Towards a Declarative Language for Parallel and Concurrent Programming

Silvia Breitinger, Rita Loogen
Philipps-Universität Marburg*

Yolanda Ortega-Mallén
Universidad Complutense de Madrid†

Abstract

We define a new language 'EDEN' by extending a functional language by constructs for the explicit specification of dynamic process systems. Following Shapiro [Sha89] we distinguish transformational and reactive systems. Simple annotation-like notions for the explicit definition of processes are sufficient to describe the class of transformational systems. For the definition of reactive systems extra-functional constructs must be introduced. We show that a concept similar to the "incomplete message principle" from concurrent logic programming is a vital and powerful mechanism for the treatment of dynamically specified communication channels. Time-dependencies are handled by special nondeterministic processes.

1 Introduction

Most parallel programming languages used nowadays are based on the imperative programming paradigm. In such languages synchronization of and communication between processes must be handled by the programmer on a very low level of abstraction and the verification of programs is a tedious and complex task.

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. Our main intention is the programming of systems with distributed memory. In order to preserve the characteristics of the purely functional kernel language, a clean (semantic) separation between processes and functional objects is required. Therefore, in our approach, processes are "2^{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.

Our approach 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 handled on a separate level of the language. Nondeterminism and

*Fachbereich Mathematik/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@dia.ucm.es

time-dependency are modelled by special process abstractions like `MERGE` and `SPLIT`. We start with some important definitions:

Following the terminology of [Sha89] we use the following classification of *concurrent* systems. A *transformational* system or program receives some input at the beginning of its operation and yields an output at its end. Even if some basic interactive input/output is performed, 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 yield 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*.

There exist two main streams of parallelism in declarative languages: either parallelism is kept *implicit* or it is made *explicit*. 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 [vEP93] or para-functional programming [Hud91] 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 an extension of the semantics as well. Usually the semantics is defined operationally only.

In [GC92] Gelernter and Carriero 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.

Plan of the paper

In Section 2 we present the kernel of our language EDEN which has been developed from the functional process calculus CFP introduced in [BLOM94]. We show the programming of transformational systems in Section 3 and discuss concepts for the modeling of reactive systems in Section 4. After a short sketch of the semantics of the language, we comment on related work and draw conclusions.

2 The vanilla EDEN

The computational kernel of EDEN consists of a lazy functional language. This kernel is extended by a coordination language which provides means for

(1) the definition and creation of processes

(2) communication and synchronization

(3) the specification of interconnections in the process system.

We keep the extensions to a minimum and preserve the characteristics of the computation language as far as possible.

2.1 Process abstractions and process 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 instantiations* or *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	$inport_1 \ldots inport_m$
output	$outport_1 \ldots outport_n$
body	$equation_1 \ldots equation_r$ **end**

It specifies a general process scheme with a tuple of parameter variables or terms after the keyword `process`, a number of `input` and `output` ports[1] which specify the interface of a process (generated by use of this abstraction), and a `body` consisting of a list of equations which define outports, auxiliary functions and common subexpressions. There is at least one defining equation for every outport. 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 inports and outports respectively. The tuple type which wraps up the inports of a process indicates that the corresponding channels must be provided 'in one piece' while currying is admitted for the process parameters.

Communication channels transmit completely evaluated values of arbitrary structured type (i.e. neither suspensions nor functions are transmitted). In order to model the transmission of a stream of values we introduce a new unary type constructor *strm*. A channel of type

$$\boxed{strm\ \tau}$$

transmits values of type τ one by one. This is the only difference between streams and lists. A channel of type *list* τ is assumed to transfer exactly one list[2] of type τ. Except for the context of communication channels, streams correspond to lazy lists and accordingly, stream types are fully compatible with the corresponding list types.

[1] We denote by *port* the local abstraction of a physical channel linking the process to another. Thus a channel connects one *inport* of a process to one *outport* of another process.

[2] Note that the transmitted list is finite because its evaluation must be completed before the transmission. The transmission of a potentially infinite list is only possible by component-wise transmission via a stream channel.

Denotationally a process abstraction is nothing else than an annotated λ-abstraction. They 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[3], inports or auxiliary definitions. Thereby it is guaranteed that a process is an independent unit of computation which communicates via its ports 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. *Process creation* takes place when a process abstraction (with no more unbound parameters) is applied to a tuple of inport expressions. This is called *process instantiation*:

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

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. Process instantiations are only allowed as right hand sides of equations. Exactly this restriction prevents processes from being first class citizens. Thus, while process abstractions may be passed as argument values to functions or abstractions, process instantiations are not allowed as parameters of other expressions in order to avoid the duplication of running processes which are dynamic entities with an internal state.

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* outports of a process. Thus processes are divided into several *concurrent threads*, one for each outport. So we get a natural distinction of two levels of concurrency: processes and threads within processes. Computations performed for auxiliary definitions are shared by all threads. Our notion of a process has several advantages. It can be simply extended by annotations for processor placement and provides a natural interface to foreign language processes. Additionally it gives the programmer explicit control of the process granularity. The latter point can be seen in the following simple example which makes use of the constructs introduced so far.

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 list $[1, n]$, it yields the factorial of n on its outport. 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)
```

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

```
                    where  mid  = (l + h) / 2
                           (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 denotationally equivalent to the pure functional definition one receives by replacing process abstractions by corresponding λ-abstractions or functions.

2.2 Communication and synchronization

Communication between processes takes place via communication channels only. It is asynchronous and $1:1$, i.e. each communication channel connects exactly one sender and one receiver process. Each channel provides a transmission buffer into which fully evaluated objects are transferred by the sender process. As soon as a value of the corresponding type has been transmitted in full, the channel is closed and the communication port is abolished. For implementation reasons only finite, non-partial values of structured type can be transferred via communication channels.

Stream channels pass values one by one. They are processed using ordinary list processing functions. A stream channel is closed by [] ("nil"). Note that the evaluation of outports is independent of the consumption of the produced values. Thus, the channel buffers of stream channels must ideally be unbounded.

Some communication channels are established on process creation (input and output ports specified in the process abstraction), but some others are established later on in the "life" of the process e.g. when subprocesses are instantiated.

If a concurrent thread within a process needs some value from an inport whose channel buffer is empty, this thread will be suspended until the corresponding sender process writes some value into the channel buffer. Thus, communication via channels is done via a non-blocking send and a blocking receive. Interprocess synchronization is performed by the exchange of information via channels only. Notice that the communication and synchronization mechanisms are implicit, i.e. we do not provide any *send*, *receive*, or *wait* operators.

2.3 Interconnecting processes

In principle, direct communication is only possible between parents and children. In [BLOM94] an explicit *bypass* operator was used to redirect channels of newly created processes to sibling processes or predecessors. Although this operator was rather elementary, its use was unnatural and tedious. We decided to remove it and to create general process topologies by recursive equations of process instantiations. Thereby we assume that an appropriate bypassing 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 outport, values will be passed via the parent process.

Example (Twodimensional Grid of Processes). The process abstraction `grid2` generates a twodimensional grid of processes. The process abstraction for the grid processes is passed as a parameter. The channels are oriented from left to right and from top to bottom.

```
grid : process<process < (*,*)  -> (*,*) >
                -> ((*,*),(*,*)) -> ((*,*),(*,*))>
grid = process pabs :: process < (*,*)  -> (*,*) >
       input (left11,left21), (top11,top12) :: (*,*)
       output (right12,right22), (bot21,bot22) :: (*,*)
       body
          (right11,bot11) = pabs (left11,top11)
          (right12,bot12) = pabs (right12,top12)
          (right21,bot21) = pabs (left21,bot11)
          (right22,bot22) = pabs (right21,bot12)
       end
```

2.4 Outlook on additional language features

This chapter has shown the basics of our approach to declarative parallel programming. As you already might have expected, the case is far more complex than that. We decided to introduce additional constructs:

- the possibility to dynamically create *reply* channels via the principle of incomplete messages (arbitrary terms with free 'channel variables');
- special nondeterministic process constants with nondeterministic behaviour.

In the subsequent chapters we will demonstrate both the necessity of additional language elements and the advantages of the above-mentioned features over possible other solutions. We will use two of the Salishan problems[4] as case studies for these investigations. The viability of our approach with regard to both *transformational systems* and *reactive systems* will be shown.

Note that our language additionally incorporates an explicit control of demand propagation by means of a demand control store and two predefined operations *ask* and *tell* to access and to extend this store. This feature will not be presented in this paper. For a discussion on this topic, the interested reader is referred to [BLOM95].

3 Transformational systems

In this section we will investigate in how far the language constructs presented in Section 2 are sufficient for programming transformational systems efficiently. A typical representative of a transformational system is a *pipelining* computation. We will take the well-known *hamming problem* as an example.

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

[4]The set of Salishan problems has been proposed at the 1988 Salishan High-Speed Computing Conference and is intended as a standard by which to compare parallel programming notations (see e.g. [Tho93]).

This problem can be expressed in EDEN by a set of processes with linear communication topology, where there is a subprocess for every prime. 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 sorting it. Although this is an obvious realization 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, we decide to extend EDEN by a more convenient mechanism for the definition of arbitrary communication topologies: we allow the dynamic creation of *reply* channels. A process may send a message to another process which contains the name of a reply channel, which can be used by the receiving process to return some information to the sender process, or be passed further on to another process. These possibilities exclude each other and are termed *receive and use* and *receive and pass*, respectively. Syntactical restrictions ensure that each dynamically created channel is used to establish a one-to-one connection between a unique writer process and the process which creates the channel. Channel variables can neither be duplicated nor be sent to several processes.

This method of dynamic channel creation corresponds to the concept of *incomplete messages* known from concurrent logic programming. Channel names can be seen as free variables, which are passed to other processes and which will be bound to some reply information.

We introduce a new unary type constructor *chan_name* for dynamically created reply channels. Objects and parameters of type *chan_name* τ will syntactically be named by identifiers which start with the symbol $.

A process which receives such a reply channel name **$chan** and wants to reply on it, uses a conditional expression of the following form:

$$\boxed{\langle\texttt{expression1}\rangle \texttt{ <- \$chan = } \langle\texttt{expression2}\rangle}$$

Before **expression1** is evaluated, a new concurrent thread for the evaluation of **expression2** is generated. The result of this concurrent evaluation is sent via the received reply channel.

Now we will illustrate the use of this construct by a solution to the hamming problem.

```
hamming :: process < Int -> [Int] -> () -> (strm Int) >
hamming = process  n :: Int, primes :: [Int],  input  output  result :: strm Int
   body  result = if (primes = [ ]) then [ ]
                  else $C
                       where (son) = (powers n primes) ([ ], $C)
                                               % reply channel generation
      powers = process n :: Int, (prim1:primT) :: [Int]
         input  in  :: strm Int, $C :: chan_name strm Int
         output out :: Int  % dummy
         body   out = if (primT = [ ]) then 0 <-  ($C = local)
                                               % receive and use
                         else next
                              where (next) = (powers n primT)  (local, $C)
                                               % receive and pass
                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
```

This program assigns the result stream to the output of process **hamming** by using the reply channel **$C**. The pipeline stages are instantiations of the abstraction **powers** and every such process creates its successor process. They pass on the channel **$C** which the last pipeline stage uses for directly transmitting the result stream back to the **hamming** process.

4 Reactive systems

A very graphic example of a reactive system is the *doctor's office problem*, which features patients and doctors as asynchronous processes with circular dependencies. Despite its seeming simplicity this problem turned out to be a real challenge for languages with a functional style (cf [Tho93]). We consider it important to find an elegant and clear solution to this problem because it can straightforwardly be expressed in imperative languages and it incorporates some typical coordination problems of reactive systems.

The doctor's office problem can be described as follows: Given a set of m patients, a set of n doctors and a receptionist, the following interactions have to be modelled (see Figure 1). The receptionist handles a queue of patients waiting for medical treatment and a queue of unoccupied doctors waiting for patients. He assigns patients to doctors in a FIFO manner and notifies the respective patient. Circular dependencies are caused by doctors and patients, since a doctor rejoins the queue of free doctors after he has cured his current patient, and a patient after a while will again fall ill and show up at the reception and ask for a doctor.

Now we will discuss what is needed in order to model these agents. As Figure 1 shows, the receptionist receives input streams which contain the inputs from all doctors and all patients respectively. This form of n:1 (resp. m:1) communication involves the merging of streams. As we need a *fair* merge of streams, we could not use a pure function to implement this. Instead of admitting 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 set stream channels with type **strm Msg**:

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

In addition, we introduce an analogous process abstraction SPLIT that distributes the contents of its input stream nondeterministically to its output streams. For a more detailed discussion of the latter, we refer the reader to [BLOM94].

The receptionist and his office

The application of `MERGE` is illustrated by the `office` process that provides the receptionist with the input channels `fromPats` and `fromDocs`. It creates the receptionist and all patient and doctor processes.

```
office :: process < Int -> Int -> () -> strm (Int,Int) >
office =  process  m, n :: Int
          input
          output protocol :: strm (Int,Int)
          body
             (protocol) = receptionist (fromPats,fromDocs)
             (fromPats) = MERGE (patChans)
             patChans = gen_agents m patient
             (fromDocs) = MERGE (docChans)
             docChans = gen_agents n doctor
             gen_agents :: Int -> process <() -> (X)> -> [X]
             gen_agents k abstr = if k=0 then []
                                  else fromagent : gen_agents (k-1) abstr
                                       where (fromagent) = abstr ()
          end
```

The `receptionist` waits for both a patient and a free doctor to arrive and blocks if at least one of the input streams is empty. The blocking of the receptionist is achieved automatically in EDEN because the access to an empty input channel leads to the suspension of a computation. Both doctors and patients send messages to the receptionist which contain a dynamic reply channel. In case of a doctor-to-patient assignment the problem specification requires that the receptionist uses the patient's channel to send a message with the doctor's id and reply channel back to the patient. The subsequent treatment is then handled by patient and doctor autonomously (marked by a dashed connection in Figure 1).

```
receptionist :: process < (strm patToRecMsg,strm docToRecMsg)
                          -> (strm (Int,Int)) >
receptionist = process
    input   fromPats :: strm patToRecMsg, fromDocs :: strm docToRecMsg
    output  protocol :: strm (Int,Int)
    body
      protocol = (handle fromPats fromDocs)
      handle :: strm patToRecMsg -> strm docToRecMsg -> strm (Int,Int)
      handle ((AskForDoc patId $patChan) : pTail)
             ((AskForPat docId $docChan) : dTail)
             = (patId, docId):handle pTail dTail
                                <- ($patChan = (ProvideDoc docId $docChan))
    end
```

Patients and doctors

With the patient and doctor processes we find a more complicated coordination problem. Patients and doctors have to communicate with different partners in a fixed sequential order. No natural data dependencies enforce the order of the following actions to be taken by the patient: sending his id to the receptionist, waiting for a doctor's id, contacting the corresponding doctor, and later on running through this cycle again. In order to prevent the patient for instance from demanding a new doctor from the receptionist too early, synchronization operations or artificial data dependencies have to be introduced.

The concept of reply channels allows us to express these temporal dependencies in an elegant and concise way: A sick patient first receives a message containing doctor information. He then uses the included reply channel to send a request for cure to his doctor and waits for the arrival of a cure message. Recursion is used to model the state of the patient. We present a solution with such data connections.

```
patient :: process < num -> () -> (strm patToRecMsg) >
patient =  process  myid :: Int
           input          % only via dynamic channels
           output toRec :: strm patToRecMsg
           body
              toRec = sick ()
              sick() = (AskForDoc myid $Answer) :  waitforDoctor $Answer
              waitforDoctor (ProvideSymptoms docId $Doc)
                      = (waitforCure $Pat) <- ($Doc = (GetCure myid $Pat))
              waitforCure (Cure x) = well ()
              well () = sick ()
           end
```

The `doctor` has to fulfil a synchronization requirement similar to that of the patient: he may not send a message to the receptionist before he has finished curing the previous patient.

```
doctor :: process < num -> () -> (strm docToRecMsg) >
doctor =  process  myid :: Int
          input              % only via dynamic channels
          output toRec :: strm docToRecMsg
          body
           toRec = idle()
           idle() = (AskForPat myid $Doc) : (waitforPat $Doc)
           waitforPat (GetCure patId $Pat)
              = (idle() <- ($Pat = Cure cure))
                where cure = computeCure patId
                                    % with a suitable function computeCure
       end
```

This running example has shown that the proposed extensions indeed provide a convenient mechanism for the specification of reactive systems.

5 A note on semantics

The semantics of EDEN will be an extension of the standard operational semantics of the lazy functional kernel language. It will reflect the distinction between the computation and coordination language within EDEN. Similar as in [BLOM94] we will use two levels of transition systems. 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 will be so-called 'actions' by which the need for global events is communicated to the upper level.

6 Related work

Now that we have presented our own language concepts we will have a look at existing approaches and judge how EDEN compares to them.

In the class of concurrent and parallel declarative languages, *concurrent logic programming* has the longest tradition (consult for example [Sha89] or [dKC94] for a survey). Although this area is not directly connected to our work, many important concepts such as *incomplete messages* originate from there.

Concurrent constraint programming, as introduced by Saraswat [Sar93], allows for implicit parallelism. Synchronization is performed via *ask* and *tell* operations for the exchange of partial information with a global constraint store. This approach however is not very well suited to the settings we focus on since there is no control over the granularity of parallel computations and a global constraint store could not be implemented efficiently on a distributed memory system.

Closer to our work is the language **Goffin** [CGKL94], [CGK95] with a concurrent constraint coordination language and Haskell as a computation language. It offers explicit parallelism, but no explicit notion of a process. Messages are exchanged via logical variables only.

Among the parallel programming languages based on the *functional* paradigm, only approaches with *explicit* parallelism can be compared to EDEN. We will now sum up the central features of four different languages or language extensions of this type. The approach of **Concurrent ML** (CML)[Rep91] aims at a level of abstraction similar to that of EDEN. Whereas we have chosen a lazy computation language and a coordination through 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 elegant combinators for indeterministic choice and synchronization on events. The latter of which resemble the approach of **explicit parallel programming with monads** [JH93]. In this work, a special type of monad is introduced that allows the spawning 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. In similar terms we could describe the recent language **Concurrent Haskell** [PJF95], where a very simple mechanism called *mutable variables* is introduced. Although powerful enough to implement other higher level mechanism for process communication and synchronization, mutable variables result to be very low-level. The whole approach is strongly influenced by imperative programming. Another important parallel functional language is **Maude** [MW92], 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 *rewriting logic*. It will be a vital task to investigate the relevance of this very general model to our work since it subsumes a large number of other concurrent programming models.

7 Conclusions

This paper presents work which is still in progress. Our primary goal is the investigation of declarative concepts for parallel and concurrent programming. The explicit notion of a process in our language supports granularity control, annotations for processor placement as well as natural interfaces to foreign languages.

In EDEN transformational concurrent systems can be programmed without extra-functional constructs by using the special syntax of process abstractions and instantiations. For the specification of arbitrary reactive systems additional mechanisms are needed. We proposed in this paper the introduction of predefined nondeterministic processes like `MERGE` and `SPLIT`. Furthermore we transferred the incomplete message principle from concurrent logic programming to the functional context and used it to provide a convenient mechanism for the dynamic creation of reply channels.

A more detailed investigation of these concepts with regard to semantics and implementation remains to be carried out in the future.

Acknowledgements: The authors are grateful to Hendrik Lock and Manuel Chakravarty for discussions on concurrent declarative programming.

References

[BLOM94] 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.

[BLOM95] Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Concurrency in Functional and Logic Languages. In *Proceedings of the Fuji Int. Workshop on Functional and Logic Languages, Susono, Japan.* World Scientific Publishing Company, July 1995.

[CGK95] 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.

[CGKL94] 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.

[dKC94] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295 – 336, 1994.

[GC92] David Gelernter and Nicolas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96 – 107, Feb. 1992.

[Hud91] Paul Hudak. Para-Functional Programming in Haskell. In Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[JH93] Mark P. Jones and Paul Hudak. Implicit and Explicit parallel Programming in Haskell. Technical Report RR-982, Yale University, 1993.

[MW92] 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.

[PJF95] Simon Peyton Jones and Sigbjorn Finne. Concurrent Haskell: preliminary version. Technical Report G128QQ, Department of Computer Science, University of Glasgow, 1995.

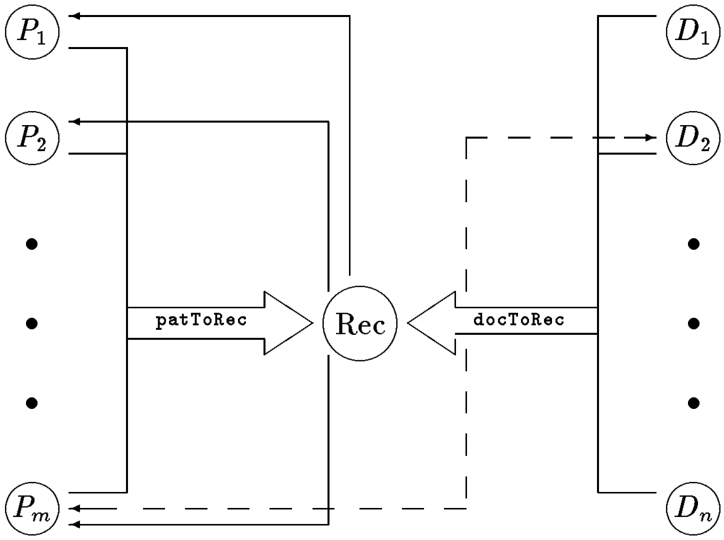
[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293 –305, 1991.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Sha89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3), 1989.

[Tho93] John Thornley. Integrating functional and imperative parallel programming: CC++ solutions to the salishan problems. Technical Report CS-TR-93-40, California Institute of Technology, 1993.

[vEP93] Marco van Eekelen and Rinus Plasmeijer. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, Int. Computer Science Series, 1993.




```
type   patToRecMsg
          = (AskForDoc Int
               (Chan_name recToPatMsg))
       docToRecMsg
          = (AskForPat Int
               (Chan_name patToDocMsg))
       recToPatMsg
          = (ProvideDoc Int
               (Chan_name patToDocMsg))
       patToDocMsg
          = (GetCure Int
               (Chan_name docToPatMsg))
       docToPatMsg
          = (Cure Int)
```

Figure 1: Interaction of agents in the doctor's office and corresponding types of messages