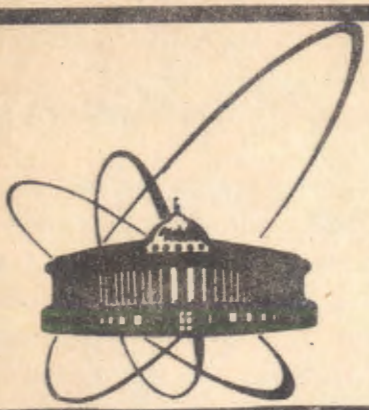
91-432



сообщения
объединенного
института
ядерных
исследований
дубна

E11 91-432

L.Kotulski*, W.Kaczorowski*, J.Rosek*

THE SEPARATE COMPILATION OF THE COLNET LANGUAGE

*Institute of Computer Science,
Jagiellonian University, Cracow, Poland

1991

1. Introduction

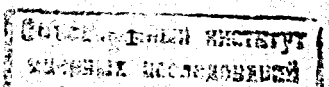
A quick development of computer systems and networks, plainly visible in the past years, has increased both a computational power and the scope of computers' usage, which caused intensification of researches for methods of creating concurrent software in distributed networks.

Owing to the carried researches, a number of concurrent programming languages have appeared. Some of them got more popularity considering either an innovatory concept (for example CSPL [1], DP [2]) or possibility to be used practically (for example Concurrent Pascal [3], Modula [4], Mesa [5], Ada [6], ARGUS[13]).

The analysis of the existing programming languages carried in [8,14] shows that most of them do not regard a specificity of computer local networks, where hardware configuration undergoes frequent changes. Particularly, these languages do not support the (dynamical) modification of the distributed system modules allocation without the necessity of recompilation of the whole concurrent program. The problem of multicompilation, in the case of heterogeneous hardware architecture of particular nodes, is not considered, too. Inconvenience mentioned above is the reason for undertaking a work on a system for supporting a concurrent software implementation for distributed systems COSID (COncurrent Software Implementation for Distributed systems) [9].

The basic idea of this system is a two level model of generation of concurrent software which includes:

- compilation level, which supplies the COLNET language (COncurrent Language for NETworks) [10], appropriate for creating concurrent programming in networks, in the compilation level software modules are defined and separately compiled,



- allocation level, on which ASNET system performs allocation in pointed computational nodes of software modules generated from patterns compiled on the previous level.

The whole information which serves a control of mutual relations between modules being compiled is maintained in a structure called environment description. A special module of the compilation level (called EDMM - Environment Description Module Manipulator) [11] is responsible for maintaining environment description information. The EDMM allows one also to insert resulting modules descriptions to software environment description without any need to recompile them. Created this way, data base for the set of modules, which are currently compiled, is a bridge between the compilation level and the allocation level of model of software generation mentioned before.

The aim of the paper is the presentation of basic mechanisms introduced in COLNET language in order to solve the problems with the separate compilation in such a two level concept.

2. The Elements of COLNET Which Enable the Separate Compilation

A unit of compilation in the COLNET language [10] is one of two modules:

- module of manager type, defined as a certain data structure and set of operations on this structure, which is an extended conception of Hoare monitor [12],
- module of process type, which can call any of entry procedures, generated from a certain pattern.

The call of manager's entry procedure in a given unit has the following syntax:

call <loc_name>,<proc_name>[<list_of_act_parameters>)].

The loc_name should be associated with the unit_pattern_name, from which manager supplying the called procedure will be generated. It is made with the help of external modules declaration made in the following form:

external <loc_name>:<unit_pattern_name>[=<global_name>];

Referring to the manager by a local name allows one, among other things, to call procedures of two different managers generated from the same pattern of a pointed unit. It is also possible to specify a global name associated with the generated manager, used in the course of allocation.

As a general rule, in programming languages it is obligatory that a unit of compilation which supplies an information for another unit of compilation is translated before this information is used. This principle forces one to accept certain order of compilation of particular units. This principle arises from context relations of their specifications. For example, in the Ada language the basic instrument to define an order of compilation is a declaration "with". The specification of each compilation unit, mentioned in the "with" declaration must be compiled before the unit including this declaration [7].

In the COLNET language, the concept of a unit superior to a module of either the manager or the process type does not exist. Any instruments imposing the order of compilation do not exist either. Compilation of units may take course in any order, but if a unit, which the unit compiled at the moment refers to, has not been compiled yet, then the compiler creates the so-called pseudo-pattern, which contains a list of demands (a list of procedures and descriptions of type

declarations together with their actual parameters). In the course of compilation of the unit, of which the pattern has already been generated, a test of its consistency with the pattern being currently constructed is performed.

On the other hand, the fact, that an environment description contains a manager pattern, which the unit being compiled refers to, enables the compiler to control fully the consistency of the corresponding lists of actual and formal parameters with regard to both, their length and type declarations.

Every inconsistencies deliver some information to a unit which is compiled and according to system designer's requirements, there takes place rejection of the already generated pattern or forcing the pattern of this module generation. If the prior is the first case, then usually a correction of detected mistakes and repeated compilation of the unit is performed. Instead, if a description of a unit pattern, which does not comply with the expectations implied from already compiled modules is forcefully added to the environment description, then this disturbs the consistency in the base.

Consequently to secure the consistency of compiled modules in the base, all module patterns, which are not consistent with the compiled module must be replaced with their pseudo-patterns. For example the above situation takes places during the repeated compilation of the module which is to be changed, if for example the number of parameters of some procedure has changed.

All the compilation units, which invoke this module are going to be inconsistent with it and must be recompiled. If, however, changes in the unit had not influenced the feedback, the algorithm had been changed but procedure's specifications (parameters and result) had not, then recompilation would not have influenced the other units.

3. The Logical Structure of a Unit Pattern

Beside import and export descriptions, which are essential for a control of invoking correctness, a description of the compiled unit must include information, which allows the allocator to find, weather in the given node the allocation of the module generated from a given pattern is possible. Consequently regarding all attributes, which any module in the COLNET language should have the description of its pattern contains:

(a) module identification, which contains:
- pattern type (manager, process, pseudo),
- pattern name, i. e. identifier of compiled module,
- global names defined as objects of the process or manager type, which are suggestions for modules names, which may be generated from the given pattern in the course of allocation,
- computer systems names, corresponding to the compiled module in the sentence of version or of exclude, where these names characterize nodes, in which allocation of the given module is (version) or is not (exclude) possible,
- name of the file, which contains result module,
- size of the result module,
- relative address of the initial point of the code,
- list of the names of importers of the given module.

(b) imports description, containing the following information:
- local name of manager, whose entry procedures are called by currently analyzed module,
- name of pattern of manager mentioned in the last point,
- list of invoked entry procedures and description of the actual parameters,

(c) exports description, which contains:
- list of entry procedures of the compiled manager,
- together with each entry procedure
- list of pointers to definitions of types of the formal parameters,
- file name containing interface module.

As have been mentioned in the case, when a currently compiled unit invokes modules which do not have their patterns in the base, after compilation is finished their pseudo-patterns are inserted to the environment description. The structure of the module's pseudo-pattern is alike, but contains only description of demands to call procedures, which are defined in the calling unit.

In the presented model of the separate compilation of the COLNET language the EDMM module (Environment Description Module Manipulator) [11] is of an essential importance. This module contains a set of procedures, which store in a disk file the descriptions of patterns and pseudo-patterns, acquired during previously performed compilations of modules of the software environment which is to be designed.

4. Co-Operation of the Compiler with EDMM

The basic operations performed on a system environment description in the course of compilation are:

(a) getting of a module description to the operating store,

(b) maintaining (modification) of module description in the file,

(c) test of consistency of import procedures by currently compiled module together with their formal definitions maintained in the description of the module exporting them or generation a pseudo-description for those procedures, based on a description of types of actual parameters,

(d) test of the consistency for procedures exported by a currently compiled module together with their pseudo-descriptions, generated during compilation of the importers of the given module.

The file containing the system environment description has a specific logical structure and is processed only under control of the EDMM module. This file must be open before any operations on the environment description are performed. This is realized by the procedure openedmm:

```
int openedmm (xname)
  char *xname; /* name of the file, which stores the
                  environment description*/
```

This procedure opens the already existing file, which stores the environment description or creates one, if it has not existed in the system yet. At the beginning of this file, parameters defining the contents of the file are placed:

- pointer to the description of the first module (they are arranged lexicographically with regard to identifiers),
- pointer to the module, which is currently compiled,
- pointer to a global list of definitions of the data types
- pointer to the ring of free areas (released and taken in the course of operations performed on the environment description).

The operation of closing of the file storing the system environment description is realized by a non-parametrical procedure: closedmm(). When this file has been closed no operation dealing with the file containing the environment description is permissible.

Most of operations performed on the environment description refer to a specific module - created, modified or verified under control of EDMM; to reduce the number of parameters of procedures of the EDMM module and facilitate

automatic processing of the environment description base, an obligation to define a work context (of the used module description) by programs using the environment description has been introduced. The definition of such a context consists in giving module name, its type and a specification of a way of access to its description - this operation is realized by the procedure with:

```
int with (who, wtype, class, op)
  char *who;  /* module name*/
  int  wtype, /* module type: process, manager*/
       op;    /* po: create or use*/
```

Procedure with - creates (create) or finds (use) in the file the module description: who, and remembers pointer to it. Hereafter the module who is accessible for processing (permissible) operations on it. The result of the procedure specifies the way, which the procedure has been realized, where:

- result > 0 - means a correct termination of the procedure and informs, that a description of module which has the statute: normal (pattern) or pseudo (pseudo-pattern),
- result<= 0 - means an erroneous performance of this procedure (usually lack of place in the system disk, preventing creation of the module description).

Considering the meaning of the full system environment description for the correctness of all stages of the definitions built by the COSID system, and particularly to secure the correctness of an independent compilation of modules - as a principle foundation is assumed, that all descriptions stored in each base containing such descriptions are fully consistent.

Because of this foundation - all operations performed on the environment description should not allow any inconsistency, in the base describing environment, to happen. This concerns, first of all, operations of adding new modules descriptions and canceling the old ones. That is why while such operations are performed, a verification of the full consistency of the environment description base should be done. Practically such a verification concerns only those descriptions, which are somehow related to the module adding to the base or canceling.

Verification of modules adding to the base is realized by procedure verify , which tests the consistency of the pointed module with the other modules inserted in the environment description.
This procedure has the following heading:

```
int verify(op, who, pr)
  int  op,      /* op  : soft, hard*/
       pr;      /* pr  : print, nonprint*/
  char *who;    /* who : name of verified module*/
```

Verification consists in:

(a) test the consistency of calls of all procedures imported by the who module with their formal definitions, placed in the modules, which export them,

(b) test the consistency of the definitions of procedures exported by the who module with demands of modules, which import them.

A new version of module description is usually added to the global list of modules, replacing, in case of need, already existing version of description or pseudo-description; if any inconsistency of the definitions of procedures exported or imported with the actual definitions placed in the base happens, then EDMM takes a decision, which depends on the value of the op parameter of the verify procedure.

If this parameter has a value of:

soft - then the pattern description is not added to the base,

hard - the pattern is added to the base the forceful way, which means, that all patterns placed in the base as well as any ones inconsistent with demands of the who module will be saved in an auxiliary list of patterns, and replaced by pseudo-patterns, consistent with the other descriptions remaining in the base.

Beside the global verification at the moment of either adding or canceling a possibility of performing a partial verification of the elements of description of the given module (pointed by the with procedure) is accepted. This concerns particularly the consistency of the formal definition of a procedure exported by one module, with its assumptive definition, constructed on the base of the call in a module importing this procedure.

This purpose serves a procedure compatible:

```
int  compatibility (px, t,ename, impname)
 int  t;            /* type of the structure being compared*/
 char *ename,       /* procedure name*/
      *impname;     /* name of the module, which exports the
                       procedure*/
 struct mem *px;/* address of the structure, which describes
                   the call*/
```

The task of the compatibile procedure is to test the consistency of types of all actual parameters with their formal definitions, specified in the module, which exports them. Considering this the compatibile procedure finds the description of the importer pattern impname (if such a

description does not exist - its pseudo-pattern is created), finds the description of the ename procedure (if such a description does not exist then creates a pseudo-description based on the description of the call of this procedure in the currently copied module) and afterwards tests the consistency of the formal description (or pseudo-description) with the description based on the description of the call of the ename procedure in the currently copied module.

5. Summary

One of the principle foundations considering a system of generation of the concurrent software for local networks COSID, was to separate a stage of generation of particular software modules , of generation the concurrent system as a whole. This idea has led to a design and implementation of the separate compiler of the COLNET language, which uses a set of procedures for software environment service and contains descriptions of patterns of compiled units.

Moreover, the set of descriptions of patterns of the compiled modules composes a base the allocation system to work, allowing the generation of the concurrent system. Independent compilation of separate modules of the currently designed concurrent system, as well as the possibility of verifying the whole system repeatedly without any need for full recompilation, allows one to increase the efficiency of the process of designing and implementing the concurrent software. Particularly valuable is the possibility of verifying incomplete concurrent system, including the possibility of getting information concerning demands of imported procedures of lacking modules.

6. References

[1] Roper T.J.: A Communicating Sequential Process Language and Implementation, Software Practice and Experience, vol 11, (1981), 1215-1234.

[2] Brinch Hansen P.: Distributed Processes: a concurrent programming concept, Comm of ACM, 21, (1978), 934-941.

[3] Brinch Hansen P.: The Programming Language Concurrent Pascal, IEEE Trans, on Soft, Eng. 1, 2, (1975).

[4] Wirth N.: Modula - A Programming Language For Modular Multiprogramming, Software Practice And Experience, vol 7, (1977), 37-52.

[5] Mitchell J.G.: Messa Language Manual, Report #CSL-79-3, Xerox Corporation, Palo Alto Research Center, California 1979.

[6] Reference Manual for Ada Programming Language, ANSI/MIL-STD-1815A-1983.

[7] Ian C. Pyle.: The ADA Programming Language - A Guide for Programmers, Prentice_Hall International, Hemel Hempstead, Hertfordshire, England.

[8] Kotulski L.: A Too Level Aproach To The Implementation LANs Software, in preparation.

[9] COSID - COncurrent Software Implementation For distributed Systems, Reports of the Institute of Computer Science, Jagiellonian University, Cracow 1989.

[10] Kotulski L. Rosek J.: COLNET - The Language Description, Reports of the Institute of Computer Science, Jagiellonian University, Cracow 1989.

[11] Kaczorowski W.: Project of the EDMM module maintaining a Software Environment Description, Reports of the Institute of Computer Science, Jagiellonian University, Cracow 1988.

[12] Hoare C A. R.: Monitors - an operating systems structuring concept, Comm of ACM, 17, 10, (1974), 549-557.

[13] Liscov B.: ARGUS Reference Manual. Programming Methodology Group memo 54, MIT Laboratory for Computer Science, Cambridge, MA, Marth 1987.

[14] Kaplan S.M., Goering Sl.K.: Visual Concurrent Object - Based Programming in GARP. LCNS, pp. 165-180.

Received by Publishing Department
on October 1, 1991.