Approaches to High-Level Programming and Prototyping of Concurrent Applications

Wilhelm Hasselbring
University of Dortmund

Concurrent programming is conceptually harder to undertake and to understand than sequential programming, because a programmer has to manage the coexistence and coordination of multiple concurrent activities. To alleviate this task several high-level approaches to concurrent programming have been developed. For some high-level programming approaches, *prototyping* for facilitating early evaluation of new ideas is a central goal.

Prototyping is used to explore the essential features of a proposed system through practical experimentation before its actual implementation to make the correct design choices early in the process of software development. Approaches to prototyping concurrent applications with very high-level programming systems intend to alleviate the development of parallel algorithms in quite different ways. Early experimentation with alternate design choices or problem decompositions for concurrent applications is suggested to make concurrent programming easier.

This paper presents a survey of approaches to high-level programming and prototyping of concurrent applications to review the state of the art in this area. The surveyed approaches are classified with respect to the prototyping process.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming, Distributed programming*; D.2.2 [**Software Engineering**]: Tools and Techniques—*Computer-aided software engineering (CASE), Petri nets, Software libraries*; D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages, Very high-level languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms: Languages

Additional Key Words and Phrases: Concurrency, Parallelism, Distribution, Rapid prototyping, Very high-level languages

Address: Computer Science Department, Informatik 10, D-44221 Dortmund, Germany; email: ⟨willi@ls10.informatik.uni-dortmund.de⟩.

Contents

1. INTRODUCTION

There has been particular attention on concurrent programming within the computer science community during the last decades. We regard *concurrency* as the potential for *parallel* execution and *distribution* of activities. Thus, concurrent programming encompasses parallel programming and distributed programming. Several motivations for concurrent programming exist:

(1) Decreasing the execution time for an application program.
(2) Increasing the fault-tolerance.
(3) Exploiting explicitly the inherent parallelism of an application.

Achieving speedup through parallelism is a common motivation for executing an application program on a parallel computer system. Usually, parallel programming aims at high-performance computing. Another motivation is achieving fault-tolerance: for critical applications like controlling a nuclear power plant, a single processor may not be reliable enough. Distributed computing systems are potentially more reliable: as the processors are autonomous, a failure in one processor does not affect the correct function of the other processors. Fault-tolerance can, therefore, be increased by replicating functions or data of the application on several processors. If some of the processors crash, the others can continue the job.

However, the main motivation for integrating explicit parallelism into high-level prototyping languages is to provide means for explicitly modeling concurrent applications. Consider, for instance, concurrent systems such as air-traffic-control and airline-reservation applications, which must respond to many external stimuli and which are therefore inherently parallel and often distributed. To deal with nondeterminism and to reduce their complexity, such applications are preferably structured as independent parallel processes.

Combining concurrent programming with prototyping intends to alleviate concurrent programming on the basis of enabling the programmer to practically experiment with ideas for concurrent applications on a high level neglecting low-level considerations of specific parallel and distributed architectures in the beginning of program development. Prototyping concurrent applications intends to bridge the gap between conceptual design of concurrent applications and practical implementation on specific parallel and distributed systems.

To be useful, prototypes must be *built* rapidly, and designed in such a way that they can be *modified* rapidly. Therefore, prototypes should be built in very high-level languages to make them rapidly available. Consequently, a prototype is usually not a very efficient program since the language should offer constructs which are semantically on a very high level, and the runtime system has a heavy burden for executing these highly expressive constructs. The above-mentioned primary goal of parallel programming — decreasing the execution time for an application program — is not the first goal for prototyping concurrent applications. The first goal is to experiment with ideas for concurrent applications before mapping programs to specific parallel architectures to achieve high speedups.

Section 2 presents a survey of approaches to high-level programming and prototyping of concurrent applications. For each approach, first the general idea is discussed. Then, a short presentation of an example system follows before some sample systems are listed. The short example is intended to give a first impression of the particular approach. For the listed sample systems a brief characterization and some references are given. We do not intend to provide a comprehensive bibliography in this paper. For each included approach just a few representative references will be given. We also do not intend to provide a comprehensive survey of concurrent programming languages. Only approaches which are designed for prototyping or appear to be good candidates for prototyping are included. The transformation of prototypes into efficient implementations is discussed in Section 3, and Section 4 draws some conclusions.

2. APPROACHES

2.1 Low-Level versus High-Level

There exist many approaches to concurrent programming. The traditional model of *message passing* is that of a group of sequential processes running in parallel and communicating through passing messages. This model directly reflects the distributed memory architecture, consisting of processors connected through a communication network. Many variations of message passing have been proposed. With *asynchronous* message passing, the sender continues immediately after sending the message. With *synchronous* message passing, the sender must wait until the receiver accepts the message. Remote procedure call and rendezvous are two-way interactions between two processes. Broadcast and multicast are interactions between one sender and many receivers. Languages based on the message passing model include occam, Ada, SR, and many others. As these languages with their variations of message passing have been studied extensively in the literature, we refer to Bal, Steiner, and Tanenbaum [1989] for an overview, and do not discuss them in detail here.

For some applications, the basic model of message passing may be the optimal solution. This is, for example, the case for an electronic mail system. For other applications, however, this basic model may be too low-level and inflexible.

Processes that are collaborating on a problem will ordinarily need to share data, but in the message-passing model data structures are sealed within processes, and so processes cannot access the others' data directly. Instead they exchange messages. This scheme adds complexity to the program as a whole: it means that each process must know how to generate messages and where to send them. This greatly increases the complexity of programs, and also restricts algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. Refer to Bal [1990] for an extensive discussion of the shortcomings of the message-passing model. To quote from Agha [1996]: "Programming using only message passing is somewhat like programming in assembler: sending a message is not only a jump, it is a concurrent one."

In contrast to the message-passing model, the shared-memory model allows application programs to use shared memory as they use normal local memory. The primary advantage of shared memory over message passing is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well.

Because of the problems with the low-level programming models for message passing, many models which emphasize some kind of *shared data* have been developed that intend to deliver a higher level of abstraction to alleviate concurrent programming. These high-level programming models appear to be good candidates for prototyping concurrent applications.

The traditional method for communication and synchronization with shared data is through shared variables. The use of shared variables for coordination of concurrent processes with, e.g., semaphores or critical sections has been studied extensively [Andrews 1991]. However, we regard shared variables as a low-level medium for coordination, because the synchronization, which is necessary to prevent multiple processes from simultaneously changing the same variable (avoiding lost updates), is difficult. Several other coordination models based on shared data exist, however, which are better suited for concurrent programming and consequently proposed for prototyping concurrent applications.

Our survey of approaches to high-level programming and prototyping of concurrent applications starts with high-level libraries for message passing (Section 2.2) to continue with data parallelism (Section 2.3), concurrent functional languages (Section 2.4), concurrent logic languages (Section 2.5), concurrent object-based languages (Section 2.6), composition/coordination languages (Section 2.7), and graphical programming systems (Section 2.8). Figure 1 classifies the approaches surveyed in the paper into a simple taxonomy.

2.2 High-Level Libraries

Idea. Mechanisms for concurrent programming are often provided to the programmer through libraries of functions on an operating-system level. As the use of many of these libraries is very complicated, it has been proposed to use some kind of high-level libraries for prototyping concurrent applications. These high-level libraries intend to alleviate their use by the provision of simple interfaces and more flexibility than the lower-level libraries provide, while relinquishing efficiency to some extent.

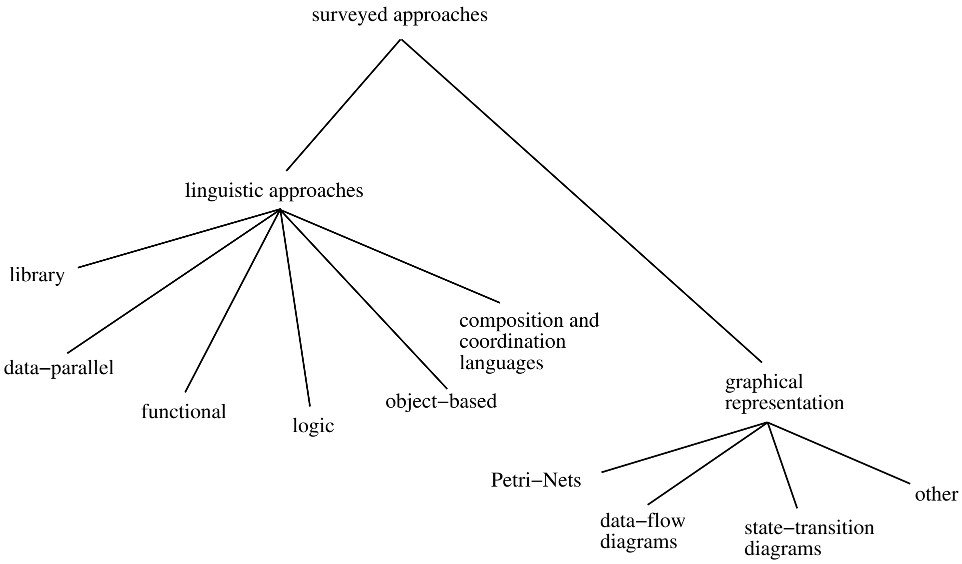

Fig. 1. A taxonomy for the approaches surveyed in the paper.

Such libraries only support unstructured programming since there exists no compiler support for checking the proper use of the libraries. It could be argued that these approaches are not really *high-level* approaches, but they are included in this survey because some are explicitly designed for prototyping concurrent applications.

An Example. IPC_FBase [Cao et al. 1993] is a library which supports prototyping on Transputer networks. The Transputer is a processor providing four serial links, which may be connected to other Transputers. Transputer processors are usually configured into a two-dimensional grid. To alleviate programming of such systems, the library IPC_FBase supports high-level functions for routing and dynamic reconfiguration to allow early prototyping and performance evaluation of different parallel algorithms on Transputer networks. Examples for topology emulation functions are:

soft_pipe(NumberOfNodes)
soft_ring(NumberOfNodes)
soft_tree(NumberOfNodes)
soft_star(NumberOfNodes)

Corresponding to each basic network topology, a routing function is provided which supports two-way communication (*routing_pipe()* etc.). These functions are intended to offer the programmer the possibility to experiment with parallel algorithms for different processor topologies. With traditional programming languages for Transputer systems (e.g. occam or Parallel-C), it is somewhat arduous to simulate and reconfigure different topologies on the grid-based architecture.

As a case study, Cao, de Vel, and Wong [1993] discuss a parallel tree search algorithm which uses the tree topology. The algorithm uses the divide-and-conquer technique, in which the overall search is broken down into a collection of smaller searches that can be undertaken in parallel. With the tree topology, the search involves communication of a search key to the root, dispersion of the key through the branches and to all the leaves, check for the existence of the key in each leaf, and to gather in the results from all the leaves via the branches. With the functions for emulation of and routing in trees, it was straightforward to implement the first versions of this parallel algorithm.

Some Sample Systems. In addition to IPC_FBase, there exist some other approaches to prototyping with high-level libraries:

—Polylith [Purtilo et al. 1988; Purtilo and Jalote 1991] is a module library for prototyping concurrent applications, that supports different communication primitives with specified delays, and provides primitives to aid debugging and evaluation. The environment also supports heterogeneous computation in which processes can execute on different hardware. Different source languages can be used for coding different modules of the processes.

—StarLite [Son and Kim 1989] provides a library of functions for prototyping transaction processing mechanisms for distributed database systems.

—The high-level library VAN (Virtual Agent Network) for prototyping computational agents in a distributed environment is presented in French and Viles [1992].

—It has also been proposed to prototype distributed information systems with remote procedure call (RPC)-based programs [Zhou 1994].

2.3 Set-Oriented Data Parallelism

Idea. Data parallelism extends conventional programming languages so that some operations can be performed simultaneously on many pieces of data. For example, all elements in a list or in an array can be updated at the same time, or all items in a database are scanned simultaneously to see if they match some criterion. For an account to data-parallel algorithms refer to Hillis and Steele [1986] and for an account to data-parallel programming refer to Quinn and Hatcher [1990]. Data-parallel operations appear to be done *simultaneously* on all affected data elements. This kind of parallelism is opposed to *control parallelism* that is achieved through multiple threads of control, operating independently.

Data parallelism is a relatively well-understood form of parallel computation as it is the simplest of all styles of concurrent programming, yet developing simple applications can involve substantial efforts to express the problem in low-level data-parallel notations. Approaches to prototyping data-parallel algorithms usually extend a sequential prototyping language with some high-level data-parallel constructs to allow experimentation with different data and problem decompositions. In contrast to most traditional data-parallel approaches, high-level data-parallel approaches usually provide some kind of set-oriented, nested data-parallel constructs.

The data-parallel approach lets programmers replace iteration (repeated execution of the same set of instructions with different data) by parallel execution. It does not address a more general case, however: performing many interrelated but *different* operations at the same time. This ability is essential in developing complex concurrent application programs. Therefore, the capabilities of the data-parallel approach for prototyping concurrent applications are limited.

An Example. Let us take a look at a data-parallel extension of SETL. SETL is a set-oriented language designed for prototyping (sequential) algorithms [Kruchten et al. 1984]. In PSETL [Hummel and Kelly 1993], parallelism is introduced into SETL through the use of *explicit parallel* iterators, which are used in iterator expressions and in loops over sets and tuples. For instance, the instructions in the following nested loop are executed in parallel and not executed in sequential iterations:

```
MySet := {};
for Range par_over {[1..1000], [3000..5000], [8500..9000]} do
    MySet := MySet + {f(Index): Index par_over Range};
end for;
```

The expression "`{f(Index): Index par_over Range}`" is a nested parallel loop over the tuple `Range`. This example computes the set of all values of some function `f` applied to the indexes in the given ranges.

Some Sample Systems. In addition to PSETL, there exist some other approaches to prototyping data-parallel algorithms:

—Parallel ISETL [Jozwiak 1993] is a variation of ISETL [Dubinsky and Leron 1993] that supports data parallelism. ISETL is an interactive implementation of SETL.

—Proteus [Mills et al. 1991] is another variation of ISETL that supports control and data parallelism for prototyping concurrent applications. In addition to the data-parallel constructs, Proteus supports *future* synchronization (see Section 2.4) and shared variables for control parallelism.

—NESL [Blelloch 1996] is a data-parallel extension to a set-oriented language which emphasizes nested data-parallel constructs. NESL is somewhat different as it has an ML-like syntax with strong typing. It has been designed for teaching and high-level programming. The NESL compiler is able to generate efficient code on the basis of an underlying performance model [Blelloch et al. 1994].

2.4 Concurrent Functional Languages

Idea. A functional program comprises a set of equations describing functions and data structures which a user wishes to compute. The application of a function to its arguments is the only control structure in pure functional languages. Functions are regarded in the mathematical sense that they do not allow side effects. As a consequence, a value of a function is determined solely by the values of its arguments. No side effects are allowed. No matter what order of computation is chosen in evaluating an expression list, the program is guaranteed to give the same result (assuming termination).

Therefore, functional programs are implicitly parallel. Because they are free of side effects, each function invocation can evaluate all of its arguments and possibly the function body in parallel. The only delay may occur when a function must wait on a result being produced by another function.

However, the real problem with efficiency in functional programs is not discovering parallelism but reducing it so as to keep the overhead on an acceptable level. Concurrent functional languages address this problem by allowing the programmer to insert annotations which specify when to create new threads of control. Multilisp

[Halstead 1985] is a typical parallel functional language, which augments Scheme with the notion of *futures* where the programmer needs no knowledge about the underlying process model, inter-process communication or synchronization to express parallelism. She or he only indicates that she/he does not need the result of a computation immediately (but only in the "future") and the rest is done by the runtime system. Refer to Szymanski [1991] for a collection of papers on several concurrent functional languages.

Processes sometimes cooperate in a way that cannot be predicted. It is impossible, for instance, to predict from which terminal of a multi-user computing system the next request for a particular service might come. Moreover, the system behavior necessarily depends on previous requests. Therefore, pure functional languages are not suitable for programming cooperating processes: they are deterministic and they do not have variables. Processes described as functions cannot include choices of alternative actions and they cannot remember their states from one action to another. Nondeterminism would destroy referential transparency in functional programming languages. Both nondeterminism of events and dependence on the process history are strong arguments for an imperative rather than applicative programming model for cooperating processes. This is due to the determinism and the lack of variables which make pure functional languages impractical for programming concurrent applications. Therefore, approaches to prototyping concurrent applications with functional languages usually support some kind of state.

An Example. In the PSP approach [Heping and Zedan 1996], a functional language has been extended with *state variables* to allow prototyping parallel responsive systems. A PSP function for changing the direction in a lift system is presented as an illustrating example:

```
fun     Change_direction()r:bool
=       let  val up_request = (exists i mem
                    (up_buttons sor down_buttons sor panel_buttons)
                    suchthat i > position)
             val down_request = (exists i mem
                    (up_buttons sor down_buttons sor panel_buttons)
                    suchthat i < position)
        in
             if (Been_served(0) and (up_request or down_request))
             then  if (direction and up_request)
                   then true
                   else if (not direction and down_request)
                        then false
                        else not direction
             else direction
        end
```

In this small example, position, direction, up_request, down_request, up_buttons, down_buttons, and panel_buttons are global state variables. Refer to Heping and Zedan [1996] for a discussion of the complete prototype implementation of this lift system.

Some Sample Systems. Another approach extends a functional language with state variables to support prototyping:

—PAISLey [Zave and Schell 1986; Nixon et al. 1994] is a functional programming language that combines asynchronous processes and functional programming to overcome the problems with pure functional languages for programming cooperating processes. Parallelism in PAISLey is based on a model of event sequences and used to specify functional and timing behavioral constraints for asynchronous parallel processes. Each process computes some function of its inputs and runs in parallel with the other processes. Each computation is activated by an event. The process then evaluates the function and returns the result. This result is used as the process state. Therefore, the life of a process is represented by a series of state changes, each of which is considered to be the result of a computation. The connection between two functions is established via *channel attributes* of the function calls.

However, some pure functional languages have also been used for prototyping:

—The Crystal approach [Chen et al. 1991] starts from pure functional programs (prototypes) through a sequence of transformations to the generation of efficient target code with explicit communication and synchronization.

—The functional subset of Standard ML has been used for prototyping parallel algorithms for computer vision [Wallace et al. 1992; Michaelson and Scaife 1995].

2.5 Concurrent Logic Languages

Idea. Logic programming languages, of which PROLOG [Clocksin and Mellish 1987] is best known, express programs as a set of *clauses*, which may be read procedurally or declaratively. For example, the following clauses:

```
A :- B,C,D
A :- E,F
```

can be interpreted as "to do `A`, do either `B`, `C`, and `D`, or do `E` and `F`". Alternatively, we can view it as "`A` is true, if either `B`, `C`, and `D` are true, or `E` and `F` are true".

PROLOG programs express two distinct forms of implicit parallelism. Firstly, several different clauses may be evaluated separately. This is called OR-parallelism, since only one of them must succeed. Secondly, each subgoal (in the above example `B`, `C`, `D`, `E`, and `F` are subgoals) can be executed in parallel, although data dependencies may limit the extent of parallelism. This is called AND-parallelism, since all of the subgoals must succeed for the clause to succeed. AND-parallelism is the simultaneous reduction of several different subgoals in a goal; OR-parallelism is the simultaneous evaluation of several clauses for the same goal. AND/OR-parallelism is implicit parallelism similar to the implicit parallelism found in functional languages, which does not give additional expressiveness.

AND-parallel committed choice logic is an approach to concurrent logic languages which uses guards in the clauses [Shapiro 1989]. Subgoals are unified in parallel. This approach uses *shared logical variables* as a communication medium. In OR-parallel committed choice logic, all clauses that match a goal are tried in parallel and when one can be unified, the execution to this clause commits. If more clauses can be unified, one is chosen nondeterministically. A clause in committed choice logic has the structure:

$$\texttt{A :- G}_1\texttt{, ..., G}_m \texttt{ | B}_1\texttt{, ..., B}_n$$

$\texttt{G}_1\texttt{, ..., G}_m$ are guards, and the rest is like ordinary PROLOG clauses. Such a clause includes a *commit* operator `|` (omitted if the guard is empty) used to separate the right-hand side of the clause into a conjunction of guard conditions and a conjunction of subgoal predicates. The guard conditions must evaluate to **`true`** to enable the evaluation of the subgoals. The commit expresses *don't-care* nondeterminism, i.e., the termination of the OR-parallel evaluation of alternative clauses: if more than one clause can apply to reduce a subgoal, one is chosen arbitrarily and no backtracking can take place if that choice later results in a failure. The underlying idea in the family of committed choice concurrent logic languages is to model synchronization between processes by imposing some constraints on the unification mechanism. Shapiro [1989], Ciancarini [1992], and de Kergommenaux and Codognet [1994] present surveys of several concurrent logic languages and systems.

As PROLOG has been used for prototyping sequential algorithms [Budde et al. 1984], concurrent logic languages seem to be good candidates for prototyping concurrent applications.

An Example. As an example for a committed choice logic program, let us consider a parallel quick-sort program in the Reactive Guarded Definite Clauses (RGDC) notation, which is copied from Huntbach and Ringwood [1995]:

```
qsort([ ], L)            :- L:=[ ].
qsort ([H|T], Q)         :- qsort(S,SS), qsort(L,SL), append(SS, [H|SL], Q),
                            part(T,H,S,L).
part([ ], X, S, L)       :- S:=[ ], L:=[ ].
part([H|T], X, S, L)     :- H<X | S:=[H|S1], part(T,X,S1,L).
part([H|T], X, S, L)     :- H≥X | L:=[H|L1], part(T,X,S,L1).
append([ ], L, L1)       :- L1:=L.
append([H|T], L, A)      :- A:=[H|A1], append(T, L, A1).
```

Parallel sorting is split into parts using the guards $H<X$ and $H \geq X$. Synchronization is implicit through data dependencies. The symbol | is overloaded: it is used for list concatenation and to separate the guards. For a detailed discussion of this compact parallel quick-sort program refer to Huntbach and Ringwood [1995].

At least for developers who are familiar with logic programming, this approach provides a way for experimenting with concurrent applications on a very high level.

Some Sample Systems

—Strand [Foster and Taylor 1989] is a commercial system based on committed choice logic which comprises a language, a development environment and concurrent programming libraries to support prototyping of parallel algorithms.

—PROLOG has been used for simulation and prototyping of Estelle protocol specifications in the Véda system [Jard et al. 1988].

2.6 Concurrent Object-Based Languages

Idea. An approach to imperative programming which has gained widespread popularity is that of object-oriented programming [Meyer 1997]. In this approach, an object is used to integrate both data and the means of manipulating that data. Objects interact exclusively through message passing by method invocation and the data contained in an object is visible only within this object itself. The behavior of an object is defined by its class, which comprises a list of operations that can be invoked by sending a message to an object. All objects must belong to a class. Objects in a class have the same properties and can be manipulated using similar operations. The definition of an object class can act as a template for creating instances of the class. Each instance has a unique identity, but has the same set of data properties and the same set of operations which can be applied to it.

Inheritance allows a class to be defined as an extension of another (previously defined) class. Typically when a new class is created, a place for it is defined within the class hierarchy. The effect of this is that the new class inherits the state attributes and operations of its superclass in the hierarchy. Objects may inherit features from more than one class in some approaches (multiple inheritance).

There are several possibilities for the introduction of concurrency into object-oriented languages, viz.:

—Objects are active without having received a message.

—Objects continue to execute after returning results.

—Messages are sent to several objects at the same time.

—Senders proceed in parallel with receivers.

These possibilities can be realized by associating a process with each object. Just as a parallel-processing environment implies multiple processes, a concurrent object-oriented system spawns multiple active objects, each of which can start a thread of execution. Objects are usually addressed by an object *reference* (returned upon creation of the object) or by a global object name.

Concurrent object-oriented languages use three types of communication: synchronous, asynchronous, and eager invocation. Synchronous communication uses remote procedure calls. It is easiest to implement, but sometimes inefficient because of the necessity for both the sender and receiver to *rendezvous*. Asynchronous communication eliminates the wait for synchronization and can increase parallel activity. Eager invocation, or the *futures* method, is a variation of asynchronous communication (see also our discussion of futures in Multilisp in Section 2.4). As in asynchronous operation, the sender continues executing, but a *future variable* holds a place for the result. The sender executes until it tries to access the future variable. When the result has been returned, the sender continues; if not, it blocks and waits for the result.

Probably the most difficult aspect of integrating parallelism into object-oriented languages is that inheritance greatly complicates synchronization. When a subclass inherits from a base class, programs must sometimes redefine the synchronization constraints of the inherited method. If a single centralized class explicitly controls message reception, all subclasses must rewrite this part each time a new operation is added to the class. The subclass cannot simply inherit the synchronization code, because the higher-level class cannot invoke the new operation of the subclass. The concurrent object-oriented languages resolve these synchronization problems in different ways as discussed in Agha [1990]. Matsuoka and Yonezawa [1993] provide a detailed discussion of this so-called *inheritance anomaly* in concurrent object-oriented languages, where re-definitions of inherited methods are necessary in order to maintain the integrity of parallel executing objects. Inheritance anomaly represents the situation where the synchronization on a parent class needs to be changed as a result of the extension on that class via inheritance [Mitchell and Wellings 1996]. This anomaly could eliminate the benefits of inheritance. Meyer introduces the subject with the warning:

> "Judging by the looks of the two parties, the marriage between concurrent computation and object-oriented programming — a union much desired by practitioners in such fields as telecommunications, high performance computing, banking and operating systems — appears easy enough to arrange. This appearance, however, is deceptive: the problem is a hard one." [Meyer 1993, page 56]

Due to the problems for synchronization which are caused by inheritance, concurrent *object-based* systems have been suggested for prototyping concurrent applications. Object-based languages do not support inheritance. The actor model is a classical example for an object-based model [Agha 1986; Agha 1996]. Actors extend the concept of objects to concurrent computation: Each actor potentially executes in parallel with other actors and may send messages to actors of which it knows the addresses.

To our knowledge there exists no published approach to prototyping concurrent applications with concurrent object-oriented languages, in spite of the fact that object-oriented languages — in particular Smalltalk — appear to be good candidates for prototyping sequential systems [Barry 1989; Budde et al. 1992; Bischofberger and Pomberger 1992].

An Example. PDC [Weinreich and Ploesch 1995] is a concurrent object-based system that extends C++ to use operating system processes as active objects. The communication between active objects is handled by executing so-called *remote method calls* which may be synchronous or asynchronous. It is not allowed to modify or redefine *remote methods* through inheritance to avoid the problems with the inheritance anomaly. Figure 2 displays a snapshot of active objects in PDC. Each active object is a sequential process implemented in C++. The prototyping idea here is to provide simple mechanisms for communication between active objects. PDC is accompanied by the graphical tool ProcessBuild, which offers a graphical representation for configuring the active objects (see also Section 2.8).

Some Sample Systems. Some other concurrent object-based systems have been suggested for prototyping parallel algorithms:

—RAPIDE [Luckham et al. 1993] is a concurrent object-based language specifically designed for prototyping parallel systems that combines the partially ordered event set (poset) computation model with an object-oriented type system for the sequential components.

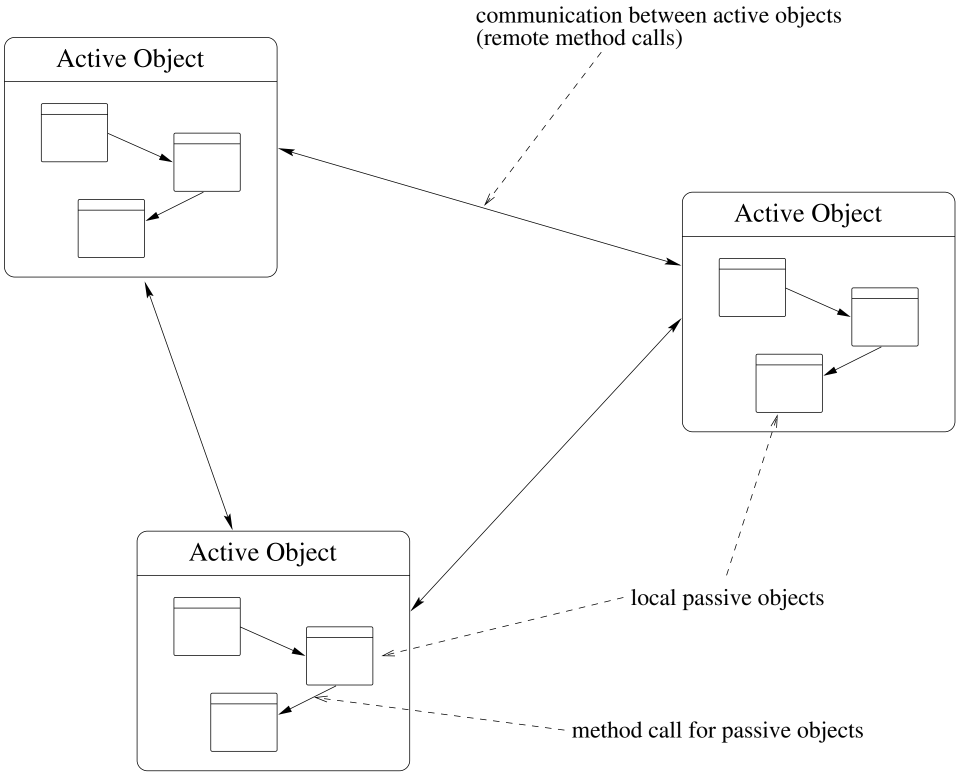

Fig. 2. A snapshot of active objects in PDC. Each active object may contain several passive objects, but no active objects.

—The PROTOB [Baldassari et al. 1991] approach considers nodes in PROT nets [Bruno and Marchetto 1986] to be communicating objects (see also Section 2.8). Inheritance is not supported by PROTOB.

This is not an exhaustive list of concurrent object-based languages. Other examples are the ABCL [Yonezawa 1990] series of languages and HAL [Houck and Agha 1992], which could be used for prototyping concurrent applications. Agha, Wegner, and Yonezawa [1993] provide a collection of papers on various concurrent object-oriented approaches, and Wyatt, Kavi, and Hufnagel [1992] provide a survey of concurrent object-oriented languages.

2.7 Composition and Coordination Languages

Idea. With composition and coordination languages, programming is split in two separate activities: a sequential language is used to build single-threaded computations, whereas a coordination language is used to coordinate the activity of several single-threaded computations. A *coordination language* provides means for process creation and inter-process communication which may be combined with sequential *computation languages* [Carriero and Gelernter 1992]. The concurrent extensions of logic, functional and object-oriented languages discussed in the preceding subsections are coordination languages. The present subsection discusses coordination languages which are somewhat independent of the computation language.

With composition and coordination languages, concurrent systems are described in terms of processes that comprise a system and the communication and control interconnections between these processes. As discussed in Ciancarini [1996], there is a need for high-level coordination languages to simplify the design and implementation of concurrent applications, because most software engineers currently develop concurrent applications using low-level communication primitives.

An Example. A coordination language should orthogonally combine two languages: one for coordination (the inter-process actions) and one for (sequential) computation [Carriero and Gelernter 1992; Ciancarini 1996]. ProSet-Linda [Hasselbring 1994; Hasselbring 1997] combines the sequential prototyping language ProSet [Doberkat et al. 1992] with the coordination language Linda [Gelernter 1985] to obtain a concurrent programming language as a tool for prototyping concurrent applications. ProSet is an acronym for PROtotyping with SETs. The procedural, set-oriented language ProSet is a successor to SETL [Kruchten et al. 1984]. The high-level structures that ProSet provides qualify the language for prototyping.

To support prototyping of concurrent applications, a prototyping language must provide simple and powerful means for dynamic creation and coordination of parallel processes. In ProSet-Linda, the concept for process creation via Multilisp's futures [Halstead 1985] (see also Section 2.4) is adapted to set-oriented programming and combined with Linda's concept for synchronization and communication. The parallel processes in ProSet-Linda are decoupled in time and space in a simple way: processes do not have to execute at the same time and do not need to know each other's addresses (this is necessary with synchronous point-to-point message passing). The shared data pool in the Linda concept is called *tuple space*, because its access unit is the tuple, similar to tuples in ProSet; thus, it is rather natural to combine both models on this basis. Reading access to tuples in tuple space is *associative* and not based on physical addresses, but rather on their expected content described in *templates*. This method is similar to the selection of entries from a data base. ProSet-Linda supports multiple tuple spaces. Several library functions are provided for handling multiple tuple spaces dynamically [Hasselbring 1994].

ProSet-Linda provides three tuple-space operations. The `deposit` operation deposits a tuple into a tuple space:

```
deposit [ "pi", 3.14 ] at TS end deposit;
```

`TS` is the tuple space at which the tuple `[ "pi", 3.14 ]` has to be deposited. The `fetch` operation tries to fetch and remove a tuple from a tuple space:

```
fetch ( "name", ? x ) at TS end fetch;
```

This template only matches tuples with the string `"name"` in the first field and integer values in the second field. The optional l-values specified in the formals (the variable `x` in our example) are assigned the values of the corresponding tuple fields, provided matching succeeds. Formals are prefixed by question marks. The selected tuple is removed from tuple space. The `meet` operation is the same as `fetch`, but the tuple is not removed and may be changed. Tuples which are met in tuple space can be regarded as shared objects since they remain in tuple space irrespective of changing them or not. With `meet`, in-place updates of specific tuple components are supported. For a detailed discussion of prototyping parallel algorithms with ProSet-Linda refer to Hasselbring [1994]. Some application experience with the ProSet-Linda approach is discussed in Hasselbring and Kröber [1997].

Some Sample Systems. Several other approaches to prototyping concurrent applications suggest the use of composition and coordination languages:

—ISETL-Linda [Douglas et al. 1995] is a control-parallel extensions to ISETL very similar to ProSet-Linda.

—The focus of the Parallel Composition Notation (PCN) [Chandy and Taylor 1992] approach, which is based on committed choice logic (see Section 2.5), is the development of programs by the *parallel composition* of simpler components, in such a way that the resulting programs preserve the properties of the components that they compose.

—The parallel constructs of Compositional C++ (CC++) [Chandy and Kesselman 1993] are based on the ideas of Strand and PCN, while using C++ for the sequential portions of the code. CC++ uses pure single assignment variables and not logical variables as found in other concurrent logic languages such as Strand or PCN. PCN and CC++ are explicitly designed for rapid prototyping of concurrent applications [Chandy and Taylor 1992; Chandy and Kesselman 1993].

—Durra [Barbacci and Lichota 1991] provides a configuration language through which one can specify the structure of Ada programs in conjunction with the behavior, timing, and implementation dependencies. These specifications may be validated by a run-time interpreter to allow prototyping.

2.8 Graphical Programming Systems

Idea. Graphical representations of parallel programs are annotated graphs: data flow graphs, control flow graphs etc. In particular for message-passing programs, such multi-dimensional representations appear to be a good way to get over the complex architecture of concurrent applications. The animation and simulation features, as well as the code generation from the graphical representations may be used to prototype some aspects of concurrent applications. Petri-Nets [Reisig 1985], state-transition diagrams like Statecharts [Harel 1987] and data-flow diagrams [DeMarco 1978] are often used to build prototypes for message-passing programs since they can be regarded as graphical representations of message-passing programs. A variety of these and other graphical representations are suggested for prototyping concurrent applications.

An Example. Enterprise [Schaeffer et al. 1993] is a programming environment for developing parallel programs. With Enterprise, the parallelism is expressed graphically independent of the sequential code. The system automatically inserts the code necessary to correctly handle communication and synchronization to allow the rapid construction of parallel programs.

Enterprise supports coarse-grained parallel programs which make use of a small number of regular techniques, such as pipelines, master/slave processes, and divide and conquer. Enterprise does not directly support arbitrarily structured parallel programs, but for many applications it relieves users from the tedious details of distributed communication to let them concentrate on algorithm development. The user specifies the desired technique by manipulating icons using the graphical user interface. The user interface is implemented in Smalltalk and allows program animation for prototyping [Lobe et al. 1993].

Figure 3 displays a snapshot while animating a parallel program with Enterprise. Program animation is used to monitor and to identify weak points in the implemented parallel algorithms. The double-line rectangle represents the *enterprise*, which is the entire program. Each icon represents the state of a parallel process. Message queues are displayed as connecting lines between the process icons. The system assumes that the displayed events are partially ordered. It is possible to monitor the parallel program while it is running or to replay the events. For a more detailed discussion of this example refer to Lobe, Szafron, and Schaeffer [1993].

Szafron and Schaeffer [1996] provide an experimental comparison of parallel programming with the Enterprise system and with message-passing libraries based on the experience in a graduate parallel and distributed computing course. This comparison supported the claim that higher-level tools can be more usable than low-level message-passing libraries.

Some Sample Systems. Petri-Nets, state-transition diagrams, data-flow diagrams and some own notations are suggested for prototyping concurrent applications:

(1) Petri-nets are a popular formalism for designing and analyzing parallel algorithms. Their simulation and animation can be used for prototyping concurrent applications:
   —In Breant [1991], it is proposed to use Petri-net prototypes for developing occam programs. Petri-nets are used to validate and evaluate a model before its implementation.

   While providing a valuable formalism with which to describe and analyze concurrent systems, plain Petri-Nets cannot help with system decomposition and structuring for large, complex systems. Several extensions of place-transition nets are proposed for prototyping concurrent applications:
   —Communicating Petri-nets (CmPNs) [Bucci and Vicario 1992] have been proposed for prototyping distributed systems. With CmPNs, subsystems are modeled as Petri-nets. Connection diagrams model the global system structure. A tool supports animation and code generation for prototyping.
   —The specification formalism Concurrent Object Oriented Petri Nets (CO-OPNs) [Buchs et al. 1992] combines algebraic specifications with Petri-nets. The specification is prototyped using a translation of the specification into PROLOG.
   —The G-Net [Deng et al. 1993] formalism extends Petri-nets with modules to allow prototyping of complex information systems.
   —Generalized Stochastic Petri-nets (GSPNs) [Donatelli et al. 1994] are proposed for prototyping functionality and performance of parallel algorithms.
   —The PARSE [Gorton et al. 1995] process graph notation allows to describe a parallel system in terms of a hierarchy of interacting components. These components are either passive function or data servers, or active control processes.

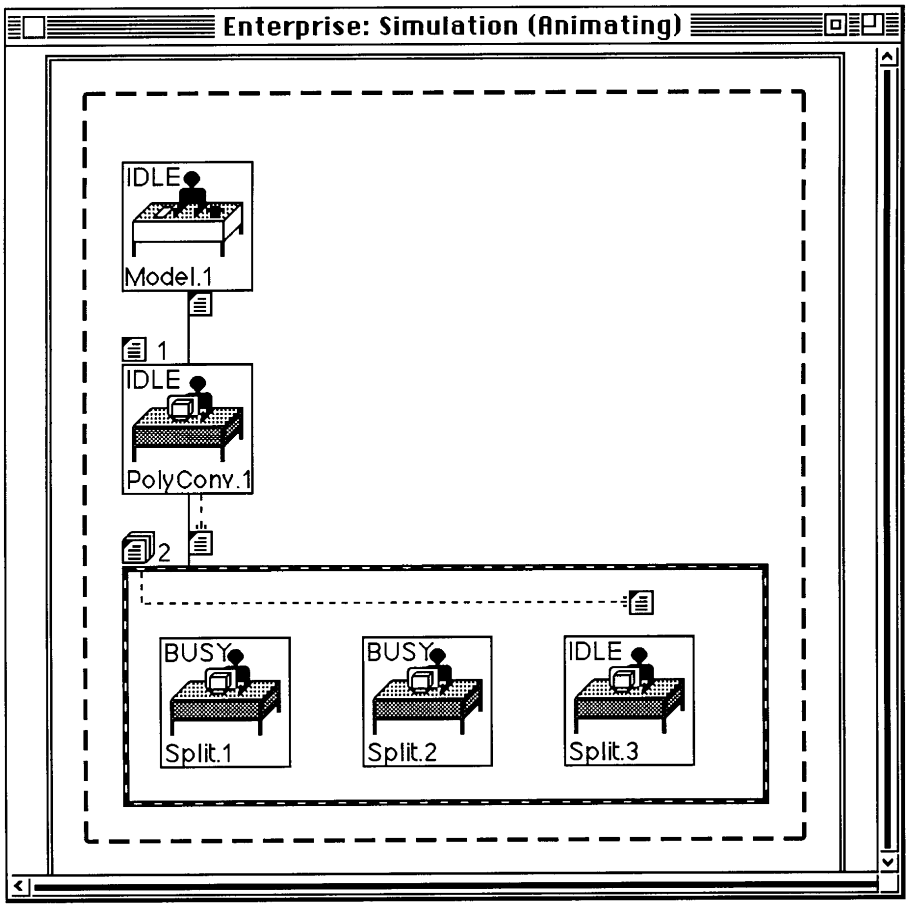

Fig. 3. Animating a parallel program with Enterprise. This is a copy of Figure 7 in Lobe, Szafron, and Schaeffer [1993].

Processes interact by message passing on designated communication paths. Within the PARSE project, the integration of behavioral analysis techniques involving Petri-Nets for design validation has been explored through (manual) translation of process graphs into Petri-Nets to offer a path to formal verification [Gorton et al. 1995]. Alternative architectural approaches can be derived and expressed in the PARSE process graph notation, and performance prototyping and formal validation tools allow various aspects of the proposed solution to be rapidly explored [Hu and Gorton 1997].

—PROT net [Bruno and Marchetto 1986] is a high-level Petri-net formalism for prototyping parallel systems. With PROT nets, tokens represent processes.

—The language SEGRAS [Krämer 1991] is based on an integration of algebraic specifications and high-level Petri-nets. Data objects are specified as abstract data types, while dynamic behavior is specified graphically by means of high-level Petri-nets. A subset of the language is executable to support prototyping. Execution is based on term rewriting and Petri-net simulation.

(2) Animation and simulation of communicating state-transition diagrams may be used for prototyping concurrent applications:

—Communicating Real-time State Machines (CRSMs) [Raju and Shaw 1994] are proposed for prototyping real-time systems. Individual CRSMs are similar to Statecharts [Harel 1987].

—For prototyping protocols, it has been proposed to translate LOTOS specifications into a set of Extended Finite State Machines (EFSMs) [Valenzano et al. 1993].

(3) Data-flow diagrams are a popular formalism in software engineering [Ghezzi et al. 1991]. They are proposed for prototyping concurrent applications, as well:

—Extended data-flow diagrams (EDFGs) [Levy et al. 1990] are proposed to support prototyping of distributed systems with an emphasis on client-server applications. With EDFGs, a parallel system is described as a set of communicating graphs, where each graph is either a server or a client.

—Formal data-flow diagrams (FDFDs) [Fugetta et al. 1993] with precise semantics for synchronization in combination with E/R-diagrams are proposed for prototyping parallel systems. Execution of FDFDs is based on their formal semantics.

—Data-flow diagrams are specified in Jones, Dowdeswell, and Hintz [1990] with IDE's Software Through Pictures tool (StP) to develop occam programs. The occam code is generated from StP's internal data dictionary to provide a first executable program for prototyping.

—The Prototype System Description Language PSDL [Luqi et al. 1988] uses data-flow diagrams with associated timing and control constraints to compose reusable components from a software library for prototyping real-time systems. The retrieval of reusable components is based on a term rewriting system. The components are implemented with Ada.

(4) Similar to Enterprise, many systems for prototyping concurrent applications use their own notation: JADE [Unger 1988], Transim [Hart and Flavell 1990], MCSE [Calvez et al. 1994], and ProcessBuild [Weinreich and Ploesch 1995] are

tools which support several graphical representations for prototyping through simulation and animation of parallel algorithms. These graphs represent data and control flow in various ways.

3. TRANSFORMING PROTOTYPES INTO EFFICIENT IMPLEMENTATIONS

A Prototype may be classified as throwaway, experimental or evolutionary [Floyd 1984]. A throwaway prototype describes a product designed to be used only to help identify requirements for a new system. Experimental prototyping focuses on the technical implementation of a development goal. In evolutionary prototyping, a series of prototypes is produced that converges to an acceptable behavior, according to the feedback from prototype evaluations. Once the series has converged, the result may be turned into a software product by *transformations*. Evolutionary prototyping is a continuous process for adapting the model of an application system to changing organizational constraints.

This raises issues of software engineering: once we are satisfied with the prototype, how do we transform it systematically into a production efficient program? Such transformations are usually accomplished manually or semi-automatically with some kind of tool support. Some manual transformations of prototypes into efficient implementations are discussed in the literature:

—The Crystal [Chen et al. 1991] approach starts from a high-level functional problem specification, through a sequence of optimizations tuned for particular parallel machines, leading to the generation of efficient target code with explicit communication and synchronization. This approach to automation is to design a compiler that classifies source programs according to the communication primitives and their cost on the target machine and that maps the data structures to distributed memory, and then generates parallel code with explicit communication commands. Regarding those classes of problems for which the default mapping strategies of the compiler are inadequate, Crystal provides special language constructs for incorporating domain specific knowledge by the programmer and directing the compiler in its mapping.

—Nixon and Croll [1993] manually transform PAISLey prototypes into occam programs.

—The stepwise refinement of PCN programs is discussed in [Chandy and Taylor 1992].

—In Hummel, Talla, and Brennan [1995], high-level parallel algorithm specifications are refined within PSETL [Hummel and Kelly 1993]. High-level PSETL code is successively transformed manually into lower-level architecture-specific PSETL code.

—The manual transformation of ProSet-Linda prototypes into efficient C-Linda and message-passing implementations is discussed in Jodeleit [1996] and in Kirsch [1996].

—It has been proposed to transform Standard ML prototypes into occam programs [Wallace et al. 1992].

Some kind of tool support to assist with the transformation has also been discussed:

—In Breant [1991], occam programs are produced semi-automatically from Petri-nets.

—Ada program skeletons are automatically derived from PROT nets, a high-level Petri-net formalism, to assist the programmer with the transformation [Bruno and Marchetto 1986].

—The semi-automatic refinement system for the Proteus language [Mills et al. 1991] is based on algebraic specification techniques and category theory to transform prototypes to implementations on specific architectures. For the time being, these transformations are restricted to the data-parallel constructs of Proteus [Prins and Palmer 1993]. Nyland, Prins, Goldberg, Mills, Reif, and Wagner [1996] discusses the transformation of data-parallel Proteus programs to low-level data-parallel systems and to message-passing libraries.

—Tool support for the transformation of PSDL [Luqi et al. 1988] prototypes is discussed in [Berzins et al. 1993].

It is unlikely that fully automatical transformation tools as they are known for parallelization of imperative sequential languages such as C and Fortran [Bacon et al. 1994] can be built for control-parallel prototypes, but some kind of tool support is conceivable. Before building such transformation tools, it appears to be reasonable to get an assessment of the requirements on such tools through practical experience and to develop a theoretical foundation for such tools. The automatic or semi-automatic transformation of *control-parallel* prototypes into efficient low-level programs is, therefore, still an unsolved problem and subject to further research.

4. CONCLUSIONS

To develop a concurrent application, you should start with executable prototypes to experiment with ideas (neglect the execution performance in the first instance). Powerful tools are needed to make prototyping of concurrent applications feasible.

Table 1 presents an overview of the linguistic approaches and Table 2 presents an overview of the graphical approaches surveyed in the paper. The tables classify the surveyed approaches with respect to the process of prototyping concurrent applications. The structure of the tables corresponds to the taxonomy in Figure 1. The information is extracted from the available literature. Note, that some approaches belong to multiple categories. This is the case for the linguistic approaches PDC and PROTOB which are accompanied by the graphical representations of Process-Build and PROT nets, respectively. As we can see, most of the included approaches have been designed for prototyping, some have methods for transformations, and few are accompanied with tools to help with the transformation into efficient implementations.

Figure 4 illustrates the historical development of some of the surveyed linguistic approaches. The graphical approaches are not included in this (simplified) illustration, because there would be no connections to the other approaches and the dependencies among the graphical approaches already become apparent through the structure of Table 2.

This paper surveys several approaches to high-level programming and prototyping of concurrent applications and classifies them with respect to the prototyping process to review and structure the state of the art in this area. The approaches intend to solve the problems in quite different ways. The essential prototyping activities are

(1) programming,

(2) evaluation and

(3) transformation of prototypes into efficient implementations.

Linguistic approaches emphasize programming concerns. Animation and simulation of graphical representations concentrate on evaluation concerns. The systematic transformation of prototypes into efficient low-level programs is still an unsolved problem.

Performance evaluation of specific embedded parallel hardware systems with concrete real-time requirements is not covered by this survey. We restrict ourselves to software and refer the interested reader to Jelly and Gray [1992] for the discussion of performance evaluation approaches to prototyping parallel hardware systems. However, several approaches which are surveyed in this paper are used for prototyping concrete *real-time* systems on a software basis: CRSM [Raju and Shaw 1994], Durra [Barbacci and Lichota 1991], MCSE [Calvez et al. 1994], PAISLey [Nixon et al. 1994], PSDL [Luqi et al. 1988], RAPIDE [Luckham et al. 1993], and Transim [Hart and Flavell 1990].

Despite the large number of publications and progress made with prototyping concurrent applications and concurrent programming in general, the techniques for systematically engineering concurrent applications are still not fully developed. In particular the systematic transformation of prototypes into efficient implementations is not well understood as yet and, therefore, a subject for further research. However, the appearance of workshops on the subject is promising [PDSE 1996; PDSE 1997]. Prototyping is one important concern for software engineering of concurrent applications.

REFERENCES

AGHA, G. 1986. *Actors: A model of concurrent computation in distributed systems.* The MIT Press.

AGHA, G. 1990. Concurrent object-oriented programming. *Commun. ACM 33*, 9 (Sept.), 125–141.

AGHA, G. 1996. Linguistic paradigms for programming complex distributed systems. *ACM Computing Surveys 28*, 2 (June), 295–296.

AGHA, G., WEGNER, P., AND YONEZAWA, A. Eds. 1993. *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press.

ANDREWS, G. 1991. *Concurrent Programming.* Benjamin/Cummings.

BACON, D., GRAHAM, S., AND SHARP, O. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4 (Dec.), 345–420.

BAL, H. 1990. *Programming Distributed Systems.* Silicon Press.

BAL, H., STEINER, J., AND TANENBAUM, A. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, 3 (Sept.), 261–322.

BALDASSARI, M., BRUNO, G., AND CASTELLA, A. 1991. PROTOB: An object-oriented CASE tool for modelling and prototyping distributed systems. *Software: Practice and Experience 21*, 8 (Aug.), 823–844.

BARBACCI, M. AND LICHOTA, R. 1991. Durra: An integrated approach to software specification, modeling, and rapid prototyping. In N. KANOPOULOS Ed., *Proc. Second International*

Workshop on Rapid System Prototyping (Research Triangle Park, NC, June 1991), pp. 67–81. IEEE Computer Society Press.

BARRY, B. 1989. Prototyping a real-time embedded system in Smalltalk. *ACM SIGPLAN Notices 24*, 10.

BERZINS, V., LUQI, AND YEHUDAI, A. 1993. Using transformations in specification-based prototyping. *IEEE Trans. Softw. Eng. 19*, 5, 436–452.

BISCHOFBERGER, W. AND POMBERGER, G. 1992. *Prototyping oriented software development concepts and tools*. Springer-Verlag.

BLELLOCH, G. 1996. Programming parallel algorithms. *Commun. ACM 39*, 3 (March), 85–97.

BLELLOCH, G., HARDWICK, J., SIPELSTEIN, J., ZAGHA, M., AND CHATTERJEE, S. 1994. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing 21*, 1 (April), 4–14.

BREANT, F. 1991. Rapid prototyping from Petri-Net on a loosely coupled parallel architecture. In T. S. DURRANI, W. A. SANDHAM, J. J. SORAGHAN, AND J. HULSKAMP Eds., *Proc. International Conference on Applications of Transputers* (Glasgow, UK, 1991), pp. 644–649. IOS Press.

BRUNO, G. AND MARCHETTO, G. 1986. Process-translatable Petri Nets for the rapid prototyping of process control systems. *IEEE Trans. Softw. Eng. 12*, 2 (Feb.), 346–357.

BUCCI, G. AND VICARIO, E. 1992. Rapid prototyping through communicating Petri nets. In N. KANOPOULOS Ed., *Proc. Third International Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June 1992), pp. 58–75. IEEE Computer Society Press.

BUCHS, D., FLUMET, J., AND RACLOZ, P. 1992. Producing prototypes from CO-OPN specifications. In N. KANOPOULOS Ed., *Proc. Third International Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June 1992), pp. 77–93. IEEE Computer Society Press.

BUDDE, R., KAUTZ, K., KUHLENKAMP, K., AND ZÜLLIGHOVEN, H. 1992. *Prototyping — An Approach to Evolutionary System Development*. Springer-Verlag.

BUDDE, R., KUHLENKAMP, K., MATHIASSEN, L., AND ZÜLLIGHOVEN, H. Eds. 1984. *Approaches to Prototyping*. Springer-Verlag.

CALVEZ, J., PASQUIER, O., AND HERAULT, V. 1994. A complete toolset for prototyping and validating multi-transputer real-time applications. In M. BECKER, L. LITZLER, AND M. TRÉHEL Eds., *TRANSPUTERS'94 Advanced Research and Industrial Applications* (Saline Royale d'Arc et Senans, France, Sept. 1994), pp. 71–86. IOS Press.

CAO, J., DE VEL, O., AND WONG, K. 1993. Supporting a rapid prototyping system for distributed algorithms on a transputer network. In J. KERRIDGE Ed., *Transputer and Occam Research: New Directions* (Sheffield, UK, 1993), pp. 115–130. IOS Press.

CARRIERO, N. AND GELERNTER, D. 1992. Coordination languages and their significance. *Commun. ACM 35*, 2 (Feb.), 96–107.

CHANDY, K. AND KESSELMAN, C. 1993. CC++: A declarative concurrent object-oriented programming notation. In G. AGHA, P. WEGNER, AND A. YONEZAWA Eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 281–313. The MIT Press.

CHANDY, K. AND TAYLOR, S. 1992. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers.

CHEN, M., CHOO, Y., AND LI, J. 1991. Crystal: Theory and pragmatics of generating efficient parallel code. In B. SZYMANSKI Ed., *Parallel Functional Languages and Compilers*, Frontier Series, pp. 255–308. ACM Press.

CIANCARINI, P. 1992. Parallel programming with logic languages: a survey. *Computer Languages 17*, 4 (April), 213–240.

CIANCARINI, P. 1996. Coordination models and languages as software integrators. *ACM Computing Surveys 28*, 2 (June), 300–302.

CLOCKSIN, W. AND MELLISH, C. 1987. *Programming in Prolog* (3rd ed.). Springer-Verlag.

DE KERGOMMENAUX, J. AND CODOGNET, P. 1994. Parallel logic programming systems. *ACM Computing Surveys 26*, 3 (Sept.), 295–336.

DeMarco, T. 1978. *Structured Analysis and System Specification*. Yourdon Press.

Deng, Y., Chang, S., de Figueired, J., and Perkusich, A. 1993. Integrating software engineering methods and Petri nets for the specification and prototyping of complex information systems. In M. Marsan Ed., *Proc. 14th International Conference on Application and Theory of Petri Nets*, Volume 694 of *Lecture Notes in Computer Science* (Chicago, IL, June 1993), pp. 206–223. Springer-Verlag.

Doberkat, E.-E., Franke, W., Gutenbeil, U., Hasselbring, W., Lammers, U., and Pahl, C. 1992. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos Ed., *Proc. Third International Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June 1992), pp. 235–248. IEEE Computer Society Press.

Donatelli, S., Franceschinis, G., Ribaudo, M., and Russo, S. 1994. Use of GSPNs for concurrent software validation in EPOCA. *Information and Software Technology 36*, 7 (July), 443–448.

Douglas, A., Rowstron, A., and Wood, A. 1995. ISETL-Linda: parallel programming with bags. Technical Report YCS 257 (April), University of York, Dept. Computer Science, Heslington, York, UK.

Dubinsky, E. and Leron, U. 1993. *Learning abstract algebra with ISETL*. Springer-Verlag.

Floyd, C. 1984. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven Eds., *Approaches to Prototyping*, pp. 1–18. Springer-Verlag.

Foster, I. and Taylor, S. 1989. *Strand: New Concepts in Parallel Programming*. Prentice-Hall.

French, J. and Viles, C. 1992. A software toolkit for prototyping distributed applications. Technical Report CS-92-26 (Sept.), University of Virginia, Dept. Computer Science.

Fugetta, A., Ghezzi, C., Mandrioli, D., and Morzenti, A. 1993. Executable specifications with data-flow diagrams. *Software: Practice and Experience 23*, 6, 629–653.

Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. on Programm. Lang. Syst. 7*, 1 (Jan.), 80–112.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. 1991. *Fundamentals of Software Engineering*. Prentice-Hall.

Gorton, I., Gray, J., and Jelly, I. 1995. Object based modelling of parallel programs. *IEEE Parallel and Distributed Technology 3*, 2 (Summer), 52–63.

Halstead, R. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programm. Lang. Syst. 7*, 4 (Oct.), 501–538.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3, 231–274.

Hart, E. and Flavell, S. 1990. Prototyping transputer applications. In H. Zedan Ed., *Real-Time Systems with Transputers* (Amsterdam, 1990), pp. 241–247. IOS Press.

Hasselbring, W. 1994. *Prototyping Parallel Algorithms in a Set-Oriented Language*. Ph. D. thesis, Department of Computer Science, University of Dortmund. (Published by Verlag Dr. Kovač, Hamburg).

Hasselbring, W. 1997. The ProSet-Linda approach to prototyping parallel systems. *The Journal of Systems and Software*. (to appear).

Hasselbring, W. and Kröber, A. 1997. Combining OMT with a prototyping approach. *The Journal of Systems and Software*. (to appear).

Heping, H. and Zedan, H. 1996. An executable specification language for fast prototyping parallel responsive systems. *Computer Languages 22*, 1, 1–13.

Hillis, W. and Steele, G. 1986. Data parallel algorithms. *Commun. ACM 29*, 12 (Dec.), 1170–1183.

Houck, C. and Agha, G. 1992. HAL: a high-level actor language and its distributed implementation. In *Proc. 21st International Conference on Parallel Processing (ICPP 91)*, Volume II (St. Charles, IL, Aug. 1992), pp. 158–165.

Hu, L. and Gorton, I. 1997. A performance prototyping approach to designing concurrent software architectures. In *Proc. Second International Workshop on Software Engineering* *for Parallel and Distributed Systems (PDSE'97)* (Boston, Massachusetts, May 1997).

HUMMEL, S. F. AND KELLY, R. 1993. A rationale for parallel programming with sets. *Journal of Programming Languages 1*, 3, 187–207.

HUMMEL, S. F., TALLA, S., AND BRENNAN, J. 1995. The refinement of high-level parallel algorithm specifications. In *Proc. Working Conference on Programming Models for Massively Parallel Computers (PMMP '95)* (Berlin, Germany, Oct. 1995). IEEE Computer Society Press.

HUNTBACH, M. AND RINGWOOD, G. 1995. Programming in concurrent logic languages. *IEEE Software 12*, 6 (Nov.), 71–82.

JARD, C., MONIN, J.-F., AND GROZ, R. 1988. Development of Véda, a prototyping tool for distributed algorithms. *IEEE Transactions on Software Engineering 14*, 3 (March), 339–352.

JELLY, I. AND GRAY, J. 1992. Prototyping parallel systems: a performance evaluation approach. In *Proc. International Conference on Parallel and Distributed Computing and Systems* (Pittsburgh, PA, Oct. 1992), pp. 7–12. ISSM Press.

JODELEIT, P. 1996. Implementing the Salishan and Cowichan problems based on prototype evaluation and transformation. Master's thesis, Department of Computer Science, University of Dortmund. (in German).

JONES, D., DOWDESWELL, S., AND HINTZ, T. 1990. A rapid prototyping method for parallel programs. In T. BOSSOMAIER, T. HINTZ, AND J. HULSKAMP Eds., *The Transputer in Australia (ATOUG-3)* (Sydney, Australia, June 1990), pp. 121–128. IOS Press.

JOZWIAK, J. 1993. Exploiting parallelism in SETL programs. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

KIRSCH, M. 1996. Implementing parallel model matching algorithms for 3-D computer vision with Linda and message passing based on prototype evaluation and transformation. Master's thesis, Department of Computer Science, University of Dortmund.

KRÄMER, B. 1991. Prototyping and formal analysis of concurrent and distributed systems. In *Proc. Sixth International Workshop on Software Specification and Design* (1991), pp. 60–66. IEEE Computer Society Press.

KRUCHTEN, P., SCHONBERG, E., AND SCHWARTZ, J. 1984. Software prototyping using the SETL programming language. *IEEE Software 1*, 4 (Oct.), 66–75.

LEVY, A., VAN KATWIJK, J., PAVLIDES, G., AND TOLSMA, F. 1990. SEPDS: A support environment for prototyping distributed systems. In P. NG, C. RAMAMOORTHY, L. SEIFERT, AND R. YEH Eds., *Proc. First International Conference on Systems Integration* (Morristown, NJ, April 1990), pp. 652–661. IEEE Computer Society Press.

LOBE, G., SZAFRON, D., AND SCHAEFFER, J. 1993. Program design and animation in the Enterprise parallel programming environment. Technical Report TR 93-04 (March), University of Alberta, Dept. Computing Science.

LUCKHAM, D., VERA, J., BRYAN, D., AUGUSTIN, L., AND BELZ, F. 1993. Partial orderings of event sets and their application to prototyping concurrent timed systems. *The Journal of Systems and Software 21*, 3 (June), 253–265.

LUQI, BERZINS, V., AND YEH, R. T. 1988. A prototyping language for real-time software. *IEEE Trans. Softw. Eng. 14*, 10 (Oct.), 1409–1423.

MATSUOKA, S. AND YONEZAWA, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent languages. In G. AGHA, P. WEGNER, AND A. YONEZAWA Eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 107–150. The MIT Press.

MEYER, B. 1993. Systematic concurrent object-oriented programming. *Commun. ACM 36*, 9 (Sept.), 56–80.

MEYER, B. 1997. *Object-oriented Software Construction* (second ed.). Prentice Hall.

MICHAELSON, G. AND SCAIFE, N. 1995. Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming 5*, 3 (July), 345–382.

MILLS, P., NYLAND, L., PRINS, J., REIF, J., AND WAGNER, R. 1991. Prototyping parallel and distributed programs in Proteus. In *Proc. Third IEEE Symposium on Parallel and Distributed Processing* (Dallas, TX, Dec. 1991), pp. 26–34.

Mitchell, S. and Wellings, A. 1996. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. *Computer Languages 22*, 1, 15–26.

Nixon, P., Birkinshaw, C., Croll, P., and Marriot, D. 1994. Rapid prototyping of parallel fault tolerant systems. In *Proc. Euromciro Workshop on Parallel and Distributed Systems* (Malaga, Spain, 1994), pp. 202–210. IEEE Computer Scociety Press.

Nixon, P. and Croll, P. 1993. The functional specification of occam programs for time critical applications. In J. Kerridge Ed., *Transputer and Occam Research: New Directions* (Sheffield, UK, 1993), pp. 131–145. IOS Press.

Nyland, L., Prins, J., Goldberg, A., Mills, P., Reif, J., and Wagner, R. 1996. A refinement methodology for developing data-parallel applications. In *EuroPar'96 Parallel Processing*, Volume 1123 of *Lecture Notes in Computer Science* (Lyon, France, Aug. 1996), pp. 145–150. Springer-Verlag.

PDSE. 1996. *First International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'96)* (Berlin, Germany, March 1996).

PDSE. 1997. *Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)* (Boston, Massachusetts, May 1997).

Prins, J. and Palmer, D. 1993. Transforming high-level data-parallel programs into vector operations. In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 119–128.

Purtilo, J. and Jalote, P. 1991. An environment for prototyping distributed applications. *Computer Languages 16*, 3/4, 197–207.

Purtilo, J., Reed, D., and Grunwald, D. 1988. Environments for prototyping parallel algorithms. *Journal of Parallel and Distributed Computing 5*, 4 (Aug.), 421–437.

Quinn, M. and Hatcher, P. 1990. Data-parallel programming on multicomputers. *IEEE Software 7*, 5 (Sept.), 69–76.

Raju, S. and Shaw, A. 1994. A prototyping environment for specifying, executing and checking communicating real-time state machines. *Software: Practice and Experience 24*, 2 (Feb.), 175–195.

Reisig, W. 1985. *Petri Nets*. Springer-Verlag.

Schaeffer, J., Szafron, D., Lobe, G., and Parsons, I. 1993. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology 1*, 3 (Aug.), 85–96.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Computing Surveys 21*, 3 (Sept.), 412–510.

Son, S. H. and Kim, Y. 1989. A software prototyping environment and its use in developing a multiversion distributed database system. In E. Plachy and P. Kogge Eds., *Proc. 1989 International Conference on Parallel Processing*, Volume II (University Park, PA, Aug. 1989), pp. 81–88. Pennsylvania State University Press.

Szafron, D. and Schaeffer, J. 1996. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience 8*, 2, 147–166.

Szymanski, B. Ed. 1991. *Parallel functional languages and compilers*. Addison-Wesley.

Unger, B. 1988. Distributed software via prototyping and simulation – JADE. Technical Report 88/300/12 (March), University of Calgary, Dept. Computer Science.

Valenzano, A., Sisto, R., and Ciminiera, L. 1993. Rapid prototyping of protocols from LOTOS specifications. *Software: Practice and Experience 23*, 1 (Jan.), 31–54.

Wallace, A., Michaelson, G., McAndrew, P., Waugh, K., and Austin, W. 1992. Dynamic control and prototyping of parallel algorithms for intermediate- and high-level vision. *IEEE Computer 25*, 2 (Feb.), 43–53.

Weinreich, R. and Ploesch, R. 1995. Prototyping of parallel and distributed object oriented systems: The PDC model and its environment. In *Proc. 28th Hawaii International Conference on System Sciences (HICSS-28), Software Track* (Maui, Hawaii, Jan. 1995). IEEE Computer Scociety Press.

Wyatt, B., Kavi, K., and Hufnagel, S. 1992. Parallelism in object-oriented languages: A survey. *IEEE Software 9*, 6 (Nov.), 56–66.

YONEZAWA, A. 1990. *ABCL: An Object-Oriented Concurrent System.* MIT Press.

ZAVE, P. AND SCHELL, W. 1986. Salient features of an executable specification language and its environment. *IEEE Trans. Softw. Eng. 12*, 2 (Feb.), 312–325.

ZHOU, W. 1994. A rapid prototyping system for distributed information system applications. *Journal of Systems and Software 24*, 1 (Jan.), 3–29.

Table 1. Classification of the surveyed linguistic approaches with respect to prototyping concurrent applications

Approach	Designed for Prototyping	Methods for Transformations	Tools for Transformations
High-level Libraries			
IPC_FBase [Cao et al. 1993]	✓	–	–
Polylith [Purtilo et al. 1988; Purtilo and Jalote 1991]	✓	–	–
StarLite [Son and Kim 1989]	✓	–	–
VAN [French and Viles 1992]	✓	–	–
Zhou [1994]	✓	–	–
Set-Oriented Data Parallelism			
NESL [Blelloch et al. 1994; Blelloch 1996]	–	✓	✓
Parallel ISETL [Jozwiak 1993]	✓	–	–
Proteus [Mills et al. 1991; Prins and Palmer 1993; Nyland et al. 1996]	✓	✓	✓
PSETL [Hummel and Kelly 1993; Hummel et al. 1995]	✓	✓	–
Concurrent Functional Languages			
Crystal [Chen et al. 1991]	✓	✓	–
PAISLey [Zave and Schell 1986; Nixon and Croll 1993; Nixon et al. 1994]	✓	✓	–
PSP [Heping and Zedan 1996]	✓	–	–
Standard ML [Wallace et al. 1992; Michaelson and Scaife 1995]	–	✓	–
Concurrent Logic Languages			
RGDC [Huntbach and Ringwood 1995]	–	–	–
Strand [Foster and Taylor 1989]	✓	–	–
Véda [Jard et al. 1988]	✓	–	–
Concurrent Object-Based Languages			
ABCL [Yonezawa 1990]	–	–	–
HAL [Houck and Agha 1992]	–	–	–
PDC [Weinreich and Ploesch 1995]	✓	–	–
PROTOB [Baldassari et al. 1991]	✓	–	–
RAPIDE [Luckham et al. 1993]	✓	–	–
Composition and Coordination Languages			
CC++ [Chandy and Kesselman 1993]	✓	–	–
Durra [Barbacci and Lichota 1991]	✓	–	–
ISETL-Linda [Douglas et al. 1995]	✓	–	–
PCN [Chandy and Taylor 1992]	✓	–	–
ProSet-Linda [Hasselbring 1994; Hasselbring 1997; Hasselbring and Kröber 1997]	✓	✓	–

Table 2. Classification of the surveyed graphical approaches with respect to prototyping concurrent applications

Approach	Designed for Prototyping	Methods for Transformations	Tools for Transformations
Based on Petri-Nets			
Breant [1991]	✓	✓	✓
CmPN [Bucci and Vicario 1992]	✓	–	–
CO-OPN [Buchs et al. 1992]	✓	–	–
G-Net [Deng et al. 1993]	✓	–	–
GSPN [Donatelli et al. 1994]	✓	–	–
PARSE [Gorton et al. 1995; Hu and Gorton 1997]	✓	–	–
PROT nets [Bruno and Marchetto 1986]	✓	✓	✓
SEGRAS [Krämer 1991]	✓	–	–
Based on State-Transition Diagrams			
CRSM [Raju and Shaw 1994]	✓	–	–
EFSM [Valenzano et al. 1993]	✓	–	–
Based on Data-Flow Diagrams			
EDFG [Levy et al. 1990]	✓	–	–
FDFD [Fugetta et al. 1993]	✓	–	–
IDE [Jones et al. 1990]	✓	–	–
PSDL [Luqi et al. 1988; Berzins et al. 1993]	✓	✓	✓
Other Notations			
Enterprise [Schaeffer et al. 1993; Lobe et al. 1993; Szafron and Schaeffer 1996]	–	–	–
JADE [Unger 1988]	✓	–	–
MCSE [Calvez et al. 1994]	✓	–	–
ProcessBuild [Weinreich and Ploesch 1995]	✓	–	–
Transim [Hart and Flavell 1990]	✓	–	–

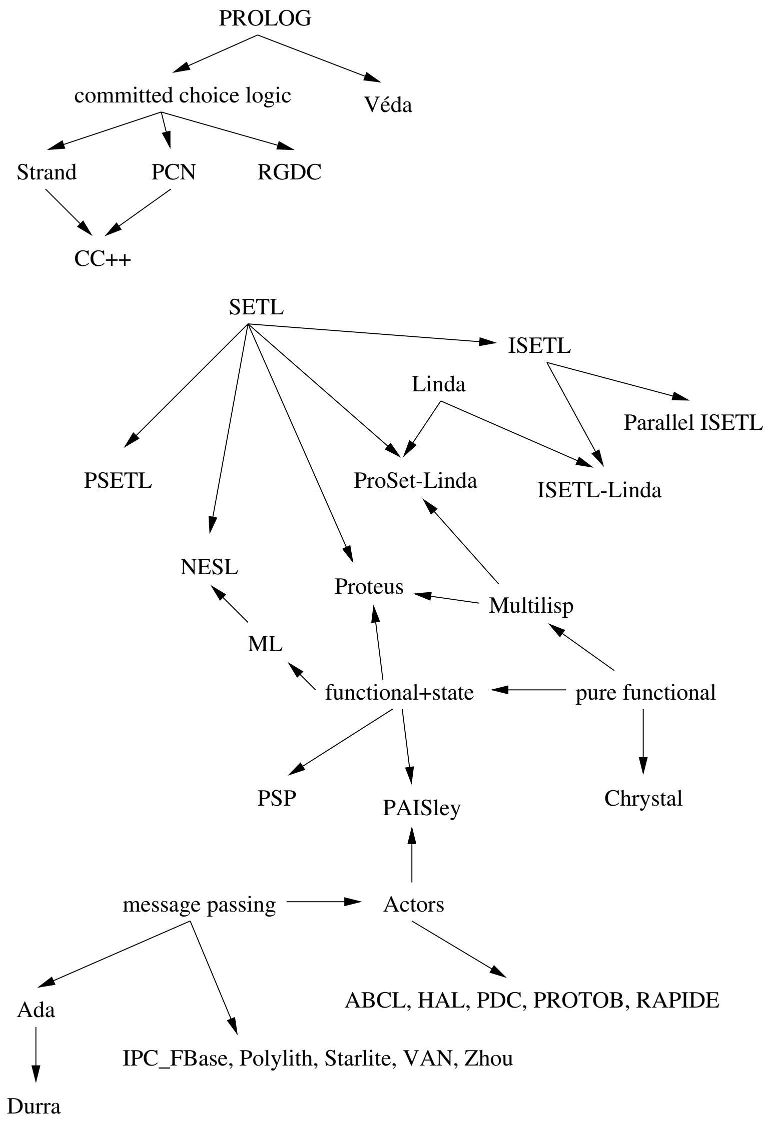

Fig. 4. An illustration of the historical development of some of the approaches surveyed in the paper. The arrows indicate dependencies.