute their result. Otherwise, they would become non-deterministic functions, an undesirable feature. Under certain restrictions, the non-determinism condition can be statically checked by the type inference system.

Eden also provides *dynamic reply channels*. A process may generate a new input channel and send a message containing the name of this new channel 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, and it should be ensured by the programmer that each dynamically created channel is used to establish a one-to-one connection between a unique writer process and the generator of the channel.

We introduce a new unary type constructor `ChanName` for the names of dynamically created reply channels:

```
new (ch_name, chan). exp
```

[1] A process is called nondeterministic, if it contains an instantiation of `MERGE` or of another nondeterministic process. Note that `MERGE` is the only source of nondeterminism.

The above expression declares a new channel name *ch_name* as reference to the new input channel *chan*. The scope of both is the body expression *exp*. The name is sent to another process to establish the communication. A process receiving a reply channel name *ch_name* and wanting to reply through it, uses an expression of the following form:

$$\boxed{ch\_name \texttt{ !* } expression_1 \texttt{ par } expression_2}$$

Before $expression_2$ is evaluated, a new concurrent thread for the evaluation of $expression_1$ is generated, whose result is transmitted via the received reply channel.

### Semantics and Implementation

The operational semantics of Eden is an extension of the standard operational semantics of Haskell and reflects the distinction between the computation and coordination sublanguages of Eden. It comprises two levels of transition systems: The lower level handles effects which are locally restricted to one process. The upper level describes global effects on the whole process system. The interface between these two levels consists of so-called 'actions' which communicate the need for global events to the upper level, e.g in the case of process generation, message exchange or process termination (see [4] for further details).

A prototype implementation of Eden on top of Concurrent Haskell exists [2]. Eden programs are transformed in order to model communication and other types of interprocess interaction by using explicitly the low-level primitives of Concurrent Haskell. A distributed implementation is currently under development.

## 3 The expressiveness of Eden

### 3.1 Comparison to different coordination mechanisms

In this section, we will shortly comment on typical communication and coordination paradigms used in imperative concurrent languages. It is not our goal to perform a simulation of imperative constructs in the declarative language Eden, but it is an interesting point in favour of Eden that the full range of these paradigms is expressible.

**Asynchronous and synchronous communication.** *Asynchronous message passing* is of course the typical communication paradigm that is used in Eden. *Synchronous message passing* which differs from the asynchronous version in that the send operation is blocking, can be implemented by exchanging explicit acknowledgement messages using a second channel from the receiver to the sender. With dynamic topologies, the sender includes the name of a reply channel in each message and only continues after it has received the acknowledgement via this reply channel:

```
Sender:   to_partner = new(chan, cont).(messageData, chan):wait cont
          wait (Ack x) = continue
Receiver: answer (data, chan2) = chan2 !* (Ack y) par work data
```

**Rendezvous and remote procedure call.** These two mechanisms form more powerful ways of synchronous communication, which allow for two-way communication between a calling and a servicing process. In both cases, the caller is blocked until the *result* of the invocation arrives. This can be easily implemented by modifying the synchronous message

passing example shown above: instead of waiting for an acknowledgement, the sender now has to wait for the result.

With rendezvous communication the service is provided by an existing thread of control and with remote procedure call by a new one. Consequently, one obtains a rendezvous solution when choosing communication via static communication channels and a remote procedure call solution when using dynamic reply channels instead. Note, however, that these two communication paradigms will not be the paradigms typically used in Eden programs, because they introduce significant communication overhead due to synchronicity.

**Nondeterministic choice.** In languages with synchronous communication, such as Ada and Occam, a choice statement is usually provided to non-deterministically select a communication between all those available at a particular moment. A general choice statement combining both input and output commands in guards has been proposed in many notations as a powerful way to express some solutions, but no practical language includes it because it is very costly to implement in distributed environments. Usually, only a restricted form of choice —in which every guard consists of a boolean expression followed by an input (or accepting) command— is provided, but it considerably limits expressiveness [12]. Furthermore, choice statements are more oriented to an imperative programming style. Its inclusion in our language would not provide more expressivity and would complicate the semantics by introducing another source of nondeterminism. Nevertheless, typical applications of choice can easily be expressed in Eden. For instance, *nondeterministic input* can be modelled using a `MERGE` process to merge the incoming requests, as it is illustrated in the disk scheduler example of Section 4. This example also shows how to handle a pool of equivalent servers without resorting to *nondeterministic output*. Boolean guards in choice can be used to defer a client task *permission* to proceed until the server reaches a convenient state. In Eden, a permission to proceed is considered an output from the server to the client and can be sent at any convenient time by the server process.

## 3.2 Eden vs. Concurrent Haskell

Concurrent Haskell is another extension of the lazy functional language Haskell which has been developed in the spirit of the previous languages Facile and Concurrent ML, which both are extensions of ML employing synchronous communication. There are a lot of subtle differences between these concurrent functional languages. But in comparison with Eden's more abstract approach to concurrency, these languages are very similar. As Concurrent Haskell is the most recent design along these lines, and in addition it is also based on Haskell and employs asynchronous communication, we decided to use this language for a more detailed comparison.

Concurrent Haskell uses IO monads to describe concurrent behaviour. It adds the following primitives to Haskell:

- `forkIO :: IO () -> IO ()` starts a new concurrent process.
- The new primitive type `MVar a` is used to introduce mutable locations which may be empty or contain a value of type a. Primitive operations are provided for manipulating new `MVars`.

On top of these primitives more abstract concurrency constructs can be defined, e.g. a channel with unbounded buffering the interface of which is as follows (see [12]).

```
type Channel a
newChan :: IO (Channel a)
putChan :: Channel a -> a -> IO ()
getChan :: Channel a -> IO a
```

Channels model infinite streams of values. In order to work with finite streams one has to extend the type of transmitted values by a special endmarker which indicates the closing of a stream:

```
type FiniteStrm a = Channel (Message a)
data Message a = EndMarker | Contents a
readFChan ::  FiniteStrm a -> IO [a]
writeFChan ::  FiniteStrm a -> [a] -> IO ()
```

Although more abstract communication constructs can be defined using Concurrent Haskell's primitives, it turns out that the programmer remains responsible for the creation of channels and the communication between processes using read and write operations on channels. Below a Concurrent Haskell version of the sorting network example of Section 2 is given, which reveals the explicit management of concurrent behaviour.

```
sortNet ::  Channel a -> Channel a -> IO ()
sortNet listChan slistChan
    = readFChan listChan >>= \ list ->
      case list of
        [] -> writeFChan slistChan []
        [x] -> writeFChan slistChan [x]
        _ -> let (l1,l2) = unshuffle list
             in newChan >>= \ l1chan ->
                newChan >>= \ l2chan ->
                newChan >>= \ sl1chan ->
                newChan >>= \ sl2chan ->
                forkIO (sortNet l1chan sl1chan) >>
                forkIO (sortNet l2chan sl2chan) >>
                writeFChan l1chan l1 >>
                writeFChan l2chan l2 >>
                merger (sl1chan,sl2chan) slistChan
merger ::  (Channel a, Channel a) -> Channel a -> IO ()
merger (s1,s2) out = readFChan s1 >>= \ l1 ->
                     readFChan s2 >>= \ l2 ->
                     writeFChan out (smerge l1 l2)
```

Programmers using Eden are liberated from the task of explicitly sending messages to a receiver specified during compile time. The process instantiation:

```
merger # (sortNet # l1, sortNet # l2)
```

specifies the instantiation of three processes with the corresponding interconnecting channels. In Concurrent Haskell nine IO actions are necessary in order to specify the same straightforward behaviour.

In Eden, the data contained in the outputs of one process is transparently transferred to the respective inputs of the receiver(s). Channel buffers contents do not need to be constructed message by message and higher order functions such as `map`, `filter` and `zipWith` can be conveniently used. The evaluation mechanism and the implicit buffers provided by the implementation take care of the execution. The name of the receiver will in any case be inferred from the process instantiations and not from the process abstractions.

# 4 Modular development of concurrent systems

## 4.1 A synchronous heartbeat algorithm: the traveling salesman

A system with a manager process which is responsible for the distribution of work to a set of worker processes can be organized as a deterministic system by adopting a synchronous heartbeat scheme controlled by the cycles of the manager process (cf. [1]). In each cycle the manager distributes work to every worker process, thereafter it collects the results from the workers, updates its internal state and tests whether there is more work to distribute or not. In the following we develop such a system to solve the traveling salesman problem with a parallel branch and bound algorithm.

Given is a completely connected graph of $n$ cities. The problem is to find an order of visits of each city (a circular path), minimizing the total travelled distance. The idea is to maintain a set of partial solutions shared by the $m$ workers. Initially, the set contains $n-1$ partial solutions representing the different edges starting at city 1. Each worker repeatedly takes a partial solution from the shared set and tries every possibility of extending it with a city that has not yet been visited in that path. A cost function taking into account the actual length of the extended path, and an underestimation of the cost of further extending it up to a complete solution is applied and compared with the actual cost of the best complete solution found so far. If the new path turns out to be too costly, the worker discards it. If not, it is added to the set of partial solutions, or used to update the best solution in case it was a complete one. Thus the best path found so far is used as an upper bound to prune the search tree so that only promising paths are examined. As an initial value, a greedy solution computed in polynomial time could be used.

Let us give now the corresponding master-worker system in Eden. The program defines a function `salesman` that, given the graph and the number of desired workers as parameters, returns a solution path together with its cost. To compute the solution, the `salesman` instantiates a `manager` process and the given number of `worker` processes. This is done with the help of a skeleton *farm* [6]:

```
farm :: Process a b -> Process [<a>] [<b>]
farm p = process xs -> f xs
         where f []     = []
               f (x:xx) = (p # x) : f xx
type Path     = [Int]          -- sequence of visited cities
type Solution = (Path,Int)     -- visited cities, (estimated) cost of the solution
type Task     = (Path,Int)     -- sequence to be extended, best cost up to now
salesman :: Graph -> Int -> Solution
worker   :: Graph -> Process [Task] [[Solution]]
manager  :: Graph -> Int -> Process [<[[Solution]]>] ([<[Task]>],Solution)
salesman g m = s where (tasks,s) = manager g m # results
                       results   = farm (worker g) # tasks
```

Function `f` in the farm skeleton instantiates a copy of process `p` for every channel in the input list of `farm`. In our case, the manager is responsible for generating a `tasks` channel list of length $m$. Workers do exist as long as there is some work to do, repeatedly taking a partial path, together with the current best solution cost, and producing a (possibly empty) list of extended plausible paths, together with their estimated costs. We will not show workers, but concentrate on the manager, which maintains a set `pend` of pending partial solutions, the best complete solution `bSol` found so far, and the cost `bCost` of this solution.

```
manager g m = process results -> cycle g m iniPending iniCost iniSol results where
    (iniSol,iniCost) = greedySol g
    iniPending       = [ [1,i] | i <- [2..n]]  -- Initial one-edge paths
    n = numVertex g
    cycle :: Graph -> Int -> [Path] -> Int -> Path -> [[[Solution]]]
             -> ([[Task]],Solution)
    cycle g m pend bCost bSol solss    = (tasks,sol) where
          (tasks, restPend, w)         = distribute m pend bCost restTasks
          (rawSols, restSolss)         = collect w solss
          (newPend, newBCost, newBSol) = update restPend rawSols bCost bSol
          (restTasks, sol)  = if null newPend
                              then ( take m (repeat []), (newBSol,newBCost))
                              else cycle g m newPend newBCost newBSol restSolss
```

The whole system is synchronized and executes in a heart-beat way determined by the cycles of the manager. In every cycle the manager distributes pending partial solutions `tasks` to each worker, accompanied by the current value of `bCost`. When there are not enough pending solutions, only the first `w` workers will receive a task. Thereafter the manager collects all the solutions `rawSols` produced by the workers. Complete solutions are compared to `bSol`, which is updated correspondingly. Partial solutions are added to the pending ones, and the whole set is filtered, keeping only those solutions whose estimated cost is below `newBCost`. Moreover, it is convenient to organize pending partial solutions by increasing order of estimated costs, in such a way that the most promising ones are expanded first. All this is done by function `update`, the details of which are not shown. When there are no more pending partial solutions to be expanded, the cycle is ended and the final solution is given. The manager will therefore close every connection to the workers, and the whole system finishes. We show below the details of functions `distribute` and `collect`:

```
distribute :: Int -> [Path] -> Int -> [Task] -> ([[Task]],[Path],Int)
distribute m pend bC rtasks = (tasks,rpend,w) where
          w           = min m (length pend)
          (ps,rpend)  = splitAt w pend
          tasks       = zipwith paste (ps++repeat []) rtasks where
                        paste [] t = t
                        paste p  t = (p,bC):t
collect    :: Int -> [[Solution]] -> ([Solution],[[Solution]])
collect w rss = (rs,tail++rss2) where
  (rss1,rss2) = splitAt w rss
  (rs,tail)   = transpose [(x,xs) | x:xs <- rss1]
```

## 4.2 A client/server system: a disk scheduler

A set of users needing exclusive access to a moving head disk, address requests to a disk scheduler to access specific tracks. The scheduler stores the requests and, whenever the

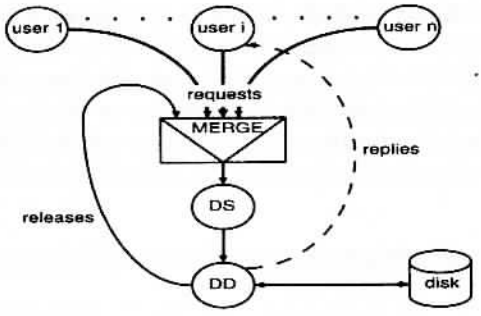


Figure 1: Disk scheduling process topology

disk is not busy, grants permission to the user whose track is nearest in the current scanning direction. Requests to access tracks above the current track are stored in the so-called *foreground* queue, whereas requests to access the current track or below it are stored in a *background* queue. When the foreground queue is exhausted, the scheduler switches to the background queue. Queues are scanned always in track ascending order (*circular scan strategy* [1]).

This example is a typical instance of a reactive system that models the time-dependent interaction of concurrent processes. The Eden solution is depicted graphically in Figure 1. There, we provide two processes to serve user requests: the $DS$ (*disk scheduler*) process, which receives user requests and stores them in track ascending order, and the $DD$ (*disk driver*) process, which actually serves requests. Process $DS$ transfers the next request to be served to process $DD$, which is supposed to do the physical connection with the disk. The response associated to each request is sent directly from process $DD$ to the user involved. The Eden dynamic reply channel facility is used for this purpose. In the drawing, this communication is represented by a dashed line. Moreover, there is a feedback communication from $DD$ to $DS$ to inform the latter by a *release message* that the disk is not longer busy. The nondeterminism in the arrival of user requests —and release messages from $DD$— to process $DS$, is modelled by connecting all these channels to a `MERGE` process which interleaves the signals into a single channel.

```
data Signal     = Request Int ParamsTrans (ChanName ReplyTrans) | Release
data Trans      = Transac ParamsTrans (ChanName ReplyTrans)
data ReplyTrans = Reply ParamsReply
type User       = Process ()       [Signal]
type DiskSched  = Process [Signal] [Trans]
type DiskDriver = Process [Trans]  [Signal]
user            :: User
diskSched       :: DiskSched
diskDriver      :: DiskDriver

user =   process () -> requests where
           (iniTrack, iniTrans) = ...
           requests = restOfTransactions iniTrack iniTrans
           resOfTransactions track trans =
             new (name,reply) . Request track trans name : waitReply reply
           waitReply (Reply replyParams)
             let (nextTrack, nextTrans) = changeState replyParams
```

```
            in  restOfTransactions nextTrack nextTrans
diskSched = process fromEnv -> cycle 0 True emptyQ  emptyQ fromEnv  where
            cycle ctrack free fq bq (s:ss) =
               case s of
                 Request track trans name ->
                   if free
                   then Transac trans name : cycle track False fq bq ss
                   else if track > ctrack
                     then cycle ctrack free (add fq (track,trans,name)) bq ss
                     else cycle ctrack free fq (add bq (track,trans,name)) ss
                 Release ->
                   if empty fq && empty bq
                   then cycle ctrack True emptyQ emptyQ ss
                   else if empty fq
                     then let (t,trans,name) = min bq
                          in Transac trans name : cycle t False (elim bq) fq ss
                     else let (t,trans,name) = min fq
                          in Transac trans name : cycle t False (elim fq) bq ss
diskDriver = process transacs -> executeTransaction transacs where
             executeTransaction (Transac params destinUser : ts) =
               let results = accessToDisk params --here accesses to disk are done
               in  destinUser !* Reply results par Release : executeTransaction ts
```

Users loop first producing a request, and then waiting for the reply. The request indicates the track to be accessed and the parameters associated to the transaction, and includes a channel name to receive the reply. The disk scheduler `diskSched` implements the circular scan strategy mentioned above. For this it records the current track being served, an indication of whether the disk is free or busy, and the foreground and background queues of pending requests. The `diskDriver` serves transactions in turn, sending back the result directly to the requesting user. The instantiation of the whole network generates a *closed* process system, the type of the result is `Process () ()`, i.e. no special value is returned.

```
instantiate :: Int -> User -> DiskSched -> DiskDriver -> Process () ()
instantiate n u ds dd = process () -> () where
      requests = [ u # () | i <- [1..n]]
      fromEnv  = MERGE # releases:requests
      transacs = ds # fromEnv
      releases = dd # transacs
```

# 5 Conclusions

The exploitation of the parallelism inherent to an algorithm is performed more efficiently by the programmer than by a compiler. But the added programming complexity which results from this design decision must be alleviated by providing a high level of abstraction. Eden integrates concurrent and functional programming in a uniform way, maintaining such a high level of abstraction. In particular, nondeterminism, which implies a loss of referential transparency, can be confined to a small part of a reactive program. The compiler can precisely identify these parts and thus infer where equational reasoning can be applied. Moreover, our language supports modular design by making explicit in the process abstraction header the external interface, corresponding to the process specification, but hiding inside the implementation the internal interface, i.e. the communications with subordinated processes. Eden's capacity to dynamically create processes and reply channels

allows for the creation of arbitrary topologies which can change at execution time. The higher order facilities of functional languages, together with the classification of process abstractions as first class values, lead to a programming style in which process schemes can be represented by generic skeletons [6]. This facilitates the concurrent programming task as programmers will only be concerned with supplying the component processes. The integration does not imply a loss of expressivity. We have shown in Sections 3 and 4 how Eden constructs can cope with typical concurrency problems and how they can simulate other coordination paradigms.

# References


[1] Gregory R. Andrews. *Concurrent programming - principles and practice.* Benjamin Cummings, Redwood City, Calif., 1991.

[2] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. Implementation Aspects of Eden. In Werner Kluge, editor, *Implementation of Functional Languages, Bonn, Germany, 1996.* Springer LNCS, to appear, 1997.

[3] Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Towards a Declarative Language for Concurrent and Parallel Programming. In David N. Turner, editor, *Functional Programming, Glasgow 1995.* Springer Verlag, 1996.

[4] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. Technical report, Philipps-Universität Marburg, 1995.

[5] Manuel M.T. Chakravarty, Yike Guo, and Martin Köhler. Goffin: higher order functions meet concurrent constraints. *First International Workshop on Concurrent Constraint Programming, Venice, Italy,* 1995.

[6] Luis A. Galán, Cristóbal Pareja, and Ricardo Peña. Functional Skeletons Generate Process Topologies in Eden. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP),* number 1140 in Springer LNCS, 1996.

[7] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Programming,* 18(2), 1989.

[8] Paul Hudak. Para-Functional Programming in Haskell. In Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[9] Paul Hudak and Phil Wadler (editors). Report on the programming language Haskell: a non-strict, purely functional language. *SIGPLAN Notices,* 27(5):1–162, 1992.

[10] M.C. Williams J.L. Armstrong and S.R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[11] Paul Kelly. *Functional Programming for Loosely Coupled Multiprocessors.* Pitman, 1989.

[12] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on Principles of Programming Languages (POPL) 96.* ACM Press, 1996.

[13] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation,* pages 293 –305, 1991.

[14] M. C. J. D. van Eekelen, E. G. J. M. H. Nocker, M. J. Plasmeijer, and J. E. W. Smetsers. Concurrent Clean. Technical Report 89-18, University of Nijmegen, 1989.