A proposal to improve reusability in a language based on the Occam-CSP model *

F. Araque(i), M. Capel(i), J.M. Mantas(ii), A. Palma(i)

(i) Dpto. de Lenguajes y Sistemas Informáticos
E.T.S. Ingeniería Informática
Universidad de Granada
Avda. Andalucía 38, 18071 - Granada
email: {faraque, mcapel, apalma}@goliat.ugr.es

(ii) Dpto. de Informática
E.P. Superior - Universidad de Jaén
Avda. Madrid 35, 23071 - Jaén
email:jmantas@apolo.ujaen.es

Abstract

The need to declare channels like global objects of the applications implemented according to the Distributed Programming Occam-CSP model limits the desirable *reusability* of the software-modules of the said applications. The integration of languages that follow the OO paradigm on this model is not easy either. In this work, easy solutions are proposed to augment the reusability of languages that follow the existing model based on a class of modules named ODAs and on a previously existing language that implement the above-mentioned modules in a multicomputer based on the Occam-Transputer platform.

1 Introduction

Eighteen years have passed since the *communicating processes* model of CSP[1] was proposed, and twelve since its most widespread distributed implementation, the Occam programming language [2]. On the other hand, it is very well known that multicomputers are nowadays the most important technological option to answer the increasing demand of speed and memory of current applications.

The inadequacy of the Ada language to be implemented efficiently, without severely restricting the language constructions in a distributed system, has failed important projects in this sense [3]. Therefore it is still not possible to count on Ada as a reference language for programming multicomputers. The absence of a general reference language for programming with this kind of architecture has been the cause of the overabundance of proposals of distributed languages in the last fifteen years. Many of these proposals have become widespread, like PVM [5], but they don't allow the verification of the total correctness of programs. Others, even though carried out by important scientists, e.g. Joyce [6], haven't achieved the necessary repercussion to be implemented in concrete architectures.

*This work has been financed by the project TIC94-0930-C02-02 of the Comisión Interministerial de Ciencia y Tecnología

None of the aforementioned problems affects the Occam-CSP model of distributed programming. Occam language is an *almost* complete implementation of the CSP model and has been used as a reference language to develop the SG-Thomson *Transputer* processor family (formerly, Inmos). The above-mentioned processors embody in *hardware* the Occam processes model, in such a way that any multicomputer, based on transputers, can execute programs developed for any transputer type (even of different *generations*). Thanks to the formal semantics of the language, derived from C.A.R. Hoare's works in the last decade[8], the basis for the development of Software Engineering tools that allow formal verification and program transformation has been established. As a result of all this, we can conclude that the Occam-CSP programming model is one of the best models for deriving high quality and secure programs for multicomputers.

Traditionally it has been argued that the lack of reusability of the Occam-CSP model is one of the drawbacks which avoids a widespread use of programming in this model. This paper proposes a possibility to overcome this drawback.

If we define *reusability* as the characteristic that allows a compiled module to be incorporated to in another program without the need of making changes, we can state that a module is reusable if it is self-contained, that is, it doesn't use variables or other objects defined externally from it. Obviously, the reusability of programs developed with the aforementioned model is limited because the code of a module, a process, etc. needs to access global channels which communicate processes in the same or in other processors. Nevertheless, this drawback may be resolved by separating the *configuration* of the channels from the programming of modules of the application, as will be demonstrated later.

The integration of Object Oriented Programming (OOP) concepts with the Occam-CSP model in order to obtain software reusability presents other drawbacks. The Occam-CSP model is mainly static (i.e. all the components should be fixed before program execution), as opposed to the *late binding* of the OOP. Additionally, the communication model of the Occam-CSP is a point-to-point synchronous message passing, whereas the OOP employs a *client-server* communication scheme. The paralell unit granularity is at instruction level in Occam, whereas the case of OO languages should be at a process or objects level. On the other hand, the object-based programming model assumes that aspects referred to configuration, distribution, mobility, etc, of the objects have to be hidden from the user programs. All this seems to point to the incompatibility of both models.

Our proposal attempts to show a technique to achieve reusability in distributed applications. This has been developed using a class of modules named ODAs that are used to encapsulate, among other information, abstract data types in SPMD-distributed applications. These modules and a language that supports them, have been presented in previous works [11], [12]. In our model, communications are carried out through global channels declared according to the Occam model, but object reusability is achieved because each ODA defines a separate module in which remote communications are specified, independent of other modules where the object implementation is programmed. Information contained in the ODA definition modules, concerning object remote communication, allows an application configuration to be automatically derived that can be transferred to an Occam2 program. This has been carried out on several platforms, at the moment, for a Telmat TN310 multicomputer with 16 T9000 processors.

In the next section, both the distributed programming model based on ODAs and its implementation in a Occam-transputer platform will be explained. In order to solve some of the problems arisen due to the direct implementation in Occam of applications based on ODAs, a previous proposal of a language that allows ODAs-description is used. [11],[12], whose most significant aspects are revised at the end of section 2.

In the third section, some of the aforementioned language characteristics that improve reusability are presented.

2 Programming model: distributed active objects

An ODA module will be implemented as an object whose code is replicated into all the nodes of the multicomputer. The different replicas of the object ("*ODA-replicas*" in the illustration) should be communicated among one another in order to:

- maintain the coherency of the global Abstract Data Type implementation
- implement the object's operations.

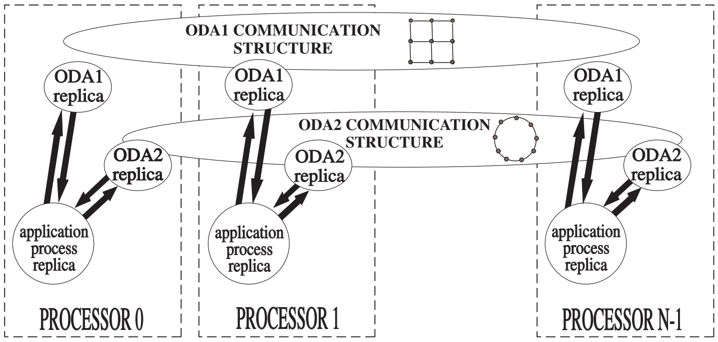

Figure 1

The communication code of the application is encapsulated in the ODAs employing the high level constructs of one specific language [12]. By means of messages between the replicas of the ODAs, a distributed computation model is achieved in that the application processes only carry out local communications and the ODAs are charged with remote communications. The way in which different ODA-replicas connect and communicate with each other will depend on the represented data type and consist of the object's own communication structure. An ODA offers to its clients (application processes and possibly other ODAs) a series of operations which operate on the implemented data type. These operations have a semantics of remote calls, i.e. one ODA-replica attends operation requests and the clients can remain hopeful for a message with results.

2.1 Distributed application building in Occam

In order to implement each ODA-replica in Occam 2, a generic procedure would be necessary whose formal parameters are:

- an array of channels (with no fixed size) for the communication with other ODA-replicas of the same type
- for each operation, an array of channels for the communication with both application processes and other client ODAs (the possibility of establishing client-server relations between different ODAs exists)
- for each ODA that could be a client of another, a pair of channels for each operation to be invoked

- two parameters that will contain the current values of each replica identifier (`REP.ID`) and the number of replicas (`NODES`).

In a similar way, the application process is described as an occam procedure whose formal parameters are as follows:

- for every ODA, a pair of channels for each operation
- the parameters `REP.ID` and `NODES`.

The procedures that implement both ODA-replicas and application processes are compiled and linked separately. The building of a complete distributed application is carried out in a configuration file, in which the parallel components to be executed in each processor are specified and the channels needed to correctly connect the different processes according to logical communication structure of the application are created. A *configuration phase* exists in which it is possible to determine the exact content of the said file, as can be seen in the following diagram.

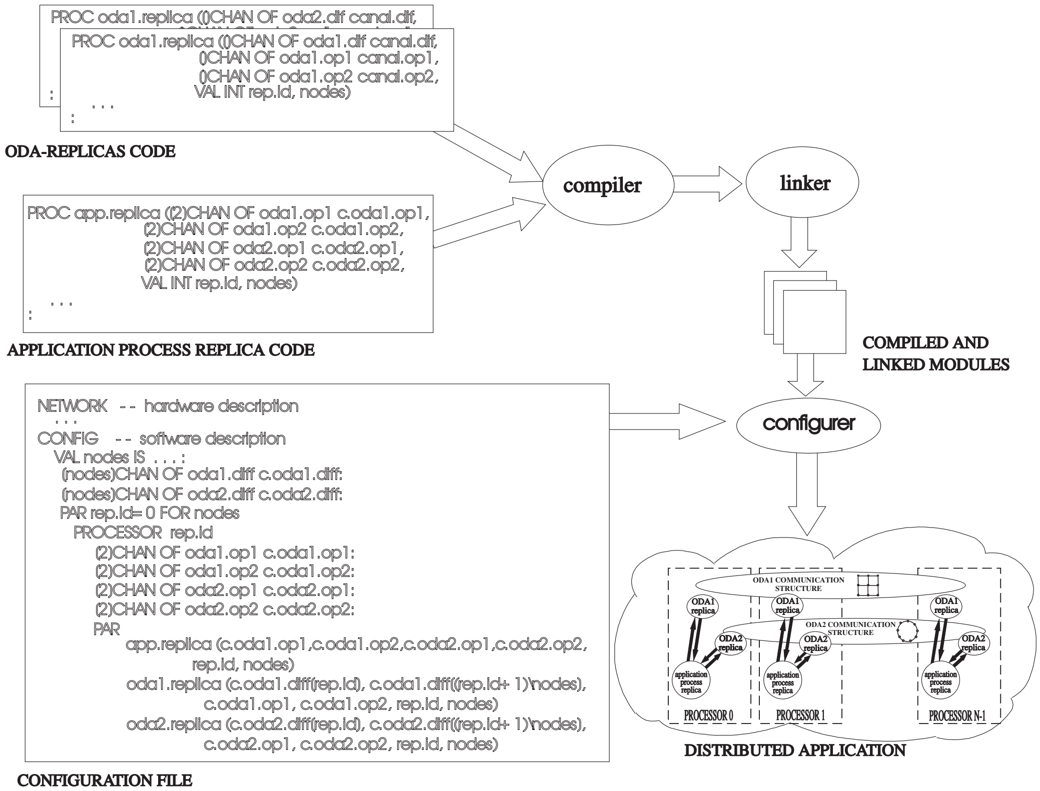

Figure 2

The existence of the configuration phase is an element that can be used to achieve the reusability of components compiled and linked to the distributed applications, even allowing to consider the possibility of having a library of said components.

2.2 ODAs description language

To use the ODA modules in the applications in a flexible way, a language to describe these modules has been proposed, which should allow the programmer of distributed applications to hide from the lowest level details, improving the programmability and code reusability.

The ODA description language has been shown in previous papers [11],[12]. In the notation of the language, the description of an ODA-module is carried out in two separate parts:

1. A definition part in which:
    - for every replica, channels that allow communication with other replicas of the same ODA assigned to different processors are declared
    - the ODA communication structure is described, that is to say, the way in which different ODA-replicas will be connected, making use of the communication channels previously mentioned
    - the interface for the ODA operations is specified in order to have a similar notation to the remote call channels of Occam 3 [14], conforming more to its semantics.
2. An implementation part, in which ODA-replicas code that implement the operations of the ODA are described.

We distinguish between the *ODA declaration* (the interface described in its definition part is imported to call its operations) and the *ODA instantiation* (an implementation part is assigned to the previously declared ODA). Declarations and instantations are made in a separate module: the application process. We decided that this module is not reusable, since it seems logical to us that a substitution of the ODA module implies a modification of the user application program code. Reusability is achieved at the ODA level, and the responsibility of generating automatically the configuration file is left to the *configurer*. Definition and implementation parts of the ODAs remain unlinked up to the point where they are declared and instantiated. This allows us to have different implementations for the same ODA-module interface.

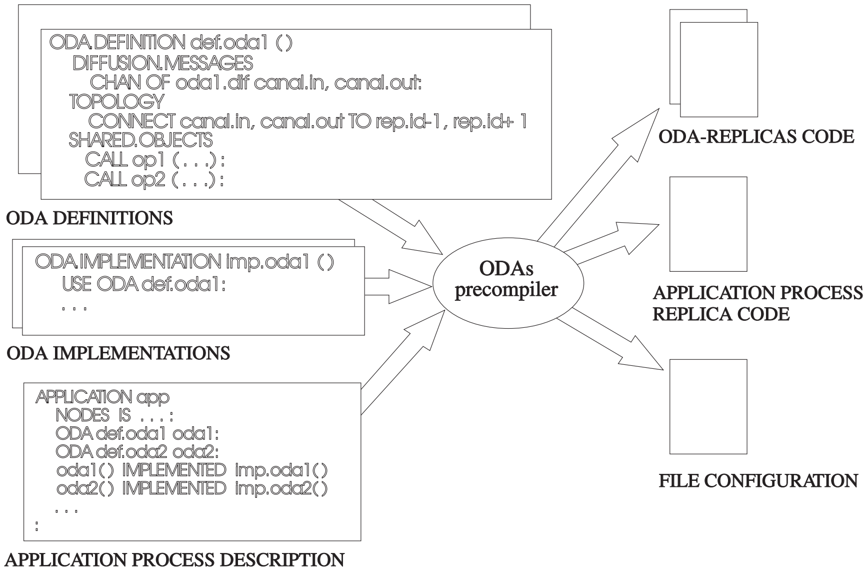

Figure 3

3 Issues of reusability in the language

3.1 Use of the same definition with different implementations

After declaring an ODA, a set of operations is offered to its possible users. In the following example, we introduce an ODA definition that represents a distributed integer with two operations `read` and `write`:

```
ODA.DEFINITION distributed.integer ()
  PROTOCOL
    integer IS INT:
```

```
  DIFFUSION.MESSAGES
    [NODES-1][2] CHAN OF integer integer.difussion:
  TOPOLOGY -- Full Connected
    DO i=0 FOR NODES-1
      DO j=0 FOR 2
        CONNECT integer.difussion[i][j] TO ((REP.ID+1)+i)\NODES
  SHARED.OBJECTS
    CALL read (RESULT INT data ):
    CALL write (VAL INT data):
```

Any possible implementation for `distributed.integer` will include a clause:

```
USE ODA distributed.integer:
```

We can have different implementations of ODA-modules that represent a distributed integer. These implementations assume both the same interface and communication structure as if they used the same definition. So for example, the most immediate implementation should allow us to manage a global variable of integer type:

```
ODA.IMPLEMENTATION global.integer.imp ()
  USE ODA distributed.integer:
  ... this implementation keeps updated an ordinary distributed integer
```

But we would be able to have other implementations that offer different functionability and that will be able to be used for determined applications. For example, it is possible to have an implementation of a global integer variable that only will be kept up to date in all the replicas when a value lower than the stored value in one replica is written. The interface and communication structure of this new implementation would be the same although its semantics would be slightly different.

```
ODA.IMPLEMENTATION smallest.imp ()
  USE ODA distributed.integer:
  ... This keep only the smallest written value
```

In the next application process, two ODAs, `e1` and `e2`, are declared with a `distributed.integer` definition. In the instatiation, we assign `global.integer.imp` implementation to `e1` and `smaller.imp` implementation to `e2`:

```
APPLICATION example
  NODES IS ...:
  ODA distributed.integer e1, e2:
  DO
    e1() IMPLEMENTED global.integer.imp()
    e2() IMPLEMENTED smaller.imp()
  SEQ
    e1{write}(0)              -- Give an initial value to e1
    e2{write}(known.value)    -- Give an initial value to e2
    ... Another things
```

The declared ODAs allow us to keep two types of global distributed variables of integer type, and both can be used in distributed applications according to the required functionability.

3.2 Topologies Redefinition

New definitions of the `TOPOLOGY` section can be created starting from previously existing definitions. For illustrating, in a easy manner, how this is possible, we consider the following definition of distributed object with ring topology:

```
ODA.DEFINITION Ring()
  PROTOCOL
    ring.protoc IS ...:
  DIFFUSION.MESSAGES
    CHAN OF ring.protoc before.c,next.c:
  TOPOLOGY
    CONNECT before.c,next.c TO (REP.ID-1) REM NODES, (REP.ID+1) REM NODES
  SHARED.OBJECTS
    CALL receive.value (RESULT INT value):
    CALL send.value (VAL INT value):
```

As can be seen in this definition, two channels `before.c` (to communicate with a previous node in the ring) and `next.c` (to communicate with the next node in the ring) are declared. The `TOPOLOGY` section establishes that these channels will be mapped onto a ring topology in the configuration phase.

From the `Ring` definition, it is possible to create an ODA definition with array linear topology in an immediate way:

```
ODA.DEFINITION Linear.Array REDEFINES Ring
  TOPOLOGY
    IF
      REP.ID=0            -- First node
        DISCONNECT before.c
      REP.ID=(NODES-1)    -- Last node
        DISCONNECT next.c
```

Using this mechanism, we can create ODA definitions from others previously existing in a hierarchical way. A topology redefinition doesn't add any new aspect to the ODA module interface, but only modifies object communication patterns.

We call the set of definitions a *family* of ODA definitions obtained from a given definition. ODA definitions in a family keep the same interface. The variations between one definition and another in a family only refer to the communication structure defined to communicate the different replicas of an ODA. In general, all the definitions in a family can be instantiated by an implementation that is compatible with communication features which characterize the said family.

3.3 ODAs Parameterization

In the ODA module definition heading, formal parameters (whose current value is fixed in the configuration phase) can be declared. The said parameters are recognized by:

- the definition part of the ODA module
- the implementation parts of ODAs which use the above-mentioned ODA module definition
- any application process that declares ODAs with the definition part in question
- the implementation parts of ODAs which use the services of some other ODA with this definition part in question.

The heading parameters of an ODA module definition, together with `NODES` and `REP.ID`, allow the description of a "skeleton" of the distributed object communication structure that will be completed when the said parameters take their current values in the ODAs instantiation.

In order to illustrate the application of many of the presented ideas, we will show some details about the implementation of an n-way filter algorithm in a multicomputer. We will implement a particular instance of the algorithm in which every leaf node in an abstractly defined tree (with a fixed branch degree for all the nodes) takes positive integer values from a file and distributes the said values toward its parent node. Internal nodes in the tree compute the largest values among those obtained from its sons and send the result to its parent node.

In order to encapsulate communication requirements of the algorithm, we build the following ODA-module definition that represents a type of distributed object with uniform n-ary tree. In this definition, the branch degree of each node (n=branch degree), is declared as a formal parameter in the heading.

As can be seen in this definition, an array of channels whose sizes are equal `branch.degree` parameter (to communicate with the sons of current node), a channel `parent.c` (to communicate with the parent node) and another two channels, `before.c` and `next.c`, are declared. These channels will be mapped onto a uniform n-ary tree topology.

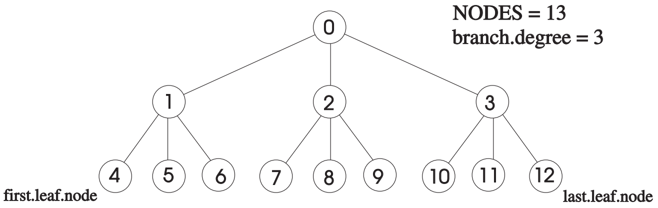

Figure 4

```
-- Uniform n-ary tree
ODA.DEFINITION n.tree (VAL INT branch.degree)
  PROTOCOL
    n.tree.protoc IS ...:
  DIFFUSION.MESSAGES
    [branch.degree]CHAN OF n.tree.protoc son.c:  -- communication with sons
    CHAN OF n.tree.protoc parent.c:         -- communication with parent
    CHAN OF n.tree.protoc before.c, next.c: -- communication with adjacent nodes
  TOPOLOGY
    DO
      IF
        is.internal.node
          DO j=0 FOR branch.degree
            CONNECT son.c[j] TO left.son.node + j
      IF
        NOT is.root.node
          CONNECT parent.c TO parent.node
  SHARED.OBJECTS
    CALL receive.subproblem (RESULT INT subproblem):
    CALL send.subproblem (VAL INT subproblem):
```

The `branch.degree` parameter, together with the `NODES` parameter, allows to define objects with tree topology that can be adapted in a flexible way for multiple distributed applications which request services of an object with this communication structure.

Once ODA definition has been specified, an implementation compatible with it, that shows the required communication in the filter algorithm, should have to be defined.

```
  ODA.IMPLEMENTATION filter.imp()
```

```
USE ODA n.tree:
...
```

Suppose that we wish to incorporate a graceful termination detection mechanism in the filter processes which will be initiated when any given leaf node receives a particular type of integer data (e.g. any negative integer). In this case, we need a communication structure that permits finishing properly the leaf nodes so that later the internal nodes are finished in an orderly way as well. This is accomplished by connecting leaf nodes in a ring shape. This can be achieved using a type of ODA in `n.tree` family and assigning `before.c` and `next.c` channels in a leaf nodes level.

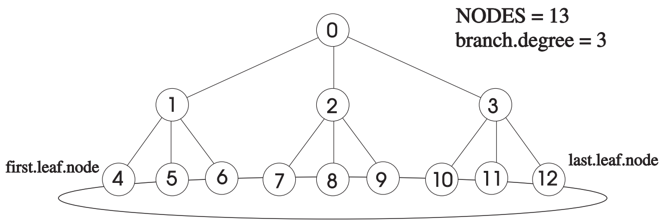

Figure 5

```
ODA.DEFINITION threading.n.tree REDEFINES n.tree
  TOPOLOGY
    DO
      IF
        is.leaf.node
          IF
            (REP.ID<>primer.nodo.hoja) AND (REP.ID<>ultimo.nodo.hoja)
              CONNECT before.c,next.c TO before.node,next.node
            (REP.ID=first.leaf.node)
              CONNECT before.c,next.c TO last.leaf.node,next.node
            (REP.ID=last.leaf.node)
              CONNECT before.c,next.c TO before.node,first.leaf.node
```

4 Conclusions

Easy solutions have been established to augment the reusability of software component developed with a language that follows the Occam-CSP model. The said solutions consist essentially of three *features of reusability* that can be applied in the distributed applications development.

1. reusability based on the definition separated from the interface: consists of replacing ODA module implementation, so it always keeps the same interface and object communication structure that the said module represents.

2. reusability based on the adaptation of the communication structure: consists of redefining statically the communication topology of an ODA, affecting the aforementioned communication structure, allowing the same implementation to remain.

3. reusability based on ODA module parameterization: consists of parameterizing the heading of the definition modules of the ODAs; these parameters will be instantiated in the configuration phase of a given application.

Also, we have tried to show the *ODA description language* proposed in previous works of the group in order to apply in a practical way a similar concept to the distributed abstract data type, which is necessary to obtain the integration between the object-based languages and others that follow the Occam-CSP model.

5 References

1. C.A.R. Hoare. "Communicating Sequential Processes". Communications of the ACM, v.21, n.8, pp.666-677, 1978.

2. Inmos, Lmtd. "Occam Programming Manual". Prentice-Hall, London, 1984.

3. Atkinson, C., Moreton, T. Natali, A. "Ada for distributed systems". Cambridge University Press, 1988.

4. Baumgarten, U. "Distributed systems and Ada-current projects and approaches comparative study's results". Ada Europe international Conference, Athens, Greece, May 1991, pp.260-278.

5. Geist, G.A., Sunderam, V.S. "Experiences with Network-Based Concurrent Computing on the PVM System". Concurrency: Practice and Experience, v.4, n.4., pp.293-311, 1992.

6. Brinch-Hansen, P. "The Joyce Language Report". Software Practice and Experience, v.19, n.6, pp.553-578, 1989.

7. Brinch-Hansen, P. "Super-Pascal a publication language for parallel scientific computing". Concurrency: practice and experience, v.6, n.5, pp.461-483, 1994.

8. Hoare, C.A.R. "Communicating Sequential Processes". Prentice-Hall, 1985.

9. Brookes, S.D., Hoare, C.A.R. ,Roscoe, A.W. "A Theory Model for Communicating Processes". Communications of the ACM, v.32, n.3, pp.560-599, 1984.

10. Goldsmith, M.H., Roscoe, A.W., Scott, B.G.O. "A Semantic Model for Occam 2". Transputer Applications and Systems'93. IOS Press, 1993, pp.965-980.

11. Capel, M., Troya, J.M. "An Object Based Tool and Methodological Approach for Distributed Programming". Software Concepts and Tools, v.15, n.4, pp.177-195,1994.

12. Capel, M., Troya, J.M., Palma, A. "Distributed Active Objects: A Methodological Proposal and Tool for Distributed programming with Transputer Systems". Microprocessing and Microprogramming, v.38, pp.197-204, 1993.

13. Sakdavong, J.C., Bahsoun, J.P., Feraud, L. "Configuration Classes: An Object Oriented Paradigm for Distributed Programming". IWOOOS'95, Lund, Sweden, Aug.1995. IEEE proceedings., pp.200-209.

14. Geoff Barret. "Occam 3 reference manual". Inmos Lmtd, 1992.