Extended Abstract for
OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures

Designing an OO Reflective Language for Massively-Parallel Processors

—for Providing Efficient & Flexible Resource Management—

Hidehiko Masuhara Satoshi Matsuoka Akinori Yonezawa
Department of Information Science, University of Tokyo*

1 Introduction

Our premise is that *computational reflection* is a useful mechanism to construct parallel/distributed systems[19]. So far, we have proposed/implemented several object-oriented concurrent reflective languages[13, 14, 9, 7], and have had programming experiences in small-scale parallel systems. Recently, experimental/commercial MPPs(massively parallel processors) having thousands of processor elements with low-latency and fast network interconnections are becoming available. Along with such advances in hardware development, practical implementations of object-oriented concurrent programming languages on MPPs are becoming feasible[15, 12, 18], and large-scale scientific applications, which repeatedly create millions of objects, are being exploited[16].

One of the major problems in MPP programming is dynamic resource management, such as load-balancing and scheduling. In general, there is a following dilemma: simple management policies sometimes lead to lackluster performance under unexpected conditions, while complicated ones, which are effective under any situation, often impose considerable overhead. Thus, successful studies on dynamic resource management for parallel computers (1) have been specific to a restricted set of applications, or (2) have provided a set of simple management policies to adapt to the current phase of the program or the current state of the system. In both cases, the requirement for programming languages is the ability to easily provide abstractions for multiple management policies. Reflective languages provide such abstraction in a clean and consistent fashion, providing high degree of reusability via encapsulating the user-level resource management policies in the meta-level.

In this paper, we present a design of a reflective architecture of an object-oriented concurrent language for MPPs. The architecture is based on ABCL/R2[9], which is extended so that the user can control the policies of the parallel programming primitives. Using reflective architecture, the user can easily build his/her own dynamic resource management system — load-balancing, scheduling, etc. — at the meta-level, controlling the policies of primitives such as object creation, object migration, and scheduling. In order to validate the viability of our approach, we also apply our architecture to a load-balancing system for a realistic application: the system is tailored for or-parallel exhaustive search problems, and uses two load-balancing policies according to the state of the system.

*E-mail: {masuhara,matsu,yonezawa}@is.s.u-tokyo.ac.jp

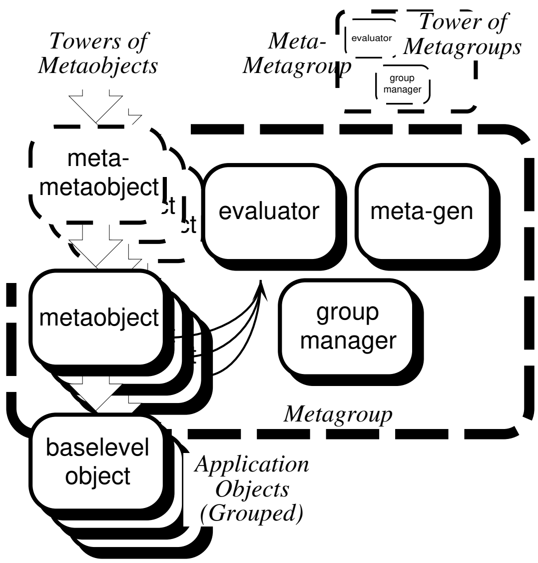

Figure 1: Basic Architecture

2 Dynamic Resource Management using Multiple Policies

Before presenting our reflective architecture, let us discuss the problem of dynamic resource management for applications on MPPs. An application on a MPP may have various computational patterns for message transmission and object creation depending on its execution 'phase.' For example, the main loop of an N-body simulation program[16] consists of two major phases; one is the construction of tree structure, and the other is the summation of the total force exerted. In most cases, different management policies are better for different computational patterns. Even for a single computational pattern, the desirable management policy could change according to the current state of the system; for example, an appropriate policy for object migration may change according to the number of objects in the system.

A general-purpose management policy that could be used for every possible computational pattern or for any system state will be marginally effective and/or have considerable overhead. A better scheme for efficient resource management would be: (1) to provide a suitable management policy for each (possible) computational pattern or state of the system, and then (2) to dynamically change the management policy according to the current computational pattern or the current system state.

3 Reflective Architecture for MPPs

We now present a reflective architecture for MPPs based on ABCL/R2[9]. The architecture is intended to provide a flexible meta-level abstraction to facilitate multiple policies of dynamic resource management. In the following subsections, the basic ABCL/R2 architecture, and the extensions for MPPs, are described.

3.1 Basic Architecture: ABCL/R2

ABCL/R2 is an object-oriented concurrent reflective language, based on the *Hybrid Group Architecture*[9], whose aim is to model meta-level encapsulation of control of limited resources. The overview of the ABCL/R2 architecture is shown in Figure 1; its characteristics are follows:

Metaobjects: For each base-level object, there is a corresponding metaobject at the meta-level. The metaobject mainly determines the structure of the object.

Group and Metagroup: A group is an abstraction of sharing limited computational resources. Objects in the same group share computational resources, which are represented as *group kernel objects* at the meta-level. The group kernel objects mainly determine the group behavior, such as script (i.e., method) execution. The default group kernel objects are: the object group manager which represents and manages a group, the object evaluator which executes scripts of the members of the group, and the object metaobject generator which creates a metaobject when a new object is created in the group.

Reflective Programming: There are two ways to perform reflective programming in ABCL/R2. One is through customization of metaobjects; since each object has its own metaobject, objects can be customized on a per-object basis. The other is through group kernel objects; since the group kernel objects are shared by the members of the group, changes to the group kernel objects affect the behavior of the group members.

From our programming experiences in ABCL/R2, we have found that there are still limitations in its reflective architecture is still limited with respect to handling of dynamic resource management on MPPs. For example, we have studied an adaptive scheduling system, which was programmed using a shared-memory multiprocessor extension of ABCL/R2[8]. From the experiments carried out under simulation, the scheduling system constructed using the meta-level programming features of ABCL/R2 exhibited 20–30% performance improvement for certain kinds of applications. However, this extension still lacked meta-level abstractions for distributed memory, and as a result, resource management on MPPs (e.g., load balancing) could not be dealt with.

3.2 Extended Architecture for MPPs

The reflective architecture of ABCL/R2 is extended so that the user can control the policies of dynamic resource management on MPPs. The extension is designed under the following principle[1]:

> No *mechanisms* of language facilities are reflective unless they are relevant to the *policy* control of dynamic resource management.

This principle is introduced for efficiency. With current compiler technology, the overhead caused by making low-level mechanisms reflective is still considerable. For example, in our architecture, the mechanism of the object migration (i.e., *how* the object should be migrated) is not reflective, while the user can control its policy (e.g., *when*). On the other hand, once the low-level mechanism of object migration is made reflective, various efficient implementation techniques (including message forwarding, address translation, etc.) could become unusable or considerably slow.

The overview of the architecture is shown in Figure 2. There is an ABCL/R2-like meta-system residing on each node (i.e., processor element). The functionality of meta-level objects metaobject, evaluator, and meta-gen are almost the same as the ones in the base ABCL/R2 architecture. Object scheduler implements a scheduling policy. Object node mgr., which corresponds to the 'group manager' in the base architecture, has several roles: (1) it decides on which node a new object is created; (2) it receives *asynchronous event messages* to cope with asynchronous events in the system; and (3) it exchanges load information with other node managers. The pseudo-code of a node manager is outlined in Figure 3.

More specifically, the architecture has the following characteristics:

[1] In most reflective languages, not all language facilities can be altered with respect to the low-level implementations. For example, in ABCL/R2, the message transmission mechanism is represented as another message transmission at the meta-level, and the underlying implementation as a function call/a queue manipulation is hidden from the programmer.

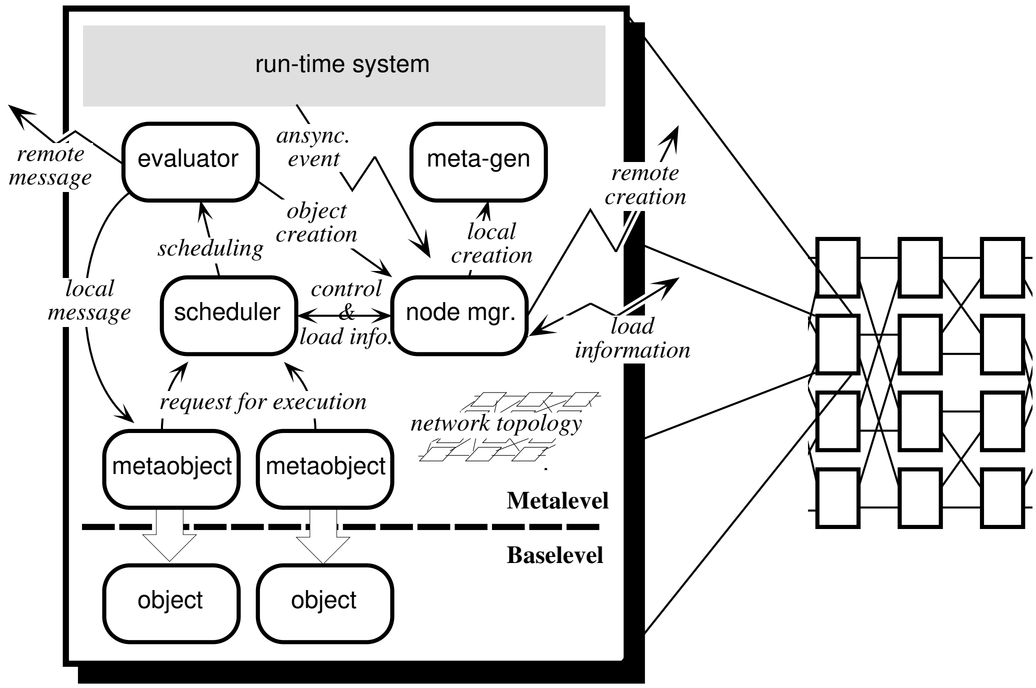

Figure 2: Extended Architecture for MPP

Transparent inter-node communication. At meta-level, a message transmission between objects on different nodes can be done transparently in the same manner as that between objects on the same node. Thanks to this property, inter-node negotiation for resource management can be easily and efficiently described. On the other hand, the implementation mechanism of inter-node communication cannot be altered, unlike many distributed reflective languages.

Object creation. An expression for object creation may designate[2] at which node it should be created. Without such designation, the node manager automatically decides an appropriate node — the local node for the default node manager(Figure 3) which could be altered by the user depending on the load distribution policy.

Scheduling. The meta-level object `scheduler` governs the intra-node object scheduling policy. It supplies the evaluator with executable scripts in a certain order, simultaneously specifying a time-quantum. The scheduler also monitors the load information of the node (e.g., utilization ratio of the evaluator and the number of active objects), and reports it to the node manager.

Object migration. The user can only control the migration policy at the meta-level; that is, *when* and *what* objects are to migrate to *which* node could be controlled by the meta-level programming, while we disallow the alternation of the low-level mechanisms for object migration (i.e., *how*).

Asynchronous event messages. Asynchronous events in the run-time system are notified as *asynchronous event messages* to the meta-level objects. Such events are used for resource management to judge the current state of the system. (The mechanisms for event management cannot be altered in our architecture.) Events notified as asynchronous event messages

[2] At the meta-level, a designation is notified as a parameter `:at` in scripts `:new` of the node-manager, whose default definition is shown in Figure 3.

```
[object node-manager
 (state [scheduler :=  scheduler] [node-id :=  node number]
        [local-meta-gen :=  metaobject-generator])
 (script
   ;; object creation
  (=> [:new ⟨parameter⟩ :at :local] @ r          ; local creation
    [local-meta-gen <= [:new ⟨parameter⟩ :scheduler scheduler] @ r])
  (=> [:new ⟨parameter⟩ :at target-node-id] @ r  ; remote creation
    [(node-manager-of target-node-id) <= [:new ⟨parameter⟩ :at :local] @ r])
  (=> [:new ⟨parameter⟩ :at :anywhere] @ r       ; default: local creation
    [local-meta-gen <= [:new ⟨parameter⟩ :scheduler scheduler] @ r])
   ;; Asynchronous Event Messages
  (=> [:timer-event time]  ... )
  (=> [:global-gc-start]  ... )
  (=> [:global-gc-end]  ... )
          ⋮
)]
```

Figure 3: Definition of the Default Node Manager

are: occurrence of global/local garbage collection, beginning/completion of object migration, timer events, etc. The user may also define his/her own event via user-level annotation.

Static program/hardware information. Information on compile-time program analysis, annotations inserted in an application program, and hardware configuration such as network topology, are available for meta-level programming. Such information is supplied as (1) asynchronous messages, (2) properties of metaobjects, and (3) reified first-class data (e.g., the network topology).

3.3 Discussions

Currently, the design of the extended architecture is not complete yet, and the following design issues are still open considerations:

- Design of the scheduler object. Currently, the internal structure of the scheduler object is not concrete. Further design is required to a modular metaobject protocol for schedulers.
- Facilities for reusable meta-level programs. To ease meta-level programming, good language facilities for reusablitiy are essential. The metaobject protocol design is of special significance.
- Introduction of Concurrent Aggregates (CA)[4] as a language facility. In meta-level programming, inter-node operations such as broadcast and reduction are frequently used. With Concurrent Aggregates, code fragments for such operations and for resource management — currently, they tend to be intermixed — can be clearly separated.

4 Example Applications

In this study, we will be examining a variety of parallel OO application to validate the effectiveness of our approach. Below, we present an outline of a dynamic load-balancing system for the or-parallel exhaustive search, which is one of the most basic class of parallel application where effective load-balancing scheme is required.

4.1 OR-Parallel Exhaustive Search

OR-parallel exhaustive search is one of the best-known examples where good dynamic load-balancing policies are effective, because of its high-degree of concurrency and the simplicity of the algorithm. The algorithm creates new objects as search subtasks[3].

```
[object search-task
 (script
  (=> [:search-start ⟨interim state⟩]
   (if ⟨it reached to a goal⟩
     [host <= [:found ⟨an answer⟩]]        ; report the answer
     (let ((children ⟨generate next interim states⟩))
       (dolist (child-parameter children)
        [(new search-task) <= [:search-start child-parameter]]))))))]
```

The load-balancing system must decide: (1) how to distribute tasks to all the processors; and (2) how to do so with minimal overhead. For this purpose, the load-balancing policies can generally be classified into centralized[5] and decentralized[11] one. In general, centralized policies can obtain good load balance, but task distribution is slow. On the other hand, the decentralized policies can quickly distribute tasks to all the nodes, but it incurs higher overheads to obtain a better balance.

Here, we propose a dynamic load-balancing system that appropriately switches between these two policies according to the current state of the system: (1) when most nodes have enough tasks, the centralized policy is used; and (2) when many nodes are idle, the decentralized policy is used. Furthermore, our system uses the notion of *generation*[11] to reduce the frequency of the monitoring process for inspecting the current state of the system[4].

Load-balancing is mainly performed by the node managers, whose pseudo-code is outlined in Figure 4. The behavior of the system is as follows:

1. A message `start-new-generation` is broadcast to all the node managers. In response to this message, every node manager (1) updates the generation counter, (2) replies its load information (e.g., number of tasks in the node) — this information is collected by the *answer-collectors* (`parent-load-collector` and `load-collector` in Figure 4), and (3) places a *sentinel*[5] into the scheduling queue.

2. After all the load information is collected from all the nodes, a global policy of new generation is determined. In the case of policy change, a message `change-policy-to` is broadcast to all the nodes.

   When the policy of the next generation is `:balance`, nodes that have enough tasks are notified IDs of nodes that are (nearly) idle.

3. The behavior of default object creation (script `new`) changes according to the current policy. When the current policy is `:distribute`, a new object is created on a remote node at some rate, and the target node is chosen randomly. When it is `:balance`, on the other hand, a new object is created locally except when there is a designation to create it on a specific node.

4. Eventually, a sentinel is scheduled at some node (script `sentinel-found`). The node then notifies the root node to start another updating of the generation (script `go-next-generation`).

[3] In this paper, we call a *node* on a search tree as 'task' or 'subtask' to avoid confusion with processor elements.

[4] Roughly, the interval of the generation updating, which invokes the monitoring process, is proportional to the number of tasks in the system. Detailed discussion on the genertaion can be found in [11].

[5] The sentinel is a boundary between generations. When the customized scheduler schedules the sentinel (i.e., all the tasks before the sentinel have been processed), it notifies the node manager.

5 Related Work

5.1 Distributed Reflective Languages/Systems

Several reflective languages/systems have been proposed for distributed computing. Although there are similarities between such reflective architectures and ours, the meta-level abstraction — what can be altered at the meta-level — is different. Our architecture is designed for flexible control of the dynamic resource management on MPPs, while they are designed to provide flexible mechanisms to implement abstraction in a distributed environment, which involves heterogeneity, fault-tolerance, etc.

- In AL-1/D[10], the mechanism for object migration can be altered. In other words, the user implements the low-level mechanisms for object migration such as message forwarding. On the other hand, the policy of migration, which is given as a list of constraints, is restricted, although it can be altered by the user to some degree.

- The design principle of RbCl[6] is to make as many low-level mechanisms reifiable as possible for maximum flexibility. On the other hand, it is not simple task to control policies of the facilities implemented by the user using the low-level mechanisms. Since such policy control operation is represented as a collective behavior of low-level mechanisms, the user must implement his/her own policies without conflict. Such discipline could be enforced by the metaobject protocol, but the MOP (metaobject protocols) for RbCl is not completely defined.

- In Open C++[3], no distributed facilities are provided in the default (base-level) language model. As a result, all such facilities are provided by the meta-level programming; i.e., they are implemented using the low-level mechanisms at the meta-level.

  An interesting feature of Open C++ is its optimization. Code analysis makes a method call in Open C++ almost equal speed to the one in native C++ in the best case. This optimization is suggestive to the compiler development, although such code analysis would be difficult in many reflective languages, which has more elaborate architectures.

- Apertos[17] is a reflective operating system for distributed environment. In Apertos, object 'migration' is a change in its *meta-space*, and is a primitive operation. The difference is that the purpose of object migration is for using different OS services, and not for load-balancing on MPPs. Moreover, inter-node migration mechanisms are open to the users.

5.2 Distributed Shared Memory

Some systems for distributed shared memory provide several memory coherence protocols and allow the user to use an appropriate one for each shared object[2]. Moreover, some systems can dynamically change the coherence protocol for a shared data.

5.3 Thread Scheduling

By providing an interface between the OS kernel and the user-level thread scheduler, significant performance improvement can be achieved[1]. Although this has similarity to the reflective system in terms of incorporating the user's policy to the kernel, there are several differences: (1) the performance improvement is mainly gained by the reduction of the interactions between the user-level and the kernel; and (2) policies that can be controlled are implemented only by a fixed set of call-back functions.

6 Conclusion

We have proposed a design of a reflective architecture for MPP based on ABCL/R2, and extended so that the policies related to the dynamic resource management are easily controlled. We also presented a load-balancing system, which uses two policies, implemented at the meta-level of our architecture.

This work is still in progress, and we are currently working on the following problems: (1) refinement of the architecture as discussed in Section 3.3; (2) other examples of dynamic resource management; (3) experiments with examples on real MPPs; (4) integration of our reflective architecture to our ABCL compiler.

Acknowledgements

We would like to thank Kenichi Asai and Jeff McAffer for numerous discussions. We would also like to thank Kenjiro Taura and Masahiro Yasugi for the daily discussions on ABCL compilers and parallel OO application programs, which inspired us.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[2] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Seattle, WA, Mar. 1990.

[3] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1993.

[4] A. Chien and W. J. Dally. Concurrent aggregates. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196, Mar. 1990.

[5] M. Furuich, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for OR-parallel exhaustive search programs on the Multi-PSI. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, volume 25, pages 50–59, Seattle, WA, Mar. 1990. ACM.

[6] Y. Ichisugi. *A Reflective Object-Oriented Concurrent Language for Distributed Environments*. Ph. D. Thesis, University of Tokyo, Mar. 1993.

[7] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 127–145, Vancouver, B.C., Oct. 1992.

[8] H. Masuhara, S. Matsuoka, and A. Yonezawa. A meta-level scheduling mechanism in ABCL/R2. In *Workshop on Object Oriented Computing (WOOC)*, Hakone, Japan, Mar. 1993. JSSST. (in Japanese).

[9] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[10] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 36–47, Tama City, Tokyo, Nov. 1992.

[11] R. Satoh, H. Sato, and K. Nakajima. A diffusional load balancing scheme on loosely coupled multiprocessors =LLS-G=. In *Joint Symposium on Parallel Processing*, pages 363–369, May 1993. (in Japanese).

[12] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1993.

[13] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 306–315, San Diego, CA, Sept. 1988. ACM.

[14] T. Watanabe and A. Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proc. REX School/Workshop on Foundation of Object-Oriented Languages (REX/FOOL)*, 1990.

[15] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object oriented concurrent computing on an extended dataflow supercomputer. In *International Conference on Supercomputing (ICS)*, Washington D.C., July 1992.

[16] M. Yasugi and A. Yonezawa. Towards performance evaluation of an N-body problem algorithm in an object-oriented concurrent language on a date-driven parallel computer. In *Workshop on Object Oriented Computing (WOOC)*, Hakone, Japan, Mar. 1993. JSSST. (in Japanese).

[17] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, B.C., Oct. 1993.

[18] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Tauro. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel & Distributed Technology*, 1(2):49–61, May 1993.

[19] A. Yonezawa and T. Watanabe. An introduction to object-based reflective concurrent computations. In *Proc. 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 50–54, 1989.

```
[object node-manager-for-exhaustive-search
 (state  ...
        [current-generation := 0]
        [current-policy := :distribute])
 (script
   ;; generation updating
  (=> [:start-new-generation new-G] @ parent-load-collector
        where (= (1+ current-generation) new-G)
    ;; if it is not a leaf, continue broadcasting
   (let ((load-collector (new answer-collector :parent parent-load-collector)))
      (dolist (child (children-of Me))
       [child <= [:start-new-generation new-G] @ load-collector]))
   [scheduler <= [:get-load] @ parent-load-collector]  ; report current state
   [scheduler <= [:put-sentinel new-G]]                ; put a sentinel
   (setq current-generation new-G))
   ;; end of generation
  (=> [:sentinel-found g] from scheduler where (= g current-generation)
   [(root-node-of Me) <= [:go-next-generation current-generation]])
   ;; start of updating generation
  (=> [:go-next-generation old-G] where (= old-G current-generation)
   [Me <= [:start-new-generation (1+ current-generation)]
       @ (new answer-collector :parent Me)])
   ;; receive global load state
  (=> [:load ⟨load information⟩]  ; from answer-collector
   (cond ((and (eq current-policy :balance)
                ⟨most nodes are starving for task⟩)
          ⟨broadcast "change policy to :distribute"⟩)
         ((and (eq current-policy :distribute)
                ⟨most nodes are satisfying⟩)
          ⟨broadcast "change policy to :balance"⟩))
   (when ⟨policy of next generation is :balance⟩
     ⟨inform nodes that have many tasks of starving nodes⟩))
   ;; policy control
  (=> [:change-policy new-P]
   (setq current-policy new-P))
   ;; object creation
  (=> [:new ⟨parameter⟩ :at :anywhere] @ r
   (if (eq current-policy :distribute)
     (if ⟨at some rate⟩
       [(node-manager-of ⟨randomly chosen node⟩) <= [:new  ... ] @ r]
       [local-meta-gen <= [:new  ... ] @ r])
     (if ⟨starving nodes are specified⟩
       [(node-manager-of ⟨one of starving node⟩) <= [:new  ... ] @ r]
       [local-meta-gen <= [:new  ... ] @ r]))))]
```

Figure 4: Definition of Node Manager for Or-Parallel Exhaustive Search