893 CV57

PAPERS PRESENTED

at the

8th EUROPEAN WORKSHOP

on

APPLICATION and THEORY of PETRI NETS

Zaragoza, Spain

June 24-26, 1987

[Ste87] Ralf Steinmetz: *Relationship between Petri Nets and the Synchronization Mechanisms of the Concurrent Languages CHILL and Occam*. In: *8th European Workshop on Applications and Theory of Petri Nets, Zaragoza, Spanien*, June 1987.

Under the auspices of:

AFCET SIG 'Systèmes Parallèles et Distribués'
AICA
BCS SIG 'Formal Aspects of Computing Science'
CNRS-C^3
EATCS
GI SIG 'Petri Nets and Related System Models'

Organised by:

Departamento de Ingeniería Eléctrica e Informática
Universidad de Zaragoza

Proceedings printed by:

IBM Germany

1 8th European Workshop on Applications and Theory of Petri Nets
Zaragoza, Spain, June 1987
pp 509-529

Relationship between Petri Nets and the Synchronisation Mechanisms of the Concurrent Languages Chill and Occam

Ralf Steinmetz
Philips Kommunikations Industrie AG,
5900 Siegen 31
Fed. Rep. of Germany

This work was supported in part by Philips Kommunikations Industrie AG, Nürnberg, Fed. Rep. of Germany, and carried out at the Institut für Übertragungstechnik, Technische Hochschule Darmstadt, Fed. Rep. of Germany.

Index Terms:
Petri nets, Occam, Chill, concurrent programming, parallel programming, process synchronisation, process communication.

Abstract:
In this paper a representation of the Chill and the Occam concurrency semantic by appropiated net models is shown. The resulting net-characteristics impact on the application of these program representations. It is demonstrated, that for analysis purposes Chill programs should be deduced from nets, while in Occam a net may be derived from the language as well.

1. Introduction

Technology, meaning VLSI (Very Large Scale Integration = 100.000 to 1.000.000 transistors) and leading towards ULSI (Ultra Large Scale Integration = 1.000.000 to 10.000.000 transistors) and WSI (Wafer Scale Integration = 10.000.000 to 100.000.000 transistors) which includes defect-tolerance, influences and spurs the development of parallel processing computers ahead /Lea 86/. Also software was and is being developed to make the parallel processing power available in a comfortable and safe manner, because many applications (digital signal processing, number crunching, switching, ..) are concurrent in nature. This includes distributed and network operating systems as well as concurrent programming techniques. Two of the early languages which introduced features of concurrency are PL/I (parts of a program executed in parallel are called task) and Algol 68 (incorporating semaphors /Dijkstra 68/). More recently f.e. ADA, Chill and Occam were and are

being developed. Also formal, mathematical methods to deal with concurrency emerged; f.e. CCS, CSP and Petri nets.

In contrast to pure sequential programming also the management of the concurrency of a programm (including synchronisation and communication between parallel processes) has to be accomplished. Therefore, new concepts were developed (semaphor, critical region, monitor, rendezvous, ...) /Andrews Schneider 83/. Here new properties related to concurrent processes (already known in the field of operating systems see e.g. /Coffman Denning 73/) arose:
A multiprogramming system can **deadlock**, that means at least one process is waiting for an event which never will occur. Perhaps the most difficult method to face deadlocks is prevention: Before a program is executed it is analysed and perhaps reconfigured in such a way, that no deadlock will occur. Another possibility is deadlock detection: These are run time checks to detect a deadlock, and then recovering strategies are employed. Having advanced information deadlock avoidance can be applied before allocating ´resources´ (the meaning of resources depends on the synchronisation primitive used) during run time.
A ´livelock´ is an infinite loop which occurs because of misuse of concurrency facilities.
Between parallel processes a property called ´fairness´ has to be taken into account. Many times this property is considered to be language immanent, but unfortunately it often is implementation dependent!

Chill (CCITT High Level Language) was designed by an international organisation incorporating ideas of many telecommunication companies /Z.200 84/. It is the compulsory language for programming public switching systems in Germany. Concerning concurrency, Chill is very complex incorporating three synchronisation mechanisms: Regions and events, buffers and signals.
The **Occam** programming language /Inmos 84, May 83, Steinmetz 87/ was developed by Inmos under the assistance of Hoare and is influenced very much by the original CSP /Hoare 78/. It was designed in conjunction with a special RISC like microprocessor called ´transputer´ /Inmos 86/. Concerning concurrency it is a very simple language incorporating a point-to-point synchronous communication principle.

Working many years in the field of Petri nets, Chill and also during the last year with Occam, questions combining the experience of these activities arose. We don´t want to compare the languages (as e.g. in

/Kropf 86/) in this paper. In addition the relationship between synchronisation mechanisms and nets has been previously studied in different contexts: The monitor concept of Modula 2 was modelled by nets in /Steinmetz 84/; implementations of nets in ADA were shown in /Colom Silva Villarroel 86/; some constructs of the Chill concurrency were modelled as predicate transition net in /Hollis Kropf 84/; the relation between CSP and nets was studied already /Czaja 84, Reisig 84, Dolezol Schübel Triebel Wirtz 84/ and the principles of deadlocks-detection in Occam programs using nets were demonstrated in /Steinmetz 87/. But here we will define some of the questions not answered up to now and we will try to give an answer by this paper:

1. Can nets describe the semantic of the concurrency mechanisms of Occam and Chill ?
2. Which net class has to be used ?
3. Where may these transformations be employed and may nets be used to analyse (with respect to the synchronisation properties) concurrent programs ?
4. Are all these questions influenced by the specific language (here Chill and Occam) ?
5. Which of the following transformation process is more useful for the analysis of synchronisation properties in concurrent programs ?

   Concurrent Language --> Net or

   Net --> Concurrent Language

2. Chill

Chill (CCITT High Level Language /Lenzer Letschert Hollis Lingen 87, Sammer Schwärzel 82, Z.200 84/) was designed primarily as a high level language for SPC (stored program control) telephone exchanges. In the CCITT (International Telegraph and Telephone Consultative Committee) a ´team of specialists´ (ITT, LME, Philips, Siemens, NTT, UKPO, Nordic PTT) was formed to develop this language.

Chill is similar to Pascal, Modula 2 and ADA and should not be considered as a ´telephone language´. But up to now the availability of compilers and programming environment is very difficult and expensive because the language has at least similar complexity as ADA /Branquart Louis Wodon 82/ (every member of the committee tried to incorporate his own language constructs into Chill).

Chill is composed of

- program structuring statements (process definition, regions,...)
- action statements (start expression, zero-adic operator, delay

action, ...) and

- description of the data objects (instance modes, synchronisation modes, ...).

The concurrency is included in these three components; we want to restrict our description to the concurrency features. Unfortunately the three communication mechanisms (regions and events, buffers, signals) are very complex that about 80 nets are required to describe the semantic in a complete form. Therefore the detailed description has to be restricted to some features of one concurrency mechanism: The features of regions, events and signals will be explained briefly and details of the buffer concept with an example will be pointed out.

Process, Process Instance and Procedure

A **process** (process declaration) is a possibly parameterised sequence of actions that may be started many times for concurrent execution from different places in the program. A process is represented by a net. A **process instance** (PI) is the sequential execution of the statements defined by a process. The PI is represented by a token in the net. Every PI is uniquely defined by an instance value: to the token this ´processinstance´ value is attached.

The body of a **procedure** is a net. If a procedure is called by a PI the respective token is introduced into the net of the procedure. In order to realize recursive procedure-calls, the depth of the call is attached to the token (´nesting level counter´).

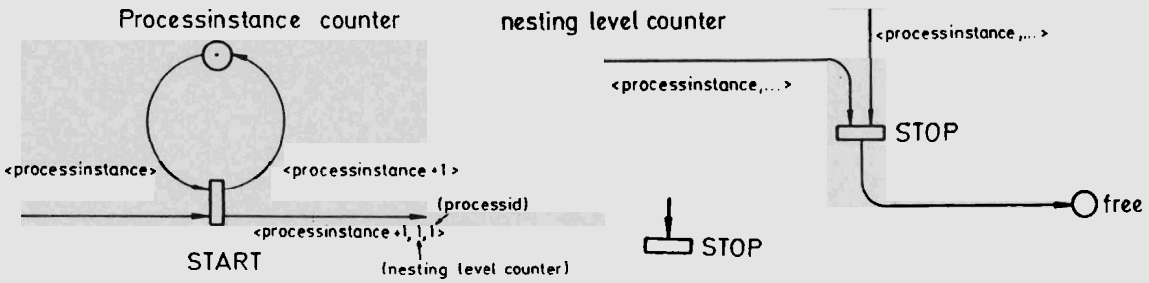

Fig. 2.1: Start action, stop action and stop action in a region

A **PI** can be created from any place out of its process definition and by the evaluation of a start expression it becomes active. In the net environment this means the introduction of a token (Fig. 2.1). The attribute ´processid´ is necessary for correct representation of the semantic of delay case actions. A PI terminates reaching the last action statement of its body or by executing a stop action. An existing PI can adopt one of the different states (handled by the scheduler and the despatcher): Running or ready to run (executing or being able to

execute), suspended or delayed (waiting for communication resources).

Region and Event

The **region** (critical region) is composed of definitions, data objects and critical procedure declarations. It provides mutually exclusive access to its variables and procedures. While a critical procedure is executed the region is locked and any PI attempting to enter the region will be delayed.

Through the **event**-locations together with the delay action, delay case action, continue action, and delayfail exception processes may synchronize and communicate. This principle is similar to the monitor concept /Hoare 74/ (for the net representation see e.g. /Steinmetz 84/).

Buffer

A **buffer** is a kind of mailbox for the communication of concurrent processes. Sending and receiving PI´s, which are considered to be executed concurrently, may send messages to and get messages from different buffers or it may be delayed to proceed later. The communicating PI´s may synchronize or not. The ADA rendezvous /ADA 83, Bishop 86, Welsh Lister 81/ and Occam channel concept are a special type of buffer with bufferlength=Ø. Additionaly ADA allows a timer to be set: During the specified time the rendezvous has to be accomplished.

The buffers can be modelled in the following ways:

1. Every buffer is represented by a place (or set of few places if needed): This leads to too many transitions and arcs,which are required to model the receive case actions.
2. Sets of buffers are different places: The sets are defined by the buffers involved in the receive case actions.

   Example:

   ```
   RECEIVE CASE (buffer_1, buffer_2) : ..
                (buffer_3) : ..        ESAC
   RECEIVE CASE (buffer_1, buffer_4) : ..
                (buffer_5) : ..        ESAC
   RECEIVE CASE (buffer_6, buffer_7) : ..
                (buffer_8) : ..        ESAC
   ```

   The buffers {buffer_1, buffer_2, buffer_3, buffer_4, buffer_5} form a set and {buffer_6, buffer_7, buffer_8} make up another set.
3. All buffers are one place: The complexity of arc inscriptions implies a loss of understanding the behavior of the net.

We adopted the first solution whenever possible or otherwise the second. With respect to the priority of the send buffer case action the

following modelling alternatives exist:

1. To each priority corresponds a place or a set of places. The Chill compiler available to us supports two different priorities, so we choose this alternative.
2. The priority is an attribute of a token. This modelling is recommended for the definition according to /Z.200 84/.

For the modelling of **buffer location** places with fifo characteristics are required. This behavior can be realized by the following simple net:

Fig. 2.2: Abbreviation of a fifo net macro

Every buffer is illustrated by three places:

- ´Delay sending process instance´ (del_sen): All tokens corresponding to a PI, which performed a send buffer action and the correspondeing buffer was full, are inserted in this place (the PI is delayed). A buffer is ´full´, if the amount of messages equals the defined bufferlength. The attribute ´process name´ of the token is used to send this token (PI) to the correct net (process) after concluding communication.
- ´Buffer location´ (buf_loc): This place is the buffer itself; the messages with a reference to the sending PI are placed here (the reference is required for the ´set´ parameter of receive buffer case actions). These messages will be fetched by the receiving PI´s.
- ´Delay receiving process instance´ (del_rec): If no message is available at the moment and no sending PI is delayed on the respective buffer location, then a PI executing a receive expression or a receive case action will be delayed. The token is inserted into the place del_rec.

The bufferlength is the capacity (abbr. Cap.) of the place buf_loc. PI´s (tokens in the place del_sen) have to be reactivated if the marking of the place buf_loc is smaller than its capacity. The message has to be transferred to the place buf_loc. The following transformations (Fig. 2.3, 2.4, 2.5 and 2.6) are shown for a buffer without priority and which is not declared in a region. Every arc starting or ending at ´sending process´ or ´receiving process´ reflects a connection to the net corresponding to the respective process.

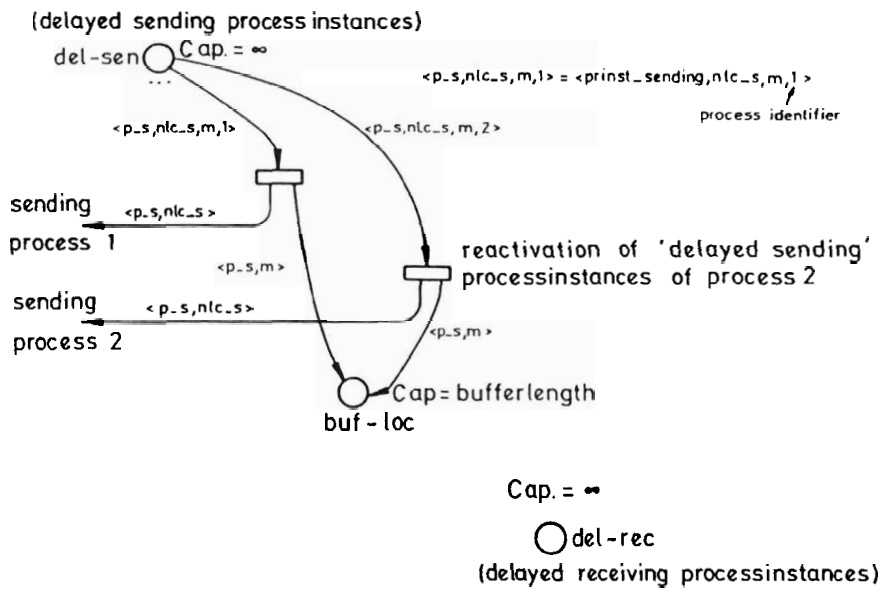

Fig. 2.3: A buffer

The semantic of the **send buffer case action** distinguishes three operations (Fig 2.4):

1. No PI is delayed in the corresponding place del_rec and the buffer is full. The sending PI will be delayed (the token is placed into ´del_sen´) and the buffer will remain full, transition t_1 occures.
2. No PI is delayed in the corresponding place del_rec and the buffer is not full. The message is placed in buf_loc by occurence of the transition t_2.
3. At least one PI is delayed being a token in the place del_rec. One of the PI´s (token in place ´del_rec´) is reactivated. Transition t_3 occures.

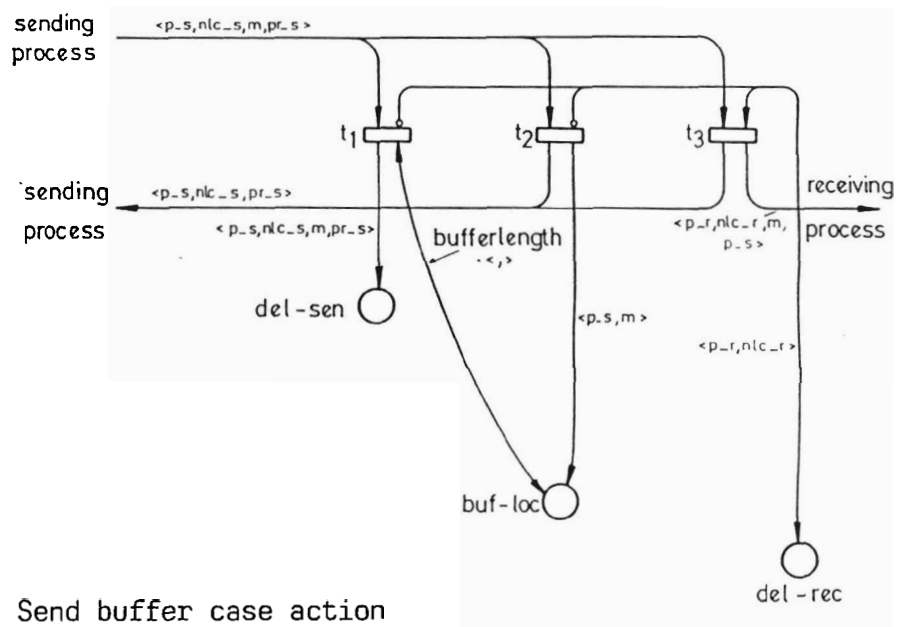

Fig. 2.4: Send buffer case action

The **receive expression** (Fig. 2.5)

- may reactivate delayed sending PI´s (transition t_1 or t_2 occures),
- may fetch a message from the place ´buf_loc´ (transition t_3) or
- the PI performing the receive expression may be delayed (t_4 occurs).

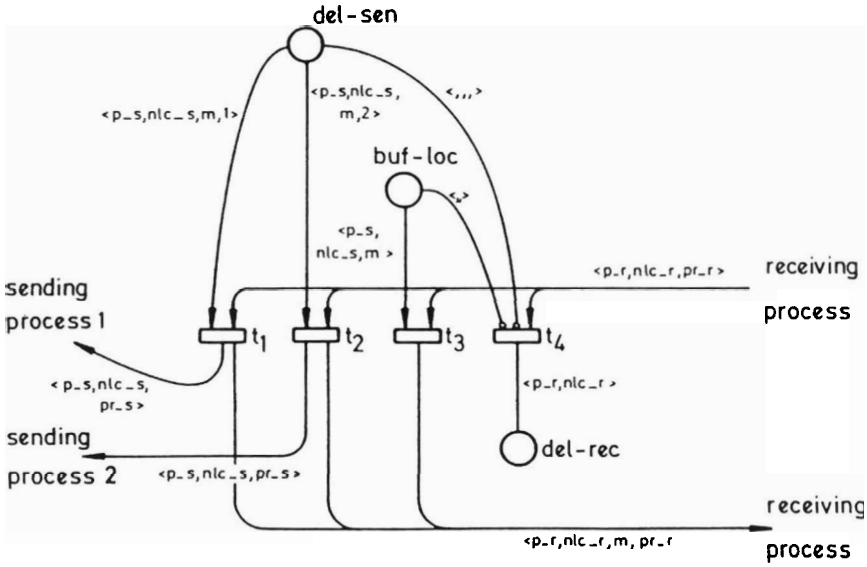

Fig 2.5: Receive expression

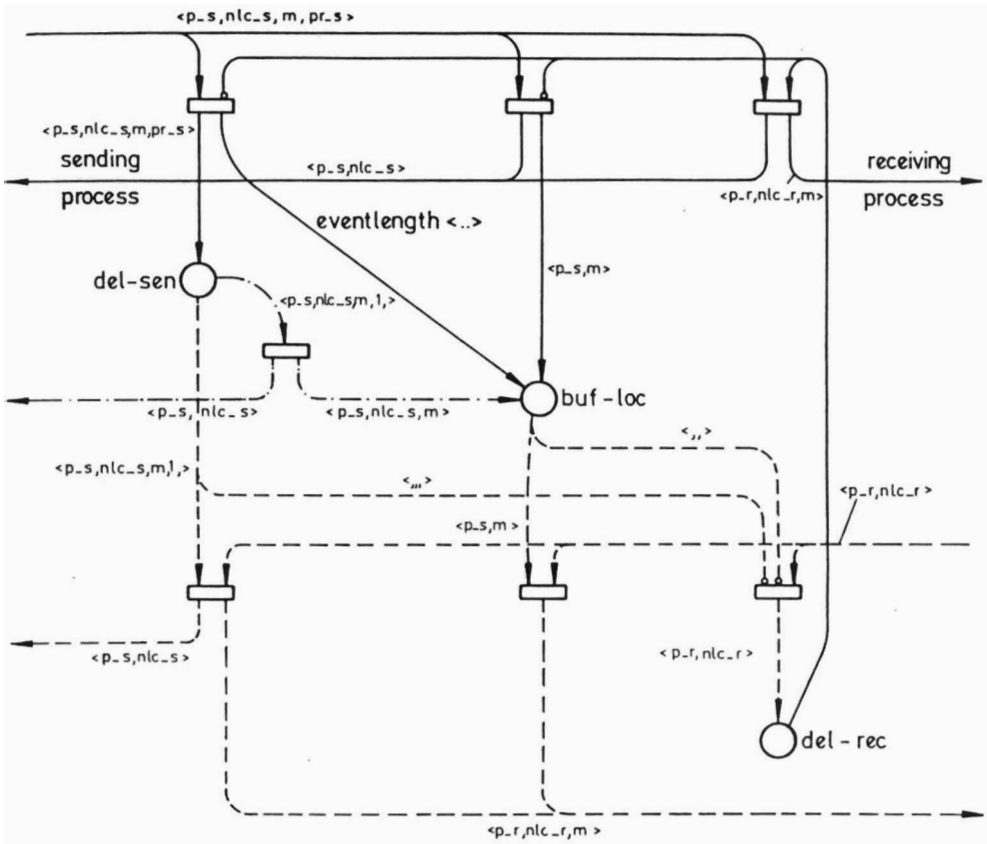

Fig. 2.6: Communication via buffer

The **receive buffer case action** is similar to the receive expression but applied to different buffer locations. If this action includes an optional ELSE-part, then the statement list following it will be executed by the PI: the PI will never be delayed.

Fig. 2.6 shows the communication of two processes (sending and receiving process) by a **send buffer case action** and a **receive expression**.
The following Chill program incorporates two sending and one receiving processes and 12 PI´s:

```
program : MODULE
  DCL store BUFFER (1 /* ansich 2 */) INT;
  putmuch:PROCESS() ...
    SEND store  (200 /* units */) PRIORITY 1;
    CWRITE(´ putmuch:  200 units delivered\n´);
  END putmuch;
  putfew:PROCESS() ...
    SEND store  (3 /* units */) PRIORITY 0;
    CWRITE(´ putfew: 3 units delivered\n´);
  END putfew;
  getall:PROCESS() ...
    DCL local_store INT INIT:= 0; /* life_time_bound init */
    CWRITE(´ getall: store empty \n´);
    DO FOR i := 1 TO 11;
      local_store := local_store + RECEIVE store;
      CWRITE(´ getall: got something\n´);
    OD;
    CWRITE(´ getall: finished\n´);
  END getall;
DO FOR i:=1 TO 10 ;
  START putmuch();
OD;
START putfew();
START getall();
END program;
```

The corresponding net (Fig 2.7) shows the communication primitives in a ´black box´. The detailed net without arc inscriptions is illustrated in Fig. 2.8. The program contains processes which execute send actions of different priorities. For a correct evaluation of the bufferlength (each priority has another place) a place called ´buffer element counter´ (b_e_c) is introduced. The sum of tokens of the places ´del_sen´ with priority 0 and ´del_sen´ with priority 1 is recorded. For every

reactivation of PI's the priority is taken into account.

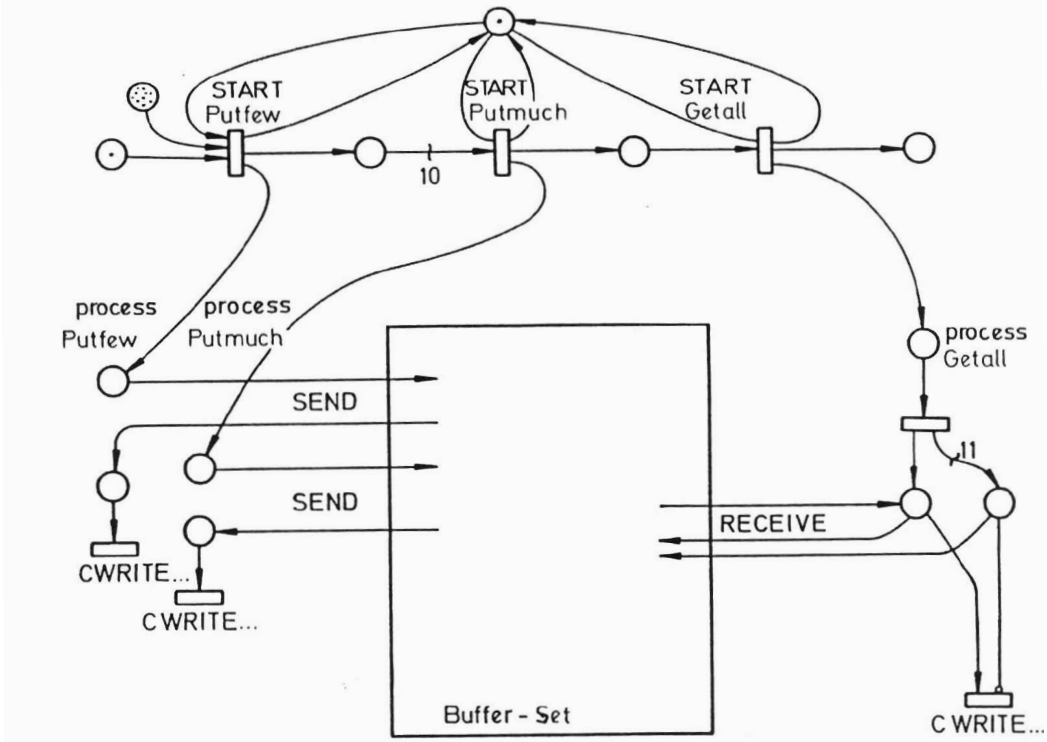

Fig 2.7: Net corresponding to the program

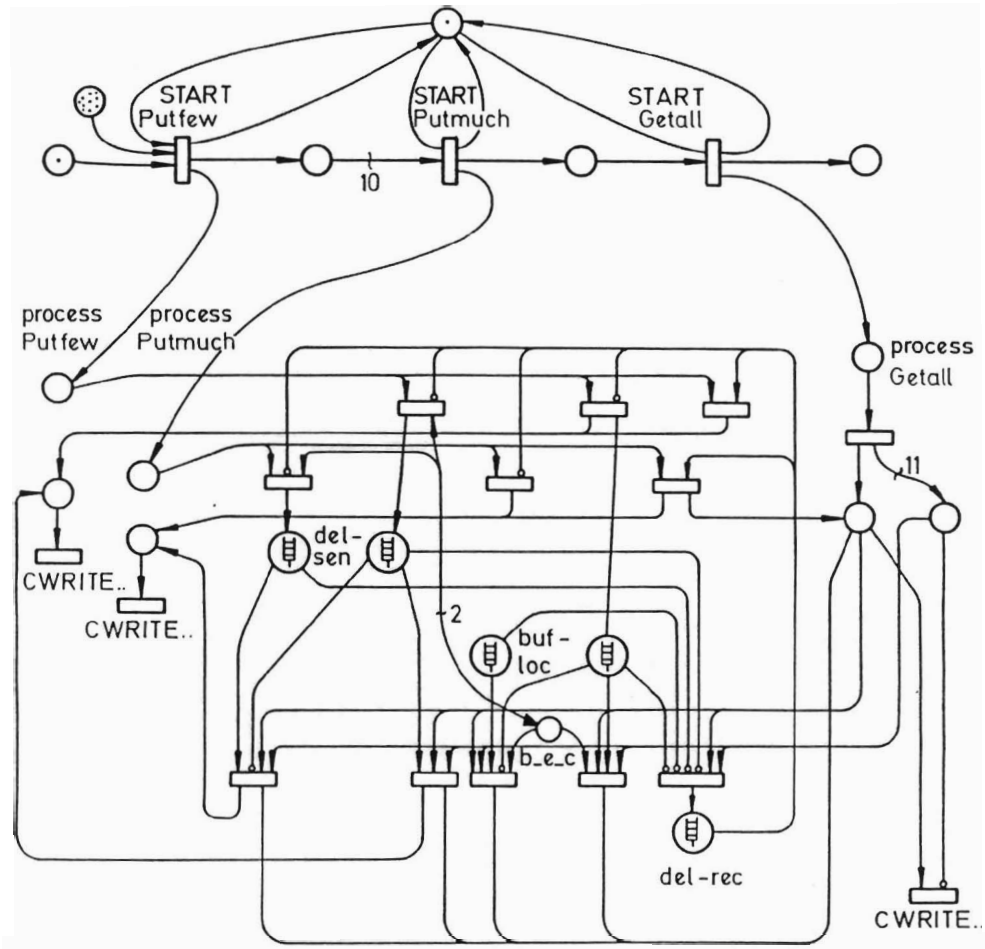

Fig. 2.8: detailed net corresponding to the program

Signal

Concurrent PI´s can communicate via sending and receiving **signals**. All signals which were sent but not received have to be queued. A sending PI will never be delayed! The send signal case action can specify a destination process and a special PI (primitive value). If none of them are specified, any PI may receive the signal. The receiving PI may specify the signals to be received. If the receive signal case action **comprises no ELSE-part and there is no appropiated signal available at** the moment, then this PI will be delayed. Signals also may have different priorities.

Chill Conclusion

The semantic of the Chill concurrency requires a high level net model (like modified predicate transition nets /Steinmetz 86a/), because tokens have to be unique for modelling recursive procedure calls and dynamic creation of PI´s. A modified predicate transition net is a predicate transition net with inhibitor and reset arcs and allowing addition, subtraction, multiplication and division being applied to the elements of the tupels of arc inscriptions. Some constructs of Chill (e.g. the receive buffer case action) demand a quite complicated net. For these tokens queues of different length and priority have to be organized. The transformations of Chill programs with about 500 lines of code, about 15 processes and extensive use of the concurrency features result in considerably complicated nets (500 places and transitions of high level nets). These results convinced us, that the analysis of Chill programs with respect to their concurrency properties should be accomplished by the following transformation concept:

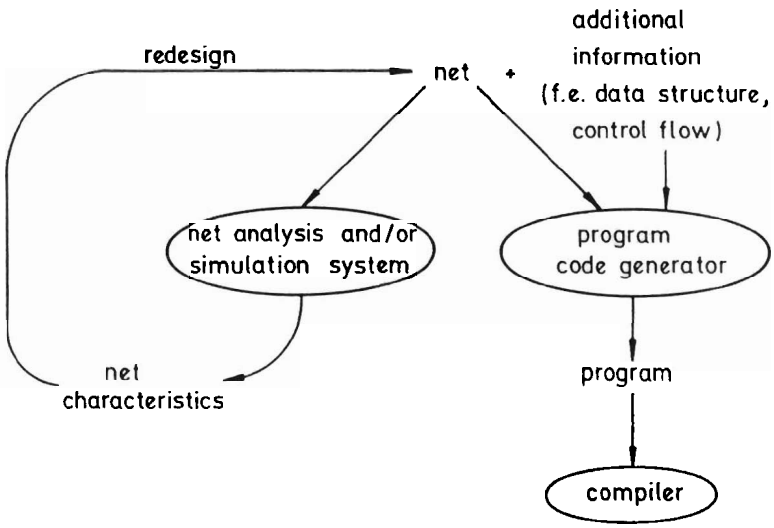

Fig. 2.9 : Appropiated transformation concept for Chill programs

3. Occam

The programming language Occam was introduced 1982 by the British company Inmos Ltd. There a special microprocessor (called transputer) architecture was conceived in conjunction with Occam. The language and the transputer as processing element is designed for concurrent processing systems. The development of Occam has proceeded and so Occam 2 was recently established as the new standard with some minor changes in the syntax and adding datatypes, multidimensional arrays and communication protocols to the original language. In this paper all references are made to the last version of Occam called also Occam 2 as defined in /May 87/ and /Steinmetz 87/.
C.A.R. Hoares original CSP /Hoare 78/ principles of synchronisation and communication were adopted in Occam in a very similar form. The relation between CSP and Petri nets, later TCSP and Petri nets, has been studied carefully (see e.g.: /Taubner 84, Dolezol Schöbel Triebel Wirtz 84, Reisig 84/). The equivalence of CSP and Occam was used to achieve the results of this paper.

For better understanding some explanations of the syntax and semantic of Occam are included in the following text. An Occam program is a **hierarchical structure of communicating sequential processes.** A process in Occam is not equivalent to a process or process instance in Chill. Every Chill action is an Occam process. An Occam process can be part of another process. With ´process´ we will denote in this chapter the Occam semantic of process.

Occam has five **primitive processes**: A SKIP causes the program to move to **the next instruction. A STOP stops a process (the following process can** never start) and can stop the whole program. Another primitive process is the assignment:

```
  var1 := var2
```

Concurrent processes communicate via channels writing values into a channel by an output process:

```
  channel ! expr
```

and reading values from a channel by an input process:

```
  channel ? var
```

Types (data structures), variables, abbreviations (including constants), channels and timers have to be specified before use. Indentation from the left hand margin is used to indicate the program structure (like

BEGIN END or { } in other languages). Comments start with double hyphen (--) and terminate at the end of the line. Three consecutive dotes (...) denote parts of the program whose internal details are not relevant at the moment. The text following these dots is a comment.

The relation between the primitive processes is defined by the constructors (SEQ, IF, WHILE, PAR, ALT, CASE). The constructs, also processes, can be part of other constructs. In Occam the structure of parallel and sequential program parts has to be formulated by the programmer.
In contrast to most of other programming languages no implicit (default) sequential relation between primitive processes exists:

```
-- incorrect, the relation between the first and the second
-- assignment process is undefined !
INT a,b:  -- specification
a := 3    -- first  assignment
b := a    -- second assignment
```

The **SEQ** construct introduces sequential ordering, one process is executed after another:

```
INT a,b :
SEQ
  a := 3
  b := a
```

Every component process is a net. The composition of these nets is a sequential ordering as shown in Fig. 3.1.

```
Program:                          Net:
  SEQ                                      a
    ... a                                  |
    ... b                                  b
    -- etc.                                |
    ... n                                  |
                                           n
```

Fig. 3.1: SEQ construct

By **PAR** the component processes are executed in parallel (together). This can be multiprogramming or multiprocessing:

```
INT a1,a2,a3:
PAR
  a1 := 1     -- these component
  a2 := 2     -- processes are
  a3 := 3     -- executed together
```

If a parallel process starts all its component processes start (meaning a,b,..,n in the example above). The process terminates if all the component processes have terminated.

Program:

```
PAR
  ... a
  ... b
  -- etc.
  ... n
```

Net:

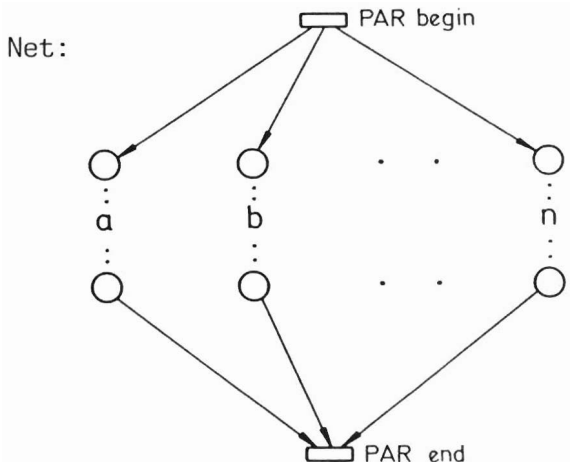

Fig. 3.2 : PAR construct

The **WHILE**, **IF** and **CASE** constructors are the repetitive and the conditional constructors respectively. The boolean expressions are modelled nondeterministically. If none of the choices (boolean expressions) of the IF construct is TRUE this process never terminates, **it stops. The corresponding transition Stop in Fig. 3.4 has only to be** incorporated if this may happen (it is considered as a programming error). Usually the last choice is TRUE and the corresponding process SKIP.

Program:

```
INT x :
SEQ
  ... other processes
  WHILE x < 100  -- boolean expr.
    ... a (loop)
```

Net:

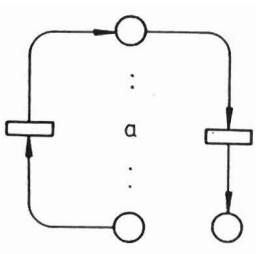

Fig. 3.3: WHILE construct

```
Program:
  INT x :
  -- to x should be assigned a
  IF        -- value before
    x < (-10)   -- 1.choice
      ... a     -- 1.process
    x = 0       -- 2.choice
      ... b     -- 2.process
    -- etc.
    x > 10000   -- n.choice
      ... n     -- n.process
```

Net:

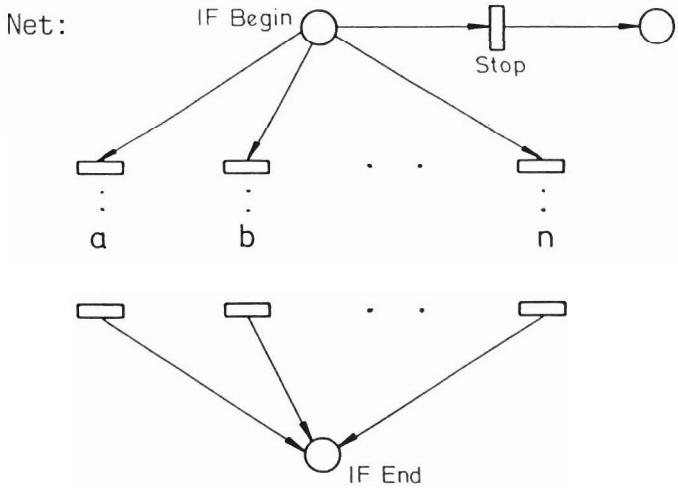

Fig. 3.4: IF construct, CASE construct leads to the same net

The communication and synchronisation method in Occam is based on channels. A **channel** always connects two (and not more) processes in one direction, point-to-point: one is an output process denoted by exclamationmark (!) and another is an input process denoted by questionmark (?). Ahead of the question- or exclamationmark the name of the channel has to be written. The transmitted value (evaluated by an expression) and the variable assigned are written after the question- or exclamationmark. If both processes are ready to communicate the value is transfered from one process to the other. A process is inactive while waiting for the communication. It makes no difference in Occam if the processes are running on one or different processors. This synchronous communication known from CSP can be represented just as a simple transition!

```
Program:
  CHAN OF INT com: -- specification
  INT x : -- of channel and variable
  PAR
    SEQ      -- process R
      com ? x
    SEQ      -- process Q
      com ! 100
```

Net:

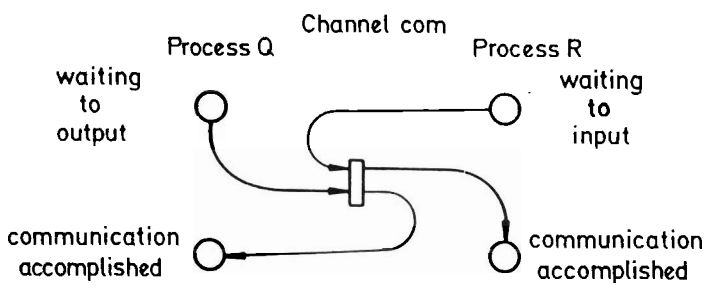

Fig. 3.5: Channel

The **ALT** construct chooses the first process ready to execute. A process is ready if the corresponding guard is active. The guard may combine a

boolean expression with an input process: e.g. (chan ? x) & (a > b). If no boolean expression is used the transition Stop can be omitted. In our programming experience we succeded in avoiding most of the boolean expressions in the ALT constructs and then usually the programs were easier to read and to understand.

Program:

```
ALT
  chan.a ? x -- guard_a
    ... a
  chan.b ? y -- guard_b
    ... b
  -- etc.
  chan.n ? z -- guard_n
    ... n
```

Net:

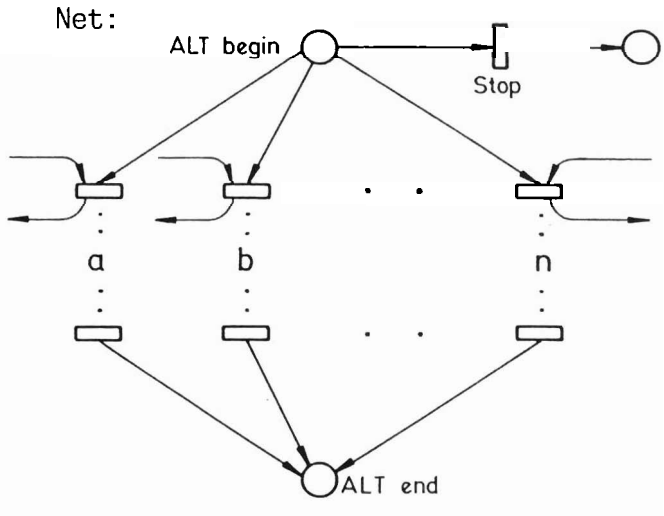

Fig. 3.6: ALT construct

The datatype of a channel is called protocol. Values of different type can also be transmitted by one channel (tagged protocol). The **input selection** construct selects one process according to a tag (selective value) received over a channel. This can be modelled by an ALT construct with nondeterministic choice of the following process (a,b,..,n in Fig. 3.6).

Like a FOR statement in Pascal a **replicator** exists in Occam. A replicator can be combined with a PAR, SEQ, ALT or IF construct to create an array of similar processes. The actual version of Occam restricts the base and count of the index of the replicator used with a PAR construct to be known before execution (at compile time). So the replicated process can be extended to an Occam processes without replicator. Knowledge of the component processes of a construct to be expanded is often needed, because in many cases just one or a few expanded processes instead of thousands or more contain the same information about synchronisation properties! This fact can be applied also to SEQ, ALT and IF constructs with base or count of a replicator to be evaluated at run time. Then these constructs can be extended in the same way as the PAR construct shown in the following example.

Program with replicator

```
[3]CHAN OF INT comm:
VAL max IS 3:
PAR
  INT i:
  [max]INT x:
  PAR i = 0 FOR max
    comm[i] ? x[i]
  INT i:
  PAR i =  0 FOR max
    comm[i] ! i
```

Expanded Program:

```
[3]CHAN OF INT comm :
PAR
  [3]INT x:
  PAR
    comm [0] ? x[0]
    comm [1] ? x[1]
    comm [2] ? x[2]
  PAR
    comm [0] ! 0
    comm [1] ! 1
    comm [2] ! 2
```

Procedures and **functions** can be expanded to a net, because recursive procedure calls are not allowed.

Program with procedure:

```
PROC add (INT result, a, b)
  SEQ
    result := a + b

INT x,y,z :
SEQ
  x := 120
  y := 130
  add (z, x, y)
```

Expanded Program:

```
INT x,y,z :
SEQ
  x := 120
  y := 130
  z := x + y
```

All assignments and specifications are not transformed into a net. If boolean expressions are required by a construct the choice is modelled nondeterministically.

The constructs and processes are composed to a new net representing the program. The following examples show an Occam program and the corresponding net. The net deadlocks, if in the net the upper right transition ´CHAN put´ fires. This is equivalent to the execution of the guard and start of the process marked with an exclamationmark.

Program with possible deadlock:

```
CHAN OF INT put, call, get:
INT ANY:
PAR
  INT info.merge, ANY:
  ALT
    call ? ANY
      SEQ
        put ? info.merge
        get ! info.merge
    put ? info.merge    -- dead-
      get ! info.merge -- lock!
  INT info.get,ANY :
  SEQ
    call ! ANY
    get ? info.get
  INT info.put :
  SEQ
    INT i, x:           -- delay
    SEQ i = 0 FOR 5000 -- delay
      x := x + 1        -- delay
    put ! info.put
```

Corresponding net:

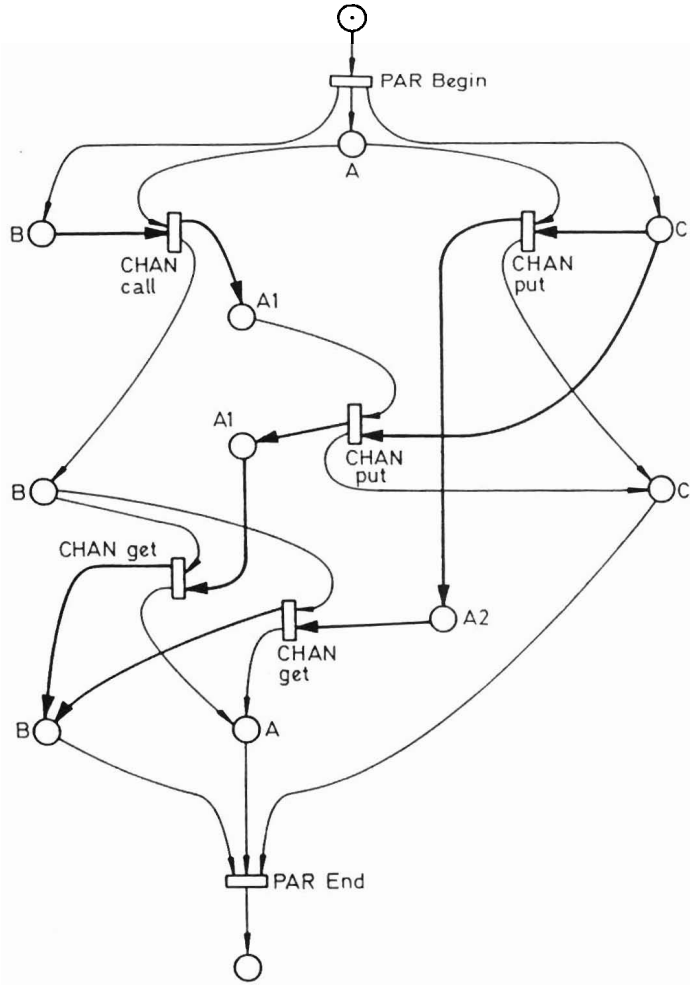

Fig. 3.7: Deadlock in an Occam program

Please note that the net model used is just the ordinary place transition net /Reisig 82, Peterson 81/. This is possible, because of some restriction on modelling level and some features of the language:

Restrictions at modelling level:

- Nondeterministic modelling of boolean expressions in guards of the IF and ALT construct.
- Evaluation of the expressions for the replicator of the SEQ and ALT construct at compile time.

Features of the language /May 87, Steinmetz 87/ which contributed to the simple net model:

- The amount of parallel processes (indicated by the replicator in the PAR construct) must be fixed at compile time.
- Procedures and functions can always be compiled either by substitution of their bodies or as closed subroutines.
- Recursions are not allowed.

We found it very appropiated to introduce the restriction concerning

nondeterminism while analysing and developing complicated Occam programs. We therefore state, that in the case of Occam the following concept (Fig. 3.8) is useful and practical. Nevertheless a scheme according to Fig. 2.9 may also be usefull.

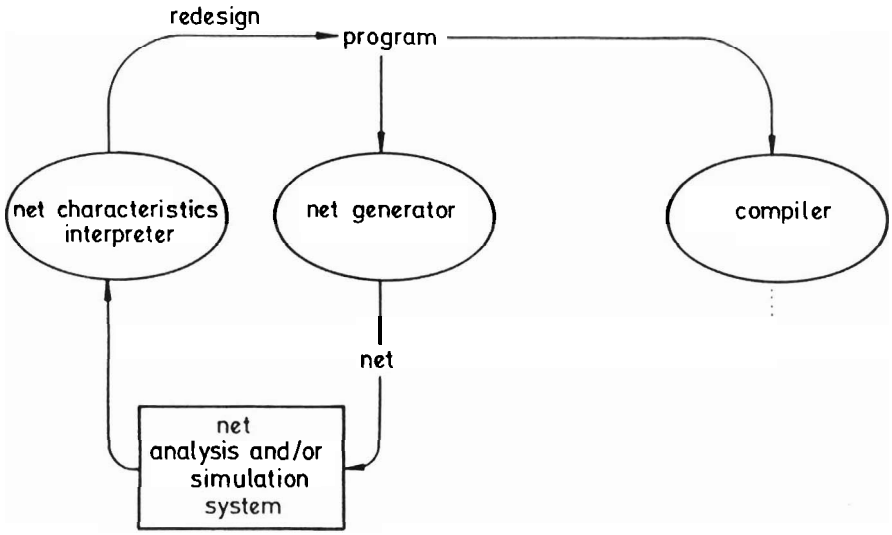

Fig. 3.8: Appropiated transformation concept for Occam programs

4. Conclusion

In Chill concurrent PI´s may start and terminate in an arbitrary manner, the amount of PI´s is not known before execution. In Occam the amount of concurrent processes is statically defined (at compile time) and the start and termination is synchronized. The rather complicated interaction of PI´s in Chill (defined with some inconsistencies /Hollis 86/) is based on shared variables (buffers, regions and events) and on message passing (signals with queuing). These mechanisms are appropiated to be executed on one central processing unit or a shared memory architecture. The message passing concept of Occam is conceived for distributed applications. A transformation of Chill programs to nets requires decomposition techniques for being analyzed /Steinmetz 86b/. The semantic of Occam can be described by algebraic laws /Roscoe Hoare 86/ or, with some abstractions, by nets as outlined in this paper.
With respect to the questions posed at the end of the introduction we found out:

1. It was shown, that the semantic of different concurrency mechanisms can be described by nets.
2. For Occam place transition nets are adequate. Chill requires a high level net model.
3. The net description of Chill may be used to describe the semantic of the concurrency in a concise form and it may be used for teaching

purposes. The application of a transformed Occam program spans from the pure description to the analysis of synchronisation properties at compile time (deadlock prevention, no detection and no avoidance). But we have to keep in mind that nondeterminisms were introduced.

4. At first glance it was very astonishing that all these results are very language dependent. But with respect to the concurrency Occam and Chill are found to be two very ´contrary´ languages: One is fairly simple and the other rather complicated. Therefore different net models and transformation processes are recommended. The two types of net needed have different Turing power. But due to the fact, that in both language-transformations data-dependencies and control-structures are modelled nondeterministically the Chill and Occam representation represent abstractions at the same level.
5. For the analysis of synchronisation properties
   - a Chill program should be derived from a net specification (Fig. 2.9),
   - an Occam program may be derived from a net or may be transformed into a net (Fig. 3.8). The advantage of the second possibility is the fairly easy incorporation into a programming environment.

The author is indebted to Dr. J. Lenzer for many valuable discussions concerning the language Chill and its concurrency and is thankful to the anonymous reviewers for their critical comments.

5. References

/ADA 83/: Reference Manual for the ADA Programming Language, US Dept. of Defence, MIL Std 1815a, 1983.

/ANDREWS SCHNEIDER 83/: G.Andrews, F.Schneider, Concepts and Notations for Concurrent Programming, Computing Surveys, vol.15, no.1, march 1983, pp.3-43.

/BISHOP 86/: J.Bishop, Practical Issues of Concurrency in ADA, Major Advances in Parallel Processing, London 9-11 dec. 1986, pp. 187-198.

/BRANQUART LOUIS WODON 82/: P.Branquart, G.Louis, P.Wodon, An Analytical Description of CHILL, the CCITT High Level Language, Lecture Notes in Computer Science, vol.128, Springer-Verlag, 1982.

/COFFMAN DENNING 73/: E.Coffman, P.Denning, Operating Systems Theory, Prentice-Hall Inc., Englewood Cliffs, N.J., USA, 1973.

/COLOM, SILVA, VILLAROEL 86/: J.Colom, M.Silva, J.Villarroel, on Software Implementation of Petri Nets and Colored Petri Nets Using High-level Concurrent Languages, 7th European Workshop on Applications and Theory of Petri Nets, Oxford, july 1986, pp.207-241.

/CZAJA 84/: L.Czaja, Nets and their Relation to CSP, Oxford University Computing Laboratory, PRG-38, june 84.

/DIJKSTRA 68/: E.Dijkstra, Cooperating Sequential Processes, In F.Genuys (Ed.) Programming Languages, Academic Press, New York, 1968, pp.43-112.

/DOLEZOL, SCHÜBEL, TRIEBEL, WIRTZ 84/: W.Dolezol, T.Schübel, J.Triebel, G.Wirtz, A Graph Grammar Providing a Predicate Transition Net Semantic for CSP-80, Proceedings of the Graphtheoretic Concepts in Computer Science, june 1984, Berlin, pp.100-124.
/HOARE 74/: C.A.R.Hoare, Monitors: An Operating System Structuring Concept, Comm. of the ACM, vol.17, no.10, Oct. 1974, pp.549-557.
/HOARE 78/: C.A.R.Hoare, Communication Sequential Process, Comm. of the ACM, vol.21, no.8, pp.323-334.
/HOARE 85/: C.A.R.Hoare, Communicating Sequential Processes, Prentice-Hall International, UK, Ltd, 1985.
/HOLLIS 86/: D.Hollis, Inconsistency, holes and problems with the CHILL concurrency definition - Comments and suggestions, Fourth Chill Conference, München, sept. 1986.
/HOLLIS KROPF 84/: D.Hollis, P.Kropf, Modelling of the Concurrency in Chill with PR/T Nets, Third CHILL Conference, Cambridge, sept. 1984.
/INMOS 84/: Occam Programming Language Manual, Prentice-Hall Inc., Englewood Cliffs, N.J., USA, 1984.
/INMOS 86/: Transputer Reference Manual, Inmos Ltd, 1986.
/KROPF 86/: P.Kropf, A Comparison between the Languages CHILL and Occam, Fourth Chill Conference, München, september 1986, pp.145-151.
/LEA 86/: R.M.Lea, On Overview of the Influence of Technology on Parallelism, Major Advances in Parallel Processing, London 9-11 dec. 1986, pp.5-15.
/LENZER LETSCHERT HOLLIS LINGEN 87/: J.Lenzer, T.Letschert, D.Hollis, Lingen, Eine Einführung in die Programmiersprache Chill, to be published at Dr. Alfred Hüthig Verlag GmbH, Heidelberg, 1987.
/MAY 83/: D.May, Occam, ACM Sigplan Notices, vol.18, no.4, april 1983, pp.69-79.
/MAY 87/: D.May, Occam 2, Formal Specification, Inmos Ltd., feb. 1987.
/PETERSON 81/: J.Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall Inc., Englewood Cliffs, N.J., USA, 1981.
/REISIG 82/:W.Reisig, Petrinetze, Springer-Verlag, 1982.
/REISIG 84/: W.Reisig, What Operational Semantics is Adecuate for Nonsequential Systems ?, Series B, Nr.30, Digital Systems Laboratory, Helsinki University of Technology, 1984.
/ROSCOE HOARE 86/: A.W.Roscoe,C.A.R.Hoare, The Laws of Occam Programming, Oxford University Computing Laboratory, PRG-53, feb. 1986.
/SAMMER SCHWÄRZEL 82/ W.Sammer, H.Schwärzel, CHILL: Eine moderne Programmiersprache für die Systemtechnik, Springer-Verlag, 1982.
/STEINMETZ 84/: R.Steinmetz, Darstellung von Monitoren durch Petri Netze, Angewandte Informatik, no.8, aug. 1984, pp.314-321.
/STEINMETZ 86a/: R.Steinmetz, Modularisierte Petri-Netze zur Beschreibung und Analyse nachrichtentechnischer Systeme mit vermaschten informationsverarbeitenden Strukturen, VDI Fortschritt-Berichte, Reihe 10, no. 56, may 1986.
/STEINMETZ 86b/: R.Steinmetz, Ein Qualitätskriterium zur Partition von Petri-Netzen, Angewandte Informatik, dec. 1986.
/STEINMETZ 87/: R.Steinmetz, Occam 2: Ein Einführungs- und Nachschlagewerk, Dr. Alfred Hüthig Verlag GmbH, Heidelberg, 1987.
/TAUBNER 84/: D.Taubner, Petri Netze als Grundlage für die Semantiken von CSP und CCS, Mitteilung Nr.127, Fachbereich Informatik, Universität Hamburg, sept. 1984.
/WELSH LISTER 81/: J.Welsh, A.Lister, A Comparative Study of Task Communication in ADA, Software-Practice and Experience, vol.11, 1981, pp.257-290.
/Z.200 84/: CCITT High Level Language (CHILL), Recommmendation Z.200, may 1984.