INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Imperative Concurrent Object-Oriented Languages: An Annotated Bibliography

Michael Philippsen*
phlipp @icsi.berkeley.edu

TR-95-049

Version 0, August 1995

Abstract

The title says it all.

*On leave from Department of Computer Science, University of Karlsruhe, Germany

Contents

1 General Comments

In this survey, we consider only languages for parallel object-oriented programming. We do not take systems into account that are focused on distributed computing. Three criteria help to differentiate between the two approaches. We consider systems to focus on distributed computing instead of being a language

- if they provide solutions for problems that arise when several programmers use the same objects (e.g. printer monitors) in otherwise unrelated programs, or
- if the objects exist even when no program is running that accesses them.
- if the system uses an IDL (interface definition language), i.e., if it is more a two-language approach.

There are a lot of systems for distributed object-based programming and a huge body of active research is directed to problems from this context. A good survey is given in [72]. But before proceeding, we would like to name and shortly characterize some of the more influential systems in that area, in particular those where there is not a static relation between objects and processes that work on them.

Amadeus [] *general distributed c++ implementation*

Argus Arguments to remote calls must be passeed by value, not by reference.

Arjuna The Arjuna approach [193] focuses on distributed transaction support for objects. It provides persistent objects, but does not offer migrating objects.

Avalon/C++ Although internally quite different, the intention of the Avalon/C++ system [79] is quite similar to that of Arjuna.

DCE++ [190] This is an extension of both the OSF Distributed Computing Environment and C++ that provides a uniform object model, location invariant invocation, remote reference parameter passing and dynamic object migration.

Peace *Could be relevant.* This is an extension of C++.

COBRA [95] This is the result of the standardization effort of the Object Management Group (OMG). COBRA, the Common Object Request Broker, aims at providing a global distributed and persistent object management framework. COBRA enables remote object invocations and offers a C++ language binding. There is no object migration in COBRA.

RDO/C++ [114, 115] RDO/C++ provides the facilities needed to implement remote server processes with C++. The interface provided by the server is described in a high-level "Interface Description Language". RDO/C++ is based on the standardization effort of OMG.

Survey articles: [17] [22] [26] [212] [171] [63] [220] [221].
Notion of "actors" was described by Hewitt [104] and further developed by Agha [4, 5]
The paper by Karaorman and Bruno [130] elaborates on the design space of parallel object-oriented programming. The thesis of Papathomas [176] and an earlier paper [175] give an first classification of concurrent object-oriented languages. However, Papathomas focussed mainly on the way of combining concurrency with objects. He does not classify the broad number of languages, we look at in this report. Neither does he take more machine-oriented details into account, e.g., the way objects or processes are mapped to the underlying parallel hardware. Hence, he is not interested in migration and scheduling. His survey is slightly biased towards languages that couple concurrency to objects, instead of having the concept of threads be orthogonal to the notion of objects.

Encapsulation in sequential object-oriented programming languages protects the internal state of objects from arbitrary manipulation and ensures its consistency. If concurrent execution is introduced in a language independently of objects it will compromise encapsulation, since concurrent execution of the operations of objects may violate the consistency of their internal state.

Reusability

Due to Wegner [215] a language that provides objects is called object-based. When classes are in the language in addition to objects, such a language is called class-based. Only if inheritance is expressible as well, the language is called object-oriented.

2 Languages

2.1 ABCL/1

Developer: University of Tokyo, Japan

Description: Actor [4, 5] language.

oo. Inheritance by delegation. Objects are active when they process an incoming message. They are waiting if they explicitly issue an receipt statement and the message did not arrive. Objects are dormant otherwise.

memory model. Each object has its own local memory that cannot be accessed from outside (data abstraction). Objects can invoke member functions of other objects if they know their name. ABCL/1 implements a system wide uniform object space. There is no inheritance in ABCL/1 but method calls can be delegated by parameterizing the return address.

parallelism. By declaration of an object potential parallelism is provided. ABCL/1 has

three types of message invocation, namely, synchronous, asynchronous and asynchronous with futures. A thread blocks when it tries to access a future value, that is not available. Furthermore, ABCL/1 offers a parallel block that is similar to `cobegin` in which several methods are invoked concurrently. First class futures. The caller can decide whether a method should be called asynchronously or synchronously, independent of a return value. There is no way in the language to send messages to multiple receivers, i.e., parallel execution can only be started sequentially.

scheduling. Messages can have one of two priorities. Express messages interrupt processing of ordinary messages. The programmer has a choice whether execution of the interrupted method is resumed or not. In contrast to the original Actor model, objects can explicitly wait for the arrival of certain messages.

mapping. The programmer cannot influence the mapping of objects and threads to processor nodes. Objects migrate transparently to the node of the thread that executes a method.

synchronization. One message may be active on an object at a time. (In ABCL/M there is the notion of objects without state. These objects allow multiple threads to execute methods concurrently.) Messages are accepted if they confirm to patterns given in the object specification. If an incoming message does not fit to any pattern that message is discarded. An atomic section prevents a sequence of statements of being interrupted by an express message. After a `become` has been executed, the Actor is no longer allowed to change its state during the post-processing phase.

fault tolerance. Exceptions: a complaint object can be defined, i.e., the programmer can influence to where the exception must propagate.

Availability: The Yonezawa Lab WWW Server can be reached and the ABCL/1 software can be retrieved from:

`http://web.yl.is.s.u-tokyo.ac.jp`
`ftp://camille.is.s.u-tokyo.ac.jp`

Email addresses:
group address ⟶ `abcl@is.s.u-tokyo.ac.jp`
Akinori Yonezawa ⟶ `yonezawa@is.s.u-tokyo.ac.jp`

References: [224]

2.2 ABCL/f

Developer: University of Tokyo, Japan

Description:

oo. ABCL/f is based on the Actor model [4, 5] and is an extension of ABCL/1 (see section 2.1). Unlike its predecessor, ABCL/f is typed and is class-based, i.e., methods for classes can be defined. ABCL/f does not provide inheritance but inherits the delegation mechanism of ABCL/1.

memory model. In ABCL/f the programmer faces a purely object-based approach. Objects can transparently be addressed via their network-wide identifiers.

parallelism. See ABCL/1 (section 2.1).

scheduling. The programmer cannot influence the order in which messages are accepted by objects. The runtime and operating system schedules threads that are ready to execute.

mapping. By default functions and procedures are executed where the caller resides. Methods are executed on the node where the receiver object resides. The programmer can change this behavior by explicitly providing the node number where the execution has to take place.

APCL/onAP1000: When objects are created the programmer can choose between local and remote creation.

synchronization. See ABCL/1 (section 2.1). *Are express msg and interrupt masks still available?*

fault tolerance.

Availability: A prototype implementation of ABCL/f on a distributed memory multicomputer AP1000 has been nearly completed. The ABCL/f software is not yet available.

The Yonezawa Lab WWW Server can be reached at:
`http://web.yl.is.s.u-tokyo.ac.jp`

Email addresses:
group address ⟶ `abcl@is.s.u-tokyo.ac.jp`
Akinori Yonezawa ⟶ `yonezawa@is.s.u-tokyo.ac.jp`

References: [202]

2.3 ABCL/R2

Developer: University of Tokyo, Japan

Description: ABCL/R2 is a descendant of ABCL/R. Whereas ABCL/R was implemented on top to ABCL/1, ABCL/R2 is implemented directly in Common Lisp.

Actor [4, 5] language.

oo. As in ABCL/R, each object has its own meta-object. In addition, each object always belongs to some group. A group represents a shared resource. Since there is no explicit receipt statement, objects are either active or dormant.

memory model. See ABCL/1 (section 2.1).

parallelism. See ABCL/1 (section 2.1).

scheduling. There is no explicit receipt of messages and thus no waiting state of objects. Since the meta-object explicitly implements the behavior of the objects, the message queue is visible and reaction can be programmed differently.

mapping. See ABCL/1.

synchronization.

fault tolerance. See ABCL/1.

Availability: The Yonezawa Lab WWW Server can be reached under and the ABCL/R2 software can be retrieved from:
`http://web.yl.is.s.u-tokyo.ac.jp`
`ftp://camille.is.s.u-tokyo.ac.jp`

The predecessor ABCL/R is still available from ftp, however, it is neither supported nor recommended to use.

Email addresses:
group address $\longrightarrow$ `abcl@is.s.u-tokyo.ac.jp`
Akinori Yonezawa $\longrightarrow$ `yonezawa@is.s.u-tokyo.ac.jp`

References: [156] [224]

2.4 ABCL/R3

Developer: University of Tokyo, Japan

Description:

oo. ABCL/R3 is an extension of ABCL/R2. The main idea is to make per physical processor node objects and scheduler objects visible as meta objects, which are accessible from each object that resides on the same node.

memory model. Scheduler and node objects are shared for those objects that reside on the same node.

parallelism. See ABCL/1 (section 2.1).

scheduling. The default scheduler queues messages that address objects of the same node and invokes corresponding methods on a FIFO basis. The programmer can implement specific schedulers that behave differently and thus the user can influence the order of message acceptance. He might even implement conditional acceptance in the scheduler.

mapping. The node object is consulted when a new object is to be created. The node object either creates the new object locally or remotely depending on the average load. The programmer might implement different schemes.

synchronization.

Availability: A prototype implementation of ABCL/R3 is under construction and not yet available.

The Yonezawa Lab WWW Server can be reached at:
`http://web.yl.is.s.u-tokyo.ac.jp`

Email addresses:
group address $\longrightarrow$ `abcl@is.s.u-tokyo.ac.jp`
Akinori Yonezawa $\longrightarrow$ `yonezawa@is.s.u-tokyo.ac.jp`

References: [157]

2.5 Acore

Developer: MIT AI Lab.

Description:

oo.

memory model.

parallelism. Actor language. Asynchronous call of method without return value. Methods that have a return value can only be called synchronously.

scheduling.

mapping. Locality is not an issue.

synchronization. Default behavior is one-activity at a time. The programmer can specify methods to be unserialized methods, similar to ASK (see section 2.14).

fault tolerance.

Availability:

References: [154]

2.6 ACT++

Developer: Virginia Tech

Description:

oo. C++ library; based on the Actor model [4, 5]. Compared to the Actor model there are some extensions. The main extensions are the following: Whereas the pure Actor model requires that messages that arrive at an Actor are processed in FIFO order, one can implement so-called behavior sets in ACT++. These allow to accept specific messages from the queue and postpone others until certain conditions hold. The authors claim, that the notion of behavior sets remove inheritance anomaly. A second difference is the use of CBoxes for synchronization of access to return values. The newly introduced Actor classes are not meant to be inherited. C++ inheritance is useful for defining behaviors.

memory model. ACT++ needs a shared memory because of two reasons. One reason is that Actors, message objects and behavior objects need to be addressable. ACT++ uses the absolute memory address for this purpose. Furthermore, global variables can be passed by reference. However, reference parameters mean pointers in ordinary C++, which require a single shared memory.

parallelism. An Actor that processes an incoming message can asynchronously send an arbitrary number of messages, thus starting an arbitrary number of threads. ACT++ offers three types of first class futures: the future is either a queue or stores exactly one value, namely the one that is written first or last. Methods of other Actors cannot be called synchronously. Post-processing can be used to introduce concurrency by providing the next behavior before the current method is completed. Ordinary C++ objects are not actors. Invocation of their member functions is synchronous. The programmer has to care about problems that might stem from concurrent access to non-Actor objects.

scheduling. Scheduling of threads to processors is not visible in ACT++. This is left to the underlying PRESTO thread package and the operating system. However, the programmer can influence the way in which incoming messages are processed: he can alter the default FIFO behavior.

mapping. Mapping of Actors and threads to the underlying machine is not an issue, since a shared memory is required. Hence, migration is not considered.

synchronization. Behavior abstractions. The programmer can specify how many threads are active on an Actor object (in all its concurrently existing behaviors) at a time. If synchronization is necessary only one thread is allowed, hence there is no post-processing in this case. If post-processing is used, the Actor may not change its state after the `become` statement.

fault tolerance. None.

Availability: ACT++ version 3.0 has been implemented on a Sequent Symmetry multiprocessor with shared memory using the PRESTO thread package [29, 30], see section 2.82. A port for single Sun3 and Dec5000 is planned.

Some papers and the software on ACT++ can be accessed by anonymous ftp from
`ftp://actor.cs.vt.edu/pub`

Email address:
Dennis Kafura $\longrightarrow$ `kafura@cs.vt.edu`

References: [121] [122] [123] [124] [125]

2.7 Act1

Developer:

Description:

oo.

memory model.

parallelism. Actor language. Post processing (early become), delegation. Methods can only be called asynchronously. First class futures are used to handle return values.

scheduling.

mapping. Not an issue.

synchronization. One method at a time. After the `become` statement has been issued, the Actor can no longer change its state.

fault tolerance.

Availability:

References:

2.8 Actalk

Developer: University of Paris VI, France, and Rank Xerox France.

Description:

oo. Extension of Smalltalk-80 with Actors.

memory model.

parallelism. Asynchronous message passing for Actor objects only. If there is a return value to be passed back to the caller, an additional explicit message must be used. Postprocessing is possible by using an early become.

scheduling.

mapping.

synchronization. The Actalk kernel allows only one message to be processed at a time (and for a singe behavior). However, Actalk can be used to experiment with various forms of synchronization which are available as extensions of the basic kernel in the distribution. After a `become` the state can no longer be changed.

fault tolerance.

Availability: Version 3 of Acttalk is implemented in Smalltalk-80 4.1 and is available from:
`http://web.yl.is.s.u-tokyo.ac.jp/members/briot/actalk/actalk.html`
`ftp://camille.is.s.u-tokyo.ac.jp/pub/members/briot/actalk`
`ftp://ftp.ibp.fr/ibp/softs/litp/actalk`

Email address:
Jean-Pierre Briot $\longrightarrow$ `briot@is.s.u-tokyo.ac.jp`

References: [37]

2.9 ActorSpace

Developer: University of Illinois, Urbana-Champaign and Aalborg University, Denmark

Description:

oo.

parallelism. ActorSpace is based on the Actor model [4, 5] and extends this model by the ability to address groups of actors at once. In contrast to Concurrent Aggregates (see section 2.28), groups of actors can be addressed by using pattern instead of explicit actor mail addresses. When sending a message to a group depending on the call either an arbitrary member or all members of that group respond.

mapping. In the prototype, there is no notion of location or nodes. The runtime system distributes the newly created actors evenly (in a cyclic manner) on all participating nodes. Migration is not supported, but the authors claim that since the actors are interpreted, adding migration should not be that hard. The only practical problem is that of what to do with messages for the actor that arrives while it is in the middle of moving to another node.

synchronization.

Availability: Papers on ActorSpace are available via anonymous ftp from:
`ftp://biobio.cs.uiuc.edu/pub/papers`
`ftp://biobio.cs.uiuc.edu/pub/theses` Currently a prototype is operational, but it is not yet freely available, since the main focus is to prove the concept, not an efficient implementation.

Email addresses:
Christian J. Callseen —→ `chris@iesd.auc.dk`
Gul Agha —→ `agha@cs.uiuc.edu`

References: [3] [41]

2.10 Actra

Developer: Defense Research Establishment and Carleton University, Ottawa, Canada.

Description:

oo. Extension of Smalltalk based on Actors.

memory model. Based on shared memory machine.

parallelism. Post-processing. Synchronous message passing only. In contrast to the Actor model, there is no asynchronous message passing.

scheduling.

mapping. Not an issue.

synchronization. After a `become` the state of an Actor can no longer be changed. Synchronization by synchronous communication. Only one method can be active at a time.

fault tolerance.

Availability:

References: [158] [205]

2.11 Amber

Developer: University of Washington, Seattle

Description:

oo. Subset of C++ with primitives to manage concurrency and distribution. In Amber there is a collection of mobile objects distributed among nodes in a homogeneous network. These objects interact through location independent invocation. Amber is derived from Emerald (see page 21).

memory model. The language provides a network wide shared virtual memory. References to objects can be passed as parameters and remain valid on remote nodes.

parallelism. Amber offers thread objects that operate on passive objects. Threads are first class objects that can be created dynamically. When "start" is called on a thread object a specified operation on an object is called.

mapping. Object placement is under the control of the programmer. Threads move to objects they are working on. Amber provides constructs for explicit migration and attachment of objects.

scheduling. There is a predefined scheduling policy for method invocations. The programmer can replace this by programming his favorite strategy.

synchronization. Methods are invoked synchronously. Amber offers locks, barriers, monitor objects and condition variables for synchronization. Depending on the class an object is derived from, there may be one or more threads active on an object. Synchronization of object access is thus left to the programmer. For synchronization of concurrent threads there is a `join` construct.

fault tolerance.

Availability: Amber is implemented on the Topaz operating system for the DEC FireFly [203], a multiprocessor workstation based on VAX microprocessors.

References: [61]

2.12 A-NETL

Developer: Utsunomiya University, Japan.

Description:

oo. The language differentiates between dynamic objects and indexed objects, where multiple instances are created at once. No inheritance.

memory model.

parallelism. Asynchronous method call, futures, post-processing. Synchronous method call. The caller decides about the mode to be used for the call. It is possible to kill objects (and assigned threads (e.g. post-processing)). A-NETL offers a multicast to reach several objects at once.

scheduling.

mapping. The programmer can express the relationship between objects and can express the weight of communication between objects. Moreover, he can collect objects that should reside in one node for purposes of locality. There is an allocator tool which helps in mapping to the machine.

synchronization.

fault tolerance.

Availability:

References: [81] [226]

2.13 Arche

Developer: IRISA, Institut de Recherche en Informatique et Systems Aleatoires, Rennes, France

Description:

oo. Single inheritance.

memory model.

parallelism. Methods are called synchronously, i.e., the caller is blocked until the called method is completed. To initiate concurrency, each object has a life routine, that handles incoming calls.

Arche offers a way to specify aggregate operations. Objects can be dynamically grouped into sets. The sets then can offer the methods of individual objects in their interface. By invoking such a method of the set, the method is called for all member objects of the set. In the method code, the key words `we` and `this` can be used to distinguish between the set and the object respectively.

scheduling.

mapping. Nothing published.

synchronization. Arche distinguishes between observer routines and modifier routines. The programmer is in charge of correctly labeling the routines. The default are modifier routines. Based on this classification, objects implement a reader-writer consistency protocol: there may be several concurrent invocations of observer routines, but only one invocation of a modifier routines is possible at a time.

Above that, Arche offers enabled sets. The programmer can declare states of objects, i.e., sets of available methods. In addition with sets, there is a way to specify the potential effect of transitions. Based on this effect, the compiler can make sure, that the implementation of the methods correctly sets the new synchronization state with the `become` statement.

fault tolerance.

Availability: A compiler for the Arche language has been implemented in the framework of the INRIA/Bull project Gothic at the research institute IRISA (Rennes, France). The compiler generates C code.

Email addresses:
Marc Benveniste $\longrightarrow$ `mbenveni@irisa.fr`
Valérie Issarny $\longrightarrow$ `issarny@irisa.fr`

References: [28]

2.14 ASK

Developer: University of Salerno, Italy.

Description:

oo. Based on Actor model.

memory model. Distributed memory machine (Transputers).

parallelism. Based on Actor model. At the point of the method declaration, the programmer can distinguish between serialized and unserialized methods. Only methods without return value can be called asynchronously, therefore there is no need for futures. Post-processing is available by early `become`. Afterwards the state can no longer be changed.

scheduling.

mapping. Nothing is said about placement, locality, alignment, etc.

synchronization. Serialized methods allow only one method invocation to be active at a time, until an explicit `become` statement has set the new state of the actor. (Un-)serialized methods silently assume that the following state of the actor is the same as the current state, therefore, concurrent method invocations are allowed.

fault tolerance.

Availability: ASK is running on a single Transputer, it is unclear whether the implementation of a 16 Transputer version has been completed.

Email address:
Guilia Iannello —→ `iannello@udsab.dia.unisa.it`

References: [186]

2.15 A'UM

Developer: Institute for New Generation Computer Technology, Tokyo, Japan

Description:

oo. Object with stream of incoming messages. Multiple inheritance.

memory model.

parallelism. Asynchronous message passing only. If results have to be passed back to the caller of a method, then an additional message must be sent explicitly.

scheduling. Order of messages in streams is visible part of the language (in contrast to pure Actor languages).

mapping. No details on placement, alignment, etc.

synchronization. One at at time.

fault tolerance.

Availability:

References: [225]

2.16 BETA

Developer: Aarhus University and Mjølner Informatics, Aarhus, Denmark.

Description:

Only the Mjølner BETA system currently deals with concurrency.

oo. single inheritance.

memory model.

parallelism. An object can either be used in coroutine mode by using a "resume" statement. By declaring a autonomous routine, an active object executes its own actions which are defined in an associated action part. However, the life routine is not automatically started immediately after the instantiation of the object, but can be started explicitly by use of a "fork" commands.

scheduling.

mapping. Mjølner BETA can be used in a heterogeneous environment. There exists the abstraction of a nameserver for transparent access to remote objects via automatically generated proxies. Arguments and return values are transparently marshaled and unmarshaled. Moreover, the concept of a shell, i.e. a bunch of objects that are located in one address space can be used transparently. Objects can be moved between shells, shells can be moved between physical nodes.

synchronization. The basic synchronization mechanisms are semaphores. It is possible to declare abstract high-level concurrency abstractions, which can later on be used as mixins to apply these concepts to classes.

fault tolerance. Exceptions.

Availability: The beta home page can be found on `http://www.daimi.aaudk/ beta` A wealth of information can be found in the newsgroup: `news:comp.lang.beta` A commercially available BETA system can be found under `http://www.mjolner.dk`

Email address:
information —→ `info@mjolner.dk`

References: [36] [151] [150]

2.17 Blaze 2

Developer: Purdue University, West Lafayette and ICASE, NASA Langeley Research Center.

Description:

oo. This is an object oriented extension of BLAZE, a parallel language for scientific programming.

memory model.

parallelism. The base language BLAZE contains array arithmetic, forall loops, and APL-style accumulation operators, which allow natural expression of fine grained parallelism.

Multiple concurrent threads within an object. The default behavior is that methods are serial, which results in an exclusive access to the object. They can however be declared to be parallel.

scheduling.

mapping. Nothing.

synchronization. In addition to serial methods, Variables and objects can be locked/unlocked explicitly.

fault tolerance.

Availability: The project has not really been completed.

Email address:
Piyush Mehrotra ⟶ pm@icase.edu

References: [162] [163]

2.18 Braid, Data-Parallel Mentat

Developer: University of Virginia

Description:

Although this language first appears under the name DataParallel Mentat, that authors decided to change the name to Braid.

oo. Based on Mentat, see section 2.66. The basic extension is a new keyword for creating a data-parallel class. The class description specifies the elements of the data-parallel type and the functions that work on individual elements. The data-parallel objects are one- or two-dimensional grids. Member functions are categorized by keywords depending on their purpose. There are basic element functions, reductions and overlay functions. The latter are used to spread a one- or two-dimensional ordinary array across the data-parallel grid. Member function can be invoked on all elements, on a row, or on a column.

memory model.

parallelism. Automatic virtualization of element member functions.

mapping. When creating data-parallel objects the programmer has to decide how many processors to use. He has to define the dimensions and their size. Moreover, the programmer has to provide the dominant communication patterns, i.e., what kind of neighboring elements are accessed and what is their maximal distance, and what other objects is most often used together. Depending on the pattern used to create a data-parallel type, so-called border-functions give access to neighboring elements.

synchronization. Before virtualization, element functions create local copies of the values. When finalizing the element function, local values are stored.

Availability: Email addresses:
Andrew S. Grimshaw ⟶ grimshaw@virginia.edu
group ⟶ mentat@virginia.edu

References: [216] [217]. For more references see description of Mentat in section 2.66.

2.19 C**

Developer: University of Wisconsin, Madison

Description:

oo. C** is a data-parallel language.

memory model. The memory model of C++ is transparently extended for C**. An interesting aspect of data-parallel data structures is, that they can be used as ordinary array. Access to elements are equivalent to array accesses. A change of the declaration (replacing an array with a data-parallel type or vice versa) does not result in major code rewriting.

parallelism. Member functions of data-parallel types are invoked on all elements of that type at once.

scheduling.

mapping. The language does not provide any support for mapping the array shaped data structures onto the parallel machine. Data locality is not an issue in C**. There is no way to express alignment between different arrays, or elements thereof. The compiler is in charge to invoke functions where the corresponding elements reside.

synchronization. The functions that work on all elements of a data-parallel data set in parallel have an implicit synchronization barrier at their end. All instances are executed asynchronously with respect to each other. To ensure a deterministic behavior, each instance works atomically without affecting other instances. Each instance of the parallel function works on a local copy of all visible variables. Conceptually, after termination of all instances these variables are copied back. Hence, effects produced by an instance of the function are not visible to other instances.

fault tolerance. None.

Availability: The current compiler produces code for a sequential DEC workstation and for a Sequent Symmetry shared memory computer. C** is not yet available, but a release is expected soon.

Email address:
James R. Larus ⟶ larus@cs.wisc.edu

References: [137] [138] [139]

2.20 Cantor

Developer:

Description:

oo.

memory model. single processor shared memory.

parallelism. Actor language. Asynchronous message passing only. If a result has to be passed back to the caller of a method, an explicit message has to be sent. Post-processing (early become).

scheduling.

mapping. Assignment of objects to nodes and the routine of messages between nodes is jointly handled by the compiler and runtime system.

synchronization. Actor language.

fault tolerance.

Availability: Implemented in the Reactive Kerner of the Ametek 2010 series multicomputer.

References: [19]

2.21 CEiffel

Developer: Institut für Informatik, Freie Universität Berlin, Germany

Description:

oo. Concurrency is expressed by means of annotations. Programs have two different semantics: a sequential one, if annotations are ignored, and a concurrent one, if annotations are obeyed. The author states that this annotation approach will both enhance code reusability and ease the problem of inheritance anomaly.

memory model. Shared address space. Distribution is not reflected in the language. There is no notion of a process.

parallelism. The generation of concurrency is bound to routines. There are two types of routines for concurrency. One type is the *autonomous routine*. When an object has been created and initialized, all its autonomous routines are invoked implicitly, i.e., without explicit invocation by some activity. Routines can be labeled to be *asynchronous*. Asynchronous routines are executed concurrently to the caller. Other routine calls are meant to be synchronous. CEiffel uses wait by necessity to wait on outstanding return values of asynchronous routines.

scheduling. None.

mapping. Since the distributed implementation is just starting, no work has been done on this problem.

synchronization. Synchronization issues are integrated into program code. The default semantics ensure that only one member function is active on an object at a time. This hard synchronization pattern can be loosened by the programmer. In form of annotation the programmer can provide compatibility information. Either the object is concurrently accessible in general, or certain member functions are with respect to each other. Two member functions are considered to be compatible if concurrent execution do not interfere. The author claims, that since compatibility is a symmetric relation that need not be transitive or reflexive, inheritance anomaly often is reduced.

Furthermore, CEiffel provides mechanisms to control concurrent execution of member functions even finer. Guards and delays allow to express that function invocation is only allowed if certain conditions hold. If these do not hold, the call can be delayed until the condition becomes true.

Delays even for post-conditions. Guard results in exception, delay condition results in delay. Compatibility conditions are specific to the implementation. Delay conditions are part of the specification and are independent of a particular implementation.

fault tolerance. None.

Availability: Since CEiffel is compiled to Eiffel there is some portability. On top of PVM, a distributed implementation, targeting a network of Sun workstations, is under way.

Email address:
Klaus-Peter Löhr $\longrightarrow$ `lohr@inf.fu-berlin.de`

References: [146] [147]

2.22 CFM

Developer: Keio University, Yokohama, Japan.

Description:

oo.

memory model.

parallelism.

scheduling.

mapping. Object allocation algorithm. Dynamic object grouping. Adaptive load balancing. Distance cost model and mass cost. Migration of objects. Message objects follow.

synchronization.

fault tolerance.

Availability:

References: [213]

2.23 CHARM++

Developer: University of Illinois, Urbana-Champaign

Description:

oo. Extension of C++. Three types of objects: sequential, concurrent (so-called: chares), and replicated (one per processor). Inheritance is only possible within one type of classes. The language offers the notion of modules.

memory model. The programmer can define message types which are similar to C++ structs. Instead of functions, chares have entry points for specific types of messages. Entry points define code which is executed in response to message arrival.

Since Charm++ disallows unrestricted global variables and static variables in classes, every object has its own address space. Hierarchical name space. However, shared data objects are provided that implement various forms of access regulations (reader-writer, etc.)

By using replicated objects, an object can access the value stored at the same processor. Changes to attributes of replicated objects at processor have to be propagated to different processors manually.

parallelism. The execution model is message driven only. There are no explicit send-receive pairs. After asynchronously sending a message to a chare, both the sender and the recipient work in parallel. Return values, if any, have to be sent in separate messages. Results are in the standard message queue of the client. Each chare can only process one message at a time, i.e., there is no parallelism inside of objects. One can send a message to all instances of a branched objects, thus starting parallel activity.

scheduling. Member functions are executed on that processors that stores the object they belong to. Scheduling of all active functions on one processor is beyond the scope of the language and done by the underlying system.

1st class messages. Priority Queue.

The default regime for processing incoming messages is FIFO. Other policies are predefined or can be implemented by the programmer.

mapping. Objects and parallelism are considered together. The mapping of a concurrent object decides on which processor a message directed to that object will be processed. The programmer can select from a number of load balancing and memory mapping strategies. The strategies decide on which processor new chares are created. There is no migration. There are function calls for finding out the actual processor number and the total number of available processors.

Neighboring branch offices can be addressed by indexing. The index us the number of the PE node. Pointers are not meaningful across PE boundaries. The Charm++-system offers routines to convert pointers for the transport.

synchronization. Synchronization between objects must be implemented by means of explicit messages. Since only one thread can be active on an object (monitor), access to the object's data is under mutual exclusion.

Shared data objects allow concurrent access to global data, and implement concurrency control mechanisms.

fault tolerance. None.

Availability: The runtime system (Charm) runs on Intel's iPSC/860, iPSC/2, nCUBE, Encore Multimax, Sequent Symmetry, Alliant FX/8, single-processor UNIX machines, and networks of workstations. It is being ported to the CM-5, Parsytec GCel and Alliant FX/280 and T3D.

Information about Charm++ is available from:
`http://charm.cs.uiuc.edu`
`ftp://a.cs.uiuc.edu/pub/CK`

Email address:
Laxmikant V. Kale $\longrightarrow$ `kale@cs.uiuc.edu`
Sanjeev Krishnan $\longrightarrow$ `sanjeev@cs.uiuc.edu`

References: [128]

2.24 CLIX

Developer: Korea Advanced Institute of Science and Technology

Description:

oo. inheritance by delegation

memory model. system wide unique object-id

parallelism. Communicating process model. asynchronous method call (default), specified at callee (send command). synchronous call is available (ask command). explicit reply-to statement. Post-processing is possible.

scheduling. Nothing.

mapping. Nothing.

synchronization. One activity at a time. Select statement. condition attached to method declaration, delay

fault tolerance.

Availability:

References: [110]

2.25 COB

Developer: IBM Research Tokyo.

Description:

oo. C based. multiple inheritance. separation of interface and implementation. process and class are not compatible for inheritance.

synchronization. Differentiate between active and passive processes. Active processes: communication with Ada-like rendezvous, select statement.

Passive processes: like Monitor, one-thread-at-a-time. Shared data must be implemented in a passive process, to achieve serialization.

memory model. shared. hierarchical. processes visible: communication between processes.

parallelism. objects encapsulated into processes. Processes execute concurrently. When a process object is created its init routine is executed (life routine).

scheduling. Nothing is published about scheduling.

mapping. Nothing is published about object and process placement, alignment, scheduling.

fault tolerance.

Availability: Concurrent COB has been implemented on a PS/2 system and shared-memory multiprocessor workstation called TOP-1, which has been developed at the same laboratory.

References: [107]

2.26 Compositional C++, CC++

Developer: California Institute of Technology, Pasadena

Description:

oo. Extension of C++. Six new keywords.

memory model. C++ memory model for each processor object (definition see below). Multiple processor objects have a common name space of ids of processor objects. Only those members of processor objects that are explicitly declared public can be accessed from other processor objects.

parallelism. The language offers multiple levels of parallelism. First of all, there are processor objects. A processor object is a container for running threads. One of these processor objects has a routine called main for which an initial thread is started. Other processor objects become active due to invocation of their member functions. Inside of a processor object there are three light types of parallelism: (1) several functions can be called in parallel, similar to a classical `cobegin`-block as introduced by Dijkstra in 1986 [80]. (2) In a `parfor` a concurrent thread is started for each iteration of the `for`-loop. These threads synchronize at the end of the `parfor`. Finally, (c) there is a way to `spawn` new threads, which do not impose a parent-child relation. A spawned thread cannot return a result.

scheduling. Done by thread package and operating system.

mapping. Map thread to processor object and this to processor. When creating a new logical process object, the programmer can optionally specify a placement argument. This directs the low level mapping of logical resources (processor objects) to physical resources (processors). Processor objects cannot migrate. However, one can often achieve the effect of migration by overloading the `->` operator and moving computation by creating new threads. (Note, processor objects contain no computation, they are only containers that can hold threads). Differentiate between global and local pointer. A thread runs in its processor object, RPC can be used to invoke threads in other processor objects.

synchronization. For synchronization purposes the following two mechanisms are provided. Functions can be declared `atomic` to ensure, that only one thread at a time executes them. For synchronization of multiple threads there is a special `sync` object. After declaration that object is undefined. Read accesses to it will block until exactly one write access to it has taken place. The `sync` objects are similar to futures available in other languages. It is typed and can be used to transfer return value. However, `sync` objects are no queues, i.e., only one single value can be assigned.

fault tolerance. None.

Availability: Since CC++ programs are compiled to C++ it runs on many machines. They use a self-made thread package that is based on quick-threads from the University of Washington. The authors claim that this package can easily be ported to new machines.

Available from:
`ftp://csvax.cs.caltech.edu/comp/CC++`

Email addresses:
K. Mani Chandy —→ mani@vlsi.caltech.edu
Carl Kesselman —→ carl@vlsi.caltech.edu

References: [45] [58] [59] [82]

2.27 Concurrency Class for Eiffel

Developer: University of California, Santa Barbara

Description:

oo. This is an Eiffel library. The base language is not altered.

memory model. Each active object has its own address space. Access to active objects is through local proxies, that represent the remote object in the address space of the creator. If different active objects intend to invoke methods of such an active object, they have to explicitly attach to this object first. Attachment creates a local proxy in the address space of the caller. Object references cannot be passed as arguments.

parallelism. Active objects are defined by creating classes that inherit from a process class. When creating objects of this class, they become active, i.e., a process is created. A life routine (which is called scheduling method) is started separately. Remote methods are called asynchronously. The corresponding messages are accepted explicitly by the invoked object. The object has a queue of incoming messages. The scheduling method works on messaged in a FIFO order by executing the methods that corresponds to them. Return values are enqueued in a queue at the local proxy in the address space of the calling thread.

scheduling. Active objects run the scheduling method. The default scheduling method is FIFO. This can be changed by the programmer by redefining the scheduling method.

mapping. By default, mapping of objects to processors is done by the system. The predefined concurrency class defines a split routine, that creates a remote object when a local proxy is created. By redefining this method, the programmer can implement his own mapping regime. Since processes are strongly connected to active objects, mapping of objects implies the same mapping of processes. Since access to remote is via local proxies, object migration in general is possible. However, there is no migration.

synchronization. Since there is one scheduling method per active object, only one message can be processed at a time.

Synchronization is based on futures. A call to a remote method is processed asynchronously. A calling thread blocks when trying to use the return value of a invoked remote routine that has not yet terminated.

Since every ordinary Eiffel object is effectively owned by a single active object, no synchronization hazards are possible.

fault tolerance. None.

Availability: The library is implemented using version 2.3 Eiffel running on Sun's Unix based Sun OS 3.0. However, the software is not maintained anymore.

Email addresses:
Murat Karaorman —→ murat@cs.ucsb.edu
John Bruno —→ bruno@cs.ucsb.edu

References: [130] [131]

2.28 Concurrent Aggregates, CA

Developer: University of Illinois, Urbana-Champaign. Concurrent Aggregates is part of the Concert project.

Description:

oo. Extension of the Actor model [4, 5] for massively parallel programming. The main extension is that actors can be grouped together to aggregates, and can then be addressed with messages at once. CA has single inheritance. Invocation of member functions can be forwarded by delegation and continuations. There are different interfaces for different types of propagation.

memory model. Shared name space. Each object has a unique id and its own address space. The state of an object can only be accessed via message invocations on objects. Furthermore, there are global variables, that can be used by all objects. Within aggregates, there is an additional id for each object for easy addressing of "neighboring" objects. The authors claim to have a "full blown parallel garbage collection".

parallelism. Forall in aggregate. 1st class messages. In CA method invocation is asynchronously. Therefore, thread creation is dynamic. Member functions can be invoked on all actors that belong to an aggregate at once. Methods can be invoked synchronously as well. Only with synchronous invocation, the programmer can ensure the correct synchronization before accessing returned values. Synchronous and asynchronous calls can be passed on to other methods that eventually return values. The programmer can decide whether a sequence of statements is executed sequentially, or whether they

are executed concurrently. The latter is similar to the classical `cobegin` block as introduced by Djikstra [80]. Post-processing is available.

scheduling. Scheduling is done by compiler and runtime system. Messages that arrive at an actor are processed in FIFO manner.

mapping. The mapping of objects to processors is done by the compiler and runtime system. The programmer can give hints concerning the relative locality of objects. The authors are adding support for collection placement. Threads migrate to the objects they are working on. Objects do not migrate.

synchronization. An object provides a set of abstract operations (methods), of which only one may be active at a time. The programmer can state explicitly that more than one method can be processed at a time (methods are declared to be unserialized). When using this concurrency inside of objects the programmer is in charge of dealing with all arising synchronization issues.

Global variables can only be used under a blocking reader-writer-locking.

fault tolerance. None.

Availability: The Concert system has been operational on both sequential Suns (simulated parallelism) and a CM-5 since October 1992. The next target might be the T3D.

Information, the language report, and the current release of the Concert software can be found at:
`http://www-csag.cs.uiuc.edu`
`ftp://cs.uiuc.edu/pub/csag`

Email addresses:
group $\longrightarrow$ `concert@red-herring@cs.uiuc.edu`
Andrew A. Chien $\longrightarrow$ `achien@cs.uiuc.edu`

References: [65] [66] [67] [68] [69] [70] [71] [129] [177]

2.29 ConcurrentSmalltalk

Developer:

Description:

oo.

memory model.

parallelism. Asynchronous method call plus futures (CBox). Post-processing. Synchronous messages are also available. The caller decides which mode to use.

scheduling.

mapping.

synchronization. There are two types of objects in ConcurrentSmalltalk: atomic and non-atomic objects. Atomic objects allow only one of its methods to be executed at a time. In addition, Smalltalk's semaphores can be used for activity centered coordination.

The reason for the two types of objects is the intended compatibility with Smalltalk-80 which offers objects that behave like non-atomic objects.

fault tolerance.

Availability:

References: [223]

2.30 cooC

Developer: Toshiba Corporation, Kanagawa, Japan.

Description:

oo. Extension of C (or C++).

memory model.

parallelism. Every method call is asynchronous and uses an implicit wait by necessity when return values are used. The size of the message queues is a runtime system parameter. This parameter decides how many method calls can be delayed at an object. If this queue is full, additional invocations are executed immediately. Hence, if the size is set to zero, then all invocations are concurrent.

scheduling. The programmer can influence the scheduling policy by explicitly assigning predefined policies to the run-time system handler of an object or by even reprogramming such strategies.

mapping.

synchronization. Exclusive methods. Wait by necessity. The language offers semaphores and a rendezvous.

fault tolerance.

Availability: Is implemented on a network of Sparcs using the lwp thread packets and Unix sockets. A beta version of the software is available from:
`ftp://isl.rdc.toshiba.co.jp/pub/toshiba`

Email address:
group $\longrightarrow$ `cooc@isl.rdc.toshiba.co.jp`

References: [211]

2.31 COOL (Chorus)

Developer: Chorus Systems, France.

Description:

oo. The main work went into the CHORUS object-oriented layer. An object oriented language can be mapped to this layer by trapping object creation. Objects may be declared active.

memory model. The system provides a shared address space on parallel hardware with distributed memories. To achieve that, remote objects have proxies in each memory. When a member function of such an object is called, the proxy gets the call, send a message to the remotely stored object, waits for the result and returns the values to the original caller.

Proxies are generated automatically, they do the marshaling and unmarshaling of arguments and return values.

parallelism. The system assigns a thread to objects that are declared active. Threads can be started on several methods of an object at a time, similar to classical thread packages. Such a thread starts after object creation at defined entry points.

scheduling. Done by the runtime system.

mapping. Mapping has no semantic relevance. Objects are handled by the COOL generic run time. They are persistent and can migrate transparently. When an active object migrates, the thread assigned to this object is first stopped and later continued in the new context at the defined entry point. Multiple objects can be grouped together. This indicates that they should migrate together.

synchronization. There are two general types of synchronization available. The first one is useful for threads that run in the same physical address space. Here standard synchronization mechanisms, e.g. semaphores, mutual exclusion, and reader-write locks are offered. For distributed objects there is a token based synchronization system.

fault tolerance. Persistent objects.

Availability: The only implementation reported so far runs on Intel 80386 based machines runing a Chrous UNIX clone.

`ftp://ftp.chorus.fr/pub`

`news:comp.os.chorus`

Email addresses:
group ⟶ `info@chorus.com`
Rodger Lea ⟶ `rjl@hplb.hpl.hp.com`
Christian Jacquemot ⟶ `chris@chorus.fr`
Eric Pillevesse ⟶ `pillevesse@sept.fr`

References: [6] [140] [141]

2.32 COOL (NTT), ACOOL

Since the are some collisions in language name space, the authors consider to rename their language to **ACOOL**.

Developer: NTT communication switching laboratories, Japan

Description:

oo. The language offers active and passive objects which can be declared using single inheritance. Active objects have a thread. When a function is invoked by a message, that function can delegate the task to another object's member function. If necessary, the latter returns values directly to the original caller.

memory model. Address space per object. Function calls can only have value parameters.

parallelism. By creation of active objects. Active objects have an id; knowledge of this id enables network-transparent message passing. Member functions of other active objects can be called asynchronously, as long as no return values have to be transmitted. Calls to member functions that return a value block until that value is received. Functions of passive objects are always called synchronously. There is a special receive-operation that dequeues messages.

scheduling. Active objects are implemented using SUN-light-weight-processes. There are some built-in messages for thread scheduling, e.g., threads can suspend and resume.

mapping. In ACOOL, objects are created by SETUP-statements. A SETUP-statement creates a new object only on the "same" machine. To create objects in a remote machine, the remote machine must have a "manager" object which, receiving object-create-request-messages, creates a new object in that machine. A name server is used to get the object-ID of the "manager" object in remote hosts. Semantic location transparency. However, there are no reference parameters.

synchronization. Synchronization has to be implemented by means of message passing. There is no support in the language for synchronization of multiple threads. Member functions that return values can only be called synchronously For explicit synchronization, the programmer can use suspend and resume messages. There may be only one thread active on an object at a time.

fault tolerance. None.

Availability: This language has been developed by NTT communication switching laboratories. It is running on single SparcStations. The compiler is available via anonymous ftp from
`ftp://ftp.ntt.jp/pub/lang`

Email address:
Katsumi Maruyama ⟶ `maruyama@nttmfs.ntt.jp`

References: [155]

2.33 COOL (Stanford)

Developer: Stanford University

Description:

oo. Extension of C++.

memory model. The language is implemented on shared-memory machines. A common address space is the conceptual basis.

parallelism. Functions can be declared to be parallel. After invocation, a parallel function is executed by a newly created thread that runs concurrently to the calling thread. (A serial invocation of a function that is declared parallel is possible. However, not vice versa.) Functions return condition variables immediately, all other return values have to be passed as reference parameter. The caller can choose to wait on the condition variable until the function call is completed. Manual implementation of futures.

scheduling. The runtime system manages the creation of objects and scheduling of threads to increase locality (based on the data reference information) and balances the load.

mapping. The language provides features for two layers of abstraction: parallel programming can be expressed without taking the mapping of objects and threads into account. When a programmer uses this layer, the runtime system does the mapping. On a more performance oriented layer, the programmer can give hints to the runtime system. These hints do not affect the semantics of the original program. To improve locality, the programmer can express different types of affinity: affinity of a thread relative to an object and relative to a processor, affinity of an object relative to other objects. Objects can migrate, threads can not.

synchronization. COOL provides features for different types of synchronization. (a) Synchronization of attribute access. For this purpose, functions can be declared mutex/non-mutex to enforce a kind of reader-writer monitor like locking of the object the functions are called upon. This mechanism works both for sequential and parallel functions. The second kind (b) of synchronization is between threads. Threads can block on condition variables. Furthermore, there is (c) a barrier synchronization. This is offered in a block like `waitfor` statement: the thread that enters a `waitfor` waits at its end until all threads that are created inside of the block have terminated. The difference between `waitfor` and the classical `cobegin` is, that the statements in the `waitfor` block are processed sequentially. Only threads that are created in the block are affected, whereas in `cobegin` all statements are considered to be executed concurrently.

fault tolerance. None.

Availability: COOL is available for the following architectures: Stanford Dash, silicon Graphics 4D-380, and Encore Multimax. Sources and Documentation can be found on anonymous
`ftp://cool.stanford.edu`

Email address:
Rohit Chandra ⟶ `rohit@cool.stanford.edu`

References: [55] [56] [57]

2.34 Coral

Developer: IBM Palo Alto Scientific Center

Description:

oo. Multiple inheritance.

memory model.

parallelism. Asynchronous message passing.

scheduling.

mapping. Nothing is published about object/thread placement, alignment etc.

synchronization. The author states the existence of a synchronization mechanism, however does not give any details, because a patent application is pending.

fault tolerance.

Availability: A first experimental version seems to be running on top of sequential AIX.

References: [60]

2.35 CST, Concurrent Smalltalk (MIT)

Developer: MIT

Description:

oo. Based on Smalltalk-80. Multiple inheritance.

memory model. Global virtual address space machine.

parallelism. Asynchronous method calls, futures. distinction between objects and distributed objects.

scheduling.

mapping. Nothing is published about distribution, locality etc.

synchronization. Objects: one-activity-at-a-time. Distributed objects: several replicates that can be reached by a common name. This idea is quite similar to Concurrent Aggregates (see section 2.28). Message sent to a distributed object are received by exactly one instance of the group. The programmer must ensure the consistency between the replicates by hand if needed. There are special addressing mechanisms for communication inside of a group.

fault tolerance.

Availability: A simple programming environment has been implemented on a Symbolics 3600 system. A back-end is MIT's J-machine, a message driven parallel architecture.

Email addresses:
William Dally ⟶ `dally@ai.mit.edu`
Andrew Chien ⟶ `achien@cs.uiuc.edu`

References: [75] [106]

2.36 Demeter

Developer:

Description:

oo.

memory model.

parallelism. Thread library.

scheduling.

mapping.

synchronization. Synchronization patterns can be specified separate from the any classes. Then the class code is fitted in and the final code is generated. In contrast to the technique used in Dragoon (see section 2.44) the synchronization code is not abstract, i.e., the programmer has to fill in concrete function names.

fault tolerance.

Availability: Concurrency is not directly introduced into the Demeter Tool/C++ which is distributed by anonymous ftp. However the code might be available upon request.
`http://www.ccs.neu.edu/home/lieber/demeter.html`

Email addresses:
Karl Lieberherr ⟶ `lieber@ccs.neu.edu`
Cristina Lopes ⟶ `lopes@parc.xerox.com`

References: [148]

2.37 Distributed C++, DC++

Developer: University of Utah, Salt Lake City.

Description:

oo. This is an extension of C++. In addition to ordinary C++ classes, there are value classes and gateway classes. When passing value class objects as parameter, instead of the pointer the value of that class is deeply copied and transmitted. DC++ distributes objects over a distributed memory machine. To make them remotely accessible, objects of the gateway class must be used. These proxies translate method invocation either into RPCs or into local method invocations, depending on whether the object is remote or local. Pointer references to remotely stored objects are prohibited.

memory model. The memory model of C++ is extended to reflect the semantics of the new classes: Inside of a so-called abstract processor there is the classical C++ memory model with the extension of gateway objects, which provide a global name-space for accessing remotely stored objects (in different abstract processors.)

parallelism. Objects are passive, i.e., there is no thread that is assigned to them. In contrast, there is a special thread class, hence, threads are first class objects. By addressing a gateway object a thread can invoke a member function (argument of thread creation) of an object, which in turn could create additional thread objects. Once started, a thread may be chained or passed through multiple domains.

A domain can have multiple objects and enforces the one thread per domain at a time principle. This is even stronger than the one method at a time which restricts the concurrency to individual objects.

mapping. Classes can be grouped to domains, or so-called abstract processors. When creating a domain, the programmer explicitly specifies the

number of the real processor which is to be used to store that object. Since threads work on objects of abstract processors, the programmer indirectly maps threads to processors as well.

There is no semantic location transparency. After the creation of an object the system can react differently depending on whether there exists a "gateway" to this object or not (local object).

scheduling. Scheduling of threads that are located on one processor is beyond the scope of the language. Message invocation passes through gateway objects. The gateway object can delay invocation requests and thus implement a different scheduling.

synchronization. Since only one thread can be active on an abstract processor domain at a time, access to the state of objects in that domain is synchronized. For synchronizing access to results of member functions, DC++ provides a way to wait for the availability of the return value or the termination of a thread.

1st class coordination futures. Delay Queues.

Passing value class objects as parameters may result in anomalies, when concurrent updates are made. The programmer is in charge of preventing erratic behavior.

fault tolerance. None.

Availability: Distributed C++ is available by anonymous ftp from
`ftp://cs.utah.edu/pub/dc++`

The DC++ compiler is at the proof-of-concept stage rather than being a real compiler at this time. The runtime system is stable and usable on HP Series 9000 Model 3x0 or Model 7x0 running either BSD, HP-UX, or OSF/1. The author claims that it could be ported fairly easily to other UNIX machines.

Email address:
Harold Carr $\longrightarrow$ `carr@cs.utah.edu`

References: [52] [53]

2.38 Distributed Eiffel

Developer: College of Computing, Georgia Institute of Technology

Description:

oo. This is an extension of Eiffel.

memory model.

parallelism. The programmer can declare persistent objects which are implemented on top of the Clouds operating system. These large grain objects can comprise several Eiffel objects. Each large grain object has a separate memory, the objects communicate by means of message passing. The programmer can start threads per large grain objects.

Inside of these large grain objects the programmer can initiate additional parallelism by asynchronous method calls. The caller decides whether a method is to be called synchronously or asynchronously. To use the asynchronous call for methods with return parameters, Distributed Eiffel offers first class futures.

scheduling. Distributed implementation on top of Clouds operating system.

mapping. The programmer can optionally specify where an activity should be executed. For this purpose, the programmer can use virtual processor numbers that are automatically mapped onto the underlying machine by the run-time system.

synchronization. Distributed Eiffel is based on handshake control: i.e. there is code in the class implementation that coordinates concurrent access to an object. Methods can be labeled to be reader or writer routines. In addition, method guards can be added to each method. A method call is delayed, if the reader/writer protocol requires this or if the guarding condition is evaluated to false.

In addition, there are semaphores and locks to allow for the implementation of very fine grain concurrency control.

fault tolerance. Persistent objects.

Availability: Distributed Eiffel is translated to Eiffel augmented with calls to the Clouds operating system. It is unclear whether the system is still available.

References: [97]

2.39 Distributed Smalltalk – Object

Developer:

Description:

oo. See Smalltalk-80 (section 2.92).

memory model.

parallelism. See Smalltalk-80 (section 2.92).

scheduling.

mapping. proxies are used to transparently access objects that are stored on remote node.

At least five different Distributed Smalltalk versions have been developed that bear some similarities when it comes to mapping/locality of objects. They differ in the level at which the

proxies have been added to Smalltalk. Whereas Decouchant and the system of Schelvis and Bledoeg (Océ Netherland) extended the Smalltalk virtual machine, the other projects chose to add proxy and message objects at the virtual image level.

synchronization.

fault tolerance.

Availability:

References: [27] [77] [159] [168] [189]

2.40 Distributed Smalltalk – Process

Developer:

Description:

Extension of Goldberg and Robson's Smalltalk (see section 2.92).

oo. See Smalltalk-80 (section 2.92).

memory model.

parallelism. See Smalltalk-80 (section 2.92). The "fork" message can handle the node number.

scheduling.

mapping. When creating a process object, the programmer can specify the physical node to be used. There is a distinction between shared memory processes that run on the same node and distributed memory processes that run on different nodes. Process objects that run on the same node can communicate with each other through shared objects. Process objects that run on different nodes must use call by value message passing.

synchronization. Locks for shared objects. Method can be set "serialized" (dynamically). Guards per method, but specified separately.

fault tolerance.

Availability: The implementation is based on the Smalltalk/V 286 system and runs on a network of IBM PC.

References: [144]

2.41 DoPVM

Developer: Emory University, Atlanta

Description:

oo. Extension of C++. The general idea is to have a collection of otherwise independent C++ programs work on shared objects. The programmer creates the parallelism by writing the appropriate number of programs. Each program binds itself to the DoPVM system and can then access shared data objects by knowing object ids.

Concurrent invocation of member functions of shared data is not implemented.

memory model. Each program has its own address space. The shared objects are identified and addressed by integers. If a program chooses to use one of the shared objects he has to declare a local variable and bind the shared object to it. By using this local variable, the shared object is accessed.

parallelism. By user (several individual programs).

scheduling. The user starts programs on the machines he is going to use. The operating system schedules these programs on a per machine basis.

mapping. It is unspecified, where shared objects are located. The programmer cannot influence it.

synchronization. Shared objects can be locked. When a process tries to access a locked object the process if blocked. Modification to unblocked shared objects is atomic. Some shared objects can be aggregated together to be locked at once. This helps to avoid deadlock.

Member functions can only be invoked on unlocked shared data objects.

fault tolerance. None.

Availability: The system is in beta test. Some Documentation is available via anonymous ftp from
`ftp://mathcs.emory.edu/pub/vss`

Email addresses:
Contact V. S. Sunderam $\longrightarrow$ `vss@mathcs.emory.edu`
Charles Hartley $\longrightarrow$ `skip@mathcs.emory.edu`

References: [100]

2.42 DOWL, distributed Trellis/Owl

Developer: University of Karlsruhe, Germany

Description:

Extension of Trellis/Owl (see section 2.96).

oo. Extension of Trellis/Owl (see section 2.96) for distribution. Transparent operation invocation on remote objects.

memory model. Distributed address space. Automatically generated local proxies represent objects. Calls to operations on an object transparently are redirected by the proxies.

parallelism. See Trellis/Owl (section 2.96).

mapping. The DOWL language provides constructs to express location relationships between objects. The programmer can bind objects to specific nodes. Objects that are not bound to a specific node can migrate transparently. The programmer can explicitly set the target node of an object that is to migrate.

Objects can be attached to each other. Groups of objects migrate together. It can be specified, whether objects are attached to each other at all times, whether objects should move together, or whether objects should be moved together when visited.

scheduling.

synchronization. See Trellis/Owl (section 2.96).

fault tolerance.

Availability: DOWL is running on VAXen and DECstatations under Ultrix.

Email address:
Bruno Achauer
⟶ `bruno@tk.telematik.informatik.uni-karlsruhe.de`

References: [1] [2]

2.43 dpSather

Developer: CSIRO, Australia + Monash University, Australia

Description:

oo. The language dpSather is an extension of Sather 0.5 [167, 145]. The language has a bulk data type. When declaring objects of that type, a given number of instance is created. Functions can be declared to work on instances and can be called to work on the whole bulk.

memory model. Same as Sather.

parallelism. By invoking functions on all instances of a bulk in parallel.

scheduling. Scheduling is not an issue. Since bulk data is mapped by the programmer and function invocation always affects all instances, the execution model is simple: there is a loop per processor that iterates over the instances stored at that particular processor.

mapping. The programmer can specify how bulks are mapped to the processors. For this purpose virtual topologies can be used. It is possible to declare addressing functions that give access to neighboring instances when called from an instance of the bulk. Above that, the programmer can specify how bulks are to be aligned with respect to each other. The notation borrows heavily from C* [204], Fortran D [83], HPF [109] and thus inherits some of the weaknesses, c.f. [206].

synchronization. The functions that run on the instances of a bulk in parallel are not synchronized during execution. However, the parallel activities exist only from the call of the parallel function to its termination on all instances.

fault tolerance. None.

Availability: dpSather has been implemented on a MasPar MP-1. An implementation on a Fujitsu Sparc multiprocessor is under construction.

Email address:
Heinz Schmidt
⟶ `Heinz.Schmidt@fcit.monash.edu.au`

References: [191]

2.44 Dragoon

Developer: Imperial College, London, UK

Description:

oo. Based on Ada.

memory model. Heterogeneous systems with distributed memory.

parallelism. Objects can have a thread that executes concurrently with method invocations. Objects that have this thread are called active objects; other object are referred to as passive objects.

scheduling.

mapping. The programmer can map objects to physical nodes. Heterogeneous systems are supported. There are differences when remote objects are accessed: only basic types of the language are allowed to be used as arguments (to avoid deep copying of larger structures).

synchronization. Dragoon allows a separate specification of behavior classes. The programmer can define an abstract synchronization patter, e.g. monitor, rendezvous, reader-writer etc., or can re-use predefined patterns. These synchronization patterns are abstract because the conditions are expressed by means of variables which can reflect function names. The conditions can make use of counters like `active(x)` and `requested(x)`.

To apply such a behavior class for the synchronization of concurrent accesses there is a "ruled by" primitive. The programmer must then map method names to the abstract variables used in the behavior class.

fault tolerance.

Availability: The language was available from an industrial partner (TXT, Milan, Italy). However, as it did not prove a financial success, it has been dropped. Due to Colin Atkinson TXT might be willing to donate a copy to an academic institution. Contact Marco De Michele.

Email addresses:
Colin Atkinson —→ `atkinson@cl.uh.edu`
Marco De Michele —→ `demichel@txt.it`

References: [20] [21]

2.45 DROL

Developer: Keio University, Japan.

Description:

oo. Extension of C++.

memory model.

parallelism. Parallelism is introduced by asynchronous message passing. There is a blocking receive which can be used to implement rendezvous and synchronous message passing. Since the language focuses on real-time aspects the communication protocol can be specified, including timeouts. Post-processing.

scheduling.

mapping.

synchronization. Single thread per object, i.e. monitor. In addition to a base object the programmer can specify a meta object which handles the incoming invocations and the protocol machine. This meta object knows about "enabled sets" and can reject method invocations to methods which are not currently available.

fault tolerance. Exception handling by use of timeouts.

Availability: DROL is translated to C++ and thus runs on single processor machines.

References: [199]

2.46 Eiffel//

Developer: University of Nice, Sophia Antipolis, France

Description:

oo. Slight extension of Eiffel version 2. Concurrency is introduced into the language by inheritance from special concurrency classes: For this purpose there are the classes PROCESS and REQUEST.

memory model. Common address space. There are no shared passive objects in Eiffel//. Instead of passing references to objects as parameters, Eiffel// (deep) copies the objects and passes them as value parameters.

parallelism. A process is an instance of a class inheriting directly or indirectly from PROCESS. After creation and initialization, such a process object executes its *Live* routine. This routine processes incoming requets by invoking the corresponding object feature. Although the transfer of the request object between caller and called object is a synchronous handshake, after that transfer caller and called object proceed asynchronously.

scheduling. An incoming message, i.e., a REQUEST object, is handled by an interrupt. The recipient, who might be busy executing a member function demanded by a previous request, is interrupted to accepts the new request. The process is in charge of deciding what to do with the newly arrived request. The default policy is to store this request in a list of pending request and resume the execution of the original member function. After completely serving the original request, the process proceeds to work on the next request in FIFO order. Alternatives (e.g. immediate response to arriving requests or non-FIFO policies) can be implemented manually.

mapping. Nothing in the language. Process migration/locality property for the sake of optimization and load balancing will be considered in future.

synchronization. In Eiffel// there is at most one active process per object. This holds both for process objects that work on at most one request at a time and for passive objects, that can be accessed by at most one process object.

The synchronization principle is *wait-by-necessity* [47]. A process blocks when it attempts to use the result of a feature call that has not been returned yet. This mechanism is automatic and transparent, i.e., there is no need to explicitly declared future data type or CBoxes.

The principle of Eiffel// is to prohibit shared passive objects. If one needs a shared variable, one must make it a process.

In Eiffel// it is possible to inherit synchronization abstractions. Hence, reimplementation of live routines is not always necessary. However, methods must be 1st class.

fault tolerance. None.

Availability: The Eiffel// language is still under development. It runs on networks of Sun Sparc workstations. Each language process is mapped to a Unix process. A forthcoming implementation will target both network or cluster of workstations, and parallel machines, with the language processes being implemented with both operating system processes, and light weight processes.

Email address:
Denis Caromel $\longrightarrow$ `caromel@mimosa.unice.fr`

References: [46] [47] [48] [49] [50] [51]

2.47 Ellie

Developer: University of Copenhagen, Denmark

Description:

oo. In Ellie everything is an object, e.g., classes, types, blocks, and methods. Multiple inheritance.

memory model. All objects communicate solely through method invocation. There are no globally accessible shared variables.

parallelism. Method invocation is synchronous. Methods calls either result in a return value or in the return of a first class future value. In the first case, no parallelism is induced, since the caller waits for the return value. In the latter case, the caller and the called method both continue asynchronously. The difference is specified when the routine is called.

scheduling. Done by compiler and operating system. When two objects invoke methods of the same object, it is unspecified, which will be accepted. The object that issued the one that is not accepted blocks. This policy cannot be altered, except by prohibiting the invocation of certain methods, see the description of synchronization features.

mapping. There is no way for the programmer to influence the mapping of objects, and hence processes, to the underlying machine.

synchronization. Only one process can be active on an object at a time. Since method invocation is synchronous and requires a handshake, the called object must have completed the last method invocation before the next one can be accepted.

In Ellie a method can change the set of acceptable method invocations. Dynamic interface. The interface is 1st class, Ellie offers explicit include/exclude operations to modify the interface. Calls to excluded methods block until these methods are included into the dynamic interface again. This mechanisms has similar expressive power as path expressions introduced by Campbell and Habermann in [42].

fault tolerance.

Availability: Ellie is not available outside the Distributed Systems research group in Copenhagen. Platforms are MS-DOS and transputers (with INMOS C toolkit).

Some papers and reports are available via anonymous ftp from
`ftp://ftp.diku.dk/diku/dists/ellie/papers`

Email address:
Birger Andersen $\longrightarrow$ `andersen@informatik.uni-kl.de`

References: [11] [12]

2.48 Emerald

Developer: University of Washington, Seattle, $\Longrightarrow$ DIKU at University of Copenhagen, Denkmark.

Description:

oo. Emerald is an object-based without classes and inheritance.

memory model. The programmer has the impression of a shared address space. Local and remote objects can be referenced via their unique and network-wide identifiers. There is no difference except for performance. Addresses of objects are translated as soon as they cross node boundaries.

parallelism. In Emerald there are active and passive objects. Passive objects provide member functions that can be called by processes. Active objects are similar to passive objects except that they are created together with a process which is started upon object creation. There is no way to spawn parallelism except by creating active objects. The processes are not bound to the object they have been created with.

mapping. The language offers location-independent invocation and object migration. Mobility can be restricted by fixing objects to nodes. Object can be attached to each other to achieve locality. Although in general, parameter passing is by reference, the programmer can give application dependent hints to the system by specifying *by-move* and *by-visit*. The first suggests to move the referenced object to the thread. The latter moved the thread to the object.

scheduling. Done by runtime and operating system.

synchronization. There is no way to synchronize concurrently executing processes. However, Emerald offers a monitor construct that ensures that only one process at a time can work on an object. The programmer must detect places in his code, where monitors are needed.

fault tolerance. Emerald has persistent objects and a check-pointing system to deal with node failures.

Availability: Implementations are reported for VAX, HP 9000/300, Sun3, and SunSparc machines, running on top of UNIX 4.2 BSD and compatible versions.

Papers on Emerald and the software are available from anonymous ftp from
`ftp://ftp.diku.dk/pub/diku/dists/emerald` Although the main work on Emerald has been done in the mid to later 80s at the University of Washington, Seattle, it is still alive. Most of the recent work has been done at DIKU at the University of Copenhagen, Denmark. There (and in Cracow, Poland) Emerald is used in teaching of 250 undergraduate students.

Email address:
Eric Jul —→ `eric@diku.dk`

References: [111] [119] [120]

2.49 EPEE, Eiffel Parallel Execution Environment

Developer: IRISA, France

Description:

oo. EPEE is a extension of Eiffel for data-parallel programming. The approach is based on the idea of Concurrent Aggregates as described on page 12. EPEE provides distributed classes, elements of that class are spread across the machines. The extension seems to work only for matrix data types.

memory model. Elements of the aggregate can be accessed both by their global index and their local index.

parallelism. Rather than defining element functions, the approach is library based, i.e., the library programmer has to implement the algorithm on a per processor basis. EPEE does not provide automatic virtualization. By using the higher level operations on the whole aggregate that the library programmer offers at the class interface, the operation can be invoked by an application programmer.

scheduling. Done manually by the library programmer by explicitly programming the per processor code in SPMD fashion.

mapping. The programmer can influence the distribution of data elements and the relative alignment of two aggregates by means of constructs that have a similar expressive power than HPF provides for the same purpose. In addition, the programmer can add libraries to implement the desired data distribution.

synchronization. Due to the SPMD approach all threads synchronize after a parallel step is done.

fault tolerance. None.

Availability: An EPEE prototype is running on Intel iPSC/2 and iPSC/860 and for a network of workstations above TCP/IP. However, the software is not (yet) available.

Email address:
Jean-Marc Jézéquel —→ `jezequel@irisa.fr`

References: [99] [116] [117] [118]

2.50 ES-Kit Software

Developer: Microelectronics and Computer Technology Corporation (MCC)

Description:

oo. Extension of C++. New construct "wrapper". Inheritance, but not multiple inheritance.

memory model. The focus of ES-Kit is to make a remote procedure call transparently available in the context of C++. In addition to a ordinary invocation of a method, the "wrapper" is a second way to call a member function. This second way allows the programmer to catch the call and implement additional functionality that might be used for remote procedure calls. In particular, a new thread could be started, arguments can be marshaled etc.

parallelism. Asynchronous method invocation, futures.

scheduling.

mapping. Nothing is said about location.

synchronization. Synchronization must be implemented by hand, e.g., by locking.

fault tolerance.

Availability: The ES-Kit research project was driven by the development of a parallel computer. It has been completed in March 1991. The software is not available online.
`http://www.mcc.com`

References: [194] [207]

2.51 ESP – Extensible Software Platform

Developer: Microelectronics and Computer Technology Corp., MCC

Description:

oo. Distributed C++ system. Object are considered to be passive but reactive, they are waiting for requests to execute routines.

memory model. ESP provides named objects with private address space. Public variables can only be accessed through member functions.

parallelism. Invocation of methods can be asynchronously (only if there is no return value) or synchronously (with/without return value) parallel. Method calls, that do not expect return values never block. For asynchronous calls with return values, special first class futures can be used. There is no way to aggregate objects. Methods cannot be started on multiple objects at once.

scheduling. Processes are scheduled automatically by compiler and runtime system. When more than one process invoke methods of a single object concurrently, it is unspecified, which one will be processed first. The other request is queued and processed in a FIFO order. The programmer cannot influence this scheduling policy, nor can he explicitly delay request because of manually programmed conditions.

mapping. Automatic by compiler and runtime system. However, the programmer can explicitly specify that a new object should be created on a remote processor. In the released version there is no object migration. Support for heterogeneous systems.

synchronization. Only one method can be executed at an object at a time. Futures can be declared.

Message for all objects are queued by the system. As the system detects that a message of an object has finished execution, the next message is dequeued and sent to the appropriate method.

fault tolerance.

Availability: ESP is available for a network of Sun SPARC stations and for the Motorola Pleiades multicomputer.

Email addresses:
David Croley —→ croley@mcc.com
Arun Chatterjee —→ arun@mcc.com

References: [62]

2.52 Fleng++

Developer: University of Tokyo, Japan

Description:

oo. Multiple Inheritance. Based on logic. Classes have methods which are used in an imperative way.

memory model.

parallelism. And/Or-parallelism

scheduling.

mapping. The location of objects is visible (each PE is visible).

synchronization. Only a single method can be processed at an object at a time.

fault tolerance.

Availability: Fleng++ is implemented on a PIE64 parallel computer which is built at the same laboratory.

References: [200]

2.53 Fragmented Objects, FOG/C++

Developer: INRIA, Institut National de Recherche en Informatique et en Automatique, France

Description:

oo. In addition to ordinary C++ objects, the authors introduce fragmented objects. They call them fragments, since on each node a fragment of the object exists. Two different fragments are most common: one fragment is called the provider. This is essentially an active object, that is waiting for messages requesting the execution of member functions. The other fragments are proxies in the local address space. These proxies receive ordinary C++ method invocations and convert them to message passing to the provider fragment. In contrast to similar approaches, in FOG/C++ the programmer can program both providers and local proxies explicitly. He can determine the communication model to use between client and server (1-to-1, 1-to-n), the type of message queuing, asynchronous versus synchronous behavior etc.

memory model. FOG/C++ programs run in distributed address spaces, local proxies of fragmented objects make remote objects accessible. C++ pointers cannot be passed because of the distributed address spaces. Instead, the user can implement packing and unpacking routines for copying objects by value.

parallelism. Provider fragments process incoming messages. If the programmer implements an immediate return of a call directed to a local proxy, caller and provider run concurrently. By using 1-to-n communication pattern, it is possible to spawn more than one parallel activity at once.

mapping. Fragmented objects provide mechanisms to create new proxies, i.e., making the fragmented object accessible. The programmer can implement mechanisms for replacing local proxies with provider fragments and vice versa, i.e. the programmer can migrate fragments.

scheduling. Done by the underlying SOS operating system. Since the programmer explicitly implements the behavior of provider fragments, he has the choice of implementing a strategy that decides, which of set of pending invocation requests to process next.

synchronization. A provider fragment can only process one incoming message at a time. If the caller of a remote method needs access to a return value, the call to the proxy blocks until the result is available. Otherwise, the programmer must implement a synchronization himself.

When synchronization of concurrently executing threads is needed, the programmer must implement that requirement himself.

fault tolerance.

Availability: FOG/C++ has been implemented on top of SOS, an object-oriented operating system [192].

Email address:
Yvon Gourhand $\longrightarrow$ `gourhant@corto.inria.fr`

References: [87] [152]

2.54 Guide

Developer: University of Grenoble and Bull Research Center

Description:

oo. Separation of interface and implementation. Inheritance.

memory model. Objects are persistent and stored on secondary memory. When used, these objects are bound into the address space of a job. Concurrent jobs can bind the same object. Synchronization of access to such shared objects is similar to access of concurrent activities within a single job.

parallelism. The programmer can start multiple jobs which use the same objects. Concurrency within a job is provided by `cobegin` blocks allowing for the creation of concurrently executed sub-activities. Objects are passive.

scheduling.

mapping. Transparent distribution of objects. However, the programmer can specify the storing node when creating a new object. Similarly, when an existing object is bound into the address space of a job, the programmer can optionally specify the node.

synchronization. Per default, methods of an object can be called under a mutual exclusion scheme. Above that, Guide provides boolean expressions, so-called activation conditions, attached to methods. A method may be executed when its activation condition evaluates to true. The activation conditions are specified in a special section of the class code but have a one-to-one relation to the methods. Guide provides special counters (invoked, stared, completed, etc.) Control can be inherited separately. The idea of synchronization based on counters is due to [183] and [85].

Whereas the first papers did not consider the fact that guards accessing instance variable might cause inconsistency, the paper [182] discusses this problem.

fault tolerance. Persistent objects. Exceptions.

Availability: Implemented for a network of 486 PCs on Mach 3.0 micro kernel (Guide-2).

The French speaking reader might find more information about IMAG on
`http://www.imag.fr`

Information about Guide can be retrieved via anonymous ftp from
`ftp://ftp.imag.fr/pub/GUIDE`

References: [64] [78] [98] [133] [134] [182]

2.55 HAL

Developer: University of Illinois at Urbana-Champaign

Description:

oo. inheritance. Forwarding of messages.

memory model.

parallelism. Both synchronous and asynchronous method calls. Asynchronous calls only when there is no return value. Post-processing by early `become`.

scheduling.

mapping. Target distributed memory machine. Placement of objects automatic.

synchronization. Methods can be *disabled* upon a condition, `all-except` disabled almost all methods if a given condition holds. Conditions can be arbitrary boolean functions without side-effects. The disabling conditions are part of the class definition and can be redefined in subclasses.

fault tolerance.

Availability: HAL is based on CHARM which is implemented both on shared and non-shared memory machines, including Sequent Balance and Symmetry, Encore Multimax, Aliant FX/8, Intel iPSC/2 and iPSC/i860 and NCUBE/2. The software and some documentation is available via anonymous ftp from
`ftp://biobio.cs.uiuc.edu/pub/Hal`

Email addresses:
Chris Houck ⟶ `chouck@ncsa.uiuc.edu`
Wooyoung Kim ⟶ `wooyoung@cs.uiuc.edu`
Gul Agha ⟶ `agha@cs.uiuc.edu`

References: [108] [132]

2.56 Harmony

Developer: Laboratory for Intelligent Systems, Ottawa, Canada/

Description:

oo. Class based operating system (no inheritance). Library/OS Kernel approach.

memory model. Shared.

parallelism. Active task object. The programmer provides a function which is to be executed concurrently as an argument when creating the task object. Message passing.

scheduling.

mapping. Targeting shared memory machine, i.e., no distribution.

synchronization. Synchronization primitives, e.g., semaphores and blocking receive.

fault tolerance.

Availability:

References: [149]

2.57 Heraklit

Developer: University of Erlangen-Nürnberg, Germany.

Description:

oo. Single inheritance. Method calls can be delegated.

memory model.

parallelism. An object can have an algorithm whose execution is called the object activity. This activity is started by a synchronous or an asynchronous call. The caller decides whether a call is asynchronous or not by adding the key word `ACTIVATE` to the call. Return values are not allowed for asynchronous calls. An asynchronous call returns a task object. This task object can be used to identify and modify the task object. For example, it is possible to terminate a task.

scheduling.

mapping. Global object space. The compiler or an additional tool is in charge of transparently mapping objects to the underlying distributed machine.

synchronization. Heraklit handles objets in a monitor like fashion, i.e., there is at most one activity working on an object. Heraklit offers special consistency points, where activities that are working on an object can delay their execution. During the delay, other method calls can be executed. The programmer thus can suggest a defer, i.e., he can inform the scheduler that a good point is reached to preempt the running thread.

fault tolerance.

Availability: Heraklit is no longer under development. Some documentation and a Sun Sparc (SunOS 4.1) implementation can be found on
`http://www2.informatik.uni-erlangen.de/IMMD-II/Research/Projects/`

Email address:
Peter Arius ⟶ `arius@informatik.uni-erlangen.de`
Wolfgang Betz ⟶ `betz@informatik.uni-erlangen.de`

References: [105]

2.58 HoME

Developer:

Description:

oo. See Smalltalk-80 (section 2.92).

memory model.

parallelism. See Smalltalk-80 (section 2.92).

scheduling.

mapping.

synchronization. See Smalltalk-80 (section 2.92).

fault tolerance.

Availability: The multi-processor used is the shared-memory multiprocessor OMRON LUNA 88K on which the Mach operating system runs.

References: [172]

2.59 Hybrid

Developer: University of Geneva, Switzerland

Description:

oo. Active objects communicate and synchronize by message passing. The basic model of communication is that of remote procedure call. RPC passes parameters by value.

memory model. Each object has its own address space. Object attribute values can only be accessed through member functions.

parallelism. Objects are grouped to domains. Domains are the unit of parallelism. There may be at most on thread active on at most one object of a domain. Other threads may be blocked or idle. Objects are active while they are responding to a message. New activities are created by invoking a special kind of operation, called a reflex. A reflex is a method which can be called asynchronously. Other methods are called synchronously. Hence, the calling mode is determined at the method declaration.

scheduling. If not specified otherwise, there is one queue of incoming messages per object. When a message is processed the corresponding member function is invoked. If two messages arrive at the queue the programmer cannot influence which of the messages is processed first.

Scheduling of threads on the real processor is done by the runtime and the operating system.

mapping. Not an issue, since implemented on a single processor machine.

synchronization. Sending an RPC message is synchronous, i.e., the sending thread blocks. Multiple thread can work on a domain in a quasi-concurrent fashion (one at a time)

However, the programmer can declare multiple (delay) queues and attach these to member functions. By explicitly opening and closing these delay queues, the programmer can implement a dynamically changing behavior of an object. But even with multiple queues it remains beyond the programmers control which of the messages from open queues is to be processed first.

There is no predefined construct in the language that allows to synchronize multiple active threads.

fault tolerance. None.

Availability: A prototype implementation of Hybrid runs on a single UNIX machine with shared-memory with pseudo concurrent processes. Concurrency is supported by a custom made package for lightweight processes.

Email address:
Oscar Nierstrasz ⟶ `oscar@iam.unibe.ch`

References: [169] [170] [176]

2.60 Java

Developer: Sun Microsystems Computer Corporation.

Description:

oo. Very similar to C++. Only single inheritance. Interface and class definition may be separated.

memory model. shared, flat.

parallelism. Special thread class. When an object of a class is created that is under this thread class, then a special function can be invoked separately, i.e., Java offers objects with an autonomous routine.

scheduling.

mapping. not an issue, since shared memory machine is targeted.

synchronization. Per default, all methods of a class can be executed concurrently. However, individual methods and code blocks can be marked "synchronized". In this case, a lock that is associated with every object is used to ensure that only one of the methods is executed at a time.

fault tolerance. exceptions.

Availability: The alpha version has been released (Release 1.0 Alpha 3) on May 11, 95. The current version and the documentation can be accessed from
`http://java.sun.com`

Email address:
group ⟶ `java@java.sun.com`

References: The documentation is available from the Web-page.

2.61 Karos

Developer: CE Saclay DEIN/SIR, France

Description:

oo. C++ library extension.

memory model. flat shared uni-processor.

parallelism. Asynchronous messages only.

scheduling.

mapping. Since only a single PE is targeted, no thoughts on mapping, placement, etc. are published.

synchronization. Synchronization is based on a 2 phase commit protocol on objects. The programmer can link subtasks together, if one of them fails to succeed, all of them are canceled. For this purpose the system keeps copies of earlier object states.

The programmer can/must re-implement the control method to decide whether an operation of a subtask has succeeded/failed.

fault tolerance. Atomicity of transactions.

Availability: There exists an implementation on a single PE.

References: [96]

2.62 LO

Developer: European Computer-Industry Research Center, Munich, Germany.

Description:

oo. Declarative state transitions.

memory model.

parallelism. The transition rules can split up into several successors, which can then be processed concurrently. A similar construct is available to combine those successors again.

scheduling.

mapping.

synchronization.

fault tolerance.

Availability:

References: [13]

2.63 Maude

Developer:

Description:

oo.

memory model.

parallelism.

scheduling.

mapping.

synchronization.

fault tolerance.

Availability: Email addresses:
—→

References:

2.64 Mediators

Developer:

Description:

oo.

memory model.

parallelism. A mediated object has its own thread which is dedicated to executing its synchronization code. This thread causes an invocation to start executing via one of the statements "exec" or "spawn". When "exec" is used, the mediator thread executes the method itself; "spawn" creates a new thread that asynchronously executes the invoked method. A "release" statement returns to the caller and removes the invocation from the mediated object.

scheduling.

mapping.

synchronization. Separation of synchronization code and instance variables used for synchronization from method code and regular instance variables. Mediators uses a "cycle" construct which is in fact a combination of a loop with a traditional select statement. Inside of this cycle statement, guards are used to start activities.

The guarded commands can use conditions which access instance variables of the calling thread (which function is called, information useful for scheduling purposes, instance variables etc.) Moreover, the conditions have access to the queues of pending invocations. Basically the same information used in Guides counters (see section 2.54).

fault tolerance.

Availability:

References: [88]

2.65 MeldC

Developer: Columbia University, New York

Description:

oo. This is an object-oriented language that understands all concepts as objects of a meta-class. A program is a collection of active objects which send and receive messages to and from other (local and remote) objects.

memory model. Each object has its own address space. Communication is only via message passing, i.e., invocation of member functions.

parallelism. Member functions are called synchronously or asynchronously. Only synchronous calls can be used to return results. The underlying thread concept works as follow: A thread is converted into a message which is sent to the addressed object. The callee accepts the messages and converts it back into a thread that then executes the called function. In a synchronous call, the thread is transformed back into a message that contains the return value and sent back to the caller. After receiving the message, return value and thread are extracted.

scheduling. When two messages arrive at an object at the same time, it is unspecified, which is executed first. The programmer cannot influence this directly. However, by implementing shadow objects he can. Shadow objects work like proxies. Calls that are directed to an object are trapped by the shadow object and considered there first.

mapping. Although there are no mechanisms in the language for object mapping (and hence process mapping), the programmer can implement these features by using shadow objects and implementing local proxies. This mechanisms easily extends to object migration, although migration is not available in the current version of MeldC.

synchronization. The shadow object could use a data structure to delay or re-arrange the request to the object. Many threads can be active on an object at one time. To implement synchronization, the language provides an atomic block that is guaranteed to be entered by only a single thread at a time. Hence, the programmer must ensure that the state of an object is only changed consistently.

MeldC offers a semaphore object to synchronize threads that concurrently work on different objects.

fault tolerance.

Availability: Currently the system is running on single processor Sun4 and Dec workstations. Depending on compiler flags, a MeldC program is compiled into a set of Unix programs or into one Unis program that uses a thread package to simulate concurrency.

Email addresses:
group $\longrightarrow$ `MeldC@cs.columbia.edu`
Gail E. Kaiser $\longrightarrow$ `kaiser@cs.columbia.edu`

References: [126] [127] [178]

2.66 Mentat

Developer: University of Virginia

Description:

oo. Extension of C++. The keyword **mentat** in the class definition specifies that objects of this class are used in parallel.

memory model. Each mentat object possesses a unique name, an address space, and a single thread of control. Mentat classes are address space disjoint. Thus all communication is via member function invocation and value parameters, i.e., data-driven. Vanilla C++ classes belong to the address space of a mentat object.

parallelism. The caller of a Mentat object member function is unaware of whether the implementation of the member function is sequential or parallel. The Mentat system analyzes the call graph and independently decides which member function calls can be executed concurrently.

scheduling. The Mentat programming language comes together with the Mentat runtime system. An object is represented as a process. The paper [91] gives information about Mentat's scheduling algorithms.

mapping. Generally Mentat automatically schedules objects for the application programmer. Though it is possible for the application programmer to give location hints (optional parameter for object creation), this is generally not done. There is no migration of objects.

synchronization. The programmer can label an object to be a stateless (regular) mentat object. For stateless objects, arbitrary many concurrent threads can execute member functions.

For other objects only one member function of a mentat object may execute at a time. This synchronizes access to the object's attributes. Since all communication must go through the parameters of function invocation, the compiler can analyze the call graph and detect data dependencies. Therefore, the compiler decides where and whether the caller needs to block. In case of dependencies, synchronizing and communication code is issued.

Mentat provides guarded statement execution similar to Ada's select statement, which can be used to implement high level synchronization needs and life routines.

fault tolerance. None.

Availability: Mentat has been implemented on networks of Sun3's, Sun4's, the IBM RS 6000, Silicon Graphics, and Gamma's. Mentat is available from
`http://www.cs.virginia.edu/ mentat`
`ftp://uvacs.cs.virginia.edu`

Email addresses:
group ⟶ mentat@virginia.edu
Andrew S. Grimshaw ⟶ grimshaw@virginia.edu

References: [89] [90] [91] [92] [93] [94]

2.67 Meyer's Proposal

Developer: Proposal by Betrand Meyer

Description:

oo. Extension of Eiffel. Meyer considers the *Design by Contract* [164] as the basic principle of object-oriented programming. He introduces the new keyword *separate* into Eiffel. That keyword indicates that an object is handled by a different processor. Due to the parallelism introduced, some semantic changes are necessary, to retain the principle of design-by-contract.

memory model. Common address space. No other changes to Eiffel.

parallelism. No notion of process and active object. A separate object can be busy, idle, or blocked. Calls to separate objects block, if the precondition is not fulfilled. Called functions are executed, while temporary disabling other calls to that object. Member function calls are synchronous if a return value is expected and asynchronous otherwise.

mapping. When creating a separate object, a virtual processor is assigned to this object. Meyer does not give any hint, of how virtual processors may be mapped to physical ones. There is no way to specify attachment of objects to each other because of locality.

scheduling. None. Left to the implementations.

synchronization. A separate object can only process one call of an operation at a time. Concurrent calls block. They block, too, if they cannot fulfill the preconditions of a feature.

fault tolerance. None.

Availability: This is just a proposed language. The author tries to retain the object-oriented paradigm, by minimally extending an existing language (Eiffel).

Email address:
Betrand Meyer ⟶ bertrand@eiffel.com

References: [165]

2.68 Micro C++, μC++

Developer: University of Waterloo, Waterloo, Canada

Description:

oo. μC++ is an extension of C++. In addition to ordinary C++ objects, there are monitor objects, coroutine objects, coroutine-monitor objects, and tasks.

memory model. Since only shared memory machines are considered, the memory model is that of standard C++.

parallelism. By declaring task objects the programmer creates new threads that proceed concurrently to the calling thread. There is a close connection between block boundaries and thread liveliness: A block is left only after all threads that have been declared within the syntactic boundaries of that block have ceased to exist. A thread terminates either if it completes the main routine of the task object or if the task object is deleted explicitly. Multiple threads can be declared – and started at once by declaring an array of tasks.

In contrast to many other systems, invocation of member function always is synchronous. However, by using synchronous routine calls and mutex objects the programmer can easily program asynchronous behavior or future objects if these are needed. The life routine of a thread handles the synchronization.

mapping. This is not an issue, since shared-memory architectures are targeted.

scheduling. The compiler and runtime system schedule thread execution. The programmer can explicitly control which message to accept next. For this purpose μC++ provides both guarding conditions and explicit accept statements. Moreover, the there are statements to delay a currently executed thread and resume execution later on.

synchronization.
Monitor objects, monitor-coroutine objects and tasks enforce an implicit mutual execution of the threads that access their member functions. If mutual exclusion is not needed, the programmer can use classical C++ classes and coroutines.

To use the autonomous routine of a task in a life form to govern synchronization, a guarded accept is provided.

fault tolerance.

Availability: μC++ has been implemented on single processor workstations and shared memory multiprocessors. Version 4.1 is available via anonymous ftp from ftp://plg.uwaterloo.ca/pub/uSystem

Email addresses:
group ⟶ usystem@maytag.uwaterloo.ca
Peter A. Buhr ⟶

References: [39] [38] [40]

2.69 Modula-3*

Developer: University of Karlsruhe, Germany.

Description:

oo. Extension of Modula-3.

memory model. Global address space.

parallelism. Modula-3* introduces a forall statement to spawn new activities.

scheduling.

mapping. It is intended to automatically map objects and concurrent activities to the parallel target machines.

synchronization. There are two versions of the forall statement, namely a synchronous version and an asynchronous version. In the synchronous version, the concurrent activities execute the statements in the body of the forall statement in complete unison, i.e., in lock step fashion. The synchronous execution is defined with a multiple-SIMD concept for branching statements. In contrast to this, the activities are free to proceed in their own speed when the asynchronous forall statement is used. Other than that there are no synchronization mechanisms.

fault tolerance.

Availability: The Modula-3* system is currently under construction and not yet available.

Email address:
Ernst A. Heinz $\longrightarrow$ `heinze@ira.uka.de`

References: [101]

2.70 MPC++

Developer: Tsukuba Research Center, Real World Computing Partnership, Japan

Description:

oo. Extension of C++. New is that functions can be called asynchronously. Furthermore, there are tokens and message entry points.

memory model. There is an address space per object. For communication between objects, messages and entry points are provided. Threads that access the same object may communicate via attributes of that object. Assumed memory model: processors with local memory, no clusters.

parallelism. For each asynchronously called function a thread is created that runs concurrently to the caller. If the method returns a result, there is a choice between asynchronous and synchronous call. In case of the asynchronous call, MPC++ offers a future-like concept: the code has entry-points, where a variable is either set to the return value or the thread is waiting until the value is available. Since there is no way to create more than one thread at once, scalability might be a problem for massively parallel applications.

scheduling. Automatic by runtime system.

mapping. The programmer is responsible to provide the processor number on which a thread has to be executed. For this purpose, MPC++ uses the @-notation. Since objects are created where a thread that executes the create function runs, the programmer can influence the mapping of objects to processors. No migration.

Process layer, virtual process layer, HPF like mapping. Explicit coding of mapping (block, cyclic) data parallel functions. The programmer knows about local variables per hardware PE and codes mappers where necessary.

synchronization. If threads that access the same object should communicate via attributes of that object, the programmer has to make sure, that access is synchronized properly. For this purpose, the language provides an atomic region. Statements that are grouped together in atomic regions are executed by a thread that has entered the region atomically, i.e., without another thread interfering.

The token mechanism for inter-object message passing ensures, that at most one thread may send a message to an entry point at a time. This sets up pairs of send and receive operations that lead to synchronization.

fault tolerance. None.

Availability: MPC++ is now running on top of the RWC1 functional simulator. At this time, just control parallel features have been implemented but data parallel features are still under construction. RWC1 is a message driven multi-threaded architecture. The testbed machine will be operational next year. The RWC1 machine consisting of 1,204 PE's will be installed around 1996-1997. MPC++ will be implemented on a CM-5 and an Intel Paragon.

Further information can be found
`http://www.rwcp.or.jp`

Email address:
Yutaka Ishikawa $\longrightarrow$ `ishikawa@rwcp.or.jp`

References: [112] [113]

2.71 Multiprocessor Smalltalk

Developer:

Description:

oo. See Smalltalk-80 (section 2.92).

memory model.

parallelism. See Smalltalk-80 (section 2.92).

scheduling.

mapping.

synchronization. See Smalltalk-80 (section 2.92).

fault tolerance.

Availability: Multiprocessor Smalltalk is implemented on the FireFly multiprocessor [203].

References: [174]

2.72 NAM

Developer: National Tsing-Hua University, Taiwan

Description: The proposed language is very similar to pC++, which is described on page 33.

Availability: The language has been implemented on a 32 node nCUBE2.

References: [142]

2.73 Obliq

Developer: Digital Equipment Corporation, Palo Alto

Description:

oo. Object based language, related to Modula3. There are no classes. Inheritance is implemented by object cloning, i.e., Obliq is a prototype-based language. (See [34] for the original prototype based language proposal.)

memory model.

parallelism. Additional activities are introduced by fork and join commands. The join command can wait upon the completion of names threads and returns the return value of the forked procedure.

scheduling. A thread can decide to pause.

mapping.

synchronization. The programmer can declare objects to be serialized, to achieve a monitor like behavior. In addition, there are condition variables (signal and watch). If a thread waits at a condition variable, it releases the mutex lock of a serialized object. A thread that executes a method of a serialized object can call other methods of the same object without causing deadlock. Above condition variables, Obliq offers mutexes and a corresponding lock statement.

fault tolerance. exception handling.

Availability: A version of Obliq which is implemented on top of Modula-3 Network Objects has been available since 1994 and has been used in several projects. The implementation and documentation are available from
`http://www.research.digital.com/SRC/home.html`

Email address:
Luca Cardelli $\longrightarrow$ `luca@src.dec.com`

References: [43] [44]

2.74 Orca

Developer: Vrije Universiteit, Amsterdam, The Netherlands

Description:

oo. Orca is object-based. There are no inheritance and no dynamic binding.

memory model. Distributed Shared Memory. Communication is based on shared data-objects. Other than shared data objects there no globally shared objects. (Orca's shared objects are similar to *protected objects* as introduced in Ada9x. See [23] for a comparison.)

parallelism. Parallelism is introduced by forking processes on a processor which can be specified optionally. Shared objects are parameters to process invocation.

scheduling. Done by the run-time system.

mapping. The Orca compiler determines the access patterns of processes to shared objects. A summary of this is passed to the runtime system, which uses it to make good decisions about which objects to replicate and where to store nonreplicated objects. Objects can be migrated and replicated without any intervention from the user.

Whereas storage of shared objects is transparent, the programmer must explicitly specify where a process is to be started. If the programmer does not provide the number of the processor when forking a process the new process runs on the same processor as the forking process.

synchronization. Implicitly done by the compiler. Since the compiler knows which shared data objects are accessed by a process, it issues synchronization code to mutually exclude processes from accessing the same object. Operations on an shared object appear to be indivisible. Beside this implicit synchronization there is a way to explicitly program guarding conditions. Operation calls may block if guards are evaluated to false.

fault tolerance.

Availability: Documentation can be found at
`ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers`
`ftp://ftp.cs.vu.nl/pub/papers/orca`

Email address:
Henri E. Bal ⟶ `bal@cs.vu.nl`

References: [22] [23] [25] [24] [102] [201]

2.75 Oz, Perdio

Developer: German Research Center for Artificial Intelligence, DFKI, Saarbrücken.

Description:

Oz is an object-oriented concurrent constraint programming language. Multiple inheritance. A project Perdio is currently being discussed which uses Oz in a distributed environment.

Oz determines automatically what to execute concurrently and applies a Wait by necessity mechanism to coordinate. Oz's logical variables are very similar to first class futures. The `batch` is a way to send a collection of method calls to an object which then are executed without interruption from other callers. Object can be closed, a closed object does not respond to incoming messages; these are queued until the object becomes open again.

Based on Monitor concept, i.e., one method at a time.

Perdio proposes transparent automatic distribution of objects.

Availability: Oz is available for many platforms running Unix-X, including Sparcs and 486 PCs. More information can be retrieved from
`ftp://ps-ftp.dfki.uni-sb.de`
`http://ps-www.dfki.uni-sb.de/oz`

Email addresses:
group ⟶ `oz@dfki.uni-sb.de`
Gerd Smolka ⟶ `smolka@dfki.uni-sb.de`

References: [103] [195] [196] [197]

2.76 Panda

Developer: University of Kaiserslautern, Germany

Description:

oo. Panda is a run-time package based on a very small operating system kernel which supports distributed applications written in C++.

memory model. All nodes of a Panda system share a single virtual address space, references transmitted across node boundaries do not lose their meaning. The programmer can declare objects in the shared memory and locally, the latter prevents access from remote nodes but increases the efficiency of local access.

parallelism. Panda offers a class for user level threads. Each instantiation of this class generates a new activity. This class has a member function called `code` that is executed by the thread. By declaring an array of user level threads, there is a way to create and start multiple parallel threads art once.

scheduling. The operating system is in charge of scheduling user processes and user level threads. (The programmer can redefine the scheduling policy by explicitly providing his own implementation.)

mapping. A non-local object is moved to the thread that invokes an operation on it. For thread mobility the use level thread class offers a explicit method for migration. There is no way to determine, where threads and objects are located at first.

synchronization. Shared objects are stored in Distributed Shared Memory. Currently an exclusive-read exclusive-write protocol has been implemented.

fault tolerance. Panda offers persistent objects.

Availability: Currently Sun Sparcs, Motorola 680xx and Intel i386/i486 are supported that communicate via TCP/IP and raw Ethernet. Sources and some documentation are available via anonymous ftp from
`ftp://ftp.uni-kl.de/reports_uni-kl/computer_science/system_softwar`
`http://www.uni-kl.de/AG-Nehmer/panda/panda.html`

Email addresses:
Holger Assenmacher
⟶ `assen@informatik.uni-kl.de`
Thomas Breitbach
⟶ `breitbac@informatik.uni-kl.de`
Peter Buhler ⟶ `buhler@informatik.uni-kl.de`
Volker Hübsch ⟶ `huebsch@informatik.uni-kl.de`
Holger Peine ⟶ `peine@informatik.uni-kl.de`
Reinhard Schwarz
⟶ `schwarz@informatik.uni-kl.de`

References: [18]

2.77 Parallel C++, pC++

Developer: University of Indiana

Description:

oo. The language is an extension of C++ for data-parallel programming. The language has a aggregate data type. When declaring objects of that type, a number of instance is created. Functions can be declared to work on instances and can be called to work on the whole aggregate. Aggregates are arrays of arbitrary dimension.

memory model. Same as C++. Elements can access other elements of the same aggregate, but may see outdated data. The programmer has to do necessary synchronization.

parallelism. By invoking functions on all instances of an aggregate at once. Above that, pC++ offers the triple notation of HPF [109] [*start-index, stop-index, stride*] to start member function on a specific subset of the aggregate.

scheduling. Scheduling is not an issue. Since aggregates are mapped by the programmer and function invocation always affects a regular subset of these aggregates, the execution model is simple: there is a loop per processor that iterates over the instances stored at that particular processor. (This scheduling strategy obeys the Owner-Computes-Rule [] used in the Fortran world.)

mapping. The programmer specifies a mapping of a template to the available processors. Then he can align his aggregates relative to this template. Hence, the programmer must try to achieve data locality of elements of collections that are used together. The notation borrows heavily from C* [204], Fortran D [83], HPF [109] and thus inherits some of the weaknesses, c.f. [206].

synchronization. The functions that run on the instances of a bulk in parallel are not synchronized during execution. However, the parallel activities exist only from the call of the parallel function to its termination on all instances. Parallel C++ provides two types of member functions for collections: In one type there is an explicit synchronization barrier at the end. In the other type, the programmer must solve synchronization needs himself. For that purpose, a barrier operation is provided.

fault tolerance.

Availability: Currently (version 1.7), pC++ has runtime systems for the following parallel machines: Thinking Machines CM5, Silicon Graphics Challenge, Kendall Square KSR1, Intel Paragon, Meiko CS2, IBM SP1, BBN TC2000, Sequent Symmetry, and a homogeneous networks of workstations (PVM). A lot of papers, the program files, and additional information are available from

`http://www.extreme.indiana.edu/sage`

Email address:
Dennis Gannon ⟶ `gannon@cs.indiana.edu`

References: [31] [32] [143] [84] [153]

2.78 Parallel Computing Action

Developer: Rank Xerox France, University of Paris VI

Description:

oo. Extension of C++. Library.

memory model. Hierarchical memory model. The programmer understands the difference between local objects and remote objects. This difference is apparent when an object is created and when a member function is invoked.

parallelism. Synchronous and asynchronous calls and futures. Caller decides.

scheduling.

mapping. No migration. When it comes to object creation there is a difference between local objects and remote objects. Local objects can only be referenced locally. For remote objects there is a special pointer class. Proxies are generated automatically. No semantic location transparency.

synchronization. Per default, objects behave like monitors. However, the programmer can specify delay conditions per method. These work like negative guards.

An intersting synchronization concept is parameterized synchronization. The delay conditions that can be attached to methods can use the same parameters that are used in the procedure signature.

fault tolerance.

Availability: Implemented on a Transputer platform.

Email addresses:
Hayssam Saleh ⟶ `saleh@litp.ibp.fr`
Philippe Gautron ⟶ `gautron@litp.ibp.fr`

References: [184] [185]

2.79 Parallel Object-Oriented Fortran

Developer: Mississippi State University

Description:

oo. Extension of plain Fortran to be object-based. There are no concepts of inheritance and strong typing.

memory model. There is a global addressing space. The address of an object is the concatenation of the physical processor number and a local id of an object. However, the attributes of objects are private and can only be accessed by public functions. Hence, each object has is private addressing space. The language does not offer the notion of modules.

parallelism. Operators are called asynchronously and are not allowed to have return parameters. There is no pre-emptive scheduling. The execution model is message driven, i.e., there are no explicit send-receive pairs.

mapping. The programmer is in charge of mapping the objects to processors. No migration of objects. The programmer has to provide the object id when calling a routine of this object. The routine will execute at that object's processor. Hence, the programmer has to map parallel activity to the objects. Manual data and process alignment, manual load balancing.

scheduling. None. No pre-emptive scheduling. When a function is executed, additional messages are queued. The processing of the function is not interrupted.

synchronization. At each processor at most one operator can be active, regardless of the total number of objects stored at that processor. Synchronization between processors has to be programmed with message passing constructs explicitly. Allow statement to enable acceptable methods. The reference does not talk about a disallow, which is somewhat strange.

fault tolerance. None.

Availability: OOF is currently available on the Intel i860 and Delta, SGI, IBM and Sun networks. A threaded version for the multiprocessor SPARC's is nearing completion. It is available via anonymous ftp from `ftp://ftp.erc.msstate.edu`

Email address:
Donna Reese —→ `dreese@erc.msstate.edu`

References: [181]

2.80 PO

Developer: University of Bologna, Italy.

Description:

oo.

memory model.

parallelism. Asynchronous interaction of objects, futures. In addition life objects. Synchronous calls are also available.

scheduling.

mapping. The authors present an automatic placement algorithm which is based on a static analysis of communication cost.

synchronization. PO offers two ways to coordinate interobject parallelism. One way is called "a posteriori" synchronization. In this case concurrent activities are allowed to execute methods of an object. These activities have to synchronize themselves by means of synchronization constructs like for example semaphores. The other approach is called "a priori" synchronization. Here the concurrent activities are not allowed to unconditionally execute methods concurrently. Instead the default behavior is monitor like, i.e., one activity is allowed to execute a method of an object at a time. The programmer however can supply constraints to the methods or even explicit scheduling methods to alleviate the restrictions of the monitor behavior.

fault tolerance.

Availability: PO and the mapping algorithm is implemented for a MEIKO Computing Surface Machine, which is based on T8 Transputer technology.

References: [73] [74]

2.81 POOL, POOL-T, POOL-I

Developer: Philips Research Laboratories, Eindhoven, The Netherlands.

Description:

oo. The POOL family of languages has several members. Whereas POOL-T does not have inheritance, POOL-I is an extension thereof that has multiple inheritance and subtyping. In POOL-T every data item is considered to be an object. Objects are active, they execute code from their *body* part even when no message arrives. The default body will accept any incoming message and execute the corresponding method. By programming a body method explicitly, the programmer can implement finer synchronization patterns.

memory model.

parallelism. There are two ways to introduce parallelism. One is to create multiple objects with non-default bodies. (Default bodies that only sequentially respond to arriving messages and the synchronous message passing effectively prevents an increase of concurrency.) The other way is to use *post-processing*. After a invoked method has returned a result to the caller, it can remain active and postprocess some statements concurrently to the caller.

Since message passing is point-to-point, methods cannot be invoked on several objects at the same time.

scheduling. If the default body is used, messages are accepted in an arbitrary order. Explicit body methods allow controlled acceptance. Messages are accepted by explicit answer statements which are parametrized by a list of acceptable messages. If more than one waiting message is in the list of acceptable messages used in the answer statement, an arbitrary one is accepted. In addition, POOL provides a select construct which can be used to implement conditional acceptance. Scheduling of concurrent threads on the available hardware is done by the runtime and operating system.

mapping.

synchronization. At most one thread can be active in an object if the object has a default body or has a life routine. All objects communicate via synchronous message passing. Since every data item is considered to be an object, synchronization of concurrent access is achieved by the one-thread-at-a-time concept. There is no mechanism in the language to synchronize threads working on multiple objects.

fault tolerance. None.

Availability: Pierre America has left the field and is now interested in *Computers and Music*.

References: [7] [8] [9] [10] [198] [218]

2.82 Presto

Developer: University of Washington, Seattle

Description:

oo. Library extension of C++.

memory model. Presto is developed for shared memory multiprocessors.

parallelism. Presto offers a special thread class. Objects of this thread class have a member function "start" which starts the new thread with a function that is provided as an argument. Instantiation of thread objects and starts can be combined in a fork command. A join is available. Threads are started asynchronously. The join can be used like futures, i.e., when the function executed by the thread returns a value, then the a join will return an untyped return object.

scheduling. The programmer can declare waiting queues. Threads can decide to wait. Other threads can wake up sleeping threads. Obviously, the programmer can implement any scheduling policy based on this basic mechanism.

mapping.

synchronization. For synchronization Presto offers a variety of primitives, e.g., spin-locks, locks, mutex objects, and condition variables, and coordination futures.

fault tolerance.

Availability: The source code (Presto 1.0) is available via anonymous ftp from
`ftp://ftp.cs.washington.edu/pub`

References: [29] [30]

2.83 Procol

Developer: University of Leiden, The Netherlands

Description:

oo. Procol is an object-based language without inheritance.

memory model.

parallelism. Message passing only. Return values, if any, must be passed back by separate messages. Rendezvous.

Procol offers a multicast: it is possible to send a message to a type, i.e., to all objects of a certain type.

scheduling.

mapping. Since Procol is translated for a single workstation, mapping is not an issue.

synchronization. The synchronization mechanism of Procol is based on path expressions as introduced by Campbell and Habermann [42]. The programmer uses regular expressions and guarding conditions to express the acceptable order, i.e. the protocol, of message arrival. Per object one action can be performed at a time.

fault tolerance. None.

Availability: After completing the thesis work, Procol has ceased to exist. The University of Leiden, The Netherlands, can be reached under:
`http://www.leidenuniv.nl`

References: [35] [135] [136]

2.84 Proof

Developer: University of Florida, Gainesville.

Description:

oo. Inheritance and other oo features. Separation of class interface and class implementation.

memory model.

parallelism. Autonomous routines. cobegin.

scheduling.

mapping. Automatic object clustering.

synchronization. Synchronization with guards per method. To coordinate concurrent access to objects the user must create additional 1st class lock objects and associate those with the relevant passive objects. The programmer is responsible for correctly using the lock object (read/write/modify-mode) before accessing the object.

fault tolerance. Persistent objects.

Availability: Proof is compiled to OCCAM and runs on a 16 node Transputer System.

References: [222]

2.85 pSather

Developer: International Computer Science Institute, ICSI, Berkeley

Description:

oo.

memory model.

parallelism. Asynchronous
and synchronous method calls. First class futures (queues). Caller chooses the calling mode. `par`-construct.

scheduling.

mapping.

synchronization. Lock statements that can handle multiple locks at once. This greatly reduces the chance of unintended deadlocks.

fault tolerance.

Availability: pSather is availblbe from
`http://www.icsi.berkeley.edu/Sather`
`news:comp.lang.sather`
Email addresses:
David Stoutamire $\longrightarrow$ `davids@icsi.berkeley.edu`

References:

2.86 PVM++

Developer: University of Tennessee

Description:

oo. PVM++ extends the I/O-stream capabilities of C++ to do message passing. The main purpose is to shield the C++ user from PVM's machine oriented details.

parallelism.

synchronization.

Availability: There is a prototype based on the old PVM 2.4 interface. Currently work is under way to update PVM++ to the 3.3 interface.
Email address:
Roland Pozo $\longrightarrow$ `pozo@cs.utk.edu`

References: [179]

2.87 QPC++

Developer: University of Oldenburg, Germany.

Description:

oo. Extension of C++. Process class.

memory model.

parallelism. Special member function called "body" that is executed automatically. The activity is associated with the object. The life routine can only be interrupted to execute other member function calls at certain explicitly indicated points, namely accept statements. In addition, QPC++ offers a reply statement to implement post processing. QPC++ offers asynchronous method calls as well. This is specified in the context of the caller. If the asynchronously called method returns a result, QPC++ uses a wait-by-necessity mechanism.
QPC++ offers the concept of a processor set. Objects of a class can be (statically and dynamically) added to processor sets. With respect to addressing, a process set is treated just like a normal process. If a member function of a process set is called, this function is called for each object of that set.

scheduling.

mapping.

synchronization. QPC++ offers semaphores to coordinate concurrent access to shared variables. The body code has explicit receipt statements to accept method calls, method call and accept are modeled as rendezvous. It can even be specified, from which object a call can be accepted. In general, since a single processor machine is targeted, only one activity can be executed at a time; moreover it is impossible to execute more than one member function concurrently on an object. The accept statement can be used in a select statement.

fault tolerance.

Availability: At the moment, the language is only implemented on a uniprocessor.

Email address:
Dietrich Boles
—→ boles@informatik.uni-oldenburg.de

References: [33]

2.88 Rosette

Developer: Microelectronics and Computer Technology Corp., MCC

Description:

oo. Rosette is based on the Actor model [104, 4, 5]. It incorporates multiple inheritance and reflection. Messages that cannot be processed by an actor are passed on to its "father".

memory model. As introduced in the Actor model. In Rosette there is a distinction between primitive actors that have an immutable state and thus are passed by value and non-primitive actors which are passed by reference.

parallelism. An Actor that processes an incoming message can asynchronously send an arbitrary number of messages, thus starting an arbitrary number of threads. However, message invocation is inherently sequential itself. Rosette offers a concurrent block, which surrounds a set of expressions that can be evaluated concurrently.

Rosette offers a block construct. In this block all expressions are evaluated concurrently. The result value of the block is the result of that expression that returns first. Other expressions continue to execute, however, their return value is discarded.

scheduling. Scheduling of threads to processors is not visible in Rosette.

mapping. Mapping of Actors and threads to the underlying machine is done by the runtime and the operating system. The programmer cannot influence it.

synchronization. Rosette offers three types of methods. On is essentially a function that returns a value. Since no side-effects are allowed, multiple calls to this type of methods are allowed at one time. The second type of method changes a single internal value of the object; the object is unlocked immediately. The standard method, however, requires exclusive access to the object.

Enabled-sets in Rosette have some similarities to the mechanisms of behavior abstractions (see ACT++, section 2.6) but provide a higher flexibility. The method name (and eventually some of the actual parameter values) contained in a queued message have to match an enabled-set.

fault tolerance.

Availability: Version 1.1 of Rosette is available via anonymous ftp from
ftp://biobio.cs.uiuc.edu

References: [208] [209] [210]

2.89 SAM

Developer: The MITRE Corporation.

Description:

oo.

memory model.

parallelism. Actor language.

scheduling.

mapping. Not an issue.

synchronization. Futures. Each object that requires synchronization is associated with an instance of a synchronization manager. One method at a time.

Each method call is labeled with a global time stamp. Before the method is executed, a copy of the object's state is saved. When later a method call arrives at the object with an earlier time stamp, then the system is rolled back to this copy and the newly arrived call is executed first, before the new calls are replayed.

fault tolerance.

Availability: Implemented on an Intel iPSC/2 multicomputer.

References: [180]

2.90 Scheduling Predicates

Developer: University of Dublin, Trinity College, Ireland.

Description:

oo.

memory model.

parallelism. Scheduling Processes is mainly interested in concurrency coordination mechanisms. The discussion of those is more or less independent of the way parallelism is introduced into a language. However, there is an implementation of a language based on Scheduling Predicates. This language uses a cobegin mechanism to initiate parallelism.

scheduling. To extend the synchronization counters, Scheduling Predicates offers `there_are_no`, `there_exists` and `for_all` to express boolean conditions on other pending calls.

mapping.

synchronization. In Scheduling Predicates the focus is not only on the synchronization of concurrent accesses to an object. In addition, the authors deal with the problem of which of a collection of delayed calls is to be scheduled next. Synchronization counters provide little help for this problem.

fault tolerance.

Availability: This research was carried out as part of Ciaran McHale's PhD thesis. Now that the thesis has been completed, research into this area has ceased. Information can be found on
`http://www.dsg.cs.tcd.ie/research/sos.html`
Email addresses:
Alexis Donnelly $\longrightarrow$ `donnelly@cs.tcd.ie`
Seán Baker $\longrightarrow$ `baker@cs.tcd.ie`

References: [160] [161]

2.91 Scoop

Developer: University of Montreal, Canada.

Description:

oo. This is a logic programming language.

memory model.

parallelism. When a Scoop program is started, there is one active process. Scoop offers a statement to create a new process. The new process starts to execute the goals in the class which is an argument of the process creation. (Sort of thread object).

scheduling. Processes can explicitly be removed from and added to the ready queue.

mapping.

synchronization. Scoop does not provide any implicit synchronization for concurrent access to objects. To synchronize concurrent processes Scoop provides explicit send and receive commands. Whereas the send command is non-blocking, the receive command will block until a message has been received from the channel specified as parameter.

fault tolerance.

Availability: Scoop has been implemented in Prolog on a single processor machine.

References: [214]

2.92 Smalltalk-80

Developer:

Description:

oo.

memory model.

parallelism. By sending a "fork" message to a block of expressions a thread is started that executes these expressions concurrently.

scheduling. Special "processor" object that implements a FIFO scheduling policy for threads.

mapping.

synchronization. Semaphore.

fault tolerance.

Availability:

References: [86]

2.93 Sos

Developer: University of Dublin, Trinity College, Ireland.

Description:

oo.

memory model.

parallelism. SOS is mainly interested in concurrency coordination mechanisms. The discussion of those is more or less independent of the way parallelism is introduced into a language. However, there is an implementation of SOS which uses a cobegin mechanism to initiate parallelism.

scheduling.

mapping.

synchronization. Very similar to Mediators (see section 2.64) and an extension of Scheduling Predicates (see section 2.90). The general idea is to have a special section in the class code where guards, special instance variables, and special methods can be implemented that are used only for implementing the coordination of concurrent access to objects.

fault tolerance.

Availability: This research was carried out as part of Ciaran McHale's PhD thesis. Now that the thesis has been completed, research into this area has ceased. Information can be found on
`http://www.dsg.cs.tcd.ie/research/sos.html`

Email addresses:
Alexis Donnelly —→ `donnelly@cs.tcd.ie`
Seán Baker —→ `baker@cs.tcd.ie`

References: [160]

2.94 Synchronizing Resources, SR

Developer: University of Arizona, Tucson

Description:

oo. Inheritance and separation of interface and implementation. A resource is an object that can contain code for processes and procs. Processes are started when the resource is created, procs are started on demand (either as a procedure or as a separate thread). An object thus is either passive if no processes are defined or can have multiple concurrent threads attached to it. Processes interact by invocation of operations which is independent of the location of the resource.

memory model. The programmer can group resources together to virtual machines which are mapped onto the available processors. All resources of one virtual processor use a shared memory and hence can cooperate on shared data.

parallelism. Operations can be called both synchronously and asynchronously. The caller decides. However, the method declaration can restrict the possible calling mode. No asynchronous calls of methods that return values. Post-processing is available.

SR has a `cobegin`-like statement.

An array of activities can be created and started at once (`forall`-like). Finally, SR offers autonomous and life routines.

scheduling. All virtual processors that are mapped onto a real processor are scheduled by the runtime system and the underlying operating system.

In life routines, the guarded receive statement allows to specify the policy that is used to decide which of several pending messages is to be accepted next.

mapping. The programmer is in charge of organizing his application in form of virtual processors. By distributing the data structures needed for the application over virtual processors, he solves the mapping problem manually. For this purpose, the virtual machine can be specified at resource creation time. When a virtual machine is created, the programmer can specify the node number of the underlying machine.

Location is transparent, except for the fact that references cannot be passed between different virtual machines.

synchronization. There are synchronous and asynchronous operation invocations. The addressed resource can either explicitly accept an incoming message (rendezvous) or can start an corresponding operation.

If concurrent processes access shared data, the programmer must use basic synchronization mechanisms, e.g. semaphores or blocking communication, to synchronize access.

fault tolerance.

Availability: SR appeared first in 1981 [14] (SR version 0), changed in 1986 [16] (SR version 1) and is now available in its version 2. Version 2.3 of SR works on one or more networked machines of the same architecture. True multiprocessing is supported on Silicon Graphics, Intel Paragon, and Sequent Symmetry systems, and on Sun systems running Solaris 2.3. Multiprocessing is simulated on other platforms, which include SunOS 4.x, HP RISC, DEC Alpha and Ultrix, IBM AIX, and Linux. Documentation and more is available from
`ftp://ftp.cs.arizona.edu//sr`
`http://www.cs.arizona.edu/sr/www`

Email addresses:
group —→ `sr-project@cs.arizona.edu`
Ronald A. Olsson —→ `olsson@cs.ucdavis.edu`
Gregory R. Andrews —→ `greg@cs.arizona.edu`

References: [15] [14] [173]

2.95 Tool

Developer: Pontificia University, Rio de Janeiro, Brazil.

Description:

oo. This is an object-oriented language that is intended to be used on top of Windows 3.1. to offer a graphical user interface to object-oriented programming. Single inheritance.

memory model.

parallelism. The programmer can declare so called extended classes. If a method of an extended class is called, this call is processed asynchronously. Asynchronous calls are not allowed to have any return parameters. For objects from other classes, all calls are synchronous.

scheduling.

mapping.

synchronization. The language design is based on a one-processor architecture. Since there can only be one processor active at a time, there are no synchronization primitives in the language. In fact, there is only one method actively working at an object at any given time.

fault tolerance.

Availability: TOOL has been implemented on top of Windows 3.1. Additional information can be found on:

`http://www.inf.puc-rio.br/~sergio/tool`

Email addresses:
Sergio E. R. de Carvalho $\longrightarrow$ `sergio@inf.puc-rio.br`

References: [54]

2.96 Trellis/Owl

Developer: Eastern Research Lab, Digital Equipment Corporation.

Description:

oo. Multiple inheritance.

memory model.

parallelism. There are 1st class thread objects ("activities") to be dynamically created by the programmer. In addition there is a join-like construct (wait). The function to be executed by the new thread is provided as an argument at creation time.

Concurrent Trellis/Owl offers the concept of an "Activity Set" to combine several thread objects and wait on all of them to complete.

scheduling. The programmer can create explicit wait queues. Threads can decide to enter the wait queue, other threads must resume them. If a thread is waiting in a wait queue, all the locks that thread has acquired are temporarily released and re-acquired when the thread resumes execution.

mapping.

synchronization. Trellis/Owl is designed under the assumption that most objects are not shared. Hence, the programmer is in charge to coordinate accesses to shared objects. For this purpose concurrent Trellis offers 1st class lock objects.

fault tolerance.

Availability: The system is implemented on a VAX 11/785 and MicroVAX running VMS.

References: [166] [187] [188]

2.97 Ubik

Developer: IBM Cambridge Scientific Center.

Description:

Describes a generalization of the Actor model.

oo.

memory model.

parallelism. Actor language. Asynchronous message passing only. Early become for postprocessing. Return values, if any, have to be passed back by a separate message.

scheduling.

mapping.

synchronization. Typical Actor language.

fault tolerance.

Availability: Email address:
Peter De Jong $\longrightarrow$ `pdjong@vnet.ibm.com`

References: [76]

2.98 UC++

Developer: University College London

Description:

oo. Parallel version of C++.

memory model. Global shared address space. Pointers to local and remote objects are identical.

parallelism. When an object is created it can be labeled "active". Moreover, a class can be an active class, allowing only active objects to be created. Only member functions of active objects that do not return result values can be called asynchronously. The caller can choose the mode.

scheduling.

mapping. When creating an object, the programmer can optionally specify the physical node number. The global address space is hierarchical, addresses are a concatenation of processor number and local address. An attempt to address memory outside of the own processor is considered an error. Pointer arithmetic only works within the boundaries of a single processor. Only method calls to remote objects are handled transparently.

synchronization. One method at a time.

fault tolerance.

Availability: The system is currently running on networks of Sun, DEC alpha and SGI. A publicly available PVM is announced for the near future. The current language definition can be found at
`http://www.cs.ucl.ac.uk/coside/ucpp`

Email address:
Russel Winder ⟶ `R.Winder@cs.ucl.ac.uk`

References: [219]

References

[1] Bruno Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, September 1993.

[2] Bruno Achauer. Implementation of distributed Trellis. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 103–117, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[3] Gul Agha and Christian J. Callsen. ActorSpaces: An open distributed programming paradigm. In *Proc. of the 4th ACM Symp. on Principles & Practice of Parallel Programming*, pages 23–32, May 1993. Appears also as ACM SIGPLAN Notices 28(7), July 1993.

[4] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press Cambridge, Massachusetts, London, England, 1986.

[5] Gul A. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[6] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent object migration in COOL2. In *Proc. of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems, ECOOP'92*, Utrecht, The Netherlands, June 29, 1992.

[7] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 234–242, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[8] Pierre America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press Cambridge, Massachusetts, London, England, 1987.

[9] Pierre America. A parallel object-oriented languag with inheritance and subtyping. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–168, Ottawa, Canada, October 21–25, 1990.

[10] Pierre America. POOL: Design and experience. In *Proc. of the ECOOP OOPSLA Workshop on Object-Based Concurrent Programming*, pages 16–20, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[11] Birger Andersen. Ellie - a general, fine-grained, first class object based language. *Journal of Object-Oriented Programming*, 5(2):35–42, May 1992.

[12] Birger Andersen. Efficiency by type-guided compilation. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages e1–e5, OOPSLA'93, Washington D.C., September 27, 1993.

[13] Jean-Marc Andreoli, Remo Pareschi, and Marc Bourgois. Dynamic programming as multiagent programming. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 163–176, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[14] Gregory R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[15] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, 1993.

[16] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purin, and Gregg M. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[17] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[18] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA – supporting distributed programming in C++. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 361–383, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[19] W. C. Athas and N. J. Boden. Cantor: An Actor programming system for scientific computing. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 66–68, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[20] Colin Atkinson, Stephen Goldsack, Andrea Di Maio, and Rami Bayan. Object-oriented concurrency and distribution in DRAGOON. *Journal of Object Oriented Programming*, 4(1):11–20, March/April 1991.

[21] Colin Atkinson, Andrea Di Maio, and Rami Bayan. Dragoon: an object-oriented notation supporting the reuse and distribution of ada software. In *Ada Letters*, pages 50–59, Fall 1990.

[22] Henri E. Bal. A comparative study of five parallel programming languages. In *proc. of Spring '91 Conf. on Open Distributed Systems, EurOpen*, pages 209–228, Tromso, Norway, May 20–24 1991.

[23] Henri E. Bal. Comparing data synchronization in Ada9X and Orca. Technical Report IR-345, Vrije Universiteit, Amsterdam, The Netherlands, December 1993.

[24] Henri E. Bal and M. Frans Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proc. of OOPSLA'93, 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 162–177, Washington D.C., 26 September – 1 October, 1993. ACM SIGPLAN Notices 28(10).

[25] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[26] Henry E. Bal, Jennifer S. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[27] John K. Bennet. The design and implementation of distributed Smalltalk. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–330, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[28] Marc Benveniste and Valérie Issarny. Concurrent programming notations in the object-oriented language Arche. Technical Report 690, IRISA, Institut de Recherche en Informatique et Systems Aleatoires, December 1992.

[29] B. N. Bershad, E. D. Lazowska, and H. M. Levy. Presto: A system for object-oriented parallel programming. *Software – Practice and Experience*, 1998.

[30] Brian. N. Bershad. The PRESTO user's manual. Technical Report 88-01-04, Department of Computer Science, University of Washington, Seattle, January 1988.

[31] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[32] François Bodin, Peter Beckman, Dennis Gannon, Shelby X. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proc. of Supercomputing'93*, pages 588–597, Portland, Oregon, November 15–19, 1993.

[33] Dietrich Boles. Parallel object-oriented programming with QPC++. *Structured Programming*, 14:158–172, 1993.

[34] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proc. of the ACM/IEEE Fall Joint Computer Conf.*, 1986.

[35] Jan van den Bos and Chris Laffra. PROCOL: A parallel object language with protocols. In *Proc. of OOPSLA'89, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 95–102, New Orleans, Louisiana, October 1–6, 1989. ACM SIGPLAN Notices 24(10).

[36] Soren Brandt and Ole Lehrmann Madsen. Object-oriented distributed programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proc. of the ECOOPO'93 Workshop on Object-Based Distributed Programming*, number 791 in Lecture Notes in Computer Science, pages 185–212, Kaiserslautern, Germany, July 26–27, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[37] Jean-Pierre Briot. From objects to Actors: Study of a limited symbiosis in Smalltalk-80. Technical Report 88-58, Laboratoire Informatique Théorique et Programmation, LITP, Paris, France, September 1988.

[38] Peter A. Buhr and Glen Ditchfield. Adding concurrency to a programming language. In *Proc. of*

USENIX C++ Technical Conference, pages 207–223, Portland, OR, August 10–13, 1992.

[39] Peter A. Buhr, Glen Ditchfield, Richard A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: concurrency in the object-oriented language C++. *Software – Practice and Experience*, 22(2):137–172, February 1992.

[40] Peter A. Buhr and Richard A. Stroobosscher. *μC++ Annotated Reference Manual, Version 3.7*. University of Waterloo, June 1993.

[41] Christian J. Callsen and Gul Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, June 1994.

[42] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science*, 16:89–102, April 1974.

[43] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center, 1994.

[44] Luca Cardelli. A language with distributed scope. *Computing System*, 8(1):27–59, January 1995.

[45] Peter Carlin, Mani Chandy, and Carl Kesselman. *The Compositional C++ Language Definition, Revision 0.9*. California Institute of Technology, Pasadena, March 1, 1993.

[46] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 102–104, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[47] Denis Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.

[48] Denis Caromel. Programming abstractions for concurrent programming. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS Pacific'90*, pages 245–253, Sydney, Australia, November 1990.

[49] Denis Caromel. A solution to the explicit/implicit control dilemma. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 21–25, Ottawa, Canada, October 21–25, 1990.

[50] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[51] Denis Caromel and Manuel Rebuffel. Object-based concurrency: Ten language features to achieve reuse. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS USA'93*, pages 205–214, Santa Barbara, CA, August 1993.

[52] H. Carr, R. R. Kessler, and M. Swanson. Distributed C++. *ACM SIGPLAN Notices*, 28(1):81, January 1993.

[53] Harold Carr, Robert R. Kessler, and Mark Swanson. Compiling distributed C++. In *Proc. 5th Symp. on Parallel and Distributed Processing*, pages 496–503. IEEE Computer Society, December 1993.

[54] Sergio E. R. de Carvalho. The object and event oriented language TOOL. Technical Report MCC06-93, Pontificia University, Rio de Janeiro, Brazil, 1993.

[55] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: A language for parallel programming. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 126–148. MIT Press Cambridge, Massachusetts, London, England, 1990.

[56] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *ACM Sigplan Symp. on Principles and Practice of Parallel Programming*, pages 249–259. ACM Press, New York, September 7–8, 1993.

[57] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[58] K. Mani Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 124–144, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[59] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press Cambridge, Massachusetts, London, England, 1993.

[60] Daniel T. Chang. CORAL: A concurrent object-oriented system for constructing and executing sequential, parallel and distributed applications. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 26–30, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[61] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, Seattle, September 1989.

[62] Arun Chatterjee. Distributed execution of C++ programs. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages b1–b6, OOPSLA'93, Washington D.C., September 27, 1993.

[63] Doreen Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA, Ames Research Center, Moffett Field, CA, March 1993.

[64] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, and X. Rousset de Pina. Experience with shared object support in the Guide system. *Symp. on Experiences on Distributed Systems and Multiprocessors*, 93.

[65] Andrew A. Chien. Concurrent Aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 31–36, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[66] Andrew A. Chien, editor. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press Cambridge, Massachusetts, London, England, 1993.

[67] Andrew A. Chien. Supporting modularity in highly-parallel programs. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 175–194. MIT Press Cambridge, Massachusetts, London, England, 1993.

[68] Andrew A. Chien. Concurrent Aggregates (CA): Design and experience with a concurrent object-oriented language based on aggregates. *Journal of Parallel and Distributed Computing*, 25(2):174–106, March 1995.

[69] Andrew A. Chien and William J. Dally. Experience with Concurrent Aggregates (CA): Implementation and programming. In *Proc. of the 5th Distributed Memory Computer Conf.*, Charleston, SC, April 9–12, 1990.

[70] Andrew A. Chien, Vijay Karamcheti, and John Plevyak. The Concert system – compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, University of Illinois at Urbana-Champaign, Urbana, IL, June 1993.

[71] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbim Zhang. *Concurrent Aggregates (CA) Language Report*. Concurrent Systems Architecture Group, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1994.

[72] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming system. *ACM Computing Surveys*, 23(1):91–127, March 1991.

[73] Antonio Corradi and Letizia Leonardi. PO an object model to express parallelism. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 152–155, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[74] Antonio Corradi, Letizia Leonardi, and Daniele Vigo. Massively parallel programming environments: How to map parallel objects on transputers. In M. Becker, L. Litzler, and M. Trehel, editors, *Proc. of Transputers '92, Advanced Research and Industrial Applications*, pages 125–141, Arc et Senans, France, May 20–22, 1992. IOS Press, Amsterdam, Netherlands.

[75] William J. Dally and Andrew A. Chien. Object-oriented concurrent programming in CST. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 28–31, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[76] Peter de Jong. Concurrent organizational objects. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 40–44, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[77] D. Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 444–452, Portland, Oregon, September 29 – October 2 1986. ACM SIGPLAN Notices 21(11).

[78] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A synchronization mechanism for typed objects in a distributed system. *ACM SIGPLAN Workshop on Concurrent Object-Based Language Design, in ACM SIGPLAN Notices*, 24(4):105–107, April 1989.

[79] D. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.

[80] E. W. Dijkstra. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[81] Takanobu Baba et al. A network-topology independent task allocation strategy for parallel computers. In *Proc. Supercomputing '90*, pages 878–887, 1990.

[82] Ian Foster. *Designing and Building Parallel Programs*, pages 167–205. Addison-Wesley, Reading, Mass., 1994.

[83] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[84] Dennis Gannon. Libraries and tools for object parallel programming. In *Proc. of the CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing*, Saint Hilaire du Touvet, France, September 7–8, 1992. Elsevier, Advances in Parallel Computing, Vol. 6, 1993.

[85] A. J. Gerber. Process synchronization by conter variables. *ACM Operating Systems Review*, 11(4):6–17, October 1977.

[86] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Implementation*. Addison-Wesley, Reading, Mass., 1983.

[87] Yvon Gourhant and Marc Shapiro. FOG/C++: a fragmented-object generator. In *C++ Conf.*, pages 63–74, San Francisco, CA, April 1990.

[88] J. E. Grass and R. H. Campbell. Mediators: a synchronization mechanism. In *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, pages 468–477, Cambridge, MA, May 19–23, 1986. IEEE Comput. Soc. Press.

[89] Andrew S. Grimshaw. Easy to use object-oriented parallel programming. *IEEE Computer*, 26(5):39–51, May 1993. Also University of Virginia, Charlottesville, VA, Technical Report CS-92-32.

[90] Andrew S. Grimshaw. The Mentat computation model – data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, Charlottesville, VA, May 1993.

[91] Andrew S. Grimshaw and V. E. Vivas. FALCON: A distributed scheduler for MIMD architectures. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 149–163, Atlanta, GA, March 1991.

[92] Andrew S. Grimshaw, Jon B. Weissan, and W. Timothy Strayer. Portable run-time support for dynamic object-oriented parallel processing. Technical Report CS-93-40, University of Virginia, Charlottesville, VA, July 1993.

[93] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed C. Loyot, Jr. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, June 1994. Also University of Virginia, Charlottesville, VA, Technical Report CS-92-43.

[94] Mentat Research Group. Mentat 2.5 programming language reference manual. Technical report, University of Virginia, Charlottesville, VA, 1995.

[95] Object Management Group. The common object request broker: Architecture and specification. Technical report, OMG, December 1991.

[96] R. Guerraoui. Dealing with atomicity in object-based distributed systems. *OOPS Messenger*, 3(3):10–13, July 1992.

[97] L. Gunaseelan and R. J. LeBland. Distributed Eiffel: A language for programming multi-granular distributed objects. In *Proc. of the 4th Int. Conf. on Computer Languages (IEEE)*, San Francisco, CA, April 1992.

[98] Daniel Hagimont, P.-Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossière, and X. Rousset de Pina. Persistent shared object support in the Guide system: Evaluation & related work. In *Proc. of OOPSLA'94, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 129–144, Portland, OR, October 23–27, 1994.

[99] F. Hamelin, J.-M. Jézéquel, and T. Priol. A multiparadigm object oriented parallel environment. In H. J. Siegel, editor, *Proc. of the 8th Int. Parallel Processing Symp. IPPS'94*, Cancún, Mexico, April 1994. IEEE Computer Society Press.

[100] C. L. Hartley and V. S. Sunderam. Concurrent programming with shared objects in networked environments. In *Proc. of the 7th Int. Parallel Processing Symp.*, pages 471–478, Los Angeles, April 1993.

[101] Ernst A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for explicitly parallel problem-oriented programming. In *Joint Symp. on Parallel Processing*, pages 269–276, Waseda University, Tokyo, May 17–19, 1993.

[102] Heinz-Peter Heinzle, Henri E. Bal, and Koen Langendoen. Implementing object-based distributed shared memory on Transputers. In A. De Gloria, M. R. Jand, and D. Marini, editors, *Transputer Applications and Systems '94*. IOS Press, 1994.

[103] Martin Henz. The Oz notation. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany, 1994.

[104] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[105] B. Hindel. An object-oriented programming language for distributed systems: HERAKLIT. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 114–116, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[106] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: programming and implementation. In *Proc. of the ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation PLDI*, pages 101–109, Portland, OR, June 21–23, 1989. ACM SIGPLAN Notices 24(7).

[107] Kaoru Hosokawa and Hiroaki Nakamura. Concurrent programming in COB. In A. Yonezawa and T. Ito, editors, *Proc. of the Japan/UK Workshop on Concurrency: Theory, Language and Architecture*, pages 142–156, Oxford, UK, September 25–27. 1989. Springer-Verlag Berlin, Heidelberg, New York.

[108] Chris Houck and Gul Agha. HAL: A high-level Actor language and its distributed implementation. In *21st Int. Conf. on Parallel Processing, ICPP '92*, volume II, pages 158–165, St. Charles, IL, August 1992.

[109] High Performance Fortran (HPF): Language specification. Technical report, Center for Research on Parallel Computation, Rice University, 1992. Available from `titan.cs.rice.edu` by anonymous ftp.

[110] Jin H. Hur and Kilnam Chon. Overview of a parallel object-oriented language CLIX. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 265–273, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[111] Norman C. Hutchinson, Rajeandra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programing lanuage report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.

[112] Yutaka Ishikawa. The MPC++ programming language v1.0 specification with commentary. Technical Report TR-94014, Tsukuba Research Center, Real World Computing Partnership, Japan, June 1994.

[113] Yutaka Ishikawa, Atsushi Hori, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. MPC++: A parallel programming language and its parallel objects support. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages j1–j5, OOPSLA'93, Washington D.C., September 27, 1993.

[114] Isis Distributed Systems, Inc., Marlboro, MA. *RDO/C++ Tutorial*, April 1994.

[115] Isis Distributed Systems, Inc., Marlboro, MA. *RDO/C++ Users Guide*, April 1994.

[116] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. In *Proc. of ECOOP'92 – European Conf. on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science, pages 197–212, Utrecht, The Netherlands, June 29 – July 3, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[117] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[118] J.-M. Jézéquel. Transparent parallelisation through reuse: between a compiler and a library approach. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 384–405, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[119] Eric Jul. Migration of light-weight processes in Emerald. *IEEE Operating Sys. Technical Committee Newsletter, Special Issue on Process Migration*, 3(1):25–30, 1989.

[120] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[121] Dennis Kafura. Concurrent object-oriented real-time systems research. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 203–205, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[122] Dennis Kafura and Greg Lavender. Recent progress in combining Actor based concurrency with object-oriented programming. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 55–58, Ottawa, Canada, October 21–25, 1990.

[123] Dennis Kafura and K. H. Lee. ACT++: Building a concurrent C++ with Actors. *Journal of Object Oriented Programming*, 3(1):25–37, May 1990.

[124] Dennis Kafura, Manibrata Mukherji, and Greg Lavender. ACT++ 2.0: A class library for concurrent programming in C++ using Actors. *Journal of*

Object Oriented Programming, 6(6):47–55, October 1993.

[125] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. In *ECOOP'89 – European Conf. on Object-Oriented Programming*, pages 131–145. Cambridge University Press, 1989.

[126] Gail E. Kaiser, Wenwey Hseush, James C. Lee, Shyhtsun F. Wu, Esther Woo, Eric Hilsdale, and Scott Meyer. MeldC: A reflective object-oriented coordination language. Technical Report CUCS-001-93, Dept. of Computer Science, Columbia University, New York, January 1993.

[127] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich, and Shyhtsun F. Wu. Multiple concurrency control policies in an object-oriented programming system. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 195–210. MIT Press Cambridge, Massachusetts, London, England, 1993.

[128] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proc. of OOPSLA'93, 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–109, Washington D.C., 26 September – 1 October, 1993. ACM SIGPLAN Notices 28(10).

[129] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proc. of ACM Supercomputing'93*, pages 598–607, Portland, Oregon, November 15–19, 1993.

[130] Murat Karaorman and John Bruno. Design and implementation issues for object-oriented concurrency. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages m1–m9, OOPSLA'93, Washington D.C., September 27, 1993.

[131] Murat Karaorman and John Bruno. Introduction of concurrency to a sequential language. *Communications of the ACM*, 37(9):103–116, September 1993.

[132] WooYoung Kim and Gul Agha. Compilation of a highly parallel Actor-based language. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 1–12, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[133] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object Oriented Programming*, 3(3):11–22, September/October 1990.

[134] Serge Lacourte. Exceptions in Guide, an object-oriented language for distributed applications. In *Proc. of ECOOP'91 – European Conf. on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 268–287, Geneva, Switzerland, July 15–19, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[135] Chris Laffra and Jan van den Bos. Constraints in concurrent object-oriented environments. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 64–67, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[136] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 68–72, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[137] J. Larus. C**: A large-grain object-oriented, data-parallel programming language. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 326–341, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[138] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain object-oriented, data-parallel programming language. Technical Report UWTR-1126, Computer Science Department, University of Wisconsin, Madison, November 1992.

[139] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS'94*, pages 208–218, October 4–7, 1994. Also available as Computer Science Department, University of Wisconsin, Madison, Technical Report TR1237.

[140] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[141] Rodger Lea and James Weightman. Supporting object oriented languages in an distributed environment: The COOL approach. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS USA'91*, Santa Barbara, August 3–6, 1991. Prentice Hall, Englewood Cliffs, New Jersey.

[142] Jeng Kuen Lee and Yunn-Yen Chen. Compiler and library support for aggregate object communications on distributed memory machines. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages d1–d10, OOPSLA'93, Washington D.C., September 27, 1993.

[143] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming – experiments and results. In *Proc. of Supercomputing'91*, pages 273–282, Albuquerque, NM, November 18–22, 1991.

[144] Y. S. Lee, J. H. Huang, and F. J. Wang. A distributed Smalltalk based on process-object model. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer Software and Applications Conf.*, pages 465–471, Tokyo, Japan, September 11–13, 1991. IEEE Comput. Soc. Press.

[145] Chu-Cheow Lim. *A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions*. PhD thesis, University of California at Berkeley, October 1993. Available as technical report ICSI TR-93-063.

[146] Klaus-Peter Löhr. Concurrency annotations. *ACM SIGPLAN Notices*, 27(10):327–340, October 1992.

[147] Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, September 1993.

[148] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Mario Tokoro and Remo Pareschi, editors, *Proc. of the 8th European Conf. on Object-Oriented Programming, ECOOP'94*, number 821 in Lecture Notes in Computer Science, pages 81–99, Bologne, Italy, July 4–8, 1994. Springer-Verlag Berlin, Heidelberg, New York.

[149] S.A. MacKay, W.M. Gentleman, D.A. Stewart, and M. Wein. Harmony as an object-oriented operating system. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 209–211, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[150] Ole Lehrmann Madsen. Building abstractions for concurrent object-oriented programming. Technical report, Computer Science Department, Aarhus University, Denmark, February 1993.

[151] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Mygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, Mass., 1993.

[152] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, July 1993.

[153] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In H. J. Siegel, editor, *Proc. of the 8th Int. Parallel Processing Symp. IPPS'94*, Cancún, Mexico, April 1994. IEEE Computer Society Press.

[154] Carl Manning. A peek at Acore, an Actor core language. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 84–86, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[155] Katsumi Maruyama and Nicolas Raguideau. Concurrent object-oriented language COOL. *ACM SIGPLAN Notices*, 29(9):105–114, September 1994.

[156] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Vacnouver, Canada, October 18–22, 1992. ACM SIGPLAN Notices 27(10).

[157] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, pages 57–64, 1994.

[158] Jeff McAffer and John Duimovich. Actra – an industrial strength concurrent object oriented programming system. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 82–84, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[159] Paul L. McCullough. Transparent forwarding: First steps. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 331–341, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[160] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994.

[161] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling predicates. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc of the ECOOP'91 Workshop on Object-Based*

Concurrent Computing, pages 177–193, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[162] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[163] Piyush Mehrotra and John Van Rosendale. Concurrent object access in BLAZE 2. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 40–42, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[164] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[165] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[166] J. Elliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 171–180, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[167] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, December 1993.

[168] Claudio Nascimento and Jean Dollimore. Behavior maintenance of migrating objects in a distributed object-oriented environment. *IEEE Computer*, 25(9), September 1992.

[169] Oscar Nierstrasz. Active objects in Hybrid. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 243–253, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[170] Oscar Nierstrasz. A tour of Hybrid: A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[171] Mark Nuttal. A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, 28(4):64–80, October 1994.

[172] Kazuhiro Ogata, Satoshi Kurihara, Mikio Inari, and Norihisa Doi. The design and implementation of HoME. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages, Design and Implementation, PLDI'92*, pages 44–54, San Francisco, CA, June 17–19 1992.

[173] Ronald A. Olsson, Gregory R. Andrews, Michael H. Coffin, and Gregg M. Townsend. SR – a language for parallel and distributed programming. Technical Report TR 92-09, Dept. of Computer Science, University of Arizona, Tucson, March 1992.

[174] Joseph Pallas and David Ungar. Multiprocessor Smalltalk a case study of a multiprocessor-based programming environment. In *Proc. of SIGPLAN Conf.*, pages 268–277, 1988.

[175] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–245. University of Geneva, Switzerland, 1989.

[176] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*. PhD thesis, Université de Genève, Départment d'Informatique, January 1992.

[177] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proc. of the 22nd Annual ACM SIGACT–SIGPLAN Symp. on Principles of Programming Languages POPL'95*, pages 311–321, San Francisco, CA, January 22–25, 1995.

[178] Steven S. Popovic, Gail E. Kaiser, and Shyhtsum F. Wu. MELDing transactions and objects. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 94–98, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[179] Roland Pozo. A stream-based interface in C++ for programming heterogeneous systems. In *Proc. of the CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing*, pages 162–177, Saint Hilaire du Touvet, France, September 7–8, 1992. Elsevier, Advances in Parallel Computing, Vol. 6, 1993.

[180] Myra Jean Prelle, Ann M. Wollrath, Thomas J. Brando, and Edward H. Bensley. The impact of selected concurrent language constructs on the SAM run-time system. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 99–103, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[181] Donna S. Reese and Ed Luke. Object oriented Fortran for development of portable parallel programs. In *Proc. of the 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 608–615, Dallas, Texas, December 2–5, 1991.

[182] M. Riveill. An overview of the Guide language. In *2nd Workshop on Objects in Large Distributed Applications*, Vancouver (Canada), 18 October 1992.

[183] P. Robert and J.-P. Verjus. Toward autonomous descriptions of synchronization modules. In B. Gilchrist, editor, *proc. IFIP Congress*, pages 981–986. North-Holland Publishing Co, 1977.

[184] Hayssam Saleh and Philippe Gautron. A concurrency control mechanism for C++ objects. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. of the ECOOP'91 Workshop on object-based concurrent computing*, pages 195–210, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[185] Hayssam Saleh and Philippe Gautron. A system library for C++ distributed applications on Transputer. In *Proc. of the 3rd Int. Conf. on Applications of Transputers*, pages 638–643. IOS Press, Amsterdam, Netherlands, August 28–30, 1991.

[186] Michele Di Santo and Giulio Iannello. Implementing actor-based primitives on distributed-memory architectures. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 45–49, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[187] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 9–16, Portland, Oregon, September 29 – October 2, 1986. ACM SIGPLAN Notices 21(11).

[188] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis – object-based environment: Language reference manual. Technical Report DEC-TR-372, Eastern Research Lab, DEC, Hudson, Massachusetts, November 1985.

[189] Marcel Schelvis and Eddy Bledoeg. The implementation of a Distributed Smalltalk. In *Proc. of the European Conf. on Object-Oriented Programming, ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 212–232, Oslo, Norway, August 15–17, 1988. Springer-Verlag Berlin, Heidelberg, New York.

[190] Alexander Schill and Markus U. Mock. DC++: Distributed object-oriented system support on top of OSF DCE. *Distributed Systems Engineering Journal*, 1(2):112–125, December 1993.

[191] Heinz W. Schmidt. Data parallel object-oriented programming. In *Proc. of the 5th Australian Supercomputer Conf.*, pages 263–272, Melbourne, December 1992.

[192] Marc Shapiro, Yvon Gourhandt, Sabine Habert, Laurence Mosseri, Michel Ruffina, and Céline Valot. SOS: An object-oriented operating system – assessment and perspective. *Computer System*, 2(4), December 1989.

[193] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, January 1991.

[194] Robert J. Smith. Experimental systems kit – final project report. Technical report, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas, March 1991.

[195] Gert Smolka. The definition of kernal Oz. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany, 1994.

[196] Gert Smolka. An Oz primer. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany, April 1995.

[197] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 27–48. The MIT Press, 1995.

[198] Jan van der Spek. POOL-X and its implementation. In Pierre America, editor, *Parallel Database Systems. PRISMA Workshop*, pages 309–344, Noordwijk, The Netherlands, September 24–26, 1990. Springer-Verlag Berlin, Heidelberg, New York.

[199] Kazunori Takashio and Mario Tokoro. DROL: An object-oriented programming language for distributed real-time systems. In *Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 276–294, Vacnouver, Canada, October 18–22, 1992. ACM SIGPLAN Notices 27(10).

[200] Hidehiko Tanaka. A parallel object oriented language FLENG++ and its control system on the parallel machine PIE64. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language and Architecture. Japan/UK Workshop Proc.*, pages 157–172. Springer-Verlag Berlin, Heidelberg, New York, 1991.

[201] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henry E. Bal. Parallel programming using shared objects and braodcasting. *IEEE Computer*, 25(18):10–19, August 1992.

[202] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In G. Blelloch, M. Chandy,

and S. Jagannathan, editors, *Proc. of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[203] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[204] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, April 1991.

[205] David A. Thomas, Wilf R. LaLonde, John Duimovich, Michael Wilson, Jeff McAffer, and Brian Barry. Actra - a multitasking/multiprocessing Smalltalk. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 87–89, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[206] Walter F. Tichy, Michael Philippsen, and Phil Hatcher. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[207] Michael D. Tiemann. Solving the RPC problem in GNU C++. Technical Report ESKIT-285-88, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas, 1988.

[208] Chris Tomlinson, Won Kim, Marek Scheevel, Vineet Singh, Becky Will, and Gul Agha. Rosette: an object-oriented concurrent system architecture. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 91–93, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[209] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled-sets. In *Proc. of OOPSLA'89, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 103–112, New Orleans, Louisiana, October 1–6, 1989. ACM SIGPLAN Notices (24)10.

[210] Christine Tomlinson, Mark Scheevel, and Vineet Singh. *Report on Rosette 1.1*, August 1991. Object-Oriented and Distributed Systems Laboratory, Microelectronics and Computer Technology Corp., MCC.

[211] Rajiv Trehan, Nobuyuki Sawashima, Akira Morishita, Ichiro Tomoda, Toru Imai, and Ken ichi Maeda. Concurrent object oriented 'C' (cooC). *ACM SIGPLAN Notices*, 28(2):45–52, February 1993.

[212] Louis H. Turcotte. A survey of software environments for exploiting network computing resources. Technical report, Mississippi State University, June 11, 1993.

[213] Minoru Uehara and Mario Tokoro. An adaptive load balancing method in the computational field model. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 109–113, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[214] Jean Vaucher, Guy Lapalme, and Jacques Malenfant. SCOOP – structured concurrent object-oriented prolog. In *ECOOP'88 – European Conf. on Object-Oriented Programming*, pages 191–210, Oslo, Norway, August 15–17, 1988. Springer-Verlag Berlin, Heidelberg, New York.

[215] Peter Wegner. Dimensions of object.based language design. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 168–182, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[216] Emily A. West. *Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language*. PhD thesis, University of Virginia, Department of Computer Science, May 1994. Available as technical report CS-94-16.

[217] Emily A. West and Andrew S. Grimshaw. Braid: Integrating task and data parallelism. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 211–219, McLean, VA, February 6–9, 1995.

[218] R. H. H. Wester and B. J. A. Hulshof. The POOMA operating system. In Pierre America, editor, *Parallel Database Systems. PRISMA Workshop*, pages 396–323, Noordwijk, The Netherlands, September 24–26, 1990. Springer-Verlag Berlin, Heidelberg, New York.

[219] R. Winder, G. Roberts, and M. Wei. CoSIDE and parallel object-oriented languages. In *Addendum to the Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–213, Vacnouver, Canada, October 5–10, 1992.

[220] Barbara Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: a survey. *IEEE Computer*, 11(6):56–66, November 1992.

[221] Gao Yaoqing and Yuen Chung Kwong. A survey of implementations of concurrent, parallel and distributed Smalltalk. *ACM SIGPLAN Notices*, 28(9):29–35, September 1993.

[222] Stephen S. Yau, Xiaoping Jia, Doo-Hwan Bae, Madhan Chidambaram, and Gilho Oh. An object-oriented approach to software development for parallel processing systems. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer Software and Applications Conf.*, pages 453–5–8, Tokyo, Japan, September 11–13, 1991. IEEE Comput. Soc. Press.

[223] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 331–340, Portland, Oregon, September 29 – October 2 1986. ACM SIGPLAN Notices 21(11).

[224] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System – theory, language, programming, implementation, and application.* Computer System Series. MIT Press Cambridge, Massachusetts, London, England, 1990.

[225] Kaoru Yoshida and Takashi Chikayama. A'UM = stream+object+relation. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 55–58, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[226] Tsutomu Yoshinaga and Takanobu Baba. A parallel object-oriented language A-NETL and its programming environment. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer Software and Applications Conf.*, pages 459–464, Tokyo, Japan, September 11-13, 1991. IEEE Comput. Soc. Press.