Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
*SIGPLAN Notices Vol. 19, No. 6, June 1984*

# STATIC CHECKING OF INTERPROCESS COMMUNICATION IN ECSP

F.Baiardi, L.Ricci,
M.Vanneschi
Dept. of Computer Science
University of Pisa

ABSTRACT

The structure of a compiler for the ECSP language is described. ECSP is a concurrent language extending Hoare's CSP: it supports dynamic communication channels and nested processes. The compilation of ECPS programs is obtained by the composition of several tools of minimal functionalities.

A set of static checks on interactions between concurrent processes is described. The checks verify the mutual consistency of the interfaces of processes: an interface is given by a set of input/output channels connecting a process to its partners. It is shown that the amount and the coverage of checks depend on the entities referred to in interprocess communication constructs and that both increase with the adoption of explicit naming.

The checks on process interfaces are carried on in several tool of the compiler front-end to achieve machine independence. To support separate compilation, each tool can be applied to a subset of the processes of a program.

This work has been supported by CNR, National Program in Computer Science,Mumicro project and by Honeywell Information Systems Italia.



## 1. INTRODUCTION

Several concurrent languages based on the message-passing, or local environment, model of cooperation have been proposed in the last years /Ada 83, Bryant 78, Feldmann 79, Hansen 78, Hoare 78, Liskov 83, Mao 79/. The growing attention to these languages does not always correspond to a similar attention to the problems arising in the definition of programming environments for their support. In particular,the subject of compilation needs further research efforts. A compiler for a concurrent language is usually implemented by a set of extensions to an existing compiler for a sequential language. Using this approach, it is difficult to perform all the checks on the concurrent statements, required by the language semantics to verify the correctness of interactions. This may result in an increase of the debugging complexity as well as of the software development cost. A remarkable exception is the compiler for the PLITS language /Feldman 79/ that checks the types of data exchanged between processes and generates code to evaluate assertions on the values

of messages.

We believe that the methodology of deriving the compiler definition from the semantic definition of the language, such as /Bjiorner 77/, can be adopted also for concurrent languages, when a suitable semantic model is chosen. In our opinion the model to be adopted is the β-semantics /Francez 79/ since it supports a sharp distinction between the sequential and the concurrent aspects of a language. The work /Baiardi 82/ shows that the compiler definition can be given in terms of the operators of the β-semantics model.

This paper will focus on a single problem in the definition of a compiler for a message-passing language, namely the checks that can be executed on the interface among processes and the corresponding set of errors that may be detected. This problem is strongly related to the nature of the entities referred to in interprocess communication constructs: they may be process names or shared objects acting as communication media. This distinction is meaningful only when the communication partners can be selected by run-time decisions /Hoare 82/.
Section 2 will discuss the problem of naming in message-passing languages taking into account the aspects of homogeneity of the constructs and protection of the interactions. Section 3 will briefly present ECSP, a language that significantly extends Hoare's CSP /Hoare 78, Francez 79/ and designed according to the principles discussed in section 2. ECSP is a message-passing language oriented towards real time applications with high robustness requirements. Section 4 present the structure of a compiler for ECSP. Each tool is then described together with the checks it performs on interprocess communications and on the constructs to dynamically update the interfaces among processes.

## 2. NAMING IN A MESSAGE-PASSING LANGUAGE

The simplest interprocess communication primitives allow processes to exchange information through a single input queue associated with each process. All messages directed to a given process share this queue even when their contents are of different types or they come from distincts partners. Of course this forces the receiver to interpret the contents of each message at run time, and does not guarantee an high degree of protection since the message queue can be accessed by any process. In this case the compiler can execute only very simple checks, since communication is almost completely solved at run time.

A more structured view of interactions can be obtained when we see the processes of a program as connected by logical channels that allow a subset of processes to send messages to another distinct subset of processes. Each channel can be individuated by the ordered triple.
( sender processes , receiver processes , type)
where type is the data type of messages.

In several message passing languages, channels, called also mailboxes or ports, are considered as objects defined in the language /Mao 79, Wulf 75a, Rashid 81/ and specific constructs to operate on them are introduced. When a process P wants to communicate with a process Q, P connects itself to some channels as a sender and to other ones as a receiver, waits for Q to be connected too and then refers the channels in its send/receive commands.

An alternative view is supported by other languages, like Ada /Bjorner 80/ and CSP /Hoare 82/; in these languages the send/receive commands specify the name(s) of the partner(s) with which a process is willing to communicate. The communication occurs only when, and if, a rendez-vous is established with the partner. Following this ap-

proach, channels do not exist as passive objects distinct from the processes (of course, objects may be employed by the run time support to implement the communication) and the concept of logical channel may be employed only by the compiler to statically check the feassibility of communications.

We believe that this view is more coherent with the philosophy of message passing and that it can guarantee a greater coherence among the language constructs. It avoids the introduction of passive, shared objects and of the corresponding set of primitives to operate on them in a correct and protected fashion. Furthermore, since a program is composed of active entities only, it is easier to implement efficient abstractions of protected resources /Baiardi 81b, Bloom 79, Denning 76/.

Notice that this view does not prevent a process from dynamically selecting the partner of a communication, provided that the partner name can be specified by a variable whose value may be updated.

Further advantages can be obtained if we permit only one-to-one or many-to-one communications, i.e. the sender has to specify only one process as the receiver, while the receiver specifies that it is waiting for a message from any process belonging to a given set. Under this condition, a logical channel can be considered as a private data structure of a single process, i.e. the receiver of the channel itself. A process P can update variables used to refer partners in receive commands: this corresponds to grant/revoke the rights to use some of the channels of P.

## 3. THE ECSP LANGUAGE

ECSP /Baiardi 81a, Baiardi 81b/ is a message-passing language that adopts and extends the CSP model /Hoare 78/. We will present here only its main characteristics in connection with the topics addressed in this work.

### 3.1. Communication

ECSP permits both symmetric and asymmetric communications.

A symmetric communication is defined by a pair of corresponding send/receive (in ECSP, output/input) commands whese syntax is respectively:
<name of destination>!<cons>(<message value>)

<name of source>?<cons>(<target variable>)
where the message value and the target variable have the same type, and cons is a 'constructor': i.e. a constant string of characters used to possibly distinguish among communications of the same date type. The message type is then the pair:
(constructor,
type of message value and of target variable)

A symmetric communication may be synchronous as in CSP (a rendez-vous is established between the partners to perform the communication) or asynchronous.

By an asymmetric communication, which can be synchronous only, a process P may receive a message from any process belonging to a given set Q1, ...,Qn. The syntax of the input command is:
(x : Q1,...Qn) x?<cons>(<target variable>)
where x is a processname variable (discussed in the following) which is assigned the name of the sender of the received message: this is the first process in Q1,...,Qn which is ready to communicate.
An asymmetric channel is defined uniquely by the names of the possible source processes, by the name of the destination process and by the message type. Furthermore P cannot contain another input command with the same message type but referring a set of processes distinct from Q1,...,Qn.

In ECSP the names of processes may be either constant or variables. In the latter case a data type, processname, is introduced in the language to allow a process to dynamically modify the set of its partners. The main purpose of this facility is to support reconfiguration and dynamic protection policies to achieve higher robustness /Baiardi 81a/. The range of values of a process name variable declared in a process P is the undefined value and the names of the possible partners of P.

An input/output command where the partner name is given by the value of a processname variable x can be seen as exploiting a 'dynamic' communication channel, i.e. the partners connected to the channel can be changed by a new assigment to x. Notice that a processname variable x may be associated with a set of channels individuated by the input/output commands where x appears.

### 3.2. Process Structuring

A process P can have a hierarchical structure. This means that it can activate a set of process P1,...,Pn and wait for their termination. This is expressed by the parallel command:

activate P1,...,Pn end

where each Pi can in turn activate other processes.

The partners of P cannot observe that P has been replaced by a set of processes and thus the presence of P1,...,Pn is completely hidden to the interaction environment of P: a partner of P always refers P in its input/output commands even when it actually communicates with one of P1,...,Pn. Each Pi, instead, inherits from P the knowledge of the names of the partners of P. In terms of channels this means that P1,...,Pn can inherit channels defined in P: however this is invisible to the other processes connected to these channels.

To avoid uncontrolled nondeterminism in communication, ECSP requires that a channel of P is inherited by at most one of P1,...,Pn. If this condition were not imposed, several ambiguities could arise forcing additional synchronizations and policies difficult to express and error prone.

ECSP does not require that the channels inherited by each Pi are explicitly defined in P. For modularity reasons it is preferable that each Pi can define also new logical channels (distinct from those of P) with the partners of P. These channels are however said to be inherited since they are considered to be defined by P. Of course, to avoid ambiguities, Pi and Pj ($i \neq j$) cannot define the same channel.

For each pair of channels defined in a process P

Ci=(setj, P, Ti)

Cj=(setj, P, Tj)

if $i \neq j$ then

$seti \cap setj = \emptyset$ or $Ti \neq Tj$.

When this requirement is not met, we say that there is an ambiguity in the program. The ambiguity is called sequential when the two channels, Ci and Cj, are defined by the commands of the same process. When, instead, Ci and Cj appear in two distinct processes Pi and Pj activated by a process P, the ambiguity is called concurrent.

Notice that, because of processname variables, the absence of ambiguity cannot be completely checked at compile time, and so the compiler has to generate code to evaluate the condition at run time.

Let us discuss the following example to show that a sequential ambiguity could reveal an incongruity in the scheduling of a resource. Suppose

that R is a resource encapsulated in and managed by process P. Two operations can be executed on R: op1 with parameters par11,...,par1n and op2 with parameters par21,...,par2m. The messages requesting the execution of an operation are transmitted over two dynamic channels with types (op1, record type(par11),...,type(par1n)) and (op2, record type(par21),...,type(par2m)) respectively.

To execute an operation a userprocess sends a request over an asymmetric channel specifying the operation and waits for being connected to the appropriate channel. After the connection, the user process sends the parameters and waits for the answer. Furthermore, let us suppose that a given process P1 has always the right to execute op1; so P1 is connected by a static channel to P and does not request the permission to execute op1.

The skeleton of the program of P is shown in fig. 1.

A sequential ambiguity is signalled when the value P1 is assigned to y. This reveals both an incongruous request of P1 and that the checks on process requests fail in this situation.

A simple case of concurrent ambiguity arises when P activates two processes, Pa and Pb, working in parallel to increase efficiency. Let us suppose that Pa and Pb execute the same program and, therefore, define the same logical channels. The choice of which process receives an user request should be delegated to the implementation and this could not guarantee a well balanced load-sharing between Pa and Pb. Such a balance can be guaranteed only by policies defined by the program.

```
P:: x,y,z : processname;
    busy  : boolean;
    x:=y:=z:=undefined;
    busy:=false;
          :
* [not(busy), (x: P2,...,Pk) x?request(op) →
                    <check the rights of x>;
                if <legal request> then
                                    begin
                                     case op of
                                      op1 : y:=x
                                      op2 : z:=x
                                    endcase;
                                    x!right();
                                    busy:=true
                                   end
                                  else x!noright()
 [] notundefined(y), y?op1(par11,...,par1n)  →
                <execute op1>;
                <send result>;
                 y:=undefined
 [] notundefined(z), z?op2(par21,...,par2m)  →
                <execute op2>;
                <send result>;
                 z:=undefined
 [] P1?op1(par11,...,par1n)  →  <execute op1>;
                                <send result>
 ];
  :
  :
```

Fig. 1

## 4. THE ECSP COMPILER

In this section we will describe the choices done in the development of the ECSP compiler, stressing the influences of concurrent constructs. The main aims of the compiler structure can be summed up as follows.

a) The compiler is the kernel of the programming environment for ECSP. The compilation is obtained by the composition of a set of tools that can be exploited also in the definition of other complex tools. Thus each tool of the compiler implements a minimal functionality.

b) The compiler must support the separate compilation of processes in a program. Therefore it must be possible to analyze and translate a single process and to use these results in the compilation of the whole program.

c) The set of tools for the compilation should be partitioned into two subsets: the tools independent from any given architecture (front-end tools) and the architecture dependent ones (back-end tools).

### 4.1. Structure of the compiler

The compiler front-end is given by the following tools: the abstract syntax builder, the type checker, the channel generator, the interface checker and the intermediate code generator. The back end includes the generator of data structures used by the run-time support, i.e. process and channel descriptors, the code generator and the code optimizer.

Notice that distinct tools have been developed for the analysis of logical communication channels and for the generation of data structures implementing the channels.

We will now describe the main characteristics of those tools of the front-end that present original features when compared to the sequential case.

### 4.2. Abstract syntax builder

The abstract syntax builder (ASB) includes the lexical and the syntactic analysis of an ECSP process. It produces a set of abstract syntax trees (ast). The output is a set of ast's because the process analyzed could define other processes due to the feasibility of nesting parallel commands: in this case a distinct ast is generated for each process. This choice has been influenced by the requirement of separate compilation.

The activation relation between processes is described by a further output, the activation tree. An activation tree has two kinds of nodes corresponding to processes and to parallel commands respectively. In the activation tree, each process node has as many sons as the parallel commands in its command list are. Each parallel command node has as many sons as the processes it activates.

In ECSP, the body of a process Pi, activated by P, can be specified in the declarative part of P. During the analysis of P, when the ASB examines the declaration of P, it suspends both the analysis and the building as soon as Pi ast has been built. The activation tree is updated when the ASB meets the parallel command activating Pi (not during the analysis of Pi). To support separate compilation of processes, P can also activate a process Pi without specifying the process body. This case is signalled in the activation tree by proper information associated with the process node corresponding to Pi.

Since a distinct ast is produced for each process, each tool can be applied to a single process. So, the user can choose between a traditio-

nal compilation cycle, implemented by a further tool invoking the others, or an interactive use of the various tools.

If a syntactic error is signalled during the analysis of process P, the ast for P is not generated. After having recovered the error, only P has to be reanalyzed, since the ast's for the correct processes are always generated.

### 4.3. Type checker

The type checker visits the ast to associate each variable with its type and to verify that each statement is correct with respect to data types. Each i/o command is associated with the information, about the message type, required by the other tools.

Type analysis is simple since ECSP is a strongly typed language, and so variables have equivalent types only if their types have the same name. Besides the enriched ast, the tool produces also the symbol table of the process.

### 4.4. Channel generator

The channel generator and the interface checker, described in the following section, performe a set of consistency checks of channels. The channel generator analyzes the channels of a single process, while the interface checker verifies the mutual consistency between channels defined in distinct processes. As it will be clear in the following, this distinction makes separate compilation easier.

When applied to a process P, che channel generator (CG) first of all derives the set of input channels of P by an analysis of the input commands and of the declarations of P. By this analysis the CG derives also the dynamic channels associated with each processname variable of P. After having generated the channels, CG verifies that no sequential ambiguity exists among the static channels defined in P. This check does not suffice to guarantee that no ambiguity will arise at run time since each assignment to a processname variable modifies the set of channels in the program. Therefore the compiler has to generate code to verify that each assignment does not violate the ambiguity condition. This requires that each channel associated with the processname is compared against the other channels of the process. CG exploits the derived information to decrease the amount of comparison, thus reducing the overhead at run time. This represent an extension, to the concurrent case, of the well known technique of exploiting information deduced by a static analysis to produce a more efficient code /Wulf 75b/.

Each dynamic channel is associated with a set, PC of possible collisions. This set includes all and only the channels that, because of an assignment, could generate an ambiguity with respect to the considered channel. For a dynamic channel Ci=(seti, P, Ti), PC(Ci) includes all the channels Ci such that Tj=Ti. Each dynamic channel must be compared only against the channel in its PC.

CG analyses also the output commands of a process. From this analysis it deduces which input channels must be defined by the partners belonging to the same interaction environment. This information will be used by the interface checker.

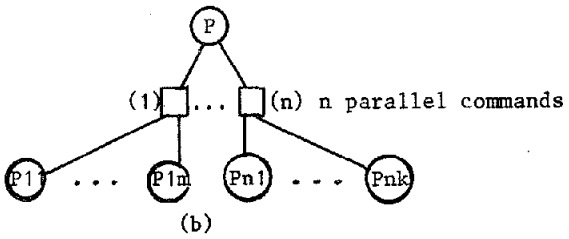
## 5. INTERFACE CHECKER

Let us describe the analysis of the interface checker (IC) by means of an example. Suppose that the tool is applied to the program fragment of fig. 2.a which corresponds to the activation

tree of fig. 2.b

```
P::
   P11::<body of P11>
      :
   Pnk::<body of Pnk>
         :
         activate P11,...,P1m end
         :
         :
         activate Pn1,...,Pnk end
         :
         :
```

(a)




(b)

Fig. 2

Initially IC analyses the subtree whose root is the first parallel command node (1). For each P1i:

a) the input/output command of P1i can refer a process Pjk only if, in the activation tree, the node corresponding to Pjk is a son of a parallel command node on the path from the process node P1i to the root. In the example P1i can refer only a process in P1z ($z \neq i$);

b) if all the senders of a channel of P1i belong to the considered parallel command, then IC checks that each sender contains at least one output command that could refer the channel, otherwise an error is signalled;

c) if at least one sender of (seti, P1i, T) does not belong to P11,..., P1m, then IC associates with the parallel command a channel (seti-(P11,...,P1m), P, T). The checks of point b) are then executed for the sender processes in P11,...,P1m;

d) for all the output commands that refers,or could refer, a process in the parallel command, IC checks that the partner has defined one input channel that could receive the message. When the command refers a process outside the parallel command, P is associated with the information that it will execute such an output command.

When this analysis has been performed for all the processes activated in a parallel command, IC analyzes the channels associated with the parallel command in step c) to verify the non ambiguity condition: i.e. it checks that two processes do not define the same input channel, or two channels that could be ambiguous after a proper assignment to a processname variable in either process. It also checks that two or more processes do not contain output commands that refer, or could refer, the same channel.

In the case of concurrent ambiguities the set of possible collisions for a channel is not introduced, as for sequential ambiguities. Now the channel that could become ambiguous are defined in distinct processes; the checks cannot be implemented by code of processes since they should examine the variables of other processes and this is in contrast with the local environment model. So the checks should be implemented in the run time support but this strongly increases the execution overhead. Thus we have chosen to consider each concurrent ambiguity as a static error.

If the analysis of the channels associated with a parallel command does not detect an error, these channels are joint to those defined by the process P executing the parallel command. A channel which has already been defined in P, or in a process activated in a parallel command previously examined, is discarded.

If no error is signalled, an extended set of input channels of P, and an extended set of information on the output command of P, have been determined. Thus the analysis can be recursively applied to the ancestors of P until the root of the

tree is reached.

After the IC analysis all the channels defined in a process, and those used by the process itself, have been determined. This information will be exploited in the generation of the data structures and of the descriptors for the run time support.

## 6. CONCLUSIONS

This paper has described the structure of a compiler for a concurrent language and the checks on process interfaces. We have shown how the adoption of explicit naming in interprocess communication has enabled the detection of several kinds of errors at compile time; this is also due to the adoption of typed messages. Other influences of the kind of naming adopted in a language are currently investigated. In particular the choice of using names of processes (not of channels) has proven to be useful in the definition of an optimizer for concurrent programs.

The tools have already been implemented and are currently under testing. They are used to develop concurrent software for the MuTEAM distributed computer.

## 7. REFERENCES


/Ada 83/ Ada Joint Program Office, "Reference Manual for the Ada Programming Language", ANSI/MIL-SID 1815 A, January 1983.

/Baiardi 81a/ F.Baiardi, A.Fantechi, M.Vanneschi, "Language Constructs for a Robust Distributed Environment and their Semantics". Internal Report, Dept. of Comp. Science, S-81-9, Univ. of Pisa, May 1981.

/Baiardi 81b/ F.Baiardi, A.Fantechi, A.Tomasi, M.Vanneschi, "Mechanisms for a Robust Multiprocessing Environment in the MuTEAM Kernel". Proc. of 11th Fault Tolerant Computing Symp., Portland (USA), June 1981, pp. 20-24.

/Baiardi 82/ F.Baiardi, L.Ricci, "Software development environment for the MuTEAM system; concurrent language and compiler". Research Report, Mumicro Project, November 1982.

/Bijorner 77/ D.Bijorner, "Programming Languages: Formal Development of Interpreters and Compilers". Int. Computing Symp. '77, North-Holland, 1977.

/Bijorner 80/ D.Bijorner, "Towards a Formal Description of Ada". Lect. Notes in Comp. Science, n. 92, Springer-Verlang, 1980.

/Bloom 79/ T.Bloom, "Synchronization Mechanisms for Modular Programming Languages". MIT TR-211, Cambridge Mass., 1979.

/Bryant 78/ R.E.Bryant, J.B.Dennis, "Concurrent Programming". MIT Report, Cambridge, Mass., 1978.

/Denning 76/ P.J.Denning, "Fault Tolerant Operating System". ACM Computing Surveys, vol. 8, n. 4, December 1976, pp. 359-389.

/Feldmann 79/ J.A.Feldmann, "High Level Programming for Distributed Computing". Comm. of the ACM, 22, n. 6, June 1979, pp. 353-368.

/Francez 79/ N.Francez, C.A.R.Hoare, D.J.Lehmann, W.P. de Roever, "Semantics of Non-Determinism, Concurrency and Communication". Journal of Comp. and System Science, n. 19, 1979, pp. 290-308.

/Hansen 78/ P.Brinch Hansen, "Distributed Processes: a Concurrent Programming Concept". Comm. of the ACM, 11, n. 21, November 1978, pp. 933-941.

/Hoare 78/ C.A.R.Hoare, "Communicating Sequential Processes". Comm. of the ACM, 21, n. 8, August 1978, pp. 666-677.

/Hoare 82/ C.A.R.Hoare, "A Calculus of Total Correctness for Communicating Processes". Science of Computer Programming, 1, n.1, March 1982, pp. 49-72.

/Liskov 83/ B.Liskov, R.Scheifer, "Guardians and Actions: Linguistic Support for Robust Distributed Programming". ACM TOPLAS, 5, n. 3, July 1983, pp. 381-404.

/Mao 79/ T.S.Mao, R.T.Yeh, "Communication Port: a Language Concept for Concurrent Programming". Proc. of 1st Int. Conf. on Distributed Computing Systems, Huntsville, October 1979.

/Rashid 81/ R.F.Rashid, G.G.Robertson, "Accent: A Communication Oriented Network Operating System". CMU Report, April 1981.

/Wulf 75a/ W.A.Wulf, R.Levin, "The Hydra Operating System". CMU Report, Pittsburgh, June 1975.

/Wulf 75b/ W.Wulf, R.K.Johnsson, C.B.Weinstok, S.O.Hobbs, C.M.Geschke, "The Design of an Optimizing Compiler", Elsevier-North Holland, New York, 1975.