Parallel Programming in Common Lisp using Actors and Parallel Abstractions*

Lothar Hotz and Michael Trowe

Universität Hamburg
Labor für Künstliche Intelligenz
Fachbereich Informatik
Vogt-Kölln-Str.30, D-22527 Hamburg, Germany
hotz@informatik.uni-hamburg.de

Abstract. In this paper we describe an extension of Common Lisp which allows the definition of parallel programs within that functional and object-oriented language. In particular, the extensions are the introducing of active objects, sending synchronous and asynchronous messages between them, automatic and manual distribution of active objects to object spaces, and transparent object managing. With these extensions object-oriented parallel programming on a workstation cluster using different Common Lisp images is possible. These concepts are implemented as an extension of Allegro Common Lisp subsumed by the name NetCLOS. Furthermore, it is shown how NetCLOS can be used to realize parallel abstractions for implementing parallel AI methods at a highly abstract level.

1 Introduction

One of the big problems of Artificial Intelligence (AI) is getting its applications answer in time. Parallel computation is one way to solve this problem. But though there are many parallel implementations of basic AI techniques, there are very few AI applications which use them. This drawback is ascertained due to two reasons:

- Most of these implementations depend on special parallel hardware (e.g., [7, 4, 12]). This hardware is expensive and not widely available. Furthermore, the specification of many applications excludes the use of special hardware (e.g., personal assistant).
- Most of them are written in special parallel programming languages unknown to the application programmer and lacking features important to develop a complete application [22]. Hence their integration into such an application is difficult.

Especially for AI methods the proposed parallel languages and models are too different from commonly used languages. They do not provide the flexibility programmers need, and are not integrated with existing languages [13]. Furthermore, in special parallel languages non-parallel aspects are not adequately expressable.

Our goal is to simplify the parallel implementation of standard AI techniques. We think this means using standard hardware (i.e., a workstation cluster) and extending a language widely used for AI programming in a way that hides any kind of explicit parallel programming from the application programmer. Thus, we extend Common Lisp [19] with two levels of features for parallel programming on workstation clusters. The upper level is intended for easy use by the AI programmer inexperienced with parallel programming, while the lower level is intended for implementing the upper level. In the lower level an integration of an actor-like language [5] in Common Lisp and its object-oriented part CLOS (Common Lisp Object System) is introduced (see Section 3). The upper level realizes parallel abstractions as complex structures and operations on them (Section 4). The use of parallel abstractions is demonstrated with a constraint filtering algorithm (Section 5). In Section 6 related work is discussed. First, we give a more detailed view of our parallel programming model.

---

* This research has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (BMBF) under the grant 01 IN 509 D 0, INDIA - Intelligente Diagnose in der Anwendung

Fig. 1. Levels of used abstractions.

3 Extending CLOS with actors

In this section we describe the lower level (NetCLOS level), which is an extension of Common Lisp and its object-oriented part CLOS (Common Lisp Object System). Features of NetCLOS are:

- Active objects, which include data, methods, a mail queue, and a process for handling incoming messages by calling methods.
- Message passing for synchronous and asynchronous communication between active objects.

---

[1] NetCLOS as an extension to Allegro Common Lisp is implemented and can be received from the authors. NetCLOS is implemented using the metaobject protocol of CLOS (see [11] and [8]).

– Synchronization operations for delaying requests.

Programming with NetCLOS is done by creating active objects and sending messages between them, which is fully integrated in the programming style of CLOS. Distribution of active objects to workstations is widely hidden to the user. Thus, active objects are distributed over a virtual machine consisting of several Lisp images residing on several workstations of a cluster. To allow flexible programming of distributed active objects automatic distributions as well as explicit moving of active objects is included in NetCLOS.

There are some approaches which include parallel programming in Lisp (see e.g., [22]) but most of them concentrate on the functional part of Lisp. In NetCLOS the object-oriented part (CLOS) is focussed as an extendable part for parallel programming. With our new approach we introduce active objects in the spirit of actors [5,1] in CLOS to make parallel object-oriented programming in Common Lisp possible.

3.1 Design decisions

Following [17] we discuss three dimensions of design issues for concurrent object-oriented programming: object model, internal concurrency, and interaction. The decisions are inspired by the concurrent object-oriented language ABCL/1 [21].

Object model. Because we extend an existing object-oriented language, where passive objects reside in the language, we use a heterogeneous object model with passive and active objects. Passive objects are normal CLOS objects, active objects are extended by a mail queue and a process. By buffering incoming messages in a mail queue active objects synchronize concurrent calls. Passive objects do not synchronize concurrent calls, i.e., they have to be saved by explicit synchronization calls or are used within a single-threaded active object.

Internal concurrency. Another design decision is whether an active object can process calls sequentially or in parallel. If calls are processed in parallel on the same active object, i.e. on one data source, a high communication rate will be necessary. Because of high communication costs in a workstation cluster, data and processes should reside on the same machine. Yet we decide to process tasks of one active object sequentially.

Interaction. In NetCLOS object identifications are used to determine the recipient of a message. Message passing can be done in three ways: *future*-messages, which are easy to integrate in a functional context, one-way messages (*past*-messages), as a more flexible but also more complicated tool for communication, and remote procedure calls (*now-messages*) for synchronous communication.

3.2 CLOS - the Common Lisp Object System

CLOS belongs to the ANSI Common Lisp standard [19] and defines the object-oriented part of the language. CLOS includes classes with multiple inheritance, generic functions, declarative method combination, and a metaobject protocol. Classes are defined by slots (instance variables or data fields) and some superclasses. All slots of all superclasses are inherited. Instead of having a message-passing concept as in other object-oriented languages, CLOS includes the more powerful concept of generic functions. A generic function describes a set of methods, i.e., a method is related to a generic function, not to one class. Because a generic function may have more than one discriminating argument, a generic function is related to a set of classes not to one specific class. Instead of passing a message to an object, the generic function is called. The classes of its arguments are used to determine (at runtime) which methods should be used to compute a value for the generic function. Declarative method combinations describe how several applicable methods should be ordered and how their results should be combined. This is done by defining different kinds of methods, e.g., *before-methods* are called before *primary-methods*, etc. The metaobject protocol is used to extend CLOS' behavior portably. For instance, the slot access can be modified to be a remote slot access. So called *metaclasses* can be defined by the user, which enhance the behavior of classes and objects (instances).

Fig. 2. Parts of an active object.

Sending messages and synchronization Active objects communicate only by using one of three message types (similar to ABCL/1): *past*, *now*, or *future*-messages. *Past*-messages are asynchronous one-way-messages. The caller can continue its work, after a message is sent. After calling the methods related to the message, the recipient does not send a reply message to the caller. Past-messages are declared by the keyword `:past` as in:

```
(defpargeneric <name> :past (<recipient-object> <argument1> ...))
```

Past-messages are used to realize complex request-reply frames.

Now-messages are remote procedure calls, i.e., the caller waits until the recipient accepts the message, computes the request, and sends the reply back. These messages are indicated by the keyword `:now`. Now-messages are used to ensure that the caller is inactive while processing the message. This can be used to realize a sequential interface to an active object or to ensure specific synchronization conditions.

When a *future*-message is sent, a *future* is created. Futures can be seen as simple active objects which can only deal with two messages: the past-message `write-result` and the now-message `touch`.

Fig. 3. Transfer of a query via a proxy.

Distribution The distribution model of NetCLOS is based on the notion of one *object space* residing on each workstation of a cluster. An object space contains all information to handle active objects, e.g., all necessary classes and functions are known to each object space.[2] Every active object resides on exactly one object space. But an active object can be referenced from each object space, not only the local one. Thus, the identity of an active object is guaranteed over all object spaces, i.e. each active object is unique and can be referenced from diverse object spaces. When messages are sent or an active object is passed as argument of a message, it makes no difference if the object is locally or remotely referenced. This holds only for active objects, other data types — like passive objects, lists, arrays, strings, or records — are only locally referenced. If objects of such data types function as arguments of a message, a copy is sent to the recipient, i.e., changes to those types made by the recipient are not known to the caller. Thus, no side effects on such datatypes are allowed. The copy

---

[2] Special features are defined for distributing new definitions of generic functions and classes and for defining systems (sets of files), which have to be known to all object spaces.

of such an object includes also nested objects (e.g. lists of lists). Cyclic data is handle correctly, by creating the same cycles in the remote object. Copying is done by generating a Lisp form, which when evaluated creates the appropriate objects, and sending the Lisp form to the recipient.

There are two alternatives for an application to distribute its active objects to object spaces: One is to move the object explicitly by calling the function `move`[3], the other is to use a predefined distribution class. There are two classes for distribution: one to realize a static distribution by deriving Task Interaction Graphs from the reference structure of the active objects to distribute and another class distributes the active objects dynamically when they are created. For the latter only a simple scheme is present for sending active objects to object spaces in a round robin manner. For distributing Task Interaction Graphs we use a combination of bisection and Kernighan-Lin (see section 4.1). But extensions of NetCLOS made by subclassing can be defined to realize more sophisticated distribution strategies.

For moving an active object explicitly the moving behavior of its slots can be specified when defining the active object. When specified with `:follow` the value of that slot (another active object) is moved in the same object space as the active object itself. When specified with `:stay` the value of the slot stays in the current object space. Instead of the value the moved active object contains a representant value (i.e., a *proxy*) as slot value.

Remote references Remote references to values is realized by proxies. A proxy knows the location of the original active object and sends a kernel message to the original active object on a proxy reference to get the referenced value. The necessary infrastructure is internal in NetCLOS. When moving an active object appropriate proxies are automatically created. If a slot is of type `:follow` a local proxy is created which referes to the remote slot value. If a slot is of type `:stay` a remote proxy is created which referes to the local slot value. Garbage collection is extended to handle proxy references, as the next paragraph describes.

Object spaces An *object space* is realized as a Lisp image and resides on one host; it is assumed that each host processes only one object space. An object space contains some features (some realized as light weight processes (lwp) inside one Lisp image), which realize the functionality of a virtual machine.

Object spaces communicate with each other by kernel messages. An object space contains one caller-lwp for each object space it wants to communicate with. The caller-lwp packs the message to be sent (i.e., creates a Lisp form which contains the message on evaluation) and sends it via TCP/IP to the other object space. There, the callee object space contains an lwp for realizing a recipient-lwp for each other object space. The recipient-lwp unpacks the message and evaluates the resulting Lisp form.

Each object space contains an object store, which contains local and remotely referenced objects. It ensures exactly one proxy for each remote object, and it realizes a garbage collection method for remote objects. This is necessary, because the internal garbage collection method of Lisp is image specific and references of proxies (residing in a remote image) to objects are not considered. Thus, with the internal garbage collection method an object would be garbage collected even if a proxy residing on another space refers to it. The remote garbage collection is carried out by counting remote references to each object. When no reference to a proxy and no local reference to the related object is present this object can be garbage collected or if the counter decrements to 0, the object can be garbage collected by the internal garbage collector contained in each Lisp image. A problem not yet attended to are garbage collecting cyclic reference structures.

Each object space contains furthermore an *object space manager* (or *object server*). Working with NetCLOS starts by loading NetCLOS in a Lisp image, which creates an initial object space on the host

---

[3] In the current implementation the function `move` can not be used in generic functions being performed in parallel on one active object, because the process synchronizing the mail queue is not moveable in Lisp.

the Lisp image is started — the master host — and an initial manager — the master[4]. This manager starts the virtual machine by giving it a number of hostnames[5]. On each host an object space is started, is initialized by some initialization forms and the communication links are established, which connect the object spaces to each other. Thus, a fully connected communication structure is created. Furthermore, the manager ensures an equal global context of classes and functions. When classes and functions are loaded in one object space, the manager sends an appropriate message to all object spaces which ensures a loading of the same classes and functions in those other spaces. If one object space stops working (e.g. because the Lisp image quits) it sends a specific message to each object space, which can react appropriately and can proceed working.

Integration of NetCLOS in CLOS There are two viewpoints to consider when integrating NetCLOS into CLOS: the implementors view and the application programmers view. From the implementors view NetCLOS is integrated in CLOS portably, i.e., without changing the implementation of CLOS. Even more, by extending the existing features, a small extension of the behavior of CLOS yields to big expressability. E.g., the slot access is extended by the possibility of defining moving behavior for slots. A slot access protocol inherent in CLOS is extended to handle this moving behavior and thus, every slot access for active objects is changed. From a programmer's point of view this is done by the same programming interface, i.e., the slot access function does not change to, e.g., special proxy access functions like `proxy-value`. Besides extending the slot access generic function metaclasses are integrated in NetCLOS for describing generic functions to be handled as messages, i.e. for each method call special methods for testing the active object's location (local or remote) and selecting the appropriate send style (now, past, future) are automatically integrated by these metaclasses. Furthermore, for each class *c*, whose instances can be moved, a subclass *proxy-c* is created. This class is of type *proxy-class*, a metaclass, which implements proxy behavior. For example, this metaclass creates only instances, which does not contain any slots, but sends slot references as messages to a remote instance, which contains the slots. Thus, with *proxy-c* the instance allocation protocol and the slot access protocol are extended.

This approach of extending CLOS is possible because of the existence of a metaobject protocol [11], which clearly specifies the behavior of diverse CLOS features, like slot access, method combination, and inheritance behavior. The extensions are portable in the sense that each CLOS implementation based on the metaobject protocol can be extended by NetCLOS. The usage of the metaobject protocol is different to a library approach where a number of functions have to be introduced and learned before a parallel program can be written. For further reading on this point see also [8].

Thus, from a programmer's point of view the extensions fit well in the programming style of CLOS. Even the programming of message passing instead of generic function calls are acceptable, because it comes as a special generic function call (i.e., to the first argument). Some Lisp specific features have to be handled with care, because they are not yet implemented in NetCLOS or are hard to integrate in a distributed environment. For instance, closures cannot be moved from one host to another and dynamically created functions are not yet handled correctly. This is due to the fact, that closures are not part of a metaobject protocol and thus, are not accessable without touching the implementation of Lisp. However, it is possible to define generic functions (i.e. named closures) and classes in NetCLOS, which are distributed to all object spaces, thus every space knows the same functions and classes. Neither are cyclic reference structures of active objects garbage collected. However, NetCLOS is used to implement parallel object-oriented programs based on CLOS.

4 Introducing parallel abstractions for programming AI applications

Parallel programming is a difficult task, because of the possibly big number of flows of control. In low-level parallel languages the handling of these flows is left to the programmer. To make parallel

---

[4] In the current implementation only the master can start object spaces, object spaces cannot connect to the master from outside. Thus, client-server structures on the object space level are not yet realizable.

[5] The current implementation does not include a user-password handling, thus, only trusted remote hosts of a workstation cluster can be given.

Fig. 4. Control abstractions and their integration in application classes.

4.1 An example structure — the relaxation net

Relaxation net is an abstraction implementing parallel discrete relaxation (see [7] for a similar approach). It consists mainly of

- a class of active objects (*value nodes*) acting as shared stores. Accesses to these stores are automatically synchronized, i.e. this is done by the NetCLOS level. These active objects can be used to implement the variables of a constraint net.

---

[6] i.e. the time when the application program is written down.

- a class of active objects (*function nodes*) which, when activated, computes a function of the content of a set of stores. These active objects can be used to implement constraints.
- a structure class which organizes stores and functional objects into a network and provides for iterated activation and parallel execution of the functional objects (i.e. a relaxation operation). This *relaxation net* can be used to implement a constraint net.

To distributed the *relaxation net* function and value nodes are modeled as tasks of a Task Interaction Graph. To distribute this graph on a workstation cluster we use a combination of bisection [18] and the Kernighan-Lin algorithm [10].

The main operation on relaxation nets is a function, which computes a fixed-point. This function can be processed in parallel if the domain of the function can be partitioned in parts and the function itself can be partitioned in independent component functions (see [20, 7] for details and Appendix A). To use the parallel abstraction *relaxation net*, the application programmer implements subclasses of the value and function node classes and the structure class *relaxation net*, redefining some methods, i.e. implements a normal object-oriented sequential interface. There is no need for any explicit parallel programming (see Appendix B).

4.2 Another example — implementing distributed AI applications with NetCLOS

In distributed AI besides others the concept of communicating agents is present. Agent structures are not yet implemented with NetCLOS, but can be realized as follows. To implement an agent an active object can be used. On which host an agent proceeds can be fixed by the user or can be decided by the system (realized by a simple distribution scheme of round robin, see section 3.4), e.g. each agent can reside on a distinct object space. Furthermore, it is possible to add new agents dynamically. For diverse agents communication schemes, e.g. direct communication of agents or blackboard architectures, necessary message protocols can be implemented by NetCLOS messages. Concrete steps may be as follows: a virtual machine consisting of n object spaces is started from the master host. Agents possibly of distinct types are created by the master and distributed to the object spaces. A past-message e.g. *do-work* starts the action of each agent, which may perform different problem solving tasks. The agents work in parallel and may communicate by further messages to each other.

5 Experimental results

We tested NetCLOS by implementing a parallel abstraction named *relaxation-net* (see also [7]), which contains a net-like reference structure of active objects and a fixpoint operation on that structure. The net is distributed on a workstation cluster by the abstraction and the operation is executed in parallel on diverse parts of the net, i.e., distribution and parallel processing is done by the abstraction. This abstraction is used for implementing a local propagation algorithm for constraint nets, i.e., on this level only the sequential interface of the parallel abstraction must be known to an application programmer.

To get a gain of parallel execution of a NetCLOS program, one has to take high communication costs into account which are related to the infrastructure of a workstation net, e.g., an ethernet or TCP/IP. Thus, as usual in such a case, the computation time on one host should be high enough to compensate the communication costs. This is also the result of experiments we made. When solving a line-diagram labeling problem [16], we only got a speed up for constraint propagation when raising the number of constraints (see Figure 5). The speed up strongly depends on the communication traffic on the ethernet and on the kind of workstations used, which are typically heterogeneous (e.g. from Sparc Classics to Ultra Sparcs). The distribution strategy does not yet consider such kind of information. Because of using a derived not a dynamic structure the constraint net is first created on one object space and than distributed to the other. This still yields to high distribution costs, which are not included in the presented results. Furthermore, all experiments are executable on only one machine. However, the experiments show that one can get a speed up for constraint propagation, when using NetCLOS.

Fig. 5. Speed-up when increasing problem size (given here in number of stairs n of diagrams like the right one) and number of workstations. For 1000 stairs we got a speed up of 3.6 for constraint propagation on 7 machines. The number of constraints is $6n + 7$ and of variables is $4n + 7$.

Because NetCLOS is integrated in Common Lisp its programming environment (profiler, debugger, editor etc.) can be used also for each object space separately. To illustrate parallel issues of programming (e.g. communication costs) specific environment extension would be useful but are still not realized (see e.g. [15]).

6 Related work

The work on NetCLOS is derived from concurrent object-oriented programming languages related to actors [5, 1, 21]. Thus, the notion of active objects, proxies, asynchronous and synchronous message passing etc. are similar. However, our main interest is to integrate such concepts in Common Lisp and CLOS as a language used for AI applications. In NetCLOS the integration of concurrent object programming is done in the CLOS programming style by introducing new subclasses, metaclasses, declarative method combination, slot options, and protocols. Thus, a CLOS programmer can use NetCLOS without learning a new parallel language.

The extension of CLOS by active objects enables parallel object-oriented programming, and thus, parallel abstractions. Other approaches [22] introduce mainly function-oriented parallel programming in Lisp by allowing parallel execution of functional arguments. A precondition for these approaches is a side effect free programming style, which is not realizable in realistic Lisp applications. Furthermore, functional approaches often generate a big number of small tasks, which increase the overhead.

Another Lisp related implementation for parallel programming is Kali Scheme [2]. Besides very similar features like adress spaces, proxies, diverse communication primitives the main difference is that in Kali Scheme first class continuations and first class procedures are supported for programming in continuation-passing style. The integration of these concepts in Lisp without non-portable access to the Lisp implementation is not possible, because the lack of first class continuations and a metaobject protocol for the functional part of Lisp. However, our interest is more a practical one: First, we use Common Lisp instead of Scheme because of its use in application programming for realizing e.g. simulation, configuration, diagnosing, and information managment systems. Second, we use Common Lisp and add the extension modul NetCLOS to it instead of defining a new language to make it possible that existing Lisp programs can still be used.

Other approaches like CMLisp [6] introduces data parallel abstractions. This showed that programming with abstractions can simplify parallel programming, but CMLisp is restricted to run on single instruction multiple data machines (i.e. the Connection Machine 2) and thus, is hardly usable for workstation clusters. This is similar to [7], where a relaxation operation is introduced to solve constraint problems, but the implementation is done on a Sequent Symmetrie, not on a more common workstation cluster.

How NetCLOS can be used for Internet programming and how CL-HTTP can support this, is part of our current work.

7 Conclusion

A fully integrated concept and implementation (called NetCLOS) of a parallel object-oriented language is presented as an extension to the Common Lisp Object System (CLOS). With NetCLOS active objects, asynchronous and synchronous message passing, synchronization features, separation of parallel programming and distribution aspects, and transparent remote access are introduced in CLOS. These extensions are integrated in the CLOS programming style by extending generic functions, slot-options and metaclasses. Thereby, a virtual machine consisting of several Lisp images residing on a workstation cluster can be programmed. This is a new extension of Common Lisp in the direction of a parallel object-oriented language using active objects. Other approaches (like, e.g., [22]) extend the functional part of Lisp.

NetCLOS was used to implement a high level programming language based on abstractions for parallelization. The main point of this structure-oriented language is to introduce control abstractions, because multiple flows of control make parallel programming difficult. These control abstractions are realized by giving diverse predefined classes (like parallel-array, net, series) to the application programmer. These classes hide specific synchronization and load balancing schemes. A constraint system is implemented with this language where constraints and variables are distributed over a workstation net and proceed in parallel. For distribution a Task Interaction Graph model in coordination with bisection and the Kernighan-Lin algorithm is used. For an appropriate problem size a speed up for constraint propagation could be achieved.

NetCLOS as an extension to Allegro Common Lisp ACL 4.3 can be received from the authors. NetCLOS and parallel abstractions might be useful for AI programmers already working with Common Lisp and who want to use a workstation cluster for computation. Especially distributed and parallel applications can be tested with the virtual machine used by NetCLOS. To implement it in other Lisp implementations than Allegro, the implementation of light weight processes and the metaobject protocol must be assumed.

A Implementation of a parallel abstraction using NetCLOS

The implementation of a parallel abstraction (here the relaxtion net and the relaxation algorithm) should be done by a programmer familiar with concurrent object-oriented programming. Though it is easy for a CLOS-programmer to use NetCLOS some synchronizations have to be done on this level (see e.g. *wait-for*, *lock*).

The generic function *relax* is implemented as a past-message, i.e. can be processed in parallel. However, the distribution of the relaxation-net is totally separated from the object-oriented part shown here, by subclassing appropriate control abstractions (see Section 4). In *relax-net* for all function nodes (active objects) the function *relax* is called, which can be processed in parallel for each node (depending on the distribution strategy). Value nodes are reserved by the first function node which performs *lock*. *apply-function* is a part of the protocol for using the parallel abstraction. This function must reduce the value-nodes. The termination of the algorithm is controled by a simple count scheme realized by *acknowledge-count* (see [14]).

```
(defpargeneric relax-net :now (net))

(defmethod relax-net ((net relaxation-net))
  (loop for f-node in (function-nodes net)
      do (relax f-node)
      do (incf (acknowledge-count net)))
  (loop until (= (acknowledge-count net) 0)
      do (wait-for (acknowledge))))

(defpargeneric relax :past (f-node))

(defmethod relax ((f-node function-node))
```

```
  (if (> (acknowledge-count f-node) 0)
      (acknowledge *caller*)
    (setf (parent f-node) *caller*))
  (let ((values ()))
    (loop for v-node in (value-nodes-by-total-order node)
        do (lock v-node)
        collect (get-value v-node) into values)
    (setf values (apply-function f-node values))
    (loop for v-node in (value-nodes-by-total-order node)
        for value in values
        do (write-value v-node value)
        do (incf (acknowledge-count f-node))
        do (unlock v-node))))

(defpargeneric write-value :now (v-node value))

(defmethod write-value ((v-node value-node) value)
  (if (new-value-p v-node value)
      (progn (if (> (acknowledge-count v-node) 0)
                 (acknowledge *caller*)
               (setf (parent v-node) *caller*)))
    (set-value v-node value)
    (loop for f-node in (function-nodes v-node)
        do (relax f-node)
        do (incf (acknowledge-count v-node))))
  (progn (acknowledge *caller*)
         (setf (parent v-node) nil)))

(depargeneric acknowledge :past (acknowledgeable))

(defmethod acknowledge ((obj acknowledgeable))
  (decf (acknowledge-count obj))
  (when (and (parent obj) (= 0 (acknowledge-count obj))
             (acknowledge (parent obj))
             (setf (parent obj) nil))))
```

B Implementation of an AI application using parallel abstractions

To use a relaxation net for implementing e.g. a constraint net following implementation by an AI application programmer should be done. The implementation consists of subclassing the classes given by the parallel abstraction and defining new methods for specific generic functions belonging to these classes. By the *bisection-K-lin-distribution-mixin* the used distribition strategy for distributing Task Interaction Graphs is introduced. Other strategies may be inserted here.

```
(defclass constraint-net (relaxation-net bisection-K-lin-distribution-mixin)
  ()
  (:metaclass netclos-obj))

(defclass constraint (function-node)
  ((relation :accessor relation :initarg :relation))
  (:metaclass netclos-obj))

(defclass variable (value-node)
  ((domain :reader get-value :writer set-value
         :initarg :domain))
  (:metaclass netclos-obj))
```

```
(defmethod function-node-class ((net constraint-net))
  (find-class 'constraint))

(defmethod value-node-class ((net constraint-net))
  (find-class 'variable))

(defmethod apply-function ((constraint constraint) domains)
  (loop for i from 0 upto (length domains)
      for domain in domains
      collect (filter domain constraint i))))

(defmethod new-value-p ((var variable) domain)
  (< (length domain) (length (get-value var))))
```

References

1. G. A. Agha. Concurrent object-oriented programming. *Communications of the ACM, 33(9)*, 125–141, September 1990.
2. H. Cejtin and S. Jajannathan and R. Kelsey. Higher-Order Distributed Objects. *Toplas, 17(5)*, September 1995.
3. Chaco. *The Chaco user's guide: Version 2.0.* Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, NM, July 1995.
4. M. Dixon, J. de Kleer. Massively Parallel Assumption-based Truth Maintenance. *Proceedings of the AAAI 88*, 199–204, 1988.
5. C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence 8*, 323–364, 1977.
6. W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
7. K. Ho. *High-Level Abstractions for Symbolic Parallel Programming.* PhD thesis, University of California at Berkeley, 1994.
8. L. Hotz and G. Kamp. Programming the Connection Machine by using the Metaobject Protocol. *Parallel Computing, Trends and Applications*, North Holland, 1994. Elsevier Science Publishers.
9. J. Kahl, L. Hotz, H. Milde and S. Wessel. Automatic Generation of Decision Trees for Diagnosis: The Mad-System. *Int. Conf. on Information Technology and Knowledge Systems*, Vienna, Budapest, August–September 1998.
10. B. Kernighan and S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs.* Bell System Technical Journal 29, 121–133, 1970.
11. G. Kiczales, D. G. Bobrow, and J. des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.
12. H. Kitano (editor). *Second International Workshop on Parallel Processing for Artificial Intelligence, PPAI'93.* Elsevier Science Publishers, 1993.
13. C. V. Lopes, and G. Kiczales. *D: A Language Framework for Distributed Programming.* PARC Technical report, February 97, SPL97-010 P9710047, Xerox PARC, Palo Alto, Ca, http://www.parc.xerox.com/spl/projects/aop/tr-d.htm, 1997.
14. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
15. S. M. Miriyala, G. Agha and Y. Sami *Visualizing Actor Programs using Predicate Transition Nets.* http://yangtze.cs.uiuc.edu/, 1996.
16. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufman, San Mateo, California, 1995.
17. M. Papathomas. Concurrency in Object-Oriented Programming Languages. O. Nierstrasz and D. Tsichritzis, (edt.), *Object-Oriented Software Composition.* Prentice Hall, 1995.
18. H. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Proc. Conference on Parallel Methods on Large Scale Structured Analysis and Physics Applications.* Pergammon Press, 1991.
19. G. L. Steele. *Common Lisp The Language Second Edition.* Digital Press, 1990.
20. M. Trowe. *An Abstraction for Parallel Programming in Lisp on a Workstation Cluster.* Diplomarbeit in German. Universität Hamburg, 1998.
21. A. Yonezawa and J.-P. Briot and E. Shibayama. *Object-Oriented Concurrent Programming in ABCL/1.* ACM SIGPLAN Notices, 21(11), 259–268, 1986.
22. C. K. Yuen. *Parallel Lisp Systems.* Chapman & Hall, 1993.