# Programming Languages and Systems for Prototyping Concurrent Applications

WILHELM HASSELBRING

*Tilburg University*

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 in 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 programming languages and systems for prototyping 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.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering* (CASE); *Petri nets*; *Software libraries*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; 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, distribution, parallelism, rapid prototyping, very high-level languages

## 1. INTRODUCTION

During the last decades, particular attention has been focused on concurrent programming within the computer science community. A *concurrent* program specifies two or more processes that cooperate in performing a task [Andrews 1991]. Each process is a sequential pro-

Author's address: INFOLAB, Tilburg University, PO Box 90153, Tilburg, 5000 LE, The Netherlands; email: hasselbring@acm.org.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 0360-0300/00/0300–0043 $5.00

CONTENTS

1. Introduction
2. Software Prototyping And Concurrent Programming
   2.1 Software Prototyping
   2.2 Prototyping Concurrent Applications
3. Languages And Systems
   3.1 Domain-Specific Libraries
   3.2 Set-Oriented Data Parallelism
   3.3 Coordination Languages
   3.4 Concurrent Functional Languages
   3.5 Concurrent Object-Based Languages
   3.6 Graphical Programming Systems
4. Transforming Prototypes Into Efficient Implementations
5. Conclusions And Directions For Future Work
   5.1 Conclusions
   5.2 Directions for Future Work

gram that executes a sequence of statements. Processes cooperate by communication and synchronization. In a *parallel* program, these concurrent processes are executed in parallel on multiple processors. A *distributed* program is a concurrent program in which processes on different computers communicate through a network. 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 fault-tolerance;

(3) explicitly exploiting 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 languages that are designed for prototyping 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 concurrent processes.

Section 2 first discusses some general issues of prototyping sequential and concurrent systems before a survey of several programming languages and systems for prototyping concurrent applications is presented in Section 3. For each approach, first the general idea is discussed. Then, a short presentation of an example system follows before some sample systems are listed for further reference. 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 as it can be found in Bal et al. [1989] for distributed programming languages. Only approaches which are designed for prototyping are included. The transformation of prototypes into efficient implementations is discussed in Section 4. Section 5 draws some conclusions and indicates directions for future work.

The focus of this paper is on rapid prototyping of concurrent programs (validation), not on the correct derivation of concurrent programs from exe-

cutable specifications (verification). Also, the focus is not on the composition and coordination of distributed software architecture. However, the discussed approaches also cover coordination and object-oriented languages, and Section 4 discusses the transformation of high-level prototypes into lower-level implementations, but these issues are not the central concern of this paper.

## 2. SOFTWARE PROTOTYPING AND CONCURRENT PROGRAMMING

Prototyping is used in the early phases of software development for requirements analysis, risk reduction, specification validation, and increased user acceptance of software systems. Considerable experience with developing *sequential* software systems using prototyping has been made [Gordon and Bieman 1995]. In particular, when combining prototyping with an evolutionary software development process, not only the quality of the product, but also the development process itself can be improved [Lichter et al. 1994]. Software prototyping is not only concerned with the rapid development of user interfaces, but also with the functionality of planned systems (e.g., with developing algorithms).

### 2.1 Software Prototyping

Prototyping refers to the phase in the process of developing software in which a model is constructed that has the *essential* properties of the final product, and which is taken into account when properties have to be checked, and when the further steps in the development process have to be determined. We define software prototypes as follows (based on Bischofberger and Pomberger [1992]; Connell and Shafer [1994]; and Pomberger and Blaschek [1996]).

*Definition:* A software prototype is an executable model of a proposed system. It must be producible with significantly less effort than the planned product and it must be readily modifiable and extensible. The prototype need not have all the features of the target system, yet it must enable testing of important system features before the actual implementation.

*Prototyping* encompasses the activities necessary to make such prototypes available. The essential prototyping activities are

(1) programming,

(2) evaluation, and

(3) transformation of prototypes into efficient implementations.

Software prototypes are used somewhat differently from hardware prototypes. For the most part, hardware prototypes are used to measure and evaluate aspects of proposed designs that are difficult to determine analytically. For example, simulation is widely used to estimate throughput and device utilization in proposed hardware architectures. Although software prototypes can be used likewise to determine time and memory requirements, they usually focus on evaluating the accuracy of problem formulations, exploring the range of possible solutions, and determine the required interactions between the proposed system and its environment. The nature of a software prototype is also different from, for example, an architectural model: a software prototype actually demonstrates features of the target system in practical use, and is not merely a simulation of them. We will use the term *prototyping* as a synonym for software prototyping in the remainder of this paper.

The idea of prototyping is being adopted in software engineering for different purposes: prototypes are used *exploratively* to arrive at a feasible specification, *experimentally* to check different approaches, and *evolutionarily* to build a system incrementally [Floyd 1984]. The order of development steps in the traditional life cycle model is mapped here into successive development cycles. Note that a prototype is a

model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Prototyping has been developed as an answer to some of the deficiencies in the traditional life cycle model, i.e., the waterfall model, where each phase is completed before the next phase is started [Ghezzi et al. 1991]; but should not be considered as an alternative to this model. It is rather optimally useful when it complements this traditional model. It is plausible that prototyping may be used during the early phases of software development.

The term *Rapid Application Development (RAD)* [Martin 1991; Card 1995] was coined to cover the involvement of the end-users in the development process. Prototyping is used in this context to help the end-users visualize adjustments to the system. Scripting languages [Ousterhout 1998] such as Perl, TCL, and Python are often used for rapid application development. These languages lack many of the features necessary to build big, modular, efficient, product-quality software systems. Their syntax and semantics are tuned to be efficient for small systems. They provide general purpose data structures like strings, lists and dictionaries and come with libraries for specific purposes such as the construction of graphical user interfaces. They are often embedded into systems to make it possible to glue components together in a flexible way to allow prototyping.

### 2.2 Prototyping Concurrent Applications

Traditionally, the emphasis of software prototyping is on the rapid construction of user interfaces [Isensee et al. 1995]. Conversely, prototyping concurrent applications emphasizes on the functionality of planned systems (particularly, developing concurrent algorithms).

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 with powerful languages and systems 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 primary goal of parallel programming mentioned above —that of 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 or distributed architectures to achieve high speedups.

The criteria for including programming languages and systems into this survey is the support for constructing *executable* models of a proposed *concurrent* system with significantly less effort than the planned product (see also the definition of software prototypes in Section 2.1). Included in this survey are high-level linguistic approaches for prototyping concurrent applications and graphical representations of concurrency which allow prototyping through animation, simulation and code generation. These graphical programming systems are not designed for prototyping user interfaces; instead they visualize the computation within concurrent systems. An essential feature of prototypes is executability.

A specific requirement for prototyping

*concurrent* systems is the integration of different techniques such as the following:

—evaluation of performance, e.g., through performance visualization;

—simulation of real-time properties;

—evaluation of fault-tolerance issues; and

—transformation of prototypes into efficient implementations.

The surveyed approaches are classified with respect to these issues. The integration of these features with more traditional techniques such as prototyping of user interfaces and interactive debugging could form a powerful means for developing concurrent applications.

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.

## 3. LANGUAGES AND SYSTEMS

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 et al. [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: "In fact, even though PVM and the MPI [Dongarra et al. 1996] are *de facto* standards in parallel programming, their related programming style looks in many respects like assembler-level programming of sequential computers."[Talia 1997]. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory [Bal 1990]. This greatly increases the complexity of programs, and also restricts algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. Therefore, several alternative models have been designed for parallel programming, which provide higher-level abstractions. These languages emphasize some kind of shared data.

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. 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:

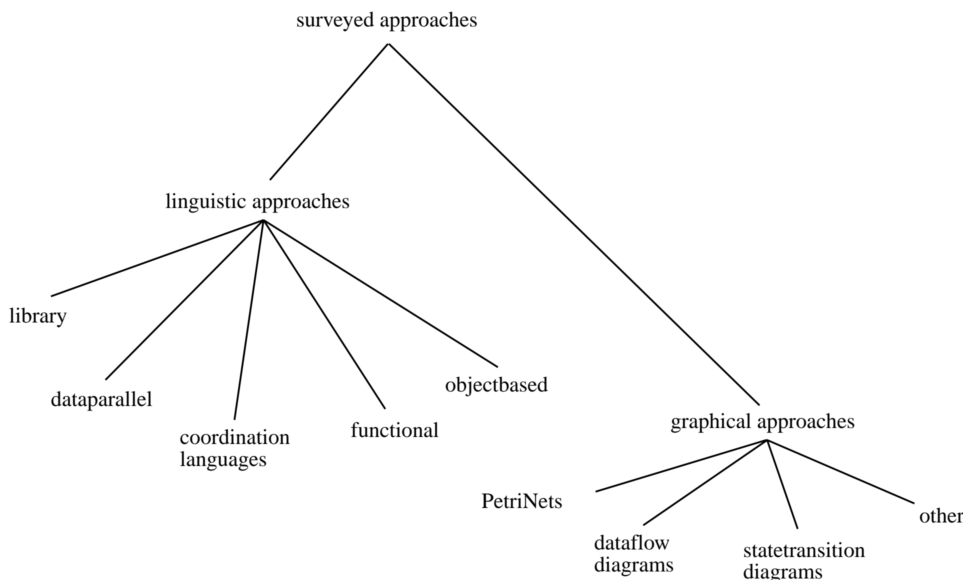


**Figure 1**. A taxonomy for the approaches surveyed.

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 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 regions 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 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 domain-specific libraries (Section 3.1); continues with set-oriented data parallelism (Section 3.2); coordination languages (Section 3.3); concurrent functional languages (Section 3.4); concurrent object-based languages (Section 3.5); and graphical programming systems (Section 3.6). Figure 1 classifies the approaches surveyed in the paper into a simple taxonomy.

As logic programming languages (of which PROLOG [Clocksin and Mellish 1987] is best known) have been used for prototyping sequential systems [Budde et al. 1984], concurrent logic languages

[Ciancarini 1992; de Kergommenaux and Codognet 1994; Shapiro 1989] seem to be good candidates for prototyping concurrent applications. For instance, 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. However, we know only one published approach on using PROLOG for simulation and prototyping of Estelle protocol specifications in the Véda system [Jard et al. 1988]; thus, concurrent logic languages are not included in this survey.

A fundamentally different approach to parallel *processing* does not propose new software models, but relies on the automatic generation of parallel versions from ordinary sequential programs [Banerjee et al. 1993]. A parallelizing compiler finds the parallelism hidden in a sequential program by analyzing its structure for operations that in fact can be done simultaneously, even though the program specifies they be done one by one. It then generates code that reflects the implicit concurrency it has found. Automatic parallelization can discover the concurrency latent in an existing algorithm on the basis of the program's expected behavior, but it cannot invent new concurrent algorithms. Programmers who can express and prototype their ideas in a concurrent way sometimes invent entirely new ways of solving problems. In order to embody their inventions in working programs, they need languages that allow concurrency to be expressed *explicitly*. Therefore, approaches to automatic parallelization are not included in this survey of prototyping approaches.

### 3.1 Domain-Specific 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 higher-level, domain-specific libraries for prototyping concurrent applications. These domain-specific libraries intend to provide simple interfaces and more flexibility than the lower-level libraries do, while relinquishing efficiency to some extent.

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***. URPC is a toolkit for prototyping remote procedure call (RPC) systems [Huang and Ravishankar 1996]. URPC is an acronym for User-level RPC. It allows programmers to provide high-level implementations of special-purpose RPC semantics and to customize supporting RPC services. The toolkit consists of a runtime library of RPC functions accompanied by a stub generator for RPC interface specifications and a name server. The name server maintains the mappings between logical names and physical addresses of processes. The library provides point-to-point RPC communication among these processes. As an example, we present the URPC implementation of a multicast RPC (one-to-many communication) from Huang and Ravishankar [1996]. First, the RPC interface specification contains some global definitions:

```
[
 aptitle = mdisp;
 server_pm = mdisp_sv_pm;
 client_pm = mdisp_cl_pm;
 transport = INET_TCP
]
```

These global definitions define the application title (`aptitle`), the names for client and server functions, and the transport-layer protocol used. The following type definitions specify types that are used for specifying the RPC interfaces:

```
urpc_msg_t *mdisp_cl_pm (int *handleP, urpc_msg_t *msg)
{
  int err = RPC_OK;
  while ((*handleP != INVALID) && (err == RPC_OK))
  {
     urpc_set_send_msg (*handleP,msg);
     err = urpc_send (*handleP);
     handleP ++; /* next server address */
  }
  if (err != RPC_OK)
     return (NULL);   /* error during multicast */
  else
     return (msg);
}
```

**Figure 2**. The protocol machine for the multicast client in URPC.

```
typedef int error_st;
typedef char string_t[1024];
```

Then, the RPC interface for multicast RPC is defined as follows:

```
[cl] error_st mdisp
([in] int *handleP,
[in] string_t msg)
```

The handles representing the addresses of servers are organized into the array pointer `*handleP`, which will be looked up in the client code when a multicast RPC is performed. The string `msg` contains the message to be sent by the client to the servers.

The stub generator of the toolkit generates stubs (headers for C functions) for ‘protocol machines’ from the RPC interface specification. These protocol machines are C functions. The protocol machine for the multicast client is displayed in Figure 2. The used library function `urpc_set_send_msg` puts a message into the send queue and the library function `urpc_send` sends a message to the given address. The header for the function `mdisp_ cl_ pm` is generated from the above specifications by the stub generator.

The use of these library functions together with the stub generation from RPC interface specifications and with the name server allows to experiment with new RPC systems meeting different application requirements. Note that the stub generator cannot check the proper use of the libraries. It provides appropriate headers for user-defined C functions on the client and server sides. The URPC library functions are then called by these user-defined C functions.

Some of the features offered by the URPC toolkit are also provided by typical CORBA implementations (stub generation from interface specifications, name services) [Mowbray and Zahavi 1995]. However, URPC is a domain-specific toolkit designed for prototyping new RPC systems, whereas CORBA is intended as a general-purpose platform for distributed applications.

***Some sample systems***. In addition to URPC, there exist some other approaches to prototyping with domain-specific libraries:

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

—Zhou [1994] proposes prototyping of distributed information systems based on some remote procedure call (RPC) functionality.

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

—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.

—Polylith [Purtilo et al. 1988; Purtilo and Jalote 1991] is a module library for prototyping distributed 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 and different source languages can be used for coding different modules. Polylith provides a software bus very similar to object request brokers in CORBA systems [Mowbray and Zahavi 1995].

The block on domain-specific libraries in Table I classifies these approaches with respect to the process of prototyping concurrent applications. Only a few issues are addressed in this category, particularly transformations are not supported.

### 3.2 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 of data-parallel algorithms, refer to Hillis and Steele [1986]; for 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 exe-

| | Methods for transformations | Tools for transformations | Performance prototyping | Real-time considered | Fault-tolerance considered | User experience reported |
|---|---|---|---|---|---|---|
| *Domain-Specific Libraries* | | | | | | |
| IPC_FBase [Cao et al. 1993] | – | – | ✔ | ✔ | – | – |
| Polylith [Purtilo et al. 1988; Purtilo and Jalote 1991] | – | – | – | ✔ | – | ✔ |
| StarLite [Son and Kim 1989] | – | – | ✔ | – | – | – |
| URPC [Huang and Ravishankar 1996] | – | – | – | – | – | – |
| VAN [French and Viles 1992] | – | – | – | – | – | – |
| Zhou [1994] | – | – | – | – | – | – |
| *Set-Oriented Data Parallelism* | | | | | | |
| DARP [Akarsu et al. 1998] | – | – | ✔ | – | – | – |
| NESL [Blelloch et al. 1994; Blelloch 1996] | ✔ | ✔ | – | – | – | ✔ |
| Parallel ISETL [Jozwiak 1993] | – | – | – | – | – | – |
| Proteus [Mills et al. 1991; Nyland et al. 1996] | ✔ | ✔ | ✔ | – | – | – |
| PSETL [Hummel and Kelly 1993; Hummel et al. 1995] | ✔ | – | – | – | – | – |
| *Coordination Languages* | | | | | | |
| Durra [Barbacci and Lichota 1991] | – | – | – | ✔ | – | – |
| ISETL-Linda [Douglas et al. 1995] | – | – | – | – | – | – |
| ProSet-Linda [Hasselbring 1998] | ✔ | – | – | – | – | – |
| *Concurrent Functional Languages* | | | | | | |
| Crystal [Chen et al. 1991] | ✔ | – | – | – | – | – |
| PAISLey [Nixon et al. 1994] | ✔ | – | – | ✔ | ✔ | – |
| PSP [Heping and Zedan 1996a; Heping and Zedan 1996b] | – | – | – | – | – | – |
| SML [Wallace et al. 1992; Michaelson and Scaife 1995] | ✔ | – | – | – | – | – |
| *Concurrent Object-Based Languages* | | | | | | |
| ActorSpace [Callsen and Agha 1994] | – | – | – | – | – | – |
| CC++ [Chandy and Kesselman 1993] | ✔ | – | – | – | – | – |
| CODB [Stytz et al. 1997] | – | – | – | – | – | ✔ |
| PDC [Weinreich and Ploesch 1995] | – | – | – | ✔ | – | – |
| PROTOB [Baldassari et al. 1991] | – | – | – | ✔ | – | – |
| RAPIDE [Luckham et al. 1993] | – | – | – | ✔ | – | – |

**Table I.** Classification of the Surveyed Linguistic Approaches with Respect to Various Issues for Prototyping Concurrent Applications

cuted in sequential iterations (specified through **par_over**):

```
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.

Data parallelism is usually applied in scientific, numerical applications, and is targeted to parallel vector processors [Quinn and Hatcher 1990]. As a typical

```
for row par_over {1,m+1 .. n} do                      -- parallel iteration
   for col par_over  {1,m+1 .. n} do                  -- parallel iteration
      for i over {row .. row+m-1} do                  -- sequential iteration
         for j over {col .. col+m-1} do               -- sequential iteration
            C(i,j) := +/ [A(i,k)*B(k,j) : k over {1..n}];
         end for;
      end for;
   end for;
end for;
```

**Figure 3**. Parallel matrix multiplication in PSETL. The unary operator +/ sums up the elements in the tuple.

example, Figure 3 presents a parallel PSETL version of matrix multiplication which is targeted to parallel processors that have local cache memories, and that share a memory where the matrices A, B and C are stored. The granulatity of parallel work is explicitly specified through the use of parallel iteration (par_over) and sequential iteration (over).

***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 3.4) 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].

—The DARP system [Akarsu et al. 1998] integrates an interpreter, source level debugger, data visualization, and data analysis packages for prototyping of High-Performance Fortran (HPF) programs. HPF supports the data-parallel programming paradigm, but not set-oriented data parallelism. However, it is included here, because an interesting feature of this system is that the user can interrupt execution of the compiled code at any point and get an interactive access to the data. For visualizations and debugging, the execution is resumed as soon as the data transfer is completed.

The block on set-oriented data parallelism in Table I classifies these approaches with respect to the process of prototyping concurrent applications. Particularly, transformations are supported in this category. Therefore, one of these approaches will be presented as an example in Section 4, where the transformation of high-level prototypes into lower-level implementations is discussed.

### 3.3 Coordination Languages

***Idea***. With 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 interprocess communication, which may be combined with sequential *computation languages* [Carriero and Gelernter 1992]. With 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 interprocess actions), and one for (sequential) computation [Carriero and Gelernter 1992; Ciancarini 1996]. ProSet-Linda [Hasselbring 1998] 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] (see also Section 3.2). In ProSet-Linda, the concept for process creation via Multilisp's futures [Halstead 1985] 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. Three tuple-space operations are provided. 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 containing two elements and with the string `"name"` in the first 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. With `meet`, in-place updates of specific tuple components are supported. 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.

As an example, we present a parallel master-worker solution to branch-and-bound search: the traveling salesman problem in which it is desired to find the shortest route that visits each of a given set of cities exactly once. Branch-and-bound search uses a tree to structure the search space of possible solutions. The root of the tree is the city in which the salesman should start. Each path from the root to a node represents

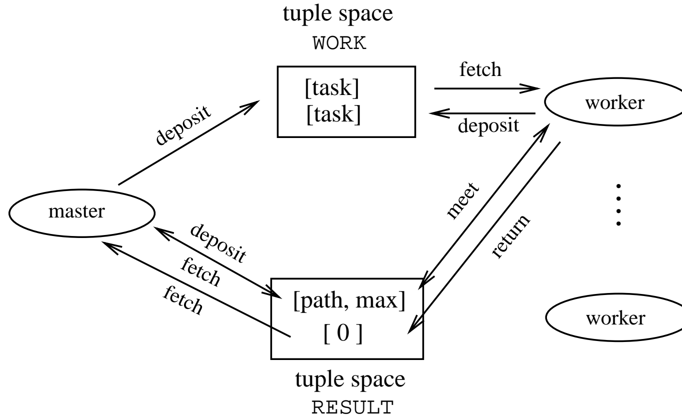


**Figure 4**. The coarse structure of the master-worker program for the traveling salesman problem.

a partial tour for the salesman. Leaf nodes represent either partial tours without connections to not yet visited cities or complete tours. Complete tours visit each city exactly once. In general, it is not necessary to search the entire tree: a *bounding rule* avoids searching the entire tree. For the traveling salesman problem, the bounding rule is simple. If the length of a partial tour exceeds the length of an already known complete tour, the partial tour will never lead to a solution better than what is already known. Parallelism in a branch-and-bound algorithm can be obtained by searching the tree in parallel.

We present below a master-worker program for this traveling salesman problem. In a master-worker program, the task to be solved is partitioned into independent subtasks. These subtasks are placed into a tuple space, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the tuple space `WORK`, solves it, and puts the solutions into tuple space `RESULT`. The master process then collects the results. An advantage of this programming approach is easy load balancing because the number of workers is variable and may be set to the number of available processors. Figure 4 displays the coarse structure of this master-worker program. Arrows indicate access to the tuple spaces. These access patterns are only shown for one of the identical worker processes.

The main program (the master process) in Figure 5 stores the cities with their connections in the global constant set `DistTable`. This set is a map that maps pairs of cities to their distance. The distances are specified for each direction. The distances between two cities may be different for different directions (e.g., for one-way connections). The set `Nodes` contains the cities involved. The string `Start` indicates the starting point.

The master (the main program in Figure 5) first deposits the *current* minimal distance together with the corresponding route into the tuple space `RESULT`. This minimal distance is initially the sum over all distances in `DistTable` (an upper limit), and the corresponding route is an empty tuple. Then, the master deposits the initial routes into tuple space `WORK`, and spawns `NumWorker` worker processes in active tuples to compute the search tree in parallel. This number is an argument to the main program. These workers execute in a loop, in which tasks are fetched from tuple space `WORK`, and results are computed and added to tuple space `RESULT`.

After spawning the workers, the master waits until all workers have done their work, and then the master fetches

```
program tsp;
  visible constant DistTable :=
          {[["Dortmund","Essen"], 32], ...},
        Nodes := domain (domain DistTable) + range (domain DistTable);
  constant WORK := CreateTS(),      -- For the workers and the work tasks
           RESULT := CreateTS(),    -- For the actual minimal route and distance
           NumWorker := argv(2);    -- Program argument: number of workers
begin
  Start := "Dortmund";
  -- The minimal distance is initially the sum over all distances:
  Max := +/ [x(2): x in DistTable];
  deposit [ [], Max ] at RESULT end deposit; -- Initialize the result

  -- Deposit the initial routes into tuple space WORK:
  for Entry in DistTable | Entry(1)(1) = Start do
    deposit [ Entry(1), Entry(2)] at WORK end deposit;
  end for;

  for i in [1..NumWorker] do -- Spawn the worker processes in active tuples:
    deposit [ || Worker (WORK, RESULT)] at RESULT end deposit;
  end for;

  for i in [1..NumWorker] do -- Wait for the workers to finish
    fetch ( 0 ) at RESULT end fetch;
  end for;
  fetch ( ? route, ? distance ) at RESULT end fetch; -- The work has been done

  if route = [] then
    put("There exists no route for the traveling salesman!");
  else
    put("Tour de Ruhr = ", route);
    put("Distance = ", distance);
  end if;
end tsp;
```

**Figure 5**. Solution for the traveling salesman problem in ProSet-Linda: main program as master process. The unary operator `domain` yields the domain of a map (a set of pairs). Accordingly, `range` yields the range of a map. For sets, + is the set union. The unary operator +/ yields the sum over all elements in a compound data structure (a tuple in our example). The function `CreateTS` creates a new tuple space.

the optimal distance together with the corresponding route from tuple space `RESULT`. Here, Multilisp's future concept is applied to synchronize the master with the workers: the workers are spawned with the operator || as components of *active* tuples. Only passive tuples can match a template; thus, the master waits for the termination of the workers, and only then fetches the results. The workers need not terminate in a specific order because each one resolves into the passive tuple `[0]` in tuple space `RESULT`. Tuple spaces are multisets.

Each worker (Figure 6) first checks whether there are more task tuples in tuple space `WORK`, and terminates when there is no more work to do. Then each worker checks whether its partial route (then stored in `MyRoute`) exceeds the length of an already known complete route: then the worker discards this partial route (according to the bounding rule) and continues to fetch another task tuple. If the length of the partial route does not exceed the length of an already known complete route, the worker checks whether its partial route is already a complete route. If the par-

```
procedure Worker (MyWORK, MyRESULT);
begin
 loop
  fetch (? MyRoute, ? MyDistance) at MyWORK
   else return 0; -- Terminate and return 0 into the comprising tuple
                  -- (become passive)
  end fetch;

  meet ( ?, ? Distance ) at MyRESULT end meet; -- to check whether we can continue
  if Distance <= MyDistance then
    continue; -- There exists already a shorter or equal long route:
              -- we prune this subtree according to the bounding rule
  end if;

  if #MyRoute >= #Nodes then
    -- We have a complete route. Change the minimum to our route if it is
    -- still the shortest one:
    meet ( ? into MyRoute, ? into MyDistance | $(2) > MyDistance  )
     xor ( ?, ?                              | $(2) <= MyDistance )
      at MyRESULT
    end meet;
  else
    -- Deposit a new task for each route which is a connection of MyRoute
    -- with a node that is not in MyRoute:
    for Entry in DistTable | (Entry(1)(1) = MyRoute(#MyRoute) and
                              Entry(1)(2) notin MyRoute) do
      deposit [MyRoute with Entry(1)(2), MyDistance + Entry(2)]
        at MyWORK
      end deposit;
    end for;
  end if;
 end loop;
end Worker;
```

**Figure 6**. Solution for the traveling salesman problem: procedure for the worker processes. The unary operator # returns the number of elements in a compound data structure. The binary operator `with` adds an element to a compound data structure. The special symbol $ is a place holder for matching tuples in tuple space.

tial route is already a complete route, the worker changes the minimal route in tuple space `RESULT` to the given route, provided that the given route is still the shortest one. If the partial route is not a complete route, the worker deposits new task tuples into tuple space `WORK` for each route which is a connection of the given route with a node that is not in the given route. There has to be a connection defined in `DistTable` between the last node in the given route and the next node that is not in the given route to constitute a new extended route.

For simplicity, we assume that there exists at least one complete route that visits each of a given set of cities exactly once. If such a complete route does not exist, the program prints a corresponding message. Often it is assumed in solutions for the traveling salesman problem that there exists a connection between each pair of cities. The presented program does not have this assumption, and also solves problems where the distances between two cities may depend on the direction. This program provides a very flexible solution to the traveling salesman problem which allows easy experimentation with alternative program and data structures.

***Some sample systems***. Other approaches to prototyping concurrent applications suggest the use of coordination languages:

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

—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. Using Durra, the application structure and the resources allocated to the individual processes are specified independently from the coding of the sequential components. These coordination specifications may be validated by a run-time interpreter to allow prototyping.

The block on coordination languages in Table I classifies these approaches with respect to the process of prototyping concurrent applications. Only a few issues are addressed in this category. The emphasis is on programming issues.

### 3.4 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, interprocess 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

```
prototype A_Lift
  (* Type definitions: *)
  POSITION          : int
  STEP              : int
  FLOOR             : int
  FLOORS            : set of FLOOR
  UP_BUTTONS        : FLOORS
  DOWN_BUTTONS      : FLOORS
  PANEL             : int
  LIFT_PANEL        : set of PANEL
STATE::      (* Definition of the state variables: *)
  position          : POSITION
  step              : STEP
  up_buttons        : UP_BUTTONS
  down_buttons      : DOWN_BUTTONS
  direction         : bool
  panel_buttons     : LIFT_PANEL

  (* Function definitions: see Figure 8. *)
end
```

**Figure 7**. The state space of the lift system in PSP.

prototyping concurrent applications with functional languages usually support some kind of state.

***An example***. In the PSP approach [Heping and Zedan 1996a; 1996b], a functional language has been extended with *state variables* to allow prototyping parallel responsive/reactive systems. As an example, let us consider the simulation of a lift system. The lift system has to react on external events that may occur concurrently. The specification of the state space for the lift system is displayed in Figure 7. In this example, *position, step, direction, up_buttons, down_buttons*, and *panel_buttons* are global state variables. *Position* denotes which floor the lift is at during that time. *Up_buttons* and *down_buttons* record which up- and down-buttons at different floors have been pushed at a time point. These variables are sets of integers, whereby each integer denotes a floor. The corresponding types are defined before the state variables are declared (see Figure 7). *Direction* denotes whether the current direction of the lift is *up*. The set *panel_buttons* records the states of the buttons on the lift panel.

A PSP function for changing the direction in this lift system is displayed in Figure 8. The local variables *up_request* and *down_request* indicate whether there exist up- and down-requests. The nested conditional expression following the keyword **in** determines the new direction (this is the function body). The auxiliary function *Been_served* ensures that the direction is changed only when the current floor has been served and the door is closed.

The local variables and the function body are defined in the style of the functional language ML [Ullman 1994]. The addition of shared state variables for parallel functions enables this functional program to remember its current state. A pure functional program without state variables would not allow such a program structure.

***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

```
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
```

**Figure 8**. A PSP function for changing the direction in the lift system.

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].

The block on functional languages in Table I classifies these approaches with respect to the process of prototyping concurrent applications. Similar to set-oriented data parallelism, transformations are supported in this category. Only a few other issues are addressed in this category. Again, the emphasis is on programming issues.

### 3.5 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 3.4). 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].

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; 1996]. Actors extend the concept of objects to concurrent computation: Each actor potentially executes in parallel with other actors and may send asynchronous 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 (i.e., supporting concurrency *and* inheritance), 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]. This survey covers only concurrent *object-based* languages.

***An example***. The ActorSpace model [Callsen and Agha 1994] extends the Actor model's point-to-point, asynchronous communication with decoupled, pattern-directed communication. ActorSpace supports *open interfaces* in so-called *actorspaces* which act as contexts for *matching* patterns. Actorspaces are somewhat similar to Linda's tuple spaces (Section 3.3), as the pattern-directed communication approach is based on the associative access in the Linda model. An important difference is that the patterns are not associated with messages (tuples), but with the message's recipients (destination patterns). Actorspaces may be nested. Patterns are matched against listed attributes of actors and actorspaces that are *visible* in an actorspace. Both visibility and attributes may change dynamically. Messages may be sent to one or all members of a *group* defined by a pattern (multicasting). An important goal is to provide a more efficient and secure communication compared to the basic Linda model. Security is provided through *capabilities* that are bound to actors and actorspaces on their creation. Only the holder of the capability for an actor or actorspace can change its *visibility*, which is required to enable pattern matching.

The ActorSpace model provides two kinds of handles to access an actor: the usual actor mail address and the attributes for an actor that are visible for matching in some actorspace. The operation '`send (pattern, message)`' sends the message to a *single* actor which is nondeterministically chosen out of the group of matching receivers. The operation "`broadcast (pattern, message)`" sends the message to *all* actors, whose attributes match the pattern.

As an example, we present a simple model of a car manufacturer adopted from Callsen and Agha [1994]. It is assumed that the individual components of a car are manufactured by subcontractors. The contractor assembles the pieces and delivers the car to the customers. There may be several subcontractors that are able to supply the same components. Figure 9 illustrates the system architecture. Actors are displayed as circles and the actorspaces that contain the actors are displayed as rectangles. Customers order cars. Then, the main contractor orders the components from the subcontractors. One of the subcontractors in a category (e.g., wheel manufacturers) delivers the requested components to the main contractor, who delivers the assembled car to the customer.

The main contractor's method for assembling a car is shown in Figure 10. The syntax is based on C++. The pattern '`wheel_space:`' in the first `send` operation identifies the matching subcontractors for wheels whose methods `make_wheel` are called on receipt (see Figure 11). The parameter `self` identifies the main contractor within its

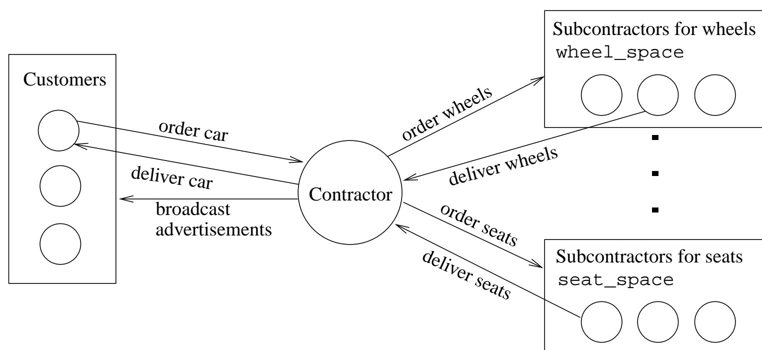


**Figure 9**. The system architecture for the car manufacturing example in ActorSpace.

method `make_car`. Note that the main contractor does not need to know the address of the actual subcontractor that serves the request, it only specifies the appropriate pattern for wheel manufacturers. The wheel manufacturer's method for making and supplying a wheel is displayed in Figure 11.

In addition, the contractor sometimes broadcasts advertisements to the customers (Figure 9). To summarize: the ActorSpace model allows a very flexible object-based system architecture by means of grouping actors in actorspaces and its pattern-directed communication.

***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.

—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. 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 3.6).

—The CODB (Common Object DataBase) approach for prototyping distributed virtual environments is based on containerization of data objects [Stytz et al. 1997].

—The PROTOB [Baldassari et al. 1991] approach considers nodes in PROT nets [Bruno and Marchetto 1986] to be communicating objects (see also Section 3.6). Inheritance is not supported by PROTOB.

—The parallel constructs of Compositional C++ (CC++) [Chandy and Kesselman 1993] are based on the ideas of committed choice logic, 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. CC++ is explicitly designed for rapid prototyping of concurrent applications.

```
method @make_car()
{
   int i;
   // Make 4 wheels:
   for (i=0; i<4; i++)
   {
      // Each invocation makes the wheel 'i' and replies
      send (wheel_space:@make_wheel, self, i);
   }
   // Make 2 seats:
   for (i=0; i<2; i++)
   {
      // Each invocation makes the seat 'i' and replies
      send (seat_space:@make_seat, self, i);
   }
   // Make the other components of a car and assemble it
   ...
}
```

**Figure 10**. The main contractor's method for assembling a car.

```
method @make_wheel (name requester, int wheel_no)
{
   struct { ... } wheel;

   // Create the wheel:
   ...
   // Reply to the requester with the wheel and its number:
   send (requester:@wheel_made, wheel, wheel_no);
}
```

**Figure 11**. The wheel manufacturer's method for making a wheel.

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 et al. [1993] provide a collection of papers on various concurrent object-oriented approaches, and Wyatt et al. [1992] provide a survey of concurrent object-oriented languages. The block on object-based languages in Table I classifies these approaches with respect to the process of prototyping concurrent applications.

### 3.6 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 multidimensional 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; Schaeffer and Szafron 1996]

is a programming environment for developing concurrent 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 concurrent programs.

Using the business analogy, concurrent structures are called *assets* for which specific icons are available in the graphical user interface. The supported assets are:

—Enterprise: it represents the complete concurrent *program*.

—Department: it represents a sequential *master* process in traditional parallel terminology.

—Individual: it represents a sequential *worker* process in traditional parallel terminology.

—Line: it represents a *pipeline* which may contain a fixed number of heterogeneous assets in a specified order.

—Division: it represents a *divide-and-conquer* computation that contains a hierarchical collection of individual assets among which the work is distributed.

—Service: it represents a *monitor* in traditional parallel terminology.

Figure 12 displays a snapshot while executing a concurrent master-worker program with Enterprise. The upper right window displays the master (Department) and two worker processes using the icons for departments and individuals. The double-line rectangle in the upper left window represents the *enterprise*, which is the entire program. Each icon represents the state of a concurrent process. Message queues are displayed as connecting lines between the process icons. The construction of such graphical models is very similar to drawing graphics with standard tools such as xfig or Powerpoint. The lower text window shows the sequential code for the master process and the window on the right displays some run-time information.

With Enterprise, the user begins by representing a program as a single enterprise asset containing a single individual (the main process). The user specifies the desired concurrent programming 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 13 displays a snapshot while animating and replaying a concurrent program with Enterprise. Replaying a concurrent program helps to reproduce nondeterministic errors.

Some basic operations are used to build a concurrent program: addition, duplication, exchange, and expansion of assets. Compared to Figure 12, the `Cube` and `Square` assets (individuals) are duplicated in Figure 13 to increase parallelism. Program animation is used to monitor and to identify weak points in the implemented parallel algorithms. The system assumes that the displayed events are partially ordered. It is possible to monitor the concurrent program while it is running or to replay the events.

Szafron and Schaeffer [1996] provide an experimental comparison of concurrent 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. Enterprise users ended up writing 66% less code than did message-passing users. Some message-passing users had problems with program correctness; some Enterprise users had problems with performance. These results indicate that the Enterprise tool can be very useful for prototyping, because performance issues should only be addressed once correctness has been established.

Each of the Enterprise assets can be

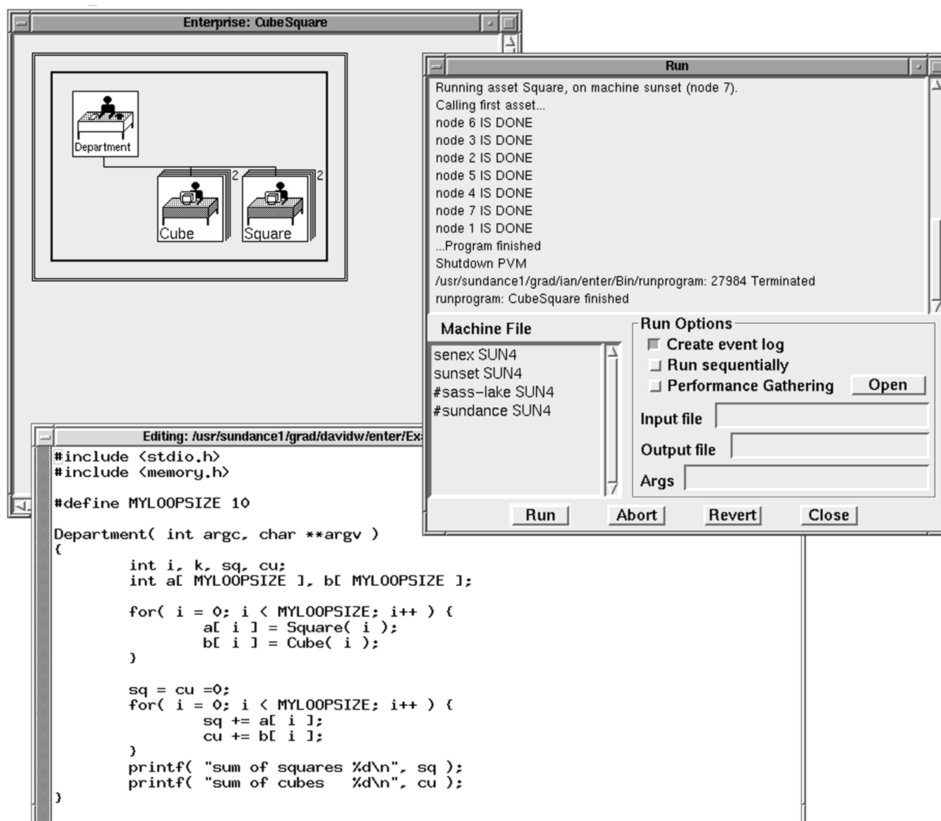


**Figure 12**. Executing a concurrent program with Enterprise.

regarded as a template, or algorithmic skeleton [Cole 1989], for parallelizing pieces of sequential source code. The example code in the lower text window in Figure 12 is sequential code. The parallelism is specified graphically. Enterprise supports coarse-grained concurrent 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 concurrent programs, but for many applications it relieves users from the tedious details of distributed communication to let them concentrate on algorithm development. An important goal in the Enterprise approach is the separation of sequential code from the concurrent constructs. This separation of concerns is a common goal in many graphical approaches to high-level concurrent programming and prototyping.

***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 concurrent 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.

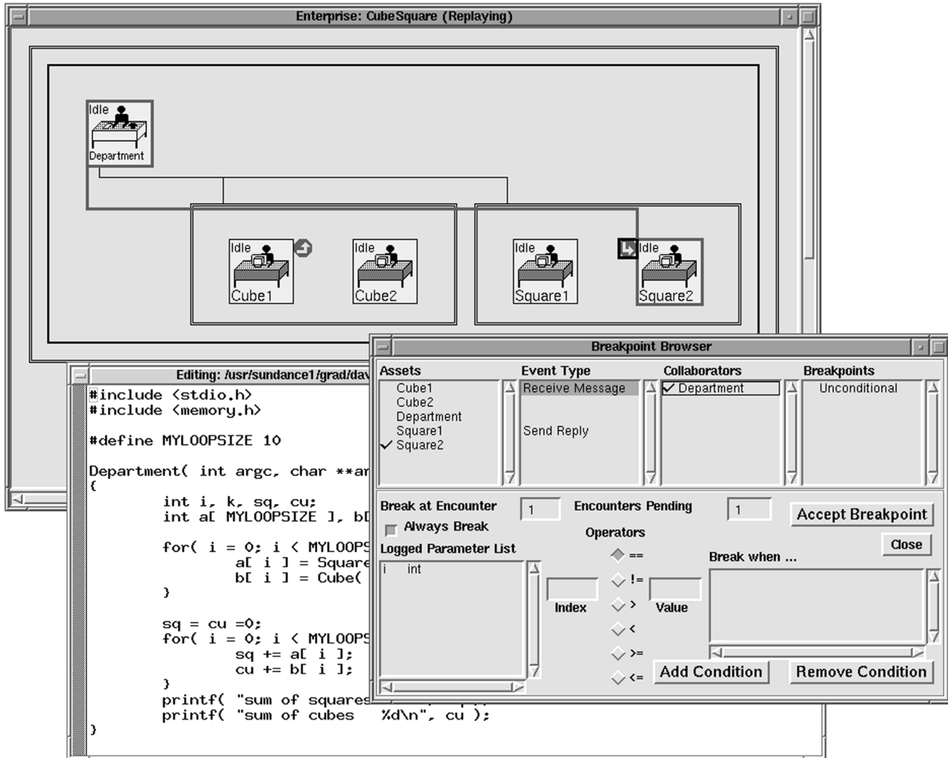


**Figure 13**. Animation and replay of a concurrent program with Enterprise.

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 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.

—The G-Net [Deng et al. 1993] formalism extends Petri-nets with modules to allow prototyping of complex information systems.

—PROT net [Bruno and Marchetto 1986] is a high-level Petri-net formalism for prototyping concurrent

systems. With PROT nets, tokens represent processes.
—Generalized Stochastic Petri-nets (GSPNs) [Donatelli et al. 1994] are proposed for prototyping functionality and performance of concurrent algorithms.
—The PARSE [Gorton et al. 1995] process graph notation allows description a parallel system in terms of a hierarchy of interacting components. These components are either passive function or data servers, or active control processes. 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].

(2) Animation and simulation of state-transition diagrams may be used for prototyping concurrent applications:
—The AutoFocus tool [Huber and Schätz 1997; Huber et al. 1998] combines state-transition diagrams with system-structure and event-trace diagrams. Prototyping is supported through generation of Java code from the graphical specification.
—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 concurrent system is described as a set of communicating graphs, where each graph is either a server or a client.
—Formal data-flow diagrams (FDFDs) [Fuggetta et al. 1993] with precise semantics for synchronization in combination with E/R-diagrams are proposed for prototyping concurrent systems. Execution of FDFDs is based on their formal semantics.
—Data-flow diagrams are specified in Jones et al. [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. Examples are EDPEPPS [Delaitre et al. 1997], JADE [Unger 1988], MCSE [Calvez et al. 1994], Process-

| | Methods for transformations | Tools for transformations | Performance prototyping | Real-time considered | Fault-tolerance considered | User experience reported |
|---|---|---|---|---|---|---|
| *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* | | | | | | |
| AutoFocus [Huber et al. 1998] | – | – | – | – | – | – |
| CRSM [Raju and Shaw 1994] | – | – | – | ✔ | – | – |
| EFSM [Valenzano et al. 1993] | ✔ | ✔ | – | – | – | – |
| *Based on Data-Flow Diagrams* | | | | | | |
| EDFG [Levy et al. 1990] | – | – | – | – | – | – |
| FDFD [Fuggetta et al. 1993] | – | – | – | – | – | – |
| IDE [Jones et al. 1990] | – | – | – | – | – | – |
| PSDL [Luqi et al. 1988; Berzins et al. 1993] | ✔ | ✔ | – | ✔ | – | – |
| *Other Notations* | | | | | | |
| EDPEPPS [Delaitre et al. 1997] | – | – | ✔ | – | – | – |
| Enterprise [Schaeffer et al. 1993] | – | – | – | – | – | ✔ |
| JADE [Unger 1988] | – | – | – | – | – | – |
| MCSE [Calvez et al. 1994] | – | – | – | ✔ | – | – |
| ProcessBuild [Weinreich and Ploesch 1995] | – | – | – | ✔ | – | – |
| REALMoD [Suarez et al. 1998] | – | – | ✔ | ✔ | – | – |
| Transim/Gecko [Hart and Flavell 1990] | – | – | – | ✔ | – | – |

**Table II**. Classification of the Surveyed Graphical Approaches, with Respect to Various Issues for Prototyping Concurrent Applications

Build [Weinreich and Ploesch 1995], REALMoD [Suarez et al. 1998], and Transim/Gecko [Hart and Flavell 1990] which are tools that support several graphical representations for prototyping through simulation and animation of concurrent systems. These graphs represent data and control flow in various ways.

Table II classifies the graphical approaches with respect to the process of prototyping concurrent applications. Only approaches that are based on formal notations such as Petri-Nets and state-transition diagrams support transformations.

## 4. TRANSFORMING PROTOTYPES INTO EFFICIENT IMPLEMENTATIONS

***Idea***. 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* [Partsch 1990]. 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 semiautomatically with some kind of tool support.

***An example***. Proteus [Mills et al. 1991] is a set-oriented data-parallel language designed for prototyping concurrent applications (see Section 3.2). The semiautomatic refinement system for the Proteus language is based on algebraic specification techniques and category theory to transform prototypes to implementations on specific architectures [Goldberg et al. 1994].

Parallelism in the data-parallel subset of Proteus comes from the iterator construct. Several semantics preserving transformation rules for iterator elimination have been defined for these constructs to replace them by calls to the Data Parallel Library (DPL) [Prins and Palmer 1993]. DPL is a collection of C functions for vector processors which provides the capability to treat nested sequences as primitive data types. This library is designed specifically to be used as a target notation for transformed Proteus programs.

**Programming:** We illustrate the transformation to low-level data-parallel code by means of the simple expression

```
[i in [1..5]: sqs(i)]
```

using the function sqs defined as follows:

```
function sqs(n: [int]) =
return [j in [1..n]: j*j];
```

This expression evaluates to a tuple containing tuples that contain the square numbers for 1 to 5: [[1],[1,4],[1,4,9] [1,4,9,16],[1,4,9,16,25].

**Evaluation:** The execution time of this small program is not optimal on vector processors, because it is not possible to translate it directly into a C program with calls to the DPL functions.

**Transformation:** In a first transformation step, a *function application* rule is applied to pull out the function call:

```
sqs1 ([i in [1..5]: i])
```

whereby $sqs_1$ is a modified version of sqs that is applicable to a tuple of integers. Instead of repeatedly evaluating single values, the function $sqs_1$ now must evaluate a sequence of values in parallel. A rule for *generating data-parallel functions* is applied to define $sqs_1$:

```
function sqs1 (V: [[int]]) =
 return [i in [1..#V]:
 [ j in [1..V[i]]: j*j] ];
```

The unary operator # returns the length of a tuple. A *syntactical* transformation writes all operations in the body of $sqs_1$ (behind the return keyword) in prefix form as follows:

```
[i in range1(length(V)): [j in
range1(index(V,i)): mult(j,j)]]
```

In the following steps this function body is transformed. Again, a *function application* rule is applied to pull out the function call to mult:

```
[i in range1(length(V)): mult1([ j
in range1(index(V,i)): j],
   [ j in range1(index(V,i)): j] ]
```

In the next transformation step, an *iterator nesting* rule is applied to simplify the inner iterator:

```
[i in range1(length(V)):
   mult1(range1(index(V,i)),
   range1(index(V,i)) ) ]
```

Next, the outer iterator is transformed through both the $\texttt{mult}_1$ and `range1` operations introducing $\texttt{mult}_2$ (function application rule):

```
mult2(range1(
[i in range1(length(V)):index(V,i) ]
      [i  in  range1(length(V)):in-
      dex(V,i) ] ))
```

The *identity* rule (`[x in [i..#Y]: Y[x]] ≡ Y`) simplifies the function body of $\texttt{sqs}_1$ to:

```
mult2(range11(V),range11(V)
```

A rule for the implementation of *deep data-parallel functions* introduces `insert` and `extract` to alter the depth of the input such that $\texttt{mult}_1$ can be used to implement $\texttt{mult}_2$:

```
insert(mult1(extract(range11(V),1),
      extract(range11(V),1),
      range11(V),1))
```

In a final transformation step, an *iterator nesting* rule is applied again to simplify the iterator from the original expression:

```
sqs1([1..5])
```

At this point we have replaced all the iterators in the example expression and the called function `sqs`. Now, this transformed version can be translated directly into a C program with calls to the DPL functions [Prins and Palmer 1993]. This step is just syntactical translation to C.

Nyland et al. [1996] discuss the transformation of data-parallel Proteus programs to low-level data-parallel systems and to message-passing libraries. For the time being, the transformation rules for Proteus programs are restricted to the data-parallel constructs of Proteus. Within the Proteus approach it has been considered to use the KIDS semiautomatic program development system [Smith 1990] to help the programmer with the transformations.

However, 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 semiautomatic transformation of *control-parallel* prototypes into efficient low-level programs is, therefore, still an unsolved problem and subject to further research.

***Some sample systems***. Some manual transformations of prototypes into efficient implementations are discussed in the literature:

—In Hummel et al. [1995], high-level data-parallel algorithm specifications are refined within PSETL (see Section 3.2). High-level PSETL code is successively transformed manually into lower-level architecture-specific PSETL code.

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

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

—The manual transformation of PROSET-Linda prototypes into efficient C-Linda and message-passing implementations is discussed in Hasselbring et al. [1997]. It has been proposed to transform Standard ML prototypes into occam programs [Wallace et al. 1992].

—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.

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

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

—A tool for the Communicating Petri-nets (CmPNs) approach supports code generation from graphical prototypes [Bucci et al. 1995].

—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].

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

## 5. CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

To develop a concurrent application, you should start with executable prototypes to experiment with ideas (neglect the execution performance in the first instance). Only when a program is running properly should performance be a consideration, but the development process for *concurrent* applications should also allow the analysis of performance issues in the prototyping phase. Powerful tools are needed to make prototyping of concurrent applications feasible.

### 5.1 Conclusions

This paper surveys several programming languages and systems for prototyping 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. Domain-specific libraries are designed for specific application domains. Set-oriented data-parallelism is usually applied in scientific, numerical applications. Coordination languages emphasize a linguistic separation of sequential and concurrent programming constructs, whereas graphical programming approaches aim at separating linguistic sequential code from the graphical concurrent constructs. Concurrent functional languages provide some extensions for states and nondeterminism. Concurrent object-based approaches aim at flexible software architectures for concurrent applications.

Table I presents an overview of the linguistic approaches and Table II 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 horizontal structure of the tables corresponds to the taxonomy in Figure 1. 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. The individual approaches are classified with respect to the following issues:

—Do methods for transforming prototypes into efficient implementations exist?

—Do tools for transforming prototypes into efficient implementations exist?

—Is performance explicitly considered in the approach (e.g., through performance visualization [Heath and Etherirdge 1991; Pancake et al. 1995])?

—Is real-time explicitly considered in the approach?

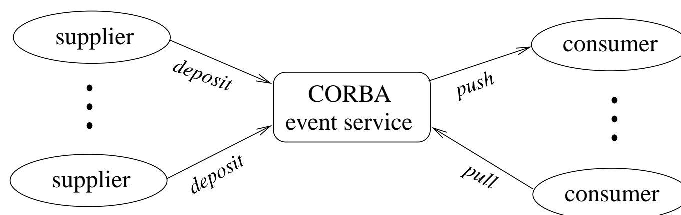


**Figure 14**. Decoupled push and pull communication with CORBA event services.

—Is fault-tolerance explicitly considered in the approach?

—Is experience from actual users reported (i.e., users who do not belong to the development teams of the particular approach)?

As we can see, some of the included approaches have methods for transformations, and few are accompanied with tools to help with the transformation into efficient implementations. Only a few approaches explicitly support performance evaluation of prototypes. Several approaches consider real-time, but only one approach aims at prototyping of fault-tolerance issues.

Only a few approaches have been used by actual users who do not belong to the development teams of the particular approach. Most of the tools have only been used as research prototypes. In particular, the lack of appropriate integration of debugging, performance evaluation and transformation of prototypes within most research tools could be a problem for real users.

The focus of the present paper is very much on 'traditional' concurrent programming languages and systems. Meanwhile, there exist a lot of work in distributed applications, where developments such as CORBA [Mowbray and Zahavi 1995] are attempting to lift the level of abstraction. For instance, the CORBA event services [Mowbray and Zahavi 1995] may be used to provide synchronous or asynchronous transfer of objects using event channels to decouple the communication between distributed objects. Consumer objects can either receive notification of events that concern them (*push* model) or can connect to the event channel to wait for their events (*pull* model). Figure 14 illustrates both mechanisms, where suppliers deposit information at the event channel and consumers fetch them in different ways. The event service can be implemented as a specialized CORBA object which means that it can be used by multiple suppliers and consumers simultaneously. In effect, this means that multiple suppliers can pass information to multiple consumers using the same event channel without any supplier or consumer having direct knowledge of each other. OrbixTalk is an implementation of the CORBA event services [IONA Technologies PLC. 1997]. Event channels provide a decoupling of the consumers and producers of events very similar to Linda's tuple spaces (Section 3.3) or the ActorSpace model (Section 3.5). Suppliers and consumers may register and unregister with an event channel with no consequences to either other suppliers who could be providing events or consumers who may be listening for events. Furthermore, the combination of CORBA with Java [Vogel and Duddy 1998] may provide a powerful tool for developing concurrent systems based on distributed active objects (see Section 3.5). Therefore, several concepts of high-level concurrent languages seem to have some influence on the design of advanced middleware platforms for distributed systems.

When developing distributed systems, software engineers often use the high-level support offered by middleware platforms or Internet techniques in order to build experimental prototypes

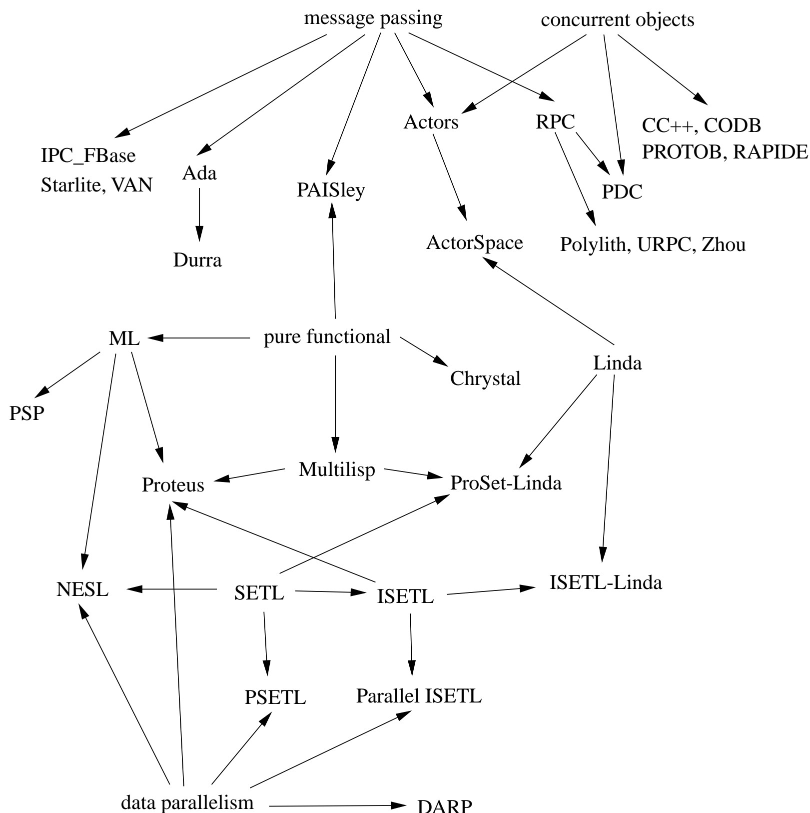


**Figure 15**. An illustration of the historical development of some of the linguistic approaches surveyed in the paper. The arrows indicate dependencies.

that can then be extended into full systems, instead of using the surveyed sytems that are specifically designed for prototyping. The reasons for these practices is that, despite of 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.

The information in the tables is extracted from the available literature. It could be the case that, in the meantime, there should be some more ticks in some of the table cells, but this would not substantially change our general conclusions which indicate some directions for future work (see Section 5.2).

To summarize the paper, Figure 15 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 II.

### 5.2 Directions for Future Work

The essential prototyping activities are programming, evaluation and transformation of prototypes into efficient implementations. Linguistic approaches emphasize programming concerns. Animation and simulation of graphical representations concentrate on evaluation concerns. The integration of the following features could form powerful means for developing concurrent applications:

—High-level linguistic approaches.

—Graphical representations of concurrency.

—Prototyping of user interfaces.

—Debugging support with replay features (post-mortem analysis).

—Performance visualization tools.

—Support for transforming prototypes into efficient implementations.

In particular, the performance evaluation of prototypes and the systematic transformation of prototypes into efficient low-level programs are still unsolved problems. For instance, DARP [Akarsu et al. 1998] goes a step into the right direction as it integrates an interpreter, a source level debugger, data visualization and data analysis packages for prototyping of High-Performance Fortran (HPF) programs.

Anyway, the appearance of workshops and symposia on the subject is promising [PDSE 1996; PDSE 1997; PDSE 1998; PDSE 1999]. Prototyping is one important concern for software engineering of concurrent applications.

## ACKNOWLEDGMENTS

The comments on drafts of this paper by the anonymous referees were a valuable source to improve the paper.

## REFERENCES

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA.

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

AGHA, G. A. 1996. Linguistic paradigms for programming complex distributed systems. *ACM Comput. Surv. 28*, 2, 295–296.

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

AKARSU, E., FOX, G., AND HAUPT, T. 1998. DARP: Java-based data analysis and rapid prototyping environment for distributed high performance computations. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing* (Palo Alto, CA, Feb./Mar.), ACM Press, New York, NY.

ANDREWS, G. R. 1991. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publ. Co., Inc., Redwood City, CA.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

BAL, H. E. 1990. *Programming Distributed Systems*. Silicon Press, Summit, NJ.

BAL, H. E., STEINER, J. G., AND TANENBAUM, A. S. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv. 21*, 3 (Sep. 1989), 261–322.

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

BANERJEE, U., EIGENMANN, R., NICOLAU, A., AND PADUA, D. A. 1993. Automatic program parallelization. *Proc. IEEE 81*, 2 (Feb.), 211–243.

BARBACCI, M. AND LICHOTA, R. 1991. Durra: An integrated approach to software specification, modeling, and rapid prototyping. In *Proceedings of the Second International Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June), N. Kanopoulos, Ed. IEEE Computer Society Press, Los Alamitos, CA, 67–81.

BARRY, B. M. 1989. Prototyping a real-time embedded system in Smalltalk. *SIGPLAN Not. 24*, 10 (Oct. 1989), 255–265.

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

BISCHOFBERGER, W. AND POMBERGER, G. 1992. *Prototyping Oriented Software Development Concepts and Tools*. Springer-Verlag, Berlin, Germany.

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

BLELLOCH, G. E., HARDWICK, J. C., SIPELSTEIN, J., ZAGHA, M., AND CHATTERJEE, S. 1994. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput. 21*, 1 (Apr. 1994), 4–14.

BREANT, F. 1991. Rapid prototyping from Petri-Net on a loosely coupled parallel architecture. In *Proceedings of the International Conference on Applications of Transputers* (Glasgow, UK), T. S. Durrani, W. A. Sandham, J. J. Soraghan, and J. Hulskamp, Eds. IOS Press, Amsterdam, The Netherlands, 644–649.

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

BUCCI, G., MATTOLINI, R., AND VICARIO, E. 1995. Automated transition from rapid prototyping to target code for distributed systems. In *Proceedings of the 2nd International Symposium on on Autonomous Decentralized Systems* (ISADS'95, Phoenix, AZ, June 1995), IEEE Computer Society Press, Los Alamitos, CA, 104–111.

BUCCI, G. AND VICARIO, E. 1992. Rapid prototyping through communicating Petri nets. In *Proceedings of the Third Interntional Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June), N. Kanopoulos, Ed. IEEE Computer Society Press, Los Alamitos, CA, 58–75.

BUCHS, D., FLUMET, J., AND RACLOZ, P. 1992. Producing prototypes from CO-OPN specifications. In *Proceedings of the Third Interntional Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June), N. Kanopoulos, Ed. IEEE Computer Society Press, Los Alamitos, CA, 77–93.

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

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

CALLSEN, C. J. AND AGHA, G. 1994. Open heterogeneous computing in ActorSpace. *J. Parallel Distrib. Comput. 21*, 3 (June 1994), 289–300.

CALVEZ, J., PASQUIER, O., AND HERAULT, V. 1994. A complete toolset for prototyping and validating multi-transputer real-time applications. In *Proceedings of the Conference on TRANSPUTERS'94: Advanced Research and Industrial Applications* (Royale d'Arc et Senans, France, Sept.), M. Becker, L. Litzler, and M. Tréhel, Eds. IOS Press, Amsterdam, The Netherlands, 71–86.

CAO, J., DE VEL, O., AND WONG, K. 1993. Supporting a rapid prototyping system for distributed algorithms on a transputer network. In *Transputer and Occam Research: New Directions*, J. Kerridge, Ed. IOS Press, Amsterdam, The Netherlands, 115–130.

CARD, D. 1995. Introduction to the special issue on rapid application development. *IEEE Software 12*, 5, 19–21.

CHANDY, K. M. AND KESSELMAN, C. 1993. CC+: a declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MIT Press, Cambridge, MA, 281–313.

CHANDY, K. M. AND TAYLOR, S. 1992. *An Introduction to Parallel Programming*. Jones and Bartlett Publ., Inc., Sudbury, MA.

CHEN, M., CHOO, Y.-I., AND LI, J. 1991. Crystal: theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed. ACM Press Frontier Series. ACM Press, New York, NY, 255–308.

CIANCARINI, P. 1992. Parallel programming with logic languages: a survey. *Comput. Lang. 17*, 4 (Apr.), 213–240.

CIANCARINI, P. 1996. Coordination models and languages as software integrators. *ACM Comput. Surv. 28*, 2, 300–302.

CLOCKSIN, W. F. AND MELLISH, C. S. 1987. *Programming in Prolog*. 3rd ed. Springer-Verlag, New York, NY.

COLE, M. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA.

CONNELL, J. AND SHAFER, L. I. 1995. *Object-Oriented Rapid Prototyping*. Yourdon Press Computing Series. Yourdon Press, Upper Saddle River, NJ.

DE KERGOMMEAUX, J. C. AND CODOGNET, P. 1994. Parallel logic programming systems. *ACM Comput. Surv. 26*, 3 (Sept. 1994), 295–336.

DELAITRE, T., RIBEIRO-JUSTO, G. R., SPIES, F., AND WINTER, S. C. 1997. A graphical toolset for simulation modelling of parallel systems. *Parallel Comput. 22*, 13, 1823–1836.

DEMARCO, T. 1978. *Structured Analysis and System Specification*. Yourdon Press, Upper Saddle River, NJ.

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 *Proceedings of the 14th International Conference on Application and Theory of Petri Nets* (Chicago, IL, June), M. Marsan, Ed. Springer-Verlag, New York, 206–223.

DOBERKAT, E.-E., FRANKE, W., GUTENBEIL, U., HASSELBRING, W., LAMMERS, U., AND PAHL, C. 1992. ProSet; A language for prototyping with sets. In *Proceedings of the Third Interntional Workshop on Rapid System Prototyping* (Research Triangle Park, NC, June), N. Kanopoulos, Ed. IEEE Computer Society Press, Los Alamitos, CA, 235–248.

DONATELLI, S., FRANCESCHINIS, G., RIBAUDO, M., AND RUSSO, S. 1994. Use of GSPNs for concurrent software validation in EPOCA. *Inf. Softw. Technol. 36*, 7 (July), 443–448.

DONGARRA, J. J., OTTO, S. W., SNIR, M., AND WALKER, D. 1996. A message passing standard for MPP and workstations. *Commun. ACM 39*, 7, 84–90.

DOUGLAS, A., ROWSTRON, A., AND WOOD, A. 1995. ISETL-Linda: parallel programming with bags. Tech. Rep. YCS 257. Department of Computer Science, University of York, York, UK.

DUBINSKY, E. AND LERON, U. 1993. *Learning Abstract Algebra with ISETL*. Springer-Verlag, Vienna, Austria.

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

FOSTER, I. AND TAYLOR, S. 1990. *Strand: New Concepts In Parallel Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ.

FRENCH, J. AND VILES, C. 1992. A software toolkit for prototyping distributed applications. Tech. Rep. CS-92-26. Department of Computer Science, University of Virginia, Charlottesville, VA.

FUGGETTA, A., GHEZZI, C., MANDRIOLI, D., AND MORZENTI, A. 1993. Executable specifications with data-flow diagrams. *Softw. Pract. Exper. 23*, 6 (June 1993), 629–653.

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

GELERNTER, D. AND CARRIERO, N. 1992. Coordination languages and their significance. *Commun. ACM 35*, 2 (Feb. 1992), 97–107.

GHEZZI, C., JAZAYERI, M., AND MANDRIOLI, D. 1991. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ.

GOLDBERG, A., MILLS, P., NYLAND, L., PRINS, J., REIF, J., AND RIELY, J. 1994. Specification and development of parallel algorithms with the Proteus system. In *DIMACS: Specification of Parallel Algorithms*, G. Blelloch, K. Chandy, and S. Jagannathan, Eds. AMS, Providence, RI.

GORDON, V. AND BIEMAN, L. 1995. Rapid prototyping: Lessons learned. *IEEE Software 12*, 1 (Jan.), 85–95.

GORTON, I., GRAY, J., AND JELLY, I. 1995. Object based modelling of parallel programs. *IEEE Parallel Distrib. Technol. 3*, 2, 52–63.

HALSTEAD, R. H. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (June 1, 1987), 231–274.

HASSELBRING, W. 1998. The ProSet-Linda approach to prototyping parallel systems. *J. Syst. Softw. 43*, 3, 187–196.

HASSELBRING, W., JODELEIT, P., AND KIRSCH, M. 1997. Implementing parallel algorithms based on prototype evaluation and transformation. Software-Technik Memo Nr. 93. Department of Computer Science, University of Dortmund, Dortmund, Germany.

HEATH, M. T. AND FINGER, J. E. 1991. Visualizing the performance of parallel programs. *IEEE Software 8*, 5 (Sept.), 29–39.

HEPING, H. AND ZEDAN, H. 1996. An executable specification language for fast prototyping parallel responsive systems. *Comput. Lang. 22*, 1, 1–13.

HEPING, H. AND ZEDAN, H. 1996. A fast prototype tool for parallel reactive systems. *J. Syst. Archit. 42*, 4, 251–266.

HILLIS, W. D. AND STEELE, G. L. 1986. Data parallel algorithms. *Commun. ACM 29*, 12 (Dec. 1986), 1170–1183.

HOUCK, C. AND AGHA, G. 1992. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing* ((ICPP 91), Volume II, St. Charles, IL, Aug.), 158–165.

HU, L. AND GORTON, I. 1997. A performance prototyping approach to designing concurrent software architectures. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems* (PDSE'97, Boston, MA, May), IEEE Computer Society Press, Los Alamitos, CA, 270–276.

HUANG, Y.-M. AND RAVISHANKAR, C. 1996. URPC: A toolkit for prototyping remote procedure calls. *Comput. J. 39*, 6, 525–540.

HUBER, F., MOLTERER, S., RAUSCH, A., SCHÄTZ, B., SIHLING, M., AND SLOTOSCH, O. 1998. Tool supported specification and simulation of distributed systems. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems* (PDSE'98, Kyoto, Japan, Apr.), B. Krämer, N. Uchihira, P. Croll, and S. Russo, Eds. IEEE Computer Society Press, Los Alamitos, CA, 155–164.

HUBER, F. AND SCHÄTZ, B. 1997. Rapid prototyping with AutoFocus. In *Formale Beschreibungstechniken für verteilte Systeme* (GI/ITG Fachgesprä. In English, St. Augustin, Germany), A. Wolisz, I. Schieferdecker, and A. Rennoch, Eds. 343–352.

HUMMEL, S. F. AND KELLY, R. 1993. A rationale for parallel programming with sets. *J. Program. Lang. 1*, 3, 187–207.

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

IONA TECHNOLOGIES PLC. 1997. White Paper: OrbixTalk. IONA Technologies PLC, Dublin, Ireland.

ISENSEE, S., RUDD, J., AND HECK, M. 1995. *The Art of Rapid Prototyping: User Interface Design for Windows and OS/2*. International Thomson Computer Press, Boston, MA.

JARD, C., MONIN, J.-F., AND GROZ, R. 1988. Development of Veda, a prototyping tool for distributed algorithms. *IEEE Trans. Softw. Eng. 14*, 3 (Mar.), 339–352.

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

JONES, D., DOWDESWELL, S., AND HINTZ, T. 1990. A rapid prototyping method for parallel programs. In *Proceedings of the Conference on The Transputer in Australia* (ATOUG-3, Syd-

ney, Australia, June), T. Bossomaier, T. Hintz, and J. Hulskamp, Eds. IOS Press, Amsterdam, The Netherlands, 121–128.

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

KRAMER, B. 1991. Prototyping and formal analysis of concurrent and distributed systems. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, IEEE Computer Society Press, Los Alamitos, CA, 60–66.

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

LEVY, A. M., KATWIJK, J. V., PAVLIDIS, G., AND TOLSMA, F. 1990. SEPDS: A support environment for prototyping distributed systems. In *Proceedings of the First International Conference on Systems Integration* (ISCI '90, Morristown, NJ, Apr. 23–26), L. Seifert, R. T. Yeh, P. Ng, and C. Ramamoorthy, Eds. IEEE Press, Piscataway, NJ, 652–661.

LICHTER, H., SCHNEIDER-HUFSCHMIDT, M., AND ZÜLLIGHOVEN, H. 1994. Prototyping in industrial software projects: Bridging the gap between theory and practice. *IEEE Trans. Softw. Eng. 20*, 11 (Nov. 1994), 825–832.

LOBE, G., SZAFRON, D., AND SCHAEFFER, J. 1993. Program design and animation in the Enterprise parallel programming environment. Tech. Rep. 93-04. University of Alberta, Edmonton, Canada.

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

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

MARTIN, J. 1991. *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN.

MATSUOKA, S. AND YONEZAWA, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MIT Press, Cambridge, MA, 107–150.

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

MEYER, B. 1997. *Object-Oriented Software Construction*. 2nd ed. Prentice-Hall, Inc., Upper Saddle River, NJ.

MICHAELSON, G. AND SCAIFE, N. 1995. Prototyping a parallel vision system in Standard ML. *J. Funct. Programm. 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 *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing* (Dallas, TX, Dec.), 26–34.

MITCHELL, S. AND WELLINGS, A. 1996. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. *Comput. Lang. 22*, 1, 15–26.

MOWBRAY, T. J. AND ZAHAVI, R. 1995. *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley and Sons, Inc., New York, NY.

NIXON, P., BIRKINSHAW, C., CROLL, P., AND MARRIOT, D. 1994. Rapid prototyping of parallel fault tolerant systems. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Systems* (Malaga, Spain), IEEE Computer Society Press, Los Alamitos, CA, 202–210.

NIXON, P. AND CROLL, P. 1993. The functional specification of Occam programs for time critical applications. In *Transputer and Occam Research: New Directions*, J. Kerridge, Ed. IOS Press, Amsterdam, The Netherlands, 131–145.

NYLAND, L., PRINS, J., GOLDBERG, A., MILLS, P., REIF, J., AND WAGNER, R. 1996. A refinement methodology for developing data-parallel applications. In *Proceedings of the Conference on EuroPar'96 Parallel Processing* (Lyon, France, Aug.), Springer-Verlag, New York, 145–150.

OUSTERHOUT, J. 1998. Scripting: Higher-level programming for the 21st century. *IEEE Computer 31*, 3, 23–30.

PANCAKE, C., SIMMONS, M., AND YAN, J. 1995. Performance evaluation tools for parallel and distributed systems. *IEEE Computer 28*, 11 (Nov.), 16–19.

PARTSCH, H. A. 1990. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer Texts and Monographs in Computer Science. Springer-Verlag, New York, NY.

POMBERGER, G. AND BLASCHEK, G. 1996. *Object-Orientation and Prototyping in Software Engineering*. Prentice-Hall Object-Oriented Series. Prentice-Hall, Inc., Upper Saddle River, NJ.

PRINS, J. F. AND PALMER, D. W. 1993. Transforming high-level data-parallel programs into vector operations. *SIGPLAN Not. 28*, 7 (July 1993), 119–128.

PURTILO, J. M. AND JALOTE, P. 1991. An environment for prototyping distributed applications. *Comput. Lang. 16*, 3/4 (1991), 197–207.

PURTILO, J. M., REED, D. A., AND GRUNWALD, D. C. 1988. Environments for prototyping parallel algorithms. *J. Parallel Distrib. Comput. 5*, 4 (Aug. 1988), 421–437.

QUINN, M. AND HATCHER, P. 1990. Data-parallel programming on multicomputers. *IEEE Software 7*, 5 (Sept.), 69–76.

RAJU, S. C. V. AND SHAW, A. C. 1994. A prototyping environment for specifying, executing and checking Communicating Real-Time State machines. *Softw. Pract. Exper. 24*, 2 (Feb. 1994), 175–195.

REISIG, W. 1985. *Petri Nets: An Introduction*. EATCS monographs on theoretical computer science. Springer-Verlag, New York, NY.

SCHAEFFER, J. AND SZAFRON, D. 1996. Software engineering considerations in the construction of parallel programs. In *High Performance Computing: Technology and Applications*, J. Dongarra, L. Grandinetti, G. Joubert, and J. Kowalik, Eds. Elsevier Science Inc., New York, NY.

SCHAEFFER, J., SZAFRON, D., LOBE, G., AND PARSONS, I. 1993. The Enterprise model for developing distributed applications. *IEEE Parallel Distrib. Technol. 1*, 3 (Aug.), 85–96.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21*, 3 (Sep. 1989), 413–510.

SMITH, D. R. 1990. KIDS: a semiautomatic program development system. *IEEE Trans. Softw. Eng. 16*, 9 (Sep. 1990), 1024–1043.

SON, S. H. AND KIM, Y. 1989. A software prototyping environment and its use in developing a multiversion distributed database system. In *Proceedings of the International Conference on Parallel Processing* (ICPP '89, Aug.), Pennsylvania State University, University Park, PA.

STYTZ, M., ADAMS, T., GARCIA, B., SHEASBY, S., AND ZURITA, B. 1997. Rapid prototyping for distributed virtual environments. *IEEE Software 14*, 5 (Sept.), 83–92.

SUAREZ, F., GARCIA, J., ENTRIALGO, J., GARCIA, D., GRAFFA, S., AND DE MIGUEL, P. 1998. A toolset for visualization and behavioral analysis of parallel real-time embedded systems based on fast prototyping techniques. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing* (PDP'98, Madrid, Jan.), IEEE Computer Society Press, Los Alamitos, CA.

SZAFRON, D. AND SCHAEFFER, J. 1996. An experiment to measure the usability of parallel programming systems. *Concurrency: Pract. Exper. 8*, 2, 147–166.

SZYMANSKI, B. K., Ed. 1991. *Parallel Functional Languages and Compilers*. ACM Press Frontier Series. ACM Press, New York, NY.

TALIA, D. 1997. Parallel computation still not ready for the mainstream. *Commun. ACM 40*, 7, 98–99.

ULLMAN, J. D. 1994. *Elements of ML Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ.

UNGER, B. 1988. Distributed software via prototyping and simulation: JADE. Tech. Rep. 88/300/12. University of Calgary, Calgary, Canada.

VALENZANO, A., SISTO, R., AND CIMINIERA, L. 1993. Rapid prototyping of protocols from LOTOS specifications. *Softw. Pract. Exper. 23*, 1 (Jan. 1993), 31–54.

VOGEL, A. AND DUDDY, K. 1998. *Java Programming with CORBA*. 2nd. ed. John Wiley and Sons, Inc., New York, NY.

WALLACE, A. M., MICHAELSON, G. J., MCANDREW, P., WAUGH, K. G., AND AUSTIN, W. J. 1992. Dynamic control and prototyping of parallel algorithms for intermediate- and high-level vision. *IEEE Computer 25*, 2 (Feb. 1992), 43–53.

WEINREICH, R. AND PLOESCH, R. 1995. Prototyping of parallel and distributed object oriented systems: The PDC model and its environment. In *Proceedings of the Hawaii International Conference on System Sciences* (HICSS '95, Maui, Hawaii, Jan.), IEEE Computer Society Press, Los Alamitos, CA.

WYATT, B., KAVI, K., AND HUFNAGEL, S. 1992. Parallelism in object-oriented languages: A survey. *IEEE Software 9*, 6 (Nov.), 56–66.

YONEZAWA, A., Ed. 1990. *ABCL: An Object-Oriented Concurrent System*. Computer systems series. MIT Press, Cambridge, MA.

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

ZEDAN, H. S. M., Ed. 1990. *Proceedings of the 13th Occam User Group Technical Meeting on Real-Time Systems with Transputers*. (OUG-13, York, UK, Sept. 18–20, 1990). IOS Press, Amsterdam, The Netherlands.

ZHOU, W. 1994. A rapid prototyping system for distributed information system applications. *J. Syst. Softw. 24*, 1 (Jan. 1994), 3–29.

1996. *Proceedings of the First International Workshop on Software Engineering for Parallel and Distributed Systems*. (PDSE'96, Berlin, Mar.). IEEE Computer Society Press, Los Alamitos, CA.

1997. *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*. (PDSE'97, Boston, MA, May). IEEE Computer Society Press, Los Alamitos, CA.

1998. *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. (PDSE'98, Kyoto, Japan, Apr.). IEEE Computer Society Press, Los Alamitos, CA.

1999. *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. (PDSE'99, Los Angeles, CA, May). IEEE Computer Society Press, Los Alamitos, CA.

Received: June 1997; revised: January 1998; accepted: January 1999