IST. EL. INF.
BIBLIOTECA
Posiz. ARCHIVIO

Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

ON COMBINING META-IV AND CCS

A. Fantechi

Nota interna B85-07

Agosto 1985

ON COMBINING META-IV AND CCS

Alessandro Fantechi*
Department of Computer Science (ID)
Technical University of Denmark (DTH)
Lyngby, Denmark

ABSTRACT

An experiment towards the development of a formal method for specification of concurrent systems is attempted, through the combination of META-IV (the meta-language of VDM) and CCS. These two formalisms are combined in two different ways: different degrees of integration of the two formalisms are exhibited. In this report we show that this kind of combination can be viable as a means for the specification of concurrent software, and we sketch the theoretical framework which is involved. As such we consider this report as a contribution towards the more general problem of combining formal methods.

1 INTRODUCTION

This report presents an experiment towards the development of a formal method for the specification of concurrent systems. The experiment consists in combining two well-known and established formalisms, the META-IV, the meta-language of the Vienna Development Method (Bjørner 78a, Bjørner 82a), and the Calculus of Communicating Systems (Milner 80a).

The reason for the choice of the two formalisms can be found in the fact that VDM is an engineered and well-understood method for specifying sequential systems, which is widely used and for which some automated tools are being produced. VDM per se is, however, not suitable for the specification of concurrent systems. On the other side, CCS is a simple and elegant formalism for specifying concurrent systems with a high deegree of modularity, but its notation is not very natural when coping with complex subsystems which are completely sequential, since it drives to decompose such subsystems in a number of processes combined by the calculus operators.

In the experiment we achieve two different degrees of integration of the function oriented approach of VDM and the process oriented structure of CCS,

---

* On leave from Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy

Visit to ID/DTH made possible by funding from C.N.R. and from Danish Ministry of Education, grant n.406.

obtaining two different formalisms (presented in sections 2 and 3, respectively). A comparison between the two formalisms is made and some related theoretical problems are addressed, for the moment only at their surface.

In section 4 we discuss the relationships between the two formalisms and VDM+CSP, which is another attempt to extend VDM to describe concurrent systems, made introducing in the meta-language some concurrency construct derived from CSP (Folkjär 80a).

Some more general considerations induced by the results of the experiment are given in section 5.

2 A SIMPLE COMPOSITION

As a first step of the experiment, we define a formalism in which a CCS-like process structure is imposed at the top level of a VDM system specification, while the functions that compute output values and new states for the processes are detailed using a VDM-style. This approach is the most natural and immediate to the problem of combining CCS with other formalisms that are suitable to the specification of sequential systems; other work has been made in this direction: see, for example, (Shields 83a), where algebraic specification has been used for the sequential part of the CCS specification of a network communication protocol.

2.1 Structure of a specification

The formulas that build up a specification are divided in six sections; the first one evidences the superimposed CCS structure.

1. Process Structure
   It gives the formulas for each component process; each process is specified by a CCS-like behaviour expression and can have parameters (corresponding to free variables in the behaviour expression defining it), that are called the "state". Its input and output sorts are specified explicitly as a set of labels, and the domain associated with them is indicated: this gives the type of the messages that can be exchanged through the associated ports. Finally, the domain of the state (i.e. the parameters to the process) is indicated.

2. Input-Output Domains
   The domains referred to in the sort specifications of processes are specified in detail with usual META-IV domain definitions. If well-formedness conditions are needed on some input-output domain, they are given in this section.

3. State Domains
The domains referred to in state specifications of processes are specified in detail with META-IV domain definitions. If well-formedness conditions are needed on some state domain, they are given in this section.

4. State Transition Functions
The functions used in the process definitions to express state transitions (i.e. to compute the parameters for processes), seen as functions from Input and State Domains to State Domains, are specified with usual META-IV notation.

5. Output Computing Functions
The functions used in process definitions to compute values to be output, seen as functions from Input and State Domains to Output Domains, are specified with usual META-IV notation.

6. Auxiliary Functions and Domains
All other functions and domains needed in the previous definitions are then specified, using META-IV notation.

This classification, which closely resembles the structure of a conventional VDM specification, built up by Synctactic Domains, Semantic Domains, Semantic Functions and Auxiliary Functions, makes clear that the CCS style remains at the "top" of the specification, without really intermixing with VDM style. In this way, the process structure of the system is always visible at the top level.

This has the advantage of showing clearly which are the main components of the system, and of clearly defining the interface of each process with the external world. However, there is no way to hide a sub-structure. That is, it is not possible for a process to be internally decomposed into sub-processes, or internally exploit a non-deterministic behaviour, without making this immediately visible at the top level. There is no means for leaving a function underspecified, or specified by means of other techniques (e.g. pre- and post- conditions), and later to refine its specification, for example to be internally engaged in a parallel computation.

2.2 Syntax and semantics of the meta-language

The syntax of the meta-language can be obtained easily from the CCS syntax by replacing the CCS categories of value expressions and variables with the META-IV categories of expressions and variables (see fig.1, in which a simplified abstract syntax is given), with some syntactic sugar added to make behaviour expressions closer to usual META-IV notation (for example, the use of the keywords in and out in front of a port-name, instead of the usual complement sign).

```
Process_def :: ProcId Var BE

BE = NIL | internal_move | Input | Output | nd_choice |
     par_comp | restriction | renaming | Process_call | Ite

                                        (usual CCS concrete repr.)

internal_move :: BE                                  τ.BE
Input      :: In_sort  Var BE                        αx.BE
Output     :: Out_sort Exp BE                        ᾱx.BE
nd_choice  :: BE BE                                  BE₁+BE₂
par_comp   :: BE BE                                  BE₁|BE₂
restriction :: BE Sort                               BE\α
renaming    :: BE Renames                            BE[map]
Process_call :: Procid Exp                           P(exp)
Ite        :: Exp BE BE                  (*)

In_sort, Out_sort :: Sort
Renames :: Sort ⇸ Sort
```

(*) Other control constructs from META-IV can be included.

fig.1

In the following we shall use only some of the simpler features of META-IV, which we assume as self-explaining, while we wish to recall briefly the intituive meaning of CCS behaviour expressions, together with the concrete syntax we use:

- τ.BE — is an internal event, i.e. its behaviour is to transform itself in the behaviour specified by BE;
- in α x.BE — is a process which inputs a value from port α, assigns it to x and then behaves like BE (free occurrences of x may be present in BE, that become bound to the input value);
- outα exp.BE — is a process which outputs the value of exp on port α, and then behaves like BE;
- BE_1+BE_2 — is a nondeterministic choice between the behaviour of BE_1 and that of BE_2;
- $BE_1|BE_2$ — is the parallel composition of the two behaviours;

BE\α — is the same as BE, but where all the Inputs and Outputs of BE referring α cannot take place: i.e. it restricts the visibility from outside.

BE[map] — is simply like BE, but with all the ports renamed, according to map;

P(exp) — has the behaviour BE associated with the definition of P (Process P(x) $\triangleq$ BE), with all free occurrences of x in BE replaced by exp.

From what said, it is clear that the semantics of the meta-language can be given completely in terms of CCS semantics, where the evaluation of value expressions is given in terms of denotational semantics of META-IV expressions. The assumption, made in (Milner 80a), p.66, that CCS value expressions compute only total functions is respected when only META-IV functions are used which are total on their definition domains or on restricted domains on which well-formedness conditions hold.

2.3 An example

As an example of this method we give here the specification of a rudimentary database. The database contains a list of identifiers (this is its state), on which four operations are defined: insert a new identifier, remove an identifier, list the contained identifiers and count them. The database accepts requests of operations from n external processes, here left unspecified, and gives them the related replies. The specification is given in fig.2. Σ_i in α_i x.BE_i is a shorthand for in α_1 x.BE_1+in α_2 x.BE_2+...+in α_n x.BE_n.

As can be seen from the example, the behaviour of the database and its interface are completely specified in the Process Structure section of the specification, which is the only section that contains behaviour expressions.

Process Structure

```
Process Service(state) ≜
    Σ_i in α_i req.out δ_i serve(req,state).
                          Service(new(req,state))
Input sort  α_1,..α_n :Request
Output sort  δ_1,..δ_n : Reply
State Content;
```

Input Output Domains

```
Request = Insertreq | Removereq | Listreq | Countreq
Insertreq, Removereq :: Id
Listreq, Countreq :: TOKEN

Reply = OK | Error | Listreply | Countreply
OK, Error :: TOKEN
Listreply :: Id*
Countreply :: Integer
```

State Domains

```
Content :: Id*
```

State Transition Functions

```
new (req,state) ≜
    (cases req
       mk-Insertreq(name) →
           if name ε elems state then state else state^<name>
       mk-Removereq(name) → <id | id ε elems state ∧ id ≠ name>
       mk-Listreq(), mk-Countreq() → state)
type Request Content → Content
```

Output Computing Functions

```
serve(req,state) ≜
    (cases req
       mk-Insertreq(name) → mk-OK()
       mk-Removereq(name) →
           if name ε elems state then mk-OK() else mk-Error()
       mk-Listreq()       → mk-Listreply(state)
       mk-Countreq()      → mk-Countreply(len state))
type Request Content → Reply
```

fig.2

The system composed by this process and its users can be simply expressed by the following process definition:

Process SYSTEM $\triangleq$ $P_1 \parallel \ldots \parallel P_n \parallel$ Service(initial_state)

where $\parallel$ is a shorthand for parallel composition of the processes and subsequent restriction on the sort common to all processes, in this case $\{\alpha_1,..\alpha_n,\delta_1,..\delta_n\}$ (we are assuming here that $\parallel$ is associative, since each P_i shares only $\{\alpha_i,\delta_i\}$ with Service).

3 AN ALTERNATIVE: THE RESULT-VALUED PROCESS CONCEPT

The meta-language proposed above allows to specify concurrent systems in a CCS style, defining some computations (typically, the sequential ones inside a process) in META-IV style. What about META-IV functions computing values using some CCS operators (e.g. nondeterministic choice or parallel composition)? Such a meta-language would achieve a greater degree of integration of CCS and VDM.

This meta-language can be based on the concept of a **Result-Valued Process** (RV-process). A RV-process is specified as a CCS-like behaviour expression which returns a value in some domain. It can be operationally seen as a function whose computation proceeds in a nondeterministic way and affects the execution of some other RV-processes. Its computation can, conversely, be influenced by the execution of other RV-processes.

This concept tries to unify both the function and process concepts: a conventional function is a deterministic RV-process without side effects, while a conventional CCS process can be seen as a RV-process returning a value in the domain whose only element is NIL, denoted in the same way.

3.1 The meta-language

The abstract syntax of RV-processes can be obtained from CCS syntax (the one of fig.1) by replacing the NIL process by a RV-process which terminates giving a result (concretely represented by: return expr), and allowing normal expressions to include calls to RV-processes, instead of pure functions (fig.3).

In this case, starting from a META-IV environment, we have that the META-IV expressions are generalised to include calls to RV-processes, whose definitions are given in terms of (result-valued) behaviour expressions, and of (extended) META-IV expressions themselves.

```
RVprocess_def :: RVprocId Var RVBE

RVBE = Result | internal_move | Input | Output |
       nd_choice | par_comp | restriction |
       renaming | RVprocess_call | Ite

Result     :: Exp
internal_move :: RVBE
Input      :: In_sort  Var RVBE
Output     :: Out_sort Exp RVBE
nd_choice  :: RVBE RVBE
par_comp   :: RVBE RVBE
restriction :: RVBE Sort
renaming   :: RVBE Renames
RVprocess_call :: RVprocid Exp
Ite        :: Exp RVBE RVBE        (*)

In_sort, Out_sort :: Sort
Renames :: Sort ⇸ Sort

Exp :: Usual_syntax_of_expressions(Exp) | RVprocess_call
```

(*) Other control constructs from META-IV can be included

fig.3

A problem arises with the parallel combination operator: since it must still represent a RV-process, its result has to return a unique value; some rules are hence to be established about which of the values of its RV-process components is to be chosen. A suitable definition would be that the operator returns the value returned by the leftmost behaviour expression; this definition maintains the associativity of the operator, but not the commutativity. An alternative definition would be that the returned value is nondeterministically chosen between the values returned by the left hand and the right end of the operator; this definition maintains both the associativity and the commutativity of the operator. In the following we use the former definition because it seems to be of more practical use (operationally, it represents a function which, during its computation, has the side effect of creating a new process, without worrying about its termination - this is a common situation in real systems, e.g. it is possible to program a similar function with the Ada tasking facilities), but this is anyway not intended to be a binding choice. The notation ⫏ denotes the (asymmetric) parallel composition operator which returns the value of the leftmost behaviour expression.

Another problem is related to the more dynamic nature of RV-processes with respect to the conventional CCS processes. In CCS the sorts of each process are defined with reference to a global set of labels. These labels

are visible at any point of a specification, i.e. any process can refer them. Since the RV-processes can be defined and called anywhere in the specification, scope rules for the port names are likely to be introduced. In the formalism we use later on, we have chosen to allow port names to be passed as parameters to RV-process; reference to such parameters naming allowed in input/output expressions (note that this is different from allowing port names to be passed in communications, as proposed in (Astesiano 84a)). Port identifiers are defined only within a parallel composition, and are passed to the component RV-processes. This is because port identifiers are needed in parallel composition to specify the ports that are to be connected. The restriction operator applies only to port identifiers (see the definition of the Service process in the example in section 3.2). The use of port parameters makes useless the renaming operator of CCS, since a renaming is implicitly done at each RV-process call.

Obviously, our choice is not binding, in the sense that more complex scope rules for port names are possible and could be included in the definition of a meta-language.

In the notation we shall use from now on, the syntax of RV-process definitions and of RV-process calls includes the port parameters, but separate from the other parameters to the RV-process (notation: [input ports; output ports]). Greek letters denote port variables, both in declaration as formal parameters, or at their use. The notation $\langle\alpha\rangle$ as a port variable denotes the array of variables $\alpha_1,..\alpha_n$, with n determined by the corresponding sort definition. Port identifiers are denoted by capital letters.

The Input-Output domains for a RV-process are now associated to its port parameters, with the same syntax used for the first formalism presented. Moreover, a type definition (in the style of META-IV) which gives the domain and codomain of the RV-process is added.

The META-IV syntax is maintained for conventional functions; notice that the RV-processes in which sorts are not used directly (i.e. in which they do not explicitly appear in any operator), but in which they are only passed to called RV-processes, are compulsory written as RV-processes, and not as conventional functions.

3.2 An example

As an example we give the specification of a system that can be seen as a sophisticated version of that of fig.2. In order to obtain a certain degree of parallelism among concurrent requests, the Service process is now decomposed into a DB process, which maintains the list of identifiers, and a Driver process which accepts external requests and delegates their serving to dynamically created Task processes. The DB process exports only three operations on the identifier list: insert, remove and list. The answer to the list request is a sequence of output values, one for each identifier. For simplicity, we assume that an external process cannot issue a new request

until the previous requests have been served. The specification is given in fig.4 (a,b).

```
RV-Process  Service(state)[<α>;<δ>] ≜
        (Driver[<α>,C;<δ>,B] ⌉ DB(state)[B;C])\B\C
Input sort  α1,..αn :Request
Output sort  δ1,..δn : Reply
type Content → NIL;

Request = Insertreq | Removereq | Listreq | Countreq
Insertreq, Removereq :: Id
Listreq, Countreq :: TOKEN
Reply = OK | Error | Listreply | Countreply
OK, Error :: TOKEN

Content :: Id*

RV-Process  Driver[<α>,γ;<δ>,β] ≜
     Σi in αi req.(Driver[<α>,γ;<δ>,β] ⌉ Task(req)[γ;β,δi])
isort α1,..αn : Request, γ: DBReply
Output sort  δ1,..δn : Reply, β: DBRequest
type       → NIL;

RV-Process  Task(req)[γ;β,ρ] ≜
              out ρ serve(req)[γ;β].return NIL
Input sort  γ : DBReply
Output sort  β : DBRequest, ρ : Reply
type Request → NIL;

RV-Process  DB(state)[β;γ] ≜
    in β req.out γ reply(req,state)[c].DB(new(req,state))[β;γ]
Input sort  β : DBRequest
Output sort  γ : DBReply
type Content → NIL;

DBRequest = Insertreq | Removereq | Listreq
DBReply = OK | Error | Item
Item :: Id

RV-Process  serve (i,req)[γ;β] ≜
 (cases req
   mk-Insertreq(name) → out β req. in γ rep. return rep
   mk-Removereq(name) → out β req. in γ rep. return rep
   mk-Listreq()       → out β req. givelist()[γ]
   mk-Countreq()      → out β mk-Listreq(). givecount()[γ] )
Input sort  γ : DBReply
Output sort  β : DBRequest
type Integer Request → Reply
```

fig.4a

```
RV-Process  reply (req,state)[γ] ≙
    (cases req
       mk-Insertreq(name) → mk-OK()
       mk-Removereq(name) →
            if name ε elems state then mk-OK()
                else  mk-Error()
       mk-Listreq()       → recreply(state)[γ] )
Output sort  γ : DBReply
type DBRequest Content → DBReply

new (req,state) ≙
    (cases req
       mk-Insertreq(name) →
            if name ε elems state then state
                else state^<name>
       mk-Removereq(name) →
            <id | id ε elems state ∧ id ≠ name>
       mk-Listreq(), mk-Countreq() → state)
type Request Content → Content

RV-Process  givelist()[γ] ≙
   in γ rep. (if ~ is-Item(rep) then return <>
                        else return  rep^givelist()[γ] )
Input sort  γ : DBReply
type      → Listreply

RV-Process  givecount()[γ] ≙
   in γ rep. (if ~ is-Item(rep) then return 0
                        else return givecount()[γ]+1 )
Input sort  γ : DBReply
type      → Countreply

RV-Process  recreply(state)[γ] ≙
    (if len state = 1 then return hd state
       else out γ recreply(tl state)[γ].return hd state)
Output sort  γ : DBReply
type  Content → Item
```

fig.4b

As we can see from the example, a top-level process structure is no longer enforced, and hiding of parts of the behaviour in lower level components is possible: this allows, for example, underspecification of some components and later refinements in an internally parallel structure.

On the other hand, a specification given by RV-processes does not always show a clear interface to the external world, since the specification of its external behaviour is spread across several components (in the example,

replies to the external requests are not sent by the same process that has received the requests).

3.3 Semantics of the meta-language

In the formalism presented in chapter 2 we had a rigid process structure visible at the top level, specifying the behaviour of the component processes of a system. This structure also gave a precise boundary between the two combined formalisms, in order to distinguish which theoretical framework applies at any point of the specification.

Conversely, the theoretical framework of the RV-processes formalism cannot be expressed as a mere combination of two separate models. Obviously, the denotational semantics by input-output functions of META-IV can no longer be maintained, neither the semantics given for conventional CCS can be directly transferred to RV-processes. Rather we need a homogeneous framework, which may be based on some of the proposed models to give the semantics of concurrent processes.

The semantics of the resulting meta-language is currently under study; a first way is to define the semantics of the meta-language operators by derivations, as made in (Milner 80a) for conventional CCS. An alternative way is to build it on the model for CCS given by Milner using Communication Trees (Milner 80a). Informally speaking, the rules given there for the construction of the Communication Tree modelling a CCS process can be extended to model RV-processes, in the following way:

a) a return expr behaviour expression is represented by as many branches in the tree as are the possible values to which expr can evaluate;

b) a RV-process call inside a behaviour expression is represented by the whole tree corresponding to the called process, and but the representation of the following part of the behaviour expression is appended to its leaves . Since those leaves all correspond to a particular value of expr in a return expr clause of the called RV-process, the appended continuation relies on the value returned by the called RV-process, as it was expected.

A third way to give the semantics of the meta-language is to provide a transformation from a specification built by RV-processes in one given by conventional CCS processes. Such a transformation allows to extract the global behaviour of the system, and to make it visible at the top level; this could be useful also, for example, for a better understanding of a specification given in terms of RV-processes. At a first analysis such a transformation seems to be possible, but a deeper study remains to be done about this topic. Having such a transformation, one could easily prove that the two presented formalisms have the same expressive power, since the inverse transformation is trivial (we have seen that conventional processes can be seen as a particular case of RV-processes).

Another open question is whether the algebraic structure of CCS is maintained and which equivalences are applicable in the case of RV-processes. This is particalarly important with regard to proof techniques.

4 RELATIONS WITH VDM+CSP

A predecessor of the present work is the combination of VDM and CSP by P.Folkjär and D.Bjørner (Folkjär 80a); they extend META-IV with CSP-like constructs.

The first formalism we have presented is less flexible than VDM+CSP, both because the former enforces a specification structure where processes are rigidly confined to the top level, and for the capability of VDM+CSP to express input-output events inside the inner functions. Conversely, our second formalism is more flexible in expressing concurrent and non-deterministic behaviours. We think also that the second formalism is "cleaner" than VDM+CSP; there no trace is kept of the communications that affect the computations of other functions.

A transformation from VDM+CSP to our second formalism seems to be possible and is currently under study. The inverse transformation is not possible in the case of the version of VDM+CSP defined in (Folkjär 80a) and used in (Løvengreen 80a), in which the mixed nondeterminism (i.e. a choice of the CCS form $\tau.P + \alpha x.Q$) is not expressible.

The experiences in the use of VDM+CSP in real systems specification (see, for example, (Løvengreen 80a)) and various experiments in the specification of real systems with CCS-based formalisms (an example is (Doeppner 83a)) give strength to our believe that the formalisms presented above can be viable in real applications, once engineered in a conveniently and widely applicable form.

5 CONCLUDING REMARKS

This report does not aim to define a comprehensive model for the specification of concurrent software; rather, its goal is to analyse the critic issues of the combination of formal methods.

The experiment presented suggests that it is possible to combine two specification formalisms that rely on different theoretical foundations. More precisely, the two formalisms discussed above show two alternative ways of combining.

The first one is almost completely a syntactical composition, let us say a juxtaposition of the two formalisms. Such a composition is theoretically sound given that the "interfaces" between the two formalisms are well-defined and well-understood: our case relies on the possibility to use META-IV expressions instead of CCS value expressions.

The second formalism is an attempt toward a greater degree of integration. What we have obtained is a formalism that has a greater flexibility with respect to the two components, and contains the two components themselves as subcases, but is no longer theoretically sound in the framework of any of the two components. Rather, it needs an ad hoc theoretical framework to be understood.

More generally, the first alternative can be seen in the direction of a specification method which enforces the use of different formalims, or metalanguages, to model different aspects of a system; the various formalisms integrate correctly once some proper assumptions about their "interfaces" are met. The second alternative is in the direction of a method providing a single meta-language, homogeneously built integrating various theoretical frameworks, in which all the aspects of a system can be modeled.

It is our opinion that these two alternative ways are the only ones to combine a formalism based on denotational semantics with in which concurrency and communication constructs play the main role. Is open to further study which way is more convenient to give formal specifications of real systems.

Another interesting general consideration about the specification of concurrent systems can be made by comparing the examples of fig.2 and fig.4. Even if we translate the former into the second formalism, they cannot be claimed to specify two equivalent processes by any of the formal equivalences defined for CCS, in spite of the fact that they can be seen as specifications of quite similar databases. The only property that applies to both is: "if a request is issued to Service, a proper reply is returned, consistent with a state of the database between the time of the request and that of the reply".

We can say that the specification of fig.2 is "more deterministic" than that of fig.4, because it allows no concurrency in the service of the requests. Relations such as "more deterministic" have been formalised as mathematical relations between processes in the framework of CSP (Brookes 84a, De Nicola 83a). We think that such formal relations, together with equivalences, have to be taken into account in any formal method for the specification of concurrent software, in order to describe properly how different phases of of a formal specification development are related.

6 ACKNOWLEDGMENTS

The author would like to thank H.H. Løvengreen for many helpful criticisms and suggestions, and D. Bjørner and A.P. Ravn for useful discussions. The final version of this report has benefited also of the careful revisions by R. De Nicola and A. Tocher.

7 REFERENCES

(Astesiano 84a) E.Astesiano, E.Zucca, "Parametric Channels via Label Expressions in CCS", to appear in Theoretical Computer Science, 1984.

(Brookes 84a) S.D.Brookes, C.A.R.Hoare, A.D.Roscoe, "A Theory of Communicating Sequential Processes", Journal of ACM, vol.31, n.3, pp.560-599, July 1984.

(Bjørner 78a) D.Bjørner, C.B.Jones, "The Vienna Development Method: The Meta-Language", Lecture Notes in Computer Sciences, 61, 1978

(Bjørner 82a) D.Bjørner, C.B.Jones, "Formal Specification and Software Development", Prentice-Hall, 1982

(De Nicola 83a) R.De Nicola, "Two complete Axiom Systems for a Theory of Communicating Sequential Processes", Internal Report CSR-154-83, University of Edinburgh, 1983. A short version also in "Foundations of Computation Theory", Lecture Notes in Computer Science, 158, pp.115-126, 1983.

(Doeppner 83a) T.W.Doeppner, A.Giacalone, "A Formal Description of the Unix Operating System", Proceedings of the 2nd Symposium on Principles of Distributed Computing, pp.241-253, Montreal 1983

(Folkjär 80a) P.Folkjär, D.Bjørner, "A Formal Model of Generalized CSP-like Language", Proceedings IFIP'80, pp.95-99, Tokyo 1980

(Løvengreen 80a) H.H.Løvengreen, "Parallelism in Ada", in D.Bjørner, O.N.Oest eds., "Towards a formal Description of Ada", Lecture Notes in Computer Sciences, 98, Springer-Verlag, pp.309-432, 1980.

(Milner 80a) R.Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Sciences, 92, Springer-Verlag, 1980

(Shields 83a) M.W.Shields,M.J.Wray, "A CCS Specification of the OSI Network Service", Internal Report CSR-136-83, University of Edinburgh, 1983.