The Eden Coordination Model for Distributed Memory Systems

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

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

Abstract

Eden is a concurrent declarative language that aims at both the programming of reactive systems and parallel algorithms on distributed memory systems. In this paper, we explain the computation and coordination model of Eden. We show how lazy evaluation in the computation language is fruitfully combined with the coordination language that is specifically designed for multicomputers and that aims at maximum parallelism. The two-level structure of the programming language is reflected in its operational semantics, which is sketched shortly.

Keywords: parallel computing, concurrency, distributed memory, functional languages

1 Introduction

In [6] Gelernter and Carriero argue that a programming model should consist of two separate pieces: the *computation* model, that allows programmers to specify transformational computations and the *coordination model* which binds separate computational activities together. This second level of the language is what distinguishes concurrent languages from sequential ones. It comprises constructs to create computational activities (generally called processes or threads) and support communication between them.

*Fachbereich Mathematik, Fachgebiet Informatik, Hans Meerwein Straße, Lahnberge, D-35032 Marburg, Germany, {breiting,loogen}@informatik.uni-marburg.de, Phone: ++49-6421-28-1520/1525, Fax: ++49-6421-28-5419

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

Following the terminology of [19] 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*.

Eden in a nutshell

Eden was designed as a declarative language for parallel and concurrent programming that facilitates the specification and the formal analysis of dynamic and reactive process systems in a distributed memory environment [3].

Eden extends the lazy functional language Haskell [8] by a coordination model which is derived from the approach of stream-based communication [10] and uses an explicit notion of a *process*. In order to preserve the characteristics of the purely functional kernel language, a clean (semantic) separation between processes and functional objects is required. As processes are dynamic entities they must be distinguished from static objects like values, functions or process abstractions, which specify the behaviour of processes without being processes themselves. For the handling of process systems, the following constructs are available:

- **process creation:** New syntactic constructs, so-called process abstractions, are used as process definitions. They are static objects which can be used by functions as arguments and results. Processes, which are dynamic objects, are *explicitly* created by the instantiation of these abstractions (see Section 3.1).
- **communication between processes:** Processes exchange values via communication channels modelled by lazy lists. This form of communication in general is *implicit*, i.e. no new syntactic constructs (like send/receive) are needed.

These concepts suffice for the specification of arbitrary transformational systems, but for the definition of general reactive systems two additional concepts are provided:

- **dynamic reply channels:** An additional mechanism for the dynamic creation of channels simplifies the generation of complex communication topologies and increases the flexibility of the language. It is related to the 'incomplete message principle' known from concurrent logic languages [19] and the 'channel name passing' principle of the π-calculus [15], but is far more restricted in order to maintain the functional flavour of the language.

- **predefined nondeterministic processes:** In order to model many-to-one and one-to-many communication in process systems, Eden provides a predefined nondeterministic `MERGE` process. This nondeterministic feature is introduced at the system level only.

Eden's operational semantics in a nutshell

The distinction between the computational and the coordination model leads to the definition of the following two levels: the level of user-defined processes and the level of process systems. User-defined processes can be seen as deterministic mappings from input channels to output channels where the interface can grow dynamically when new subprocesses are created. Nondeterminism is only handled at the system level which consists of all processes (predefined and user-defined) interacting via communication on channels.

Correspondingly, the operational semantics of Eden uses 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 is formed by so-called 'actions' which communicate the need for global events to the upper level. There are actions for input from channels, output on channels, channel closing, reply channel generation, process creation and process termination. The empty action τ corresponds to internal evaluations.

The local behaviour of a process p describes its own evolution within the whole system and is modelled by transitions on the set *State* of process states:

$$p : \vdash \subset \mathit{State} \times \mathit{Act} \times \mathit{State}.$$

A local transition may be connected with a global transition, which is expressed by the presence of a non-empty action. The internal state of a process consists of its I/O-interface (names of the active communication ports), and its local environment.

The global behaviour of process networks is defined by a transition relation on system configurations.

$$\begin{aligned} \models \; \subseteq \mathit{System} \times \mathit{System}, \text{ where } \mathit{System} &= \mathcal{P}(\mathit{Process} \cup \mathit{Channel}) \\ \mathit{Process} &= \mathit{Process\_Id} \times \mathit{Process\_Type} \times \mathit{State} \\ \mathit{Channel} &= \mathit{Channel\_Id} \times \mathit{Process\_Id}^2 \times \mathit{Contents}. \end{aligned}$$

A *system configuration* is a set of *processes* and *channels*. Each process has a unique process identifier, a process type (predefined process: `merge` and `split`, or `user`-defined process), and a state. Each channel is represented by a unique identifier, the process identifiers of its sender and its receiver, and the contents of the channel buffer.

Plan of this paper

The two following sections present the computation (Section 2) and the coordination (Section 3) languages of Eden, respectively. The coordination model will be discussed more deeply, as it is the main issue of the present paper. We will sketch the operational semantics for the features introduced and discuss the coordination of laziness and parallelism in Eden. In Section 4 we briefly comment on related work, and we finish in Section 5 by drawing conclusions and outline work in progress.

2 The Computation Language

Eden's computation language is the lazy functional language Haskell [8]. The laziness of Haskell is essential for Eden, because it makes programs very robust and flexible. In general, laziness means that expressions are only evaluated if their result is definitely needed for the overall computation, i.e. the evaluator is not doing any reduction of an expression until it is forced to do so in order to produce a result. Lazy evaluation has many benefits: it avoids unnecessary computations, allows the processing of infinite data structures and supports a modular style of programming [9]. In particular, Eden's coordination model heavily relies on the fact that (deterministic) process networks can nicely be modelled in a lazy functional language by using lazy lists as communication channels[20].

The operational semantics of local expression evaluations is based on the operational semantics of Haskell. As Eden processes are viewed as deterministic mappings from input channels to output channels, the local behaviour of processes is defined by purely functional expressions. Local expression evaluations cause no effects on the global system. They are accompanied by the empty action τ in the transition relation $\vdash$ of processes, which at the system level has no other repercussion than changing the internal state of the corresponding process.

Laziness in general can be regarded as an obstacle to parallelism, because it delays evaluations as far as possible. We want to benefit from the advantages of a lazy functional language, but nevertheless support parallel evaluation well.

In most parallel implementations of lazy functional languages *strictness annotations* are used in order to overrule lazy evaluation and to provide some potential for the exploitation of parallelism without sacrificing completeness with respect to the denotational semantics of programs [16]. But these approaches to parallelism are rather restrictive, as the process systems resulting from such annotations are usually completely hierarchical. In particular, it is not possible to program arbitrary parallel algorithms with more involved process structures, as for example a parallel grid matrix multiplication algorithm.

These problems prove that the expressivity of such annotation-based approaches is severely limited. In order to be able to specify general (especially nondeterministic) reactive process systems, Eden extends the functional language by a powerful coordination language. In the following section, we will explain how this coordination model allows the definition of arbitrary parallel process networks and show how it forms a compromise between speculative creation of processes and lazy evaluation inside processes.

3 The Coordination Language

According to [6], it is the task of the coordination language 'to bind concurrent activities into an ensemble'. In the following, we will discuss the definition of these concurrent activities and the characteristics of the communication between them.

3.1 The creation of concurrent activities

Declarative approaches with implicit parallelism tend to incur efficiency problems caused by too fine granularity, which can only be alleviated by relatively complex analysis techniques. As Eden is tailored for systems with distributed memory, the need for coarse-grained processes is especially strong. Therefore we decided to give the programmer the possibility to enclose arbitrary functional computations in a process. Such a process can have multiple inputs and outputs. If more than one output is present, the computation will be split up into separate *threads*.

Process definition by process abstraction

The process abstraction defines a general parameterized process scheme which can be used to generate processes. It is in principle a λ-abstraction, but with a special meaning of arguments. It expects a number of parameters, a tuple of so-called input ports and yields a tuple of output ports[1].

An abstraction for processes that map inputs $in_1, \ldots, in_m$ to outputs $out_1, \ldots, out_n$ has the following type and syntax, 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 of the outports, respectively:

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

$$
\boxed{\begin{array}{llllll}
p :: & \tau_1 \rightarrow & \ldots & \rightarrow \tau_k & \rightarrow & \mathit{Process}\ (\tau'_1, \ldots, \tau'_m)\ (\tilde{\tau}_1, \ldots, \tilde{\tau}_n) \\
p & \mathit{par}_1 & \ldots & \mathit{par}_k & = & \texttt{process}\ (\mathit{in}_1, \ldots, \mathit{in}_m) \rightarrow (\mathit{out}_1, \ldots, \mathit{out}_n) \\
 & & & & & \texttt{where}\ \mathit{equation}_1 \ldots \mathit{equation}_r
\end{array}}
$$

The difference between process parameters and inputs concerns the time and the method of transmission. Firstly, process parameters have to be passed before process creation, while input values can be passed during the process execution. Secondly, process parameter expressions are copied into the body of the process abstraction and are on demand evaluated by the created process itself. Input channel values are evaluated by a sender process, and received via the communication channel. The sender process is not necessarily the creator of the process, which will be explained in Section 3.2.

Inports and outports are clustered into tuples in order to make clear that they have to be supplied together, while currying is admitted for process parameters.

The optional `where`-part of a process abstraction is called *body* and contains equations defining outputs, auxiliary functions and local subexpressions which should be shared.

The right hand side expressions in the body may contain process parameters and inports, locally defined identifiers and functions. Furthermore, the name of the surrounding process abstraction itself can be used as well, so that recursive process systems can be defined. As processes have to be closed objects, which communicate with their environment via communication channels only, no free variables are allowed in the body of process abstractions. Outports are only allowed on the left hand side of equations.

Process creation by process instantiation

In a process *instantiation*, a process abstraction is supplied with input values in a similar way as a λ-abstraction is applied to argument expressions. Process creation then takes place when all parameters in the process abstraction are specified and it is applied to a tuple of input channel expressions. For type inference reasons, a special infix operator `#` for process instantiations is used.

$$\boxed{p\ e_1 \ldots e_k\ \#\ (\mathit{input\_exp}_1, \ldots, \mathit{input\_exp}_m)}$$

Often, process instantiations occur in equations that identify a tuple of names with the tuple of outports of the newly created process.

Example (matrix-vector multiplication) In this example a matrix given as a list of lists of integers is multiplied with a vector which is given as a list of its components. For every row of the matrix, a separate `agent` process is created.

```
matVect :: [[Int]] -> [Int] -> [Int]
```

```
matVect []       v = []
matVect (row:rs) v = (agent # (row, v)) : (matVect rs v)
                      -- build result vector

agent :: Process ([Int],[Int]) Int
agent  = process (as,bs) -> sprod as bs
          where sprod :: [Int] -> [Int] -> Int
                sprod []  []  = 0
                sprod (a:as) (b:bs) = a*b + sprod as bs
```

Operational semantics of process creation

The internal evaluation of a process may require the instantiation of some child processes. When this happens, not only the global system is changed by the generation of these new processes, but also the internal state of the parent is extended by the new in/outports for the communication with these children. In general, automatic bypassing (see Section 3.2) will be performed for in/outports which are newly created but "unused" by the parent. The `unused` condition of a port can be checked using the local definitions in the body of the process.

3.2 Communication between processes

Asynchronous message passing is the communication paradigm typically used in the context of distributed memory systems. We will explain how this mechanism has been integrated into a functional setting.

The types of messages are not restricted. Every structured data type can be used as the type of a communication channel. Data structures can either be transmitted as **one-value channels** or **streams**. Inside the body of a process, a stream is fully compatible to a list. However, there is an important difference: the stream can be transferred piecewise, whereas a list can only be sent if it is fully evaluated (see Section 3.3). An additional datatype is `Chan_name` τ for the definition of reply channels of type τ, see page 8. All communication channels are *hyperstrict*, i.e. values have to be evaluated in full. Consequently it is impossible to transfer a running process, i.e. a process instantiation, but only its result.

The transfer of messages does not require low-level programming. We model asynchronous message passing *implicitly*, i.e. we do not introduce primitives like send and receive, but define how output values are to be computed and how the contents of the inports is to be used to do this. Any output value produced is transferred into a channel buffer, where it can be fetched by the designated receiver process. A receiving thread is suspended if one of the required input channels does not contain a value yet. In this way we implement non-blocking send and blocking receive.

Topologies

One of the main aims of Eden is the definition of parallel algorithms and corresponding topologies. The communication structure can be built up using recursive definitions and an automatic bypassing mechanism. This mechanism is activated when a process does not access the contents of one of its communication channels itself, but passes it on to another process without using it whatsoever. In such a case a 'shortcut', i.e. a direct connection from the real producer and the real consumer of the contents is established. In cases where a process both uses input itself and sends it to another process, a form of *multicasting* is done and no redirection is used. A detailed discussion of bypassing can be found in [3].

The input and output ports specified in the process abstraction are established on process creation, but additional ones can be established later on in the "life" of the process when either child processes are created or reply channels are used. Process networks can be extended dynamically, e.g. depending on tests for sufficient task granularity.

Following the skeleton idea, useful communication topologies can be defined and instantiated appropriately, as the subsequent example illustrates.Furthermore, Eden skeletons for the definition of various topologies can be found in [5].

Example. (pipeline) The following process abstraction defines a generic pipeline stage. For interprocess communication, the lists of type `a` are interpreted as *streams* (see Section 3.3). A predicate `last` is applied to the input in order to find out how many processes have to be generated. The pipeline processes employ a transformation function `transf` and a composition function `cps`.

```
pipeStage :: ([a]->Bool) -> ([a]->[a]) -> ([a]->[a]->[a]) -> Process [a] [a]
pipeStage     last           transf         cps
     = process   inp -> out
       where out    = if   (last inp)
                      then nextin
                      else (cps inp (pipeStage last transf cps # nextin))
             nextin = transf inp
```

Dynamic reply channels

Eden provides in addition to recursive definitions a more convenient mechanism for the definition of arbitrary communication topologies: *reply channel* generation. This 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.

In analogy to the dynamic creation of *processes*, a process may dynamically generate a new input channel and send a message containing the name of this channel to another process. The other process then may either use the received channel name to return some information to the sender process, or pass the channel name further on to another process. These possibilities exclude each other and are termed *receive and use* and *receive and pass*, respectively. The programmer should ensure that each dynamically created channel is used to establish a one-to-one connection between a unique writer process and the generator process of the channel. This restriction cannot be checked statically. A runtime error would arise if a single reply channel were used by several processes.

Generating a reply channel We introduce a new unary type constructor `Chan_name` for the names of dynamically created reply channels. A new channel name of type `Chan_name` τ and a corresponding input channel of type τ are declared in the following expression, where *ch_name* gives the programmer the name of (or reference to) the newly created channel *chan*:

$$\boxed{\texttt{new (}\mathit{ch\_name}\texttt{, }\mathit{chan}\texttt{). exp}}$$

The scope of the newly generated channel *chan* and its name *ch_name* is the body expression `exp`. The reference *ch_name* can be sent to another process in order to request an answer.

Connecting to a reply channel A process receiving a reply channel name *chan* and wanting to reply through it, uses an expression of the following form:

$$\boxed{\mathit{chan}\ \texttt{!*}\ \mathit{expression}_1\ \texttt{par}\ \mathit{expression}_2}$$

Before $expression_2$ is evaluated, a new concurrent thread for the evaluation of $expression_1$ is generated. The result of this concurrent evaluation is sent via the received reply channel. When the above expression is evaluated, the connection between sender and receiver will be established and it will be checked that this reply channel is not already connected to another sender.

Operational semantics. Output operations are reflected by **send** actions. The outport used can either already be defined in the interface of the process or it can be a previously unused reply channel. The first (and maybe only) time that a process writes on such a new reply channel, the outport identifier does not yet belong to the

I/O-interface. In such cases, the new channel has to be added to the set of outports and can afterwards be handled in exactly the same way statically declared outports are.

Input operations are reflected by **receive** and **recClose** actions. When reading from an inport a value which is not the last entry on this port, a `receive`-action is carried out. If on the other hand the reader consumes the last or only value, it performs a combined `recClose` action.

3.3 Coordinating Laziness and Parallelism

Process creation Efficient handling of process creation is crucial for the programming of systems with physically distributed memory. As process creation is *expensive*, it is important not to create too many processes. As on the other hand it is *slow*, it is vital to create processes not too late, i.e. not solely on demand. The coordination model of Eden combines lazy and eager evaluation in a way that makes programs benefit most from both evaluation strategies:

- If process instantiations are placed on the top level of a running process, they are instantly executed in order to obtain maximum parallelism.

- If process instantiations are defined using an 'indirection' such as a conditional or case-expression, they are entered only on demand. The same of course applies to every usual Haskell expression.

With this straightforward rule, parallel algorithms can be programmed efficiently without any additional syntactic constructs. This is illustrated by the pipeline stage example from Section 3.2, where new pipeline stages are only created as long as the condition `last inp` does not hold.

Streams In any case, data transmitted from a sender process to some consumer process has to be fully evaluated by the sender, i.e. no suspensions can be sent. In order to make efficient interleaving of computation and communication possible, list structures by default are interpreted as *streams*, and therefore can be transmitted in 'packages'. In cases where this default interpretation is not desirable, the programmer can circumvent the type to be transmitted on a separate channel by a mixfix *channel annotation* `< >`. This is for example useful for `MERGE`, which will be explained in the following section.

Using streams, communication latency can be hidden and the implementation of stream communication can be adapted to the characteristics of the underlying distributed memory system.

3.4 Synchronisation and fair communication

In many cases, a process is expected to process certain inputs from multiple senders in an equal manner. Consider the following situation with a typical *fairness condition*:

A server process is required to answer the requests of all its clients in finite time, even if some of the clients flood it with an infinity of requests. Such a requirement for fair treatment is unable to fulfill with functional constructs only. As we want to model reactive systems, the introduction of nondeterministic elements into Eden is inevitable. We are aware that we have to restrict the use of nondeterministic components as far as possible because otherwise Eden would be expelled from the paradise of programming into the hell of semantic chaos. Therefore we provide the predefined nondeterministic process `MERGE` that introduces nondeterminism in a way that it is hidden from the user.

The `MERGE` process abstraction creates a nondeterministic *fair* merging process for a list of stream channels with type `[msg]` each. In this case, we need the channel annotations introduced in Section 3.3 in order to prevent the system from interpreting the list of list structure as *one* stream, which in the presence of infinite lists would render the implementation of fairness impossible.

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

These process abstractions are typically employed to model many-to-one and one-to-many communication, respectively. We will conclude this section with a master-worker system that can handle different list-structured workloads.

Example (master-worker algorithm) In this example generic master and worker process abstractions are presented. The master uses a function `collect` to both gather the results generated by the `n` workers and distribute the remaining work (the items except the first `n` items, which were assigned statically) to the workers that finished their previous work. The messages sent by the workers always contain the result generated and a reply channel for requesting more work. The master applies a function `order` to all the results received, which arranges them according to the values of the keys (not shown here). The key 0 is used to encode the fact that no work is left.

```
master :: Int -> [a]  -> Process [(Int, b, Chan_name(Int,a))] [b]
master    n      items =
    process inp -> order (collect (drop n items) (n+1) inp)

    where collect:: [a] -> Int -> [(Int, b, Chan_name(Int,a))]->[(Int,b)]
          collect []  m [] = []  -- no items left, no requests
          collect []  m ((key, res, next):inrest) =
                  next !* (0,[]) par (key,res) : collect [] m inrest
          collect (r:rs) m ((key, res, next):inrest) =
                  next !* (m,r) par (key,res) : collect rs (m+1) inrest
```

The workers receive messages from the master that contain a key and an item of new work. This item is cmposed with the worker's own item `bs` by means of a function `comp`, which both are given as parameters. The worker receives one pair `as = (idx,rs)` as initial work and then always receives more work when returning the previous result.

```
worker ::  a -> (a->a->b) -> Process (Int,a) [(Int, b, Chan_name(Int,a))]
worker     bs    comp     =  process  as     -> work as


    where work :: (Int, a) -> [(Int, b, Chan_name(Int,a))]
          work (0  ,rs) = []            -- key 0: no work, empty reply
          work (idx,rs) = new(chan, task).(idx, comp rs bs, chan) : work task
```

We can now implement the matrix-vector multiplication by creating `n` instances of `worker` and one of `master` to coordinate their work. The workers receive now the function `sprod` as a parameter.

```
matVect2 :: [[Int]] -> [Int] -> Int -> [Int]
matVect2      rows       v       n  = master n rows # fromworkers
         where fromworkers = MERGE # (genprc rows v n)
               genprc rs       v    0 = []
               genprc (row:rs) v  idx
                    = (worker v sprod # (idx,row)):genprc rs v (idx-1)
```

4 Related Work

Explicit parallelism can elegantly be expressed by introducing two distinct language levels: The lower level, which is also called *computation language*, is formed by a sequential programming language. The upper level, the so-called *coordination language*, introduces and controls parallelism. In the setting presented here, the isolation of this language level has the advantage that non-functional constructs can be adopted without affecting the semantics of the functional part.

This *separation* approach was advocated in a more general context by Gelernter and Carriero [6], who establish **Linda** as an all-purpose coordination language that can be combined with various computation languages. They identify the use of such an orthogonal coordination language as a prerequisite for a clear and general representation of communication issues. It is contrasted with the *integration* principle followed by parallel programming languages which incorporate both coordination and computation within the same framework.

Facile [7] was one of the first language designs to combine functional and concurrent programming. It provides an interface between process calculus expressions and functional expressions using CCS-like constructs for concurrent composition and non-deterministic choice and thus models synchronized explicit communication on a rather low level of abstraction. A further pioneer of the concurrent functional programming idea was **Caliban** [11] that defines processes by very powerful annotations, but only restricted to the case of static networks.

The younger languages **Concurrent ML** (CML)[18], **LCS**[1] and **Concurrent Haskell** [17] aim at a level of abstraction which is lower than that of Eden. The main intention of theses languages is the extension of a functional kernel language by low-level primitives like fork (spawn), send and receive which can be implemented efficiently and on top of which more powerful abstractions can be defined. CML uses the strict language ML and provides synchronous communication, while Concurrent Haskell is based on the lazy evaluation paradigm (as Eden) and provides *mutable variables*, some sort of channels with one value buffers, for synchronisation and communication purposes.

However, most languages assume a system with shared memory, as reflected for example in the global *tuple space* of Linda, the *mutable variables* of Concurrent Haskell, or the global *constraint store* of Saraswat's CP language family. A two-level approach suitable for a distributed memory system is taken in the implicitly parallel functional and object-oriented language **Maude** [14]. There are functional, system and object-oriented modules, where system modules are used to encapsulate nonfunctional elements. The semantics of Maude is based on *rewriting logic*, which is described in [13].

Another concurrent declarative language that does not rely on shared memory is **Goffin** [4], that like Eden uses Haskell as its computation language. Via a coordination language with concurrent constraints, it offers explicit parallelism, but no explicit notion of a process. Messages are exchanged via logical variables only.

5 Conclusion and Future Work

Looking for a declarative language suitable for the description of transformational systems as well as reactive ones, we have succeeded in integrating a lazy computation model with a coordination model that aims at maximum parallelism. Another advantage of our coordination model is the flexibility on designing communication topologies, achieved by dynamic reply channels, and reinforced by an automatic bypassing mechanism.

A prototype implementation of Eden on top of Concurrent Haskell has been carried out [12] and a parallel implementation on a IBM SP/2 is under way. Both are described in a separate paper on implementation aspects [2]. On the theoretical side, we aim at obtaining a denotational semantics of the whole language which can be used as a foundation for powerful analysis tools. For more information, the reader is also referred to the w3-page `http://www.mathematik.uni-marburg.de/~breiting/eden.html`.

References

[1] B. Berthomieu. Process Calculi at Work: An Account of the LCS Project. In *PSLS*, LNCS 1068. Springer, 1995.

[2] S. Breitinger, U. Klusik, and R. Loogen. Implementation Aspects of Eden. In W. Kluge, editor, *Workshop on the Implementation of Functional Languages, Bonn, Germany*. Christian-Albrechts-University Kiel, September 1996.

[3] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In D. N. Turner, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer, 1995.

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

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

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

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

[9] J. Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1992.

[10] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP 77*. North Holland, 1977.

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

[12] U. Klusik. Implementaton of the functional parallel language Eden on top of Concurrent Haskell. Diplomarbeit. Philipps-Universität Marburg, 1996. in English.

[13] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73 –155, 1992.

[14] J. Meseguer and T. Winkler. Parallel Programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 253–293. Springer, 1992.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[16] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[17] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symposium on Principles of Programming Languages (POPL) 96*. ACM Press, 1996.

[18] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

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

[20] D. Turner (ed.). *Research Topics in Functional Programming*. Addison Wesley, 1992.