Fuji Int. Workshop on Functional and Logic Progr., World Scientific Publ., 1995

Concurrency in Functional and Logic Programming

Silvia Breitinger, Rita Loogen
Philipps-Universität Marburg, Fachbereich Mathematik, Fachgebiet Informatik
Hans Meerwein Strasse, D-35032 Marburg, Germany
E-mail: {breiting,loogen}@informatik.uni-marburg.de

and

Yolanda Ortega-Mallén
Universidad Complutense de Madrid, Sec. Dept. de Informática y Automática
Facultad de C.C. Matemáticas, E-28040 Madrid, Spain
E-mail: yolanda@dia.ucm.es

ABSTRACT

This paper gives an overview of the main streams of research on concurrency in declarative languages and motivates the definition of the new concurrent functional language EDEN. Proceeding from the premise that parallelism should be expressed explicitly we have equipped the language with an explicit representation of processes and process abstractions. It is based on a lazy functional language and offers dynamic creation of processes and communication channels and a safe treatment of nondeterminism.

1. Introduction and Terminology

Today the majority of parallel programs is written in imperative languages. This programming paradigm forces the programmer to use low-level constructs for synchronization and interprocess communication and it offers no support for the verification of programs.

In contrast to imperative languages, declarative languages have the advantage that programs are able to retain parallelism inherent in the problem specification. Functional and logic programs can be evaluated sequentially or in parallel without changing their denotational and declarative semantics respectively. Such implicitly parallelizing languages are promising as they make it easy to benefit from parallel computers without reprogramming existing software, learning new programming languages or reasoning about parallel processes. But the possibilities of this way of parallel programming are quite limited, because arbitrary concurrent process systems cannot be specified.

There has been intensive research into the parallelization of functional and logic languages. An overview on parallel logic programming systems which focuses on the transparent exploitation of parallelism has been given by Chassin de Kergommeaux and Codognet[5]. A comparative study of parallel functional implementations has been carried out by Hogen[7].

As a precondition to a systematic analysis we will first introduce some important definitions.

1.1. Concurrency vs. Parallelism

Adopting the terminology used by Shapiro[15] we will classify *concurrent* systems as follows: A *transformational* system or program receives some input at the beginning of its operation and yields an output at its end. Even if it performs some basic interactive input/output, the central task of such a system is to compute a final result. The purpose of a *reactive* system is not necessarily to obtain a final result, but to maintain some interaction with its environment. Many reactive systems ideally never terminate and in this sense never return a final result, as for example operating systems. Based on this distinction, parallel systems can be defined as a special case of concurrent systems: concurrent systems with transformational behaviour are called *parallel*.

1.2. Implicit vs. Explicit Parallelism

As already mentioned, there exist two main forms of parallelism in declarative languages: either parallelism is used in an implicit way or it is shown through explicit language constructs. The aim of *implicit parallelism* is to speed up programs without troubling programmers with parallelism. We even count approaches with explicit annotations like e.g. Concurrent Clean[17] or para-functional programming[8] as implicit, because the semantics of annotated programs is usually defined as the semantics of the program without annotations, i.e. the annotations are semantically transparent. *Explicit parallelism* however provides a language with additional expressive power and requires extensions of the semantics as well. Usually the semantics is defined operationally only.

1.3. Computation Language vs. Coordination Language

Gelernter and Carriero[6] argue that a programming model should consist of two separate pieces: the *computation* model and the *coordination* model. The computation model allows programmers to specify transformational computations. The coordination model is the 'glue that binds separate activities into an ensemble'. Conventional and declarative languages embody some computational model, but in the general case only a highly restricted coordination model. Concurrent languages embody more sophisticated coordination models, because they provide constructs to create computational activities (generally called *processes* or *threads*) and support communication between them.

1.4. *Our approach*

In this paper we are mainly interested in declarative language constructs allowing programmers to express the concurrency within algorithms explicitly. In particular we aim at the development of a declarative language for parallel and concurrent programming which supports the specification and formal analysis of arbitrary dynamic and reactive process systems in a simple and elegant manner. In particular, we aim at the programming of systems with distributed memory.

Before entering into details, we will give a brief summary of the characteristic features of our concurrent language EDEN[2]: We take a lazy functional language like Haskell[9] or Miranda[16] as computation language and combine it with a coordination language which provides means for

(1) the definition and creation of processes

(2) communication and synchronization

(3) the specification of interconnection topologies.

In order to preserve the qualities of the purely functional kernel language, we keep up a clean (semantic) separation between processes and functional objects.

Therefore, in our approach, processes are '2$^{\text{nd}}$ class citizens'. As processes are dynamic entities they must be distinguished from static objects like values, functions or so-called process abstractions, which specify the behaviour of processes without being processes themselves.

The language EDEN differs from related approaches in a more explicit treatment of parallel processes. Processes are not identified with expressions or literals, but form special objects which are treated on a separate level of the language. Nondeterminism and time-dependency are handled by introducing designated process abstractions like `MERGE` and `SPLIT`.

1.5. *Organisation of the paper*

In Sections 2 and 3 we investigate several declarative concepts for parallel and concurrent programming in logic and functional programming respectively. Thereby we provide a classification of existing approaches to concurrent declarative programming. In Section 4 we present our own language EDEN. We show how transformational systems can be programmed and discuss concepts for the modelling of reactive systems.

2. Logic Constructs for Concurrent Programming

2.1. Concurrent Logic Programming

Logic programming languages were the first to be combined with concurrency. There are several representatives of this paradigm (cf. the surveys[15,5]), which have basic features in common. Every goal atom is interpreted as a separate process. Communication channels are established by introducing shared logical variables. The concept of *incomplete messages* allows the exchange of terms that contain free variables, and in this way provides an elegant mechanism to dynamically allocate and distribute communication channels.

In order to specify the behaviour of processes guarded Horn clauses are used which are evaluated using don't care nondeterminism. Furthermore, concurrent logic languages depart from sequential logic languages by replacing unification with matching. This mechanism enables concurrent processes to synchronize by waiting a shared logical variable to become sufficiently instantiated.

Obviously these languages have their strong points on the side of reactive systems. Transformational systems could of course be expressed as well, but parallel programs with controllable process granularity and arbitrary process topologies can not be defined. In the 1990s concurrent logic programming has increasingly been superseded by concurrent constraint programming which extends and optimizes the handling of partial information.

2.2. Concurrent Constraint Programming

The combination of constraints and concurrency has received much attention in the last few years. The seminal work by Saraswat[14] introduced the **CC** family of languages with an ask- and tell-mechanism for communication via a global constraint store. Concurrent agents can send partial information (i.e. a constraint) to the store by executing a *tell*-operation. Receiving of information is performed via *ask*-operations, which test if the asked constraint is *entailed* by the constraint store and otherwise block. In Saraswat's work, communication is additive, i.e. a constraint that is entailed once remains entailed at every point of time in this computation afterwards. He argues that this monotonicity condition is important for the implementation on a distributed system.

This mechanism is very general and elegant and avoids unnecessary computation by manipulating *partial* information. On the other hand, it does not coincide with our goals in two very decisive points. Firstly, we focus on the programming of distributed memory systems and therefore avoid global structures as far as possible. Secondly, this family of languages entirely relies on implicit parallelism and does not offer any possibility to control the granularity of parallel processes.

A language that is much closer to our desiderata for a parallel declarative language is the constraint functional language **Goffin**[3,4]. It offers explicit parallelism by introducing a separation between two levels of the language. Sequential subcomputations are expressed in the lazy functional language Haskell. On top of this computation language, a coordination language with concurrent constraints performs the handling of threads. These threads are introduced implicitly and they exchange messages via logical variables. In comparison to our approach, the life of a concurrent agent is rather short and its mapping to a physical processor cannot be influenced by the programmer.

3. Concurrent Functional Programming

Now we will proceed by presenting approaches based on the functional paradigm. These show a natural aptitude for the definition of transformational systems. Reactive systems however are impossible to define with a purely functional approach without any extensions. Time dependencies can not be defined because they would violate referential transparency. In the subsequent discussion we will present different amendments which have been devised in concurrent functional languages in order to compensate for this shortcoming.

An important parallel functional language is **Maude**[12], where computations are structured by the introduction of modules. Nondeterministic computations can be encapsulated inside a special type of module, the so-called *system module*. The semantics of Maude is based on a very general model called *rewriting logic* which subsumes a large number of other concurrent programming models[11].

A very interesting attempt to integrate concurrent threads into functional languages is the work on **explicit parallel programming with monads** by Jones and Hudak[10]. They extend Haskell by a special type of monad that allows the spawning of new threads to form a tree of parallel threads. However it is not clear whether this approach could be extended to represent arbitrary communication topologies. On the whole, its focus lies solely on the programming of transformational systems.

The approach of **Concurrent ML** (CML)[13] aims at a level of abstraction similar to that of EDEN. Whereas we have chosen a lazy computation language and a coordination model with asynchronous message passing, CML is based on a strict language and synchronous communication. While both combinations form a natural match, we believe that asynchronous communication is predominant in both reactive computations and efficient parallel algorithms and therefore should be chosen as basic paradigm. CML offers a full higher order concept on the parallel agents and features special combinators for indeterministic choice and synchronization on events. For example, the temporal relation between a synchronization event and the subsequent evaluation of a function can be specified using the `wrap` combinator. These combinators are good candidates for the use with asynchronous communication as well if suitable modifications to the definitions of events are introduced. A more detailed discussion of synchronization issues can be found in Section 4.

4. The Concurrent Language EDEN

EDEN uses simple annotation-like constructs for the explicit definition of processes which are sufficient to describe transformational systems. For the definition of reactive systems extra-functional constructs must be introduced. For instance, special nondeterministic processes are provided for the specification of time-dependent systems.

4.1. Process abstractions and instantiations

Processes are dynamic entities which cannot be objects of the "functional world" if we want to retain referential transparency. Therefore we distinguish in EDEN between *process abstractions*, which specify process behaviour in a purely functional way, and *process applications*, in which process abstractions are supplied with input values in a similar way as λ-abstractions are applied to argument expressions. A process is viewed as a functional mapping of (streams of) input values to (streams of) output values. Accordingly, a process abstraction has the following general structure:

process	$parameter_1 \ldots parameter_k$
input	$inchan_1 \ldots inchan_m$
output	$outchan_1 \ldots outchan_n$
body	$equation_1 \ldots equation_r$ **end**

It specifies a general process scheme with some parameter variables or terms after the keyword **process**, a number of **input** and **output** channels which specify the interface of a process (generated by use of this abstraction), and a **body** consisting of a list of equations which define output channels, auxiliary functions and common subexpressions. There is at least one defining equation for every output channel. The body can be seen as a functional program. A process abstraction is a special kind of λ-abstraction with the following type:

$$process\ \langle\ \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow (\tau'_1, \ldots, \tau'_m) \rightarrow (\tilde{\tau}_1, \ldots, \tilde{\tau}_n)\ \rangle,$$

where $\tau_1, \ldots, \tau_k$ denote the types of the parameters and $\tau'_1, \ldots, \tau'_m$ and $\tilde{\tau}_1, \ldots, \tilde{\tau}_n$ are the types of the input and output channels respectively. We allow arbitrary types for channels, although in most cases lists are used to model continuous streams of input and/or output values on channels. The tuple type which wraps up the input channels of a process indicates that these channels must be provided 'in one piece' while the parameters can be applied one by one. Denotationally a process abstraction is nothing else than an annotated λ-abstraction. The two types of abstraction mainly differ in their operational meaning.

Furthermore, we have the important condition that a process abstraction must be closed, similar to a combinator. The latter implies that the expressions in the body depend only on locally available values like parameters[a], input channels or auxiliary definitions. Thereby it is guaranteed that a process is an independent unit of computation which communicates via its channels only. This property is essential as we want to tailor a language for a distributed memory system.

Note that process abstractions are not equal to processes, in the same way as in an object-oriented setting classes are not equal to objects. We "create" processes by supplying process abstractions with input channel expressions. This operation is called *process instantiation* or *process application* and has the form:

$$\boxed{(out_1, \ldots, out_n) = (process\_abstraction\ e_1 \ldots e_k)\ (input\_exp_1, \ldots, input\_exp_m)}$$

Process instantiations are only allowed as right hand sides of equations. Exactly this restriction prevents processes from being first class citizens. The left hand side of such an equation is a tuple of terms which are bound to or matched against the output channels of the created process.

Our notion of a process has several advantages. Annotations for placement on physical processors can be added straightforwardly and it offers a natural interface to foreign languages. Moreover it is a handy tool for influencing the granularity of processes. The subsequent example illustrates the use of the constructs introduced so far and implements scalable granularity of processes.

Example (parallel operation). The recursive process abstraction **par_op** uses recursive doubling to apply a given binary function to a sequence of numbers; e.g. when applied to the function $\times$ and the input interval $[1, n]$, it yields the factorial of n on its output channel. In order to ensure a reasonable process granularity, new processes are only spawned if the sequence of numbers exceeds a threshold length specified as a parameter.

```
par_op :: process < (Int -> Int -> Int) -> Int -> ([Int]) -> (Int) >
par_op =  process op :: Int -> Int -> Int,   threshold :: Int
          input [low,high] :: [Int]          output result :: Int
          body   result    =  div&con  low high
                 div&con :: Int -> Int -> Int
                 div&con l h  =  if h-l <= threshold then  (seq_op l h)
                                 else (op son (div&con (mid + 1) h)
                                 where  mid  = (l + h) / 2
```

[a] Actual parameter expressions are copied into the body of process abstractions.

```
                              (son) = (par_op op threshold) ([l,mid])
          seq_op ::  Int -> Int -> Int
          seq_op l h     = if l = h then l
                           else (op (seq_op  l mid) (seq_op (mid+1) h))
                           where  mid  = (l + h) / 2
      end
```

In order to calculate the result a tree of processes is generated. Again note that the constructs introduced so far have a mere annotational character. The above process abstraction is semantically equivalent to the pure functional definition one receives by replacing process abstractions by corresponding λ-abstractions or functions.

4.2. Communication

Communication in EDEN is asynchronous and takes place via channels only. A channel connects exactly one *producer* to one *consumer* process, thus data cannot be broadcast implicitly. Channels may have arbitrary types. As soon as a value of the corresponding type has been transmitted in full, the channel is closed. If the type of a channel is a list, the stream of values is transmitted one by one. A list channel is closed by [] ("nil").

We distinguish between static and dynamic channels. Static channels are the input and output channels declared in the process abstraction of the process itself. Dynamic channels may be introduced on the creation of subprocesses. In the next section we will additionally propose the concept of *incomplete messages*, i.e. admit the transmission of terms with *free variables*, which can be used to establish *reply channels* from the receiver to the sender of a message dynamically.

While in lazy functional programs there is always demand for the evaluation of one expression which drives the computation, in EDEN there can be demand for *multiple* output channels. Such demand on several channels divides processes into *concurrent threads*, one for each output channel. This results in a natural distinction of two levels of concurrency: processes and threads within processes. Computations performed for auxiliary definitions are shared by all threads. Fully evaluated objects are transferred into the channel buffers.

If a process needs a value from an input channel, a transfer of data from the channel buffer is started. If the buffer is empty, the computation cannot proceed and is suspended until the sender process writes a value into it. In case that the input is received from the output channel of a child process, the demand for data leads to its creation, i.e. the process instantiation with the input variable on the left hand side is evaluated. As a result, the processes which provide the input of newly created child processes will be generated, too. Thus, there may be a cascade of process instantiations resulting from a demand for some dynamic input channel.

4.3. Specification of Interconnection Topologies

Recursive equations of process instantiations can be used to generate arbitrary process topologies. We assume that an appropriate bypassing (see below) of channels is done automatically, i.e. channels are automatically installed between the corresponding consumer and receiver processes. Only if a parent process accesses the values of a child channel directly, values between child processes will be passed via the parent process. The subsequent example shows how a two-dimensional grid topology can be defined.

Example (twodimensional grid of processes). The process abstraction **grid2** generates a grid of `2 x 2` processes. The process abstraction **pabs** for the computation to be performed by the nodes of the grid is passed as a parameter. The channels are oriented from right to left and from bottom to top, i.e. subprocesses receive input from the right and from the bottom and yield output to the left and to the top.

```
grid2 : process < process <(X,X)-> (X,X)> -> ((X,X),(X,X))-> ((X,X),(X,X)) >
grid2 = process pabs :: process <(X,X)  -> (X,X)>
        input  (fromright12,fromright22), (frombot21,frombot22) :: (X,X)
        output (toleft11,toleft21),       (totop11,totop12) :: (X,X)
        body
          (toleft11,    totop11)   = pabs (fromright11, frombot11)
          (fromright11, totop12)   = pabs (fromright12, frombot12)
          (toleft21,    frombot11) = pabs (fromright21, frombot21)
          (fromright21, frombot12) = pabs (fromright22, frombot22)
        end
```

A grid with `n x n` processes can be defined recursively by extending a grid of size `n-1` with a new row and a new column. The formal specification of this grid structure is given in the appendix. It can easily be used to define a general `n x n` torus structure which we will employ for the specification of Gentleman's parallel matrix multiplication algorithm in the following subsection.

Example (torus of processes). The process abstraction **torus** generates a two-dimensional torus with `n x n` processes, where `n` and a process abstraction **pabs** are passed as parameters.

```
torus = process n :: Int
                pabs :: process <Int -> Int -> Int -> a ->
                                                  ([b],[c])->(d,[b],[c])>
        input   matrix  :: [[a]]
        output  resultmatrix :: [[d]]
        body (resultmatrix,row,col) = (grid 1 n pabs) (matrix,row,col)
        end
```

4.4. Transformational Systems

The kernel of EDEN defined so far is sufficient for the definition of transformational systems. We will now present examples of transformational systems.

Example (matrix multiplication). Using the torus topology defined before it is rather simple to program Gentleman's parallel matrix multiplication algorithm:

```
matrixmult n a b = c where (c) = (torus n scalar_product) (ab)
                           ab  = map zip (zip (a,b))

scalar_product =
     process n, i, j :: Int  (a,b) : (Int,Int)
     input  inrow, incol :: [Int]
     output outc : Int,  outrow, outcol :: [Int]
     body   outc = iterate (n-1) (drop (i-1) inrow) (drop (j-1) incol) (a0*b0)
            outrow = a:inrow     % rotate row
            outcol = b:incol     % rotate col
            a0 = if i=1 then a else   inrow!(i-2)
            b0 = if j=1 then b else   incol!(j-2)
            iterate 0 l1 l2 val = val
            iterate (n+1) (in_a:a_tail) (in_b:b_tail) val
                                = iterate n a_tail b_tail (val + in_a * in_b)
     end
```

The only problem with transformational systems might be the appropriate generation of interconnection schemes. Consider e.g. the following pipeline computation of the Hamming numbers.

Example (Hamming problem). Given a list of primes `[a,b,c,...]` and a positive boundary $\mathtt{n}$, generate without duplication the sorted sequence of all integers $\mathtt{a}^i * \mathtt{b}^j * \mathtt{c}^k * ... \leq \mathtt{n}$ for some natural numbers i, j, k, ...

This problem can be solved in EDEN by creating a subprocess for every prime and putting them together in a linear arrangement. Every process generates its successor in the pipe. It forms its output by adding the multiples of its own prime to the input stream and keeping it sorted:

```
hamming :: process < Int -> [Int] -> () -> ([Int]) >
hamming = process  n :: Int, primes :: [Int]  input   output result :: [Int]
   body  result = if (primes = [ ]) then [ ]
                  else son
                       where (son) = (powers n primes) ([ ])
         powers = process n :: Int, (prim1:primT) :: [Int]
            input  in  :: [Int]
            output out :: [Int]
            body   out = if (primT = [ ]) then local
                         else next
```

```
                    where (next) = (powers n primT)  (local)
                           local = 1:takewhile (<= n)(sortmerge in feedback)
                           feedback = map (* prim1) local

              sortmerge [ ] l = l
              sortmerge (h:t) [ ] = h:t
              sortmerge (h:t) (h':t') = if h < h'
                                         then h : (sortmerge t (h':t'))
                                         else h': (sortmerge (h:t) t' )
end      end
```

Although this is an obvious specification of the pipeline computation, it shows an efficiency problem: the final results will be sent back to the main process through all stages of the pipe because a direct connection between main process and last pipeline stage does not exist straight away. Having identified this as a major flaw with regard to many parallel algorithms, there are two possible strategies for going on.

On the one hand we may implement a sophisticated analysis that identifies during compile time the channels that can immediately be bypassed. In the above example the output channel of each process except for the last one corresponds to the input channel from the successor process. So it can be derived that the output channel of the hamming process can immediately be filled with the output of the last process in the pipe. On the other hand we may extend EDEN with a more convenient mechanism for the definition of arbitrary communication topologies. We postpone this question until we have discussed the treatment of reactive systems.

4.5. Reactive Systems

Typical reactive systems lead to a rather imperative style of reasoning, because the temporal order of actions has to be modelled. It may be necessary to test several sources of input for the availability of data and nondeterministically choose one of the available items (cf. Dijkstra's guarded commands). In addition, there may be the requirement that output must be produced after certain other events have occurred. These time-dependencies can only be expressed with the help of special extra-functional constructs.

4.5.1. Handling nondeterminism

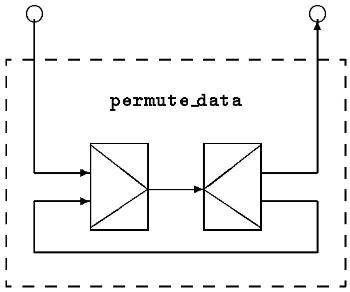
The commonly used method for introducing nondeterminism is a non-functional (fair) stream merge operator `merge`. But this conflicts with our wish to preserve the declarative flavour of our computation language. Instead of permitting nondeterministic elements inside the body of a process abstraction, we decided to extend EDEN by special predefined process abstractions. They encapsulate nondeterminism and thus do not destroy referential transparency within user-defined processes.

The MERGE process abstraction creates a nondeterministic fair merge process for a list of k stream channels with type [Msg] where k is given as a process parameter. In addition, we introduce an analogous process abstraction SPLIT that distributes the contents of its input stream nondeterministically to its output streams:

```
MERGE  ::  process < Int -> ([[Msg]]) -> ([Msg]) >
SPLIT  ::  process < Int -> ([Msg]) -> ([[Msg]]) >
```

Example: (communication medium)
The following process abstraction illustrates the use of nondeterministic processes. It models a communication medium which tends to change the order of transmitted data.

```
permute_data :: process < ([Msg]) -> ([Msg]) >
permute_data =
process   input in    output out
   body (m)        = (MERGE 2) ([in, sl])
        ([out,sl]) = (SPLIT 2) (m)
   end
```



When applied to an input stream, it creates the process system shown on the right hand side.

4.5.2. Enforcing a temporal order

A typical problem that occurs in reactive systems concerns the coordination of different output channels of a process. Everything a process does results from the demand for output channels. Note that synchronization of processes on input channels is provided automatically, as a computation that needs a value from an input channel is suspended until a value is written into the corresponding channel buffer.

Example (client/server system). Assume that we want to model a system with several client processes which ask for services from a single server process. The definition of the server process is not difficult, because the various request channels from the client processes will be merged to one single input channel of the server. The server reads the interleaved sequence of requests from its input channel and reacts by providing the requested services on the corresponding output channels. There will be one output channel for each client. Thus a process abstraction for the server has the following structure:

```
server :: process < Int -> ([Req_type]) -> ([[Service_type]]) >
server = process n :: Int                    % number of clients
         input   requests :: [Req_type]     % merged input channel
         output  to_clients :: [[Service_type]]
         body    to_clients = map (handle requests) [1..n]
```

```
            handle (request:tail) i =  if (from i request)
                                       then  service:(handle tail i)
                                       else (handle tail i)
        end
```

The server splits up into n concurrent threads, one for each client. The concurrent threads each scan the incoming requests and provide the service if the request comes from the client it serves.

The client is more difficult to model as it has to serve two different output channels in a synchronized way. One output channel is used to provide the environment with data which depend on services of the server. The other output channel is the request channel to the server. There is at least one incoming channel which supplies the client with the input from the server. These services, which are only provided on request, are necessary for the main computation. A request should only be sent when a service is really necessary to continue the main computation. A first attempt to model the client could be as follows:

```
client :: process < Int -> ([Service_type]) -> ([Data_type],[Req_type]) >
client =  process  id : Int
          input    from_server : [Service_type]
          output   to_env : [Data_type]   to_server : [Req_type]
          body     to_env    = main_computation (from_server)
                   to_server = (Req id) : to_server
          end
```

Because there is always demand on all output channels it is immediately clear that this specification does not work, as the clients flood the server with an infinity of requests. Since the transmission to the server is not controlled whatsoever, the condition that a request should only be sent on demand from the main computation cannot be fulfilled. Now we will discuss different approaches to cope with this problem.

Introducing data dependencies: The safest way to coordinate the computation of the concurrent threads within the client process would be the introduction of data dependencies. But in the case of constant demand on each output channel this is not possible. Thus, the following solution works only if we assume that there is no demand on the request channel `to_server`.

We define a function `read` which introduces a data dependency between the access to the channel `from_server` and the creation of a request on the output channel `to_server`:

```
client :: process < Int -> ([Service_type]) -> ([Data_type],[Req_type]) >
client = process  id : Int
         input    from_server : [Service_type]
         output   to_env : [Data_type]   to_server : [Req_type]
         body     to_env    = main_computation (input_values)
                  (input_values,to_server) =  read from_server
                  read l = (values, (Req id): tail_reqs)
```

```
                          where
                            (values,tail_reqs) = waitforservice l
                            waitforservice (h:t) = (h:ts,tr)
                                                   where (ts,tr) = read t
        end
```

Under these conditions this solution avoids that an arbitrary number of requests is sent to the server. But nevertheless it is not satisfying, because it is not a natural way to express the coordination of the two concurrent threads within the client process.

Using a local 'ask and tell' mechanism: The 'ask and tell' principle introduced in concurrent constraint programming may help to control the sending of requests to the server. We suggest here a strongly restricted version of the 'ask and tell' mechanism[b] of concurrent constraint programming (cf. Section 2). Note that our aim fundamentally differs from that as we do not want to implement communication between processes via 'ask' and 'tell', but only communication between different threads of the same process. The local constraint store can be placed in the private memory of one cell of the distributed memory system and is accessed by threads executing on this cell only.

If a client accesses the input channel `from_server`, it has to evaluate a 'tell' expression which generates a special entry in the local constraint store of the process. The transmission of a request to the server is delayed until in the constraint store a specific entry can be found that signals demand by the main computation.

In order to show that the evaluation of an expression depends on certain conditions, as e.g. the addition of a value to the constraint store, we use special conditional expressions of the form:

```
expression <- condition
```

```
client :: process < Int -> ([Service_type]) -> ([Data_type],[Req_type]) >
client = process id : Int
        input   from_server : [Service_type]
        output  to_env : [Data_type]   to_server : [Req_type]
        body    to_env      = main_computation (consume from_server 0)
                consume l i = ((hd l) <- tell(Req i)) : (consume (tl l) (i+1))
                to_server   = send_req 0
                send_req i  = ((Req id) <- ask(Req i)) : (send_req (i+1))
        end
```

This solution is rather simple and comprehensible, but note that the introduction of a constraint store within processes hurts the functional flavour of user-defined processes.

[b]This principle is analogous to the concept of semaphores used in imperative languages with an additional monotonicity condition.

Using 'incomplete messages': Another attempt considers a concept similar to the principle of incomplete messages known from concurrent logic programming. If we allow the transmission of terms with free variables, this can be interpreted as a means for the dynamic creation of reply channels. The receiver of a term with such a variable can use it to transfer a reply to the sender of the data. This is done by a unification of the transferred free variable with the reply message.

For each request the client produces a new free variable `$chan` which is sent to the server. As the server can reply via the dynamically created channel, the client no longer has to include his id in every request. Again we use conditional expressions to enforce the dependency.
In the client we use a local dynamic channel `$L` in order to model the static output channel to the server as a difference list. The concept of dynamic channels offers many advantages with regard to the definition of interconnection topologies as well. This aspect has been discussed in another paper[2].

```
client :: process < () -> ([Req_with_reply_chan]) >
client = process % id parameter now superfluous
         input
         output  to_env : [Data_type]
                 to_server : [Req_with_reply_chan]
         body    to_env = main_computation (get_input $L)
                 get_input $L  = $chan:(getinput $R)  <- ($L = (Req $chan):$R)
                 to_server = $L
         end
```

Dynamic reply channels are especially useful for modelling client/server systems, as the server does not need to know the clients asking for his service. As a result, the server process introduced before can be simplified and does not need static output channels any more. The introduction of systems of this type involves the need to generate demand explicitly. Here we propose a special keyword `main` for producing the demand which triggers the reading of input within the process:

```
server :: process < ([Req_with_reply_chan]) -> () >
server = process
         input   requests :: [Req_with_reply_chan]
         output                        % only dynamic output
         body main = handle requests
              handle ((Req $C):tail) = handle tail <- $C = service
         end
```

This concludes the discussion of concepts for the coordination of output within processes. Which concepts will be chosen to be incorporated into the language EDEN is left open until the consequences of their addition with respect to semantics and implementation have been investigated in more detail.

4.6. Semantics of EDEN

The semantics of EDEN is a conservative extension of the standard operational semantics of the lazy functional kernel language. It reflects the distinction between the computation and coordination language within EDEN. Similar as in our work[1] two levels of transition systems are used. On the upper level global effects on process systems are described. The lower level handles local effects within processes. The interface between the two levels are so-called 'actions' by which the need for global events is communicated to the upper level.

Process systems are modelled as *labelled process graphs*. Nodes represent processes and are labelled with process states $\pi \in \Pi$. Edges represent communication channels and are labelled with channel buffer contents. The behaviour of single processes is specified by a transition relation

$$\vdash \subseteq \Pi \times Act \times \Pi,$$

where Act denotes the set of actions. Creation of new processes, sending and receiving of values via communication channels etc. cause actions, by which the necessary information is exchanged between the system and the process level.

A comprehensive description of the semantics would be beyond of the scope of this paper.

5. Conclusions and Future Work

This paper has investigated into declarative concepts for parallel and concurrent programming with the example of the concurrent functional language EDEN. In EDEN transformational concurrent systems can be programmed without extra-functional constructs by using the special syntax of process abstractions and instantiations. The explicit notion of a process in our language supports granularity control and facilitates extensions by placement annotations and interfaces to foreign languages. For the specification of reactive systems additional mechanisms are needed. In this paper we have discussed various concepts known from other concurrent declarative languages that may be of value in this context. A more detailed analysis of these concepts with regard to semantics and implementation remains to be carried out in the future.

References

1. Karsten Bohlmann, Rita Loogen, and Yolanda Ortega-Mallén. Towards a functional process calculus. In *Proceedings of GULP-PRODE, Universita Politecnica de Valencia, Spain*, pages 234 – 250, 1994.
2. Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In *Glasgow Workshop on Functional Programming*, July 1995.
3. 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.
4. Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C.R. Lock. Two limits of purely functional parallel programming and how to overcome them. *internal report*, 1994.
5. Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295 – 336, 1994.
6. David Gelernter and Nicolas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96 – 107, Feb. 1992.
7. Guido Hogen. Parallel Functional Implementations: Graphbased vs. Stackbased Reduction. In *Proceedings of the Fuji Int. Workshop on Functional and Logic Programming, Japan.* World Scientific Publishing Company, 1995.
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. Mark P. Jones and Paul Hudak. Implicit and Explicit parallel Programming in Haskell. Technical Report RR-982, Yale University, 1993.
11. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73 –155, 1992.
12. José Meseguer and Timothy Winkler. Parallel Programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 253–293. Springer, 1992.
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. Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
15. Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3), 1989.
16. David Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, LNCS 201. Springer, 1985.
17. Marco van Eekelen and Rinus Plasmeijer. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, Int. Computer Science Series, 1993.

Appendix A

Specification of the Grid Topology

```
grid = process d, n :: Int
               pabs :: process <(Int -> Int -> Int -> a ->
                                                 ([b],[c])  -> (d,[b],[c]))>
       input   inval  :: [[a]]  fromright :: [[b]]   frombottom :: [[c]]
       output  outval :: [[d]]  toleft    :: [[b]]   totop      :: [[c]]
       body
         outval   = outrow : (combine outcol suboutval)
                    where combine [] [] = []
                          combine (x:t) (xs:ts) = (x:xs):(combine t ts)
         toleft   = leftchan : leftlist

         subinval = map tl (tl inval)
         rowval   = hd (inval)
         colval   = map hd (tl inval)
         rightchan : rightlist = fromright
         botchan   : botlist   = frombottom

         (suboutval,subleft,subtop)
                  = if n<d then ([],[],[])
                    else (grid (d+1) n pabs) (subinval,rightlist,botlist)
         (outrow,leftchan,totop)   = (gen_row d d n pabs)
                                        (rowval,rightchan,colchan:subtop)
         (outcol,leftlist,colchan) = (gen_col d (d+1) n pabs)
                                        (colval,subleft,botchan)

         gen_row d j n pabs  (rval:rowvaltail) (richan,(coltop:sbtop))
                 = if j = n
                   then ([val],leftchan,[topchan])
                        where
                        (val,leftchan,topchan) = pabs n d j rval (richan,coltop)
                   else ((val:vallist),leftchan,topchan:toplist)
                        where
                        (val,leftchan,topchan) = pabs n d j rval (rowchan,coltop)
                        (vallist,rowchan,toplist)
                         = gen_row d (j+1) n pabs rowvaltail (richan,sbtop)

         gen_col d j n pabs colval (subleft,btchan)
                  = if j > n then ([],[],botchan)
                    else ((val:vallist),h_leftlist:t_leftlist,colchan)
                    where
                    (val,h_leftlist,colchan)
                     = pabs n j d (hd colval)  (hd(subleft),rec_bot)
                    (vallist,t_leftlist,rec_bot)
                     = gen_col d (j+1) n pabs (tl colval) ((tl subleft),btchan)
      end
```