An Experiment in Implementing Process Algebra Specifications in a Procedural Language

C. Groza
Department of Computer Science, Technical University of Timişoara
Bd. V. Pârvan 2, 1900 Timişoara, România
email: calin@utt.ro

Abstract

This paper presents a method for the implementation of process algebra specifications in a (structured) imperative language. The original specification is written in the Process Specification Formalism (PSF). The actual implementation is made in an extension of the C language for concurrent programming. First we make an informal presentation and then we consider a formal approach.

Using the method presented in this paper we have implemented several communication protocols. As an example, we present the implementation of the Positive Acknowledgement with Retransmission (PAR) Protocol.

1 Introduction

The motivation to have a systematic approach for the implementation of process algebra specifications came during the development of an application which uses a communication protocol. On one side there was a formal specification of the protocol. On the other side, in order to use a protocol in the application it was required to have an implementation in a structured imperative language. The purpose of this paper is to present a method to bridge the gap between process algebra specifications and implementation in software development. For the specifications we used the Process Specification Formalism (PSF) and for the implementation the C language extended with functions for processes management and communication.

The implementation of process algebra specifications was also addressed in [Vra88]. There, the process algebraic specifications were implemented in POOL-T – an object oriented language for concurrent programming.

The next section presents a brief description of the PSF formalism. Section 3 describes the extension of the C language used in the implementation. In Section 4 we present, informally, the implementation of the basic PSF constructs. As an example we give the implementation of the Positive Acknowledgement with Retransmission Protocol in Appendix B. A formal approach to the implementation is presented in Section 5 and an example in Section C.

Contents

1 Introduction **1**

2 The Process Specification Formalism (PSF) **2**

3 EC: an extension of C for concurrent programming **3**
3.1 Tasks 3
3.2 Communications 3
3.3 Timers 4
3.4 The run-time environment of EC programs 4

4 From PSF specifications to EC programs **5**
4.1 Data 5
4.2 Processes 6
4.3 Parallel composition 6
4.4 Sequential composition 7
4.5 Alternative composition 8
4.6 Performance 10

5 Toward a formal specification of the implementation **11**

6 Conclusions **12**

A The implementation of the EC run-time environment **14**

B The implementation of the Positive Acknowledgement with Retransmission Protocol **16**
B.1 Data structures 17
B.2 Parallel composition and communications 18
B.3 Processes A and B 19
B.4 Sender 19
B.5 Channel K and Channel L 20
B.6 Receiver 22
B.7 Timer 22

C A formal specification of the implementation of PSF process definitions **23**

2 The Process Specification Formalism (PSF)

The Process Specification Formalism (PSF) is an algebraic formalism used for the specification of concurrent systems. The formal definition of PSF can be found in [MV93]. A set of tools (such as compiler, simulator) are available to analyse a specification.

PSF is based on a formalism for algebraic specification of data: Algebraic Specification Formalism (ASF) [BHK89] and a process algebra: Algebra for Communicating Processes (ACP) [BW90]. PSF supports the modular construction of specifications and parameterization of modules.

A PSF specification contains two kinds of modules:

Data modules contain the algebraic specifications of data. The major parts are: the definition of the *signature* and the description of the properties by means of *equations*.

Process modules contain the definition of PSF processes. They are defined as a series of atomic actions combined by operators. Atomic actions are the basic and indivisible elements of processes. The basic operators are: sequential composition, parallel composition, and alternative composition. Atomic actions and operators are used in the (recursive) definition of processes.

Parallel processes can communicate with each other. The atomic actions used in communications are declared in a separate section of the process modules.

PSF does not allow explicit representation of time. Nevertheless it is possible to introduce processes which represent timers. Timers are often used in communication protocols in order to deal with lost messages.

The following example is a specification of a *producer-consumer* system. There are two processes, the `Producer` and the `Consumer` which communicate with each other through a port:

```
data module Data
begin
  exports
    begin
    sorts
        DATA
    functions
        data-unit : -> DATA
    end
end Data

process module Prod-Cons
begin
  exports
    begin
      atoms
        send, recv, comm: DATA
        produce-data
        consume-data: DATA
      processes
        Producer, Consumer, Prod-Cons
      end
    imports
      Data
    sets
      of atoms
        H = { send(d), recv(d) | d in DATA}
    communications
      send(d) | recv(d) =
          comm (d ) for d in DATA
    definitions
      Producer = produce-data. send(data-unit).
                                Producer
      Consumer = sum ( d in DATA, recv(d).
                                consume-data(d)). Consumer
      Prod-Cons = encaps ( H, Producer || Consumer )
  end Prod-Cons
```

3 EC: an extension of C for concurrent programming

The language used in the implementation is C extended with functions for process management and interprocess communications. The extension (called EC) adds features usually offered by concurrent programming languages.

In order to differentiate the EC processes from PSF processes, the former will be called *tasks*. A *task* is defined by the its *code* and *data*. The code is executed sequentially, thus, it is not possible to have parallel execution of different parts of a task. The tasks communicate with each other by sending/receiving messages through *ports*. When one task issues a *send* to a port and another tasks issues a *recv* to the same port, the tasks synchronize and exchange data. The communication primitives are synchronous: when one task gets to a communication primitive it is blocked until another task executes the complementary primitive. Two boolean functions can be used to check if there is a message or a request pending at a port.

3.1 Tasks

A task is defined by both the code to be executed and the data. The code is defined as a C function.

ECtcreate - creates a task

```
#include <ec.h>
int ECtcreate( TASK (*func)(), int nargs, ... );
```

`ECtcreate` creates a new task which executes the code defined in function `func`. `nargs` is the number of simple-type (int) arguments given to the task. The function returns -1 on error and a non-negative integer on success. `TASK` is another name for the type `void`. It was introduced to increase the readability of the code. The functions which define a task code return the type `TASK`.

3.2 Communications

Communications in PSF are synchronous. We have implemented a similar mechanism in EC. Another solution could have been an event-driven mechanism with direct communication between tasks similar to that used in [Tan89].

The following functions are used for communication:

ECpcreate - creates a port

```
#include <ec.h>
int ECpcreate(PORT *p, int data_size);
```

PORT is the data structure for ports. ECpcreate initializes the data of the port pointed to by p. The second argument is size of the data transmitted through the port.

ECsend, ECrecv - sends data to port, receives data from port

```
#include <ec.h>
void ECsend(PORT *port, void *pdata);
void ECrecv(PORT *port, void *pdata);
```

ECsend sends to the port indicated by p the data stored at the address pdata. ECrecv receives data from the port pointed to by p and stores it at the address pdata. The functions block the current task until another task executes the complementary communication function.

ECexist_data, ECexist_request - these functions check if there is data at a port or a request for data respectively.

```
#include <ec.h>
bool ECexist_data(PORT *p);
bool ECexist_request(PORT *p);
```

ECexist_data returns TRUE if another process sent (using ECsend) data to the port p and FALSE otherwise. ECexist_request checks if there is a request for data (another task issued a call to ECrecv) at the port p. These functions do not block the task.

3.3 Timers

We defined the data structure TIMER and the following functions for the implementation of timers in EC programs:

ECstart_timer - starts a timer

```
#include <ec.h>
void ECstart_timer(TIMER *t, long usec);
```

TIMER is a data structure used in the implementation of the timers. ECstart_timer updates the timer data. The time constant, in microseconds, is given as the second argument of the function.

ECtimeout - indicates whether a "time-out" occurred

```
#include <ec.h>
bool ECtimeout(TIMER *t);
```

ECtimeout returns TRUE if the time elapsed since the task issued ECstart_timer is greater than the time constant of the timer.

3.4 The run-time environment of EC programs

For the implementation of the run-time environment of EC programs we have used the lightweight processes library available in the UNIX environment. At the EC level, the application is a collection of tasks which run in parallel and communicate synchronously through ports. The EC level is implemented on the top of the lightweight processes environment. At the lightweight processes level an application consists of a set of threads which are running in parallel and communicate synchronously with each other.

The main primitives offered by the lightweight processes library are:

- create and destroy lightweight processes (also called threads).

- synchronous communication between threads. A thread can send a message to another thread and then it waits for a reply or, it can wait to get a message. When the *rendezvous* happens, the threads exchange data, the receiver is unblocked, and, it can resume the sender by sending a reply.

The EC tasks are implemented by threads. The task parameters are passed to the thread when it is created. The communication between EC tasks is indirect, through ports. In the lightweight processes environment, the communication between threads is direct. Thus, in order to implement the EC communications a thread is created for every port. The EC functions `ECsend` and `ECrecv` are implemented by communications between the threads for tasks and the threads for ports. Every port contains a queue in which the messages are stored until rendezvous.

In the implementation of the timers we use an internal clock. Function `ECstart_timer` stores the current time and the time constant in the timer data. Function `ECtimeout` compares the current time with the timer data and decides if a "timeout" occured.

The Appendix A contains the C code of the EC run-time environment and a description of the data structures and functions.

4 From PSF specifications to EC programs

In this section we show, informally, how an EC implementation can be derived from a PSF specification and which are the limitations.

The primary goal is to determine the control structures in the EC implementation based on the PSF process definitions. We have identified a subset of PSF often used in the specification of real systems. We show for this subset, by means of examples, how to make an EC implementation. In section 5 we present a formalization of this method. The focus is on process definitions, but, we present also how other parts of PSF specifications (data, communications) are implemented.

4.1 Data

There are two aspects which are related to data:

1. The definition of abstract data types. In PSF specifications the data modules contain the algebraic specification of data. In EC, abstract data types are defined using the C data structures and functions.

2. The use of data. In PSF, the processes and the atomic actions can have parameters. In EC, data is used as parameters of tasks and functions and in variables, which form an environment for computation.

The abstract data types specified in the PSF data modules are implemented in EC by means of C data structures and functions. Although both C and PSF provide support for the definition of abstract data types, a number of differences makes difficult a direct conversion of PSF data definition into EC code. First, PSF uses an algebraic approach for data definition while C uses an imperative approach. Second, the syntax of the interface with the abstract data types is more flexible in PSF than in C. For example, in PSF the names of the functions can be overloaded but in C the names of the functions must be unique.

We have implemented several communication protocols from their PSF specifications. In these cases the data definition was simple and we used an ad-hoc implementation in C of the abstract data types specified in PSF. In general, the PSF sorts have corresponding C data types in the implementation and PSF functions have correspondent C functions in the implementation. For example, let us consider the data structure used in the specification of the Positive Acknowledgement with Retransmission

Protocol to describe the structure of the messages sent through a channel. On the left side is the PSF specification and on the right side is the EC implementation.

⟨*Frames - psf*⟩≡
```
data module Frames
begin
  exports
    begin
    sorts
        FRAME
    functions
        frame: DATA # BIT   -> FRAME
        ce:                 -> FRAME
        error: FRAME        -> BOOLEAN
        bit: FRAME          -> BIT
        datum: FRAME        -> DATA
    end
  imports
    Booleans, Bits, Data
  variables
    b: -> BIT
    d: -> DATA
  equations
    [fr-01] error(frame(d,b)) = false
    [fr-02] error(ce) = true
    [fr-03] bit ( frame (d,b)) = b
    [fr-04] datum ( frame (d,b)) = d
end Frames
```

⟨*Frames - ec*⟩≡
```
typedef struct {
    int error;
    BIT b;
    DATA d;
} FRAME;

DATA datum(FRAME f)
{
    return f.d;
}

BIT bit(FRAME f)
{
    return f.b;
}

int frame_error(FRAME f)
{
    return f.error;
}
```

In PSF, data occurs in process definitions as parameters of processes and atomic actions. One important difference between PSF and EC in using data is the way variables take values. In PSF this is done by matching terms while EC provides the assignment statement. We impose the following restriction on the PSF specification: matching is allowed only between a free variable and an expression. More general term matching can be converted to this simple case and conditional expressions.

4.2 Processes

If a PSF specification contains only regular process expressions it is possible to make an implementation based on the finite automaton model. The implementation of an entire system as one finite automaton has disadvantages. For example, a single finite automaton assumes that all the processes share a global memory which contains the state and a representation of the transitions. This assumption is not justified in applications in which the processes are physically distributed and do not share a global memory.

Instead of using a single finite automaton, our implementation uses a set of finite automata which communicates with each other through messages. The implementation is based on the following rules:

- the PSF specification must contain only regular process definitions;
- for every PSF process which occurs in parallel composition the implementation is an EC task.
- every task executes the code of a finite automaton which implements a collection of PSF processes.

Throughout the rest of this section we will detail these rules. First we present how the process definitions which use parallel composition determine the tasks in the implementations. Then, we present how the transitions in the finite automata can be determined from the rest of PSF definitions.

4.3 Parallel composition

The process definitions which are using parallel composition determine which are the tasks in the implementation. For example, let us consider the process `P` defined as the parallel composition of the processes `Q` and `R`:

⟨*PSF - parallel composition*⟩≡
```
    P = Q || R
```

⟨*EC - task definitions*⟩≡
```
TASK Q ( )
{
        ⟨code for the finite automaton Q⟩
}

TASK R ( )
{
        ⟨code for the finite automaton R⟩
}

TASK P ( )
{
  ECtcreate(Q, 0);
  ECtcreate(R, 0);
}
```

The code for the finite automaton has the form:

⟨*code for the finite automaton Q*⟩≡
```
enum {Q, ⟨other state names⟩} state;
state=Q;
while ( TRUE ) {
        ⟨code for transitions in the finite automata Q⟩
}
```

The implementation contains the definitions of the tasks Q and R which correspond to the operands of the PSF definition. They execute the code of finite automata. The implementation contains also the definition of task P corresponding to the PSF process P. If the operands of the parallel composition have parameters the correspondent task definitions have parameters too.

Often, the form of process definitions with parallel composition is:

```
P = encaps ( H, Q || R || S ... )
```

where H is an encapsulation set which contains the components of communications. The encapsulation operator is used to enforce the communication between parallel processes. EC does not contain an explicit element to express the *encapsulation* operator, but, because the EC communication functions are synchronous, the execution of the tasks in parallel implements not only the parallel composition but also the encapsulation.

4.4 Sequential composition

The basic elements which occur in sequential compositions are atomic actions and processes. For every PSF communication the implementation contains a declaration of a port:

```
atoms
  s3, r3, c3 : FRAME                         PORT port3;
communications
  s3(f)|r3(f) = c3(f) for f in FRAME
```

In general, it is possible to identify a *sender* and a *receiver* in communications, so, the atomic actions are divided in *send* and *receive* atomic actions. The *send* atoms are implemented by calls to the EC function ECsend.

```
s3(f)                                         ECsend(&port3, &f)
```

In the subset of PSF that we use the argument of a *recv* atom must be a free variable. Therefore, *recv* atomic actions always occur in a generalized alternative composition. The implementation is a call to the function ECrecv which assigns the value sent by ECsend to the variable given as argument to ECrecv.

```
sum(f in FRAME, r3(f))                        ECrecv(&port3, &f)
```

Although, sum(f in FRAME, r3(f)) is a process expression and not an atomic actions we will assimilate this type of process expressions to the atomic action *recv*.

For every PSF process which occurs in a sequential composition the implementation contains a state in the finite automaton. If the process has parameters, the EC state is attributed with data. The attributes are implemented as local variables within the scope of a task. We will use in the following examples the function `attrib` to express the setting of a state attribute:

`attrib ( S, Val )` - assigns the value `Val` to the attribute of state `S`.

The implementation of the sequential compositions contains a sequence of calls to EC communication functions and the transition to a new state.

⟨*PSF - sequential composition*⟩≡

```
P(f1) = s2(f1). sum(f2 in F, r3(f2). Q(f2))
```

⟨*EC - sequential composition*⟩≡

```
if ( state == P ) {
  ECsend(&port2, &f1);
  ECrecv(&port3, &f2);
  attrib(Q, f2);
  state = Q;
}
```

4.5 Alternative composition

We present the implementation of three kinds of alternative composition:

1. the operands begin with communication atomic actions,
2. the operands are conditional expressions, and,
3. the non-deterministic alternative composition.

In the first case the implementation is a loop in which the alternative ports are *polled* to see if there is available data.

⟨*PSF - alternative composition*⟩≡

```
P = sum( d1 in DATA1,
               r1(d1).Q1(d1))
    +
    sum( d2 in DATA2,
               r2(d2).Q2(d2))
```

⟨*EC - alternative composition*⟩≡

```
if ( state == P ) {
    while ( TRUE ) {
        if ( ECexist_data (&port1)) {
            ECrecv(&port1, &d1);
            attrib( Q1, d1);
            state = Q1;
            break;
        }
        if ( ECexist_data (&port2 )) {
            ECrecv(&port2, &d2);
            attrib( Q2, d2);
            state = Q2;
            break;
        }
    }
}
```

The implementation uses an asymmetric communication model. In every communication it is possible to identify an active part and a passive one. The active part "triggers" the communication. The passive part watches the port to see if the active part is present. The implementation assumes that the prefixes of the alternatives are passive atomic actions. Often, the processes expressions which occur in alternative composition start with "read" atomic actions. This means that, in general, the *receive* atoms are the passive part. In these cases we use function `ECexist_data` to check if data is available at the port. But in some cases *send* atoms are the prefix of the alternatives. For example, in the Concurrent Alternating Bit Protocol (CABP) [MV93], the process Sender can alternatively send data or receive an acknowledgement. In this case *send* will be implemented as the passive part of the communication because it occurs in an alternative composition. In the implementation we will use the function `ECexist_request` to check if there is a "request" for data from the Sender.

One disadvantage of this implementation is that it uses a *busy wait* solution for synchronization between tasks. If one is mainly interested in the use of the EC code as a pseudo code description, the implementation should be seen as a *select ... accept* structure available in concurrent programming languages based on the *rendezvous* paradigm such as Ada. If one is interested in the speed of the implementation a solution is to use a new function, `ECtresched` (tasks reschedule):

ECtresched - selects another task to run

```
#include <ec.h>
void ECtresched(void);
```

the scheduler selects a new task to run based on a round-robin policy.

The call to ECtresched is inserted at the end of the loop body. Thus, after polling the ports once, if there is not data available, a new task will take the control. The example given above becomes:

```
P =   sum( d1 in DATA1,
               r1(d1).Q1(d1))
      +
      sum( d2 in DATA2,
               r2(d2).Q2(d2))
```

```
if ( state == P ) {
    while ( TRUE ) {
        if ( ECexist_data (&port1)) {
            ECrecv(&port1, &d1);
            attrib( Q1, d1);
            state = Q1;
            break;
        }
        if ( ECexist_data (&port2 )) {
            ECrecv(&port2, &d2);
            attrib( Q2, d2);
            state = Q2;
            break;
        }
        ECtresched();  /* reschedule the tasks */
    }
}
```

Another case of alternative composition is when the operands are conditional expressions. The implementation uses conditional statements to choose one alternative.

⟨*PSF - conditional expressions*⟩≡

```
P(d) = [func(d)=true] -> Q
       +
       [func(d)=false] -> R
```

⟨*EC - conditional expressions*⟩≡

```
if ( state == P ) {
    if ( func(d) ) {
         state = Q;
    }
    if ( not(func(d)) ) {
        state = R;
    }
}
```

Let us consider the case of a process P which receives an acknowledgement. The acknowledgement can be correct or it can be damaged. This situation is specified in PSF using alternative composition:

```
P = r6(ack).Q + r6(ack-error).R
```

Because the arguments of *recv* atomic actions must be free variables, in order to make an implementation, the specification is rewritten using conditional expressions. The implementation combines the rules for sequential composition and for alternative composition of conditional expressions.

```
P = sum ( a in ACK, r6(a).(
     [error(a)=false] -> Q
     +
     [error(a)=true] -> R
   ))
```

```
if ( state == P ) {
    ECrecv(&port6, &a);
    if (  not(error(a)) ) {
        state = Q;
    }
    if ( error(a) ) {
        state = R;
    }
}
```

For the specification of internal steps the formalism provides the atomic action *skip*. It is used in specifications to express nondeterministic choices. For example, let us consider a channel which transmits a frame. The frame can be transmitted correctly, it can be lost or it can be damaged.

⟨*PSF - nondeterministic alternatives*⟩≡

```
K1(f) = skip.K2        -- lost
  + skip.s4(ce).K2     -- damaged
  + skip.s4(f).K       -- correct
```

⟨*EC - non-deterministic alternatives*⟩≡

```
if ( state == K1 ) {
    switch ( random() % 3 ) {
      case 0: state = K2;
        break;
      case 1: f.error = TRUE;
        ECsend(&port4, &f); state = K2;
        break;
      case 2: ECsend(&port4, &f);
        state = K;
        break;
    }
}
```

The implementation uses a function which generates random numbers. The result is used to select a branch in a *switch* statement.

We present the implementation for timers in the context of alternative composition. Timers occur, in general, in situations informally described by: "receive a message in maximum n microseconds" which can be transformed in "receive message OR receive time-out". A timer is specified in PSF as:

⟨*PSF - specification of a timer*⟩≡

```
T = r1(start-timer).T1
T1 = r1(start-timer).T1 + s3(time-out).T
```

In the case of the process `T` the implementation uses the rules given for sequential composition while in the case of `T1` the implementation is similar to the alternative composition when the operands start with communication atoms. The difference is that for the alternative starting with the atomic action `s3(time-out)` the implementation does not contain the test for "request" – `ECexit_request`, but the test for "timeout" – `ECtimeout`.

⟨*EC implementation of a timer*⟩≡

```
#define TIME_CONSTANT 10000 /* usec */

TIMER Timer;
TASK T ()
{
    enum { T, T1 } state;
    enum { TIME_OUT_SIGNAL,
        START_TIMER_SIGNAL } timer_signal;

    state = T;
    while ( TRUE ) {
        if ( state == T ) {
            ECrecv(&port1, &timer_signal);
            ECstart_timer(&Timer, TIME_CONSTANT);
            state = T1;
        }
        if ( state == T1 ) {
          while ( TRUE ) {
          if ( ECexist_data ( &port1 )) {
            ECrecv(&port1, &timer_signal);
            ECstart_timer(&Timer, TIME_CONSTANT);
            state=T1;
            break;
            }
          if ( ECtimeout(&Timer)) {
            timer_signal=TIME_OUT_SIGNAL;
            ECsend(&port3, &timer_signal);
            state=T;
            break;
          }
          ECtresched();
         }
       }
    }
}
```

Based on the examples presented in this section we have implemented several communication protocols: Alternating Bit Protocol (ABP), Positive Acknowledgement with Retransmission (PAR) Protocol, Concurrent Alternating Bit Protocol (CABP), Simple Token Ring (STR) Protocol [MV93]. In Appendix B we present the implementation of the Positive Acknowledgement with Retransmission Protocol.

4.6 Performance

We have compared the speed of the EC implementation with the the speed of the PSF simulator, a tool available in the PSF-Toolkit [MV93]. The test consists in the execution of three protocols (ABP, CABP and PAR without real timers) for a fixed amount of time, counting the number of transitions. The following table contains the results (`tr/sec` means transitions per second).

	PSF-simulator	EC implementation
ABP	11 `tr/sec`	2967 `tr/sec`
CABP	10 `tr/sec`	2115 `tr/sec`
PAR	11 `tr/sec`	2238 `tr/sec`

The results show that the EC implementation is significantly faster than the simulation.

5 Toward a formal specification of the implementation

Section 4 presents, informally, how the PSF specification can be used to make an EC implementation. Appendix B presents the complete implementation of the Positive Acknowledgement with Retransmission Protocol in EC. The examples are instantiations of the "rules" to transform PSF structures into EC code. We made a formal specification of this rules. The specification is not complete, in the sense that it does not generate an EC "ready-to-run" program. However, it generates the main control structures of the EC implementation and suggests the operations on data.

We make a summary of the restrictions and assumptions on the PSF specification and EC implementation introduced in Section 4:

- the examples cover only a subset of PSF;
- the implementation of data is not generated from the PSF specification;
- we used the following name conventions: *send* atoms have the form `s Nat ( Data )` , *recv* atoms `r Nat ( Data )`, and communication atoms `c Nat ( Data )`. All communications have the the form:

```
communications
  s⟨Nat⟩(⟨Data⟩) | r⟨Nat⟩(⟨Data⟩) = c⟨Nat⟩(⟨Data⟩) for ⟨Data⟩ in DATA
```

- we made the implicit assumption that the specification defines one set which contains all the components of communications:

```
sets
  of atoms
    H = { r1(d1), s1(d1) | for d1 in DATA1}
        + { r2(d2), s2(d2) | for d2 in DATA2}
        + ...
```

- the parallel compositions are "encapsulated" with `H` as encapsulation set.
- the operators used in process expressions are:
  1. parallel composition,
  2. sequential composition, and,
  3. three kinds of alternative composition: in one the alternatives begins with atoms which are part of communications, in the second the alternatives begins with the atom *skip*, in the third case the alternatives are guarded with conditions.
- the PSF specification does not make explicit use of timers. Some "extra" information is required in order to make an implementation with real timers.

For the formal specification of the implementation we used the ASF+SDF Formalism. It consists of two formalisms: the Syntax Definition Formalism (SDF) [HHKR89, HK89] for describing syntax of a context-free language and the Algebraic Specification Formalism (ASF) [BHK89] for describing semantics by means of (conditional) equations. We have made a formal specification of a transformation from the PSF subset which satisfies the conditions presented above into EC. The major parts of the specification are:

- a simple definition of the EC syntax.
- a specification of the syntax of the PSF process definitions.
- a specification of the function `map` which translates PSF structures into EC code.

The ASF+SDF specification is given in Appendix C.

We developed the ASF+SDF specification in the ASF+SDF Meta-environment [Kli93]. An important feature offered by the environment is the possibility to "execute" the specifications. In our case the input is a set of PSF process definitions and the output is a "primitive" EC implementation. We give in Appendix C the result of the "execution" of the ASF+SDF specification when the input consists of the PSF process definitions of the Positive Acknowledgement with Retransmission Protocol.

6 Conclusions

This paper presents the implementation of PSF specifications in an extension of C for concurrent programming. In order to use this method the specification must satisfy some conditions which are discussed in Sections 4 and 5. The conditions are not too restrictive for the specifications of real systems. We used the method for the implementation of process algebra specification presented above for the implementation of four protocols: Alternating Bit Protocol (ABP), Parallel Acknowledgement with Retransmission Protocol (PAR), Concurrent Alternating Bit Protocol (CABP), and Simple Token Ring Protocol (STR). The PSF specifications for these protocols are presented in [MV93].

In Section 5 we consider a formalization of the transformation from PSF to EC. In Appendix C we give a such a formal specification. Future work will be oriented in this direction. Based on the rules presented in this paper and a compiler for data, it would be possible to make a compiler of the PSF specifications into EC code.

The present implementation is based on a synchronous communication mechanism. Another direction for future work is to study the possibility to make the implementation using other mechanisms for communication such as asynchronous communication or event-driven communication.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HK89] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 283–297. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 6.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Knu84] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[Tan89] A.S. Tanenbaum. *Computer networks*. Prentice-Hall International, Englewood Cliffs, 1989.

[Vra88] J.L.M. Vrancken. The implementation of process algebra specifications in POOL–T. Report P8807, Programming Research Group, University of Amsterdam, 1988.

A The implementation of the EC run-time environment

This section presents the implementation of the EC extension for concurrent programming. The basic features of EC were presented in Section 3. The source code is organized in a set of declarations in a ".h" (header) file and a set function definitions in a ".c" file.

The header file includes standard header files and the declaration of data structures, constants and functions.

⟨*EC header file*⟩≡
```
#include <lwp/lwp.h>
#include <sys/time.h>

typedef enum {FALSE, TRUE} bool;
#define ERROR   -1
typedef void TASK;

int ECtcreate(TASK (*func)(), int nargs, ...);
void ECtresched(void);
```

`TASK` is another name for the type `void`. It was introduced to increase the readability of the code. The functions which define the code of a task return the type `TASK`.

Every port has a queue where the messages are stored until the end of the communication between two processes. The queue is implemented as a static circular queue. Because the communication is synchronous, a queue with the size larger than the number of user-defined tasks is always large enough. Every item stored in the queue contains: the address of the data to be sent or the address where the data is to be received, the identifier of the thread which sends/receives the data and the type of the message `SEND` or `RECV`. The `PORT` data structure contains the queue storage, the size of data transmitted through the port and the identifier of the thread which implements the port.

⟨*EC header file*⟩+≡
```
typedef struct {
  thread_t tid;
  void *data; int type;
} MESSAGE;

#define NMSG 32

typedef struct {
  MESSAGE m[NMSG];
  int head, tail; int data_size;
  thread_t tid;
} PORT;

int ECpcreate ( PORT *p, int data_size);
void ECrecv ( PORT *p, void *data);
void ECsend ( PORT *p, void *data);
bool ECexist_data(PORT *p);
bool ECexist_request( PORT *p);
int ECinit(void);
```

The data structure and function declarations for timers:

⟨*EC header file*⟩+≡
```
typedef struct {
  long start; /* start time */
  long last;  /* time constant */
} TIMER;

void ECstart_timer(TIMER* t, long time);
bool ECtimeout(TIMER *t);
```

The EC code starts with several declarations used only in the implementation of the EC functions:

⟨*EC code*⟩≡
```
#include <stdio.h>
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#include <stdarg.h>

#include <ec.h>

#define EC_ERROR(x)   \
        printf("\tEC ERROR:%s\n", x);

/* PRIORITIES */
#define MAXPRIO  10       /* main */
#define SCHEDULER_PRIO 3  /* scheduler  */
#define PORT_PRIO 2       /* ports   */
#define TASK_PRIO 1       /* tasks      */

/* operation on queues */
#define isempty(pptr)  \
        pptr->tail==pptr->head
#define isfull(pptr) \
        ((pptr->head + 1 ) % NMSG == pptr->tail)
#define first(pptr) \
        (pptr->m[pptr->tail])
#define enqueue(pptr, m) \
        { pptr->m[pptr->head]=m; \
        pptr->head = (pptr->head+1)% NMSG; }
#define dequeue(pptr) \
        pptr->tail = ( pptr -> tail + 1 ) % NMSG

/* MESSAGE TYPES */
#define SEND     1        /* _send_  */
#define RECV     2        /* _recv_  */
```

The type of a message is `SEND` if the message comes from a `ECsend` function or `RECV` if the message comes from a `ECrecv` function.

The scheduler is an important part of the run-time environment of EC programs. It has two functions. First, it implements a preemptive round-robin schedule policy between the tasks. A preemptive scheduler is required because otherwise the loops used for the implementation of alternative composition would block the application. The second rôle of the scheduler is to update a global variable which measure the time. This variable is used by timers. Function `ECtresched` changes the current task. Another task, if there is one ready, becomes the running task. Function `ECinit` initializes the light weight process environment and starts the scheduler. This function must be called in all the programs which are using EC primitives. The function `ECtcreate` creates the thread for a task.

⟨*EC code*⟩+≡
```
/* the time slice for a task - usec */
#define TIME_SLICE 1000

static long current_time;

static int Scheduler( void )
{
  struct timeval quantum;

  current_time=0;
  quantum.tv_sec=0;
  quantum.tv_usec=TIME_SLICE;
  while ( TRUE ) {
    lwp_sleep(quantum);
    current_time += TIME_SLICE;
    lwp_resched(TASK_PRIO);
  }
}

void ECtresched(void)
{
  lwp_yield(THREADNULL);
}
#define MAXARGS 100

int ECtcreate(TASK (*func)(), int nargs, ...)
{
  va_list ap;
  int args[MAXARGS]; int n;

  va_start(ap, nargs);
  for ( n=0; n<nargs; n++) {
      args[n]=va_arg(ap, int);
  }
  va_end(ap);

  lwp_create((thread_t *)0, (void *)func,
        TASK_PRIO, 0, lwp_newstk(), nargs, args);
}

int ECinit()
{
  (void) pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, 20);
  lwp_create((thread_t *)0, Scheduler,
    SCHEDULER_PRIO,0, lwp_newstk(), 0);
}
```

The EC run-time environment contains a thread for every port. These threads execute the code defined in function `ECPortCode`.

⟨*EC code*⟩+≡
```
static int ECPortCode( PORT *p)
{
  MESSAGE *arg, *res, m, current;
  int alen, rlen;
  thread_t tid;

  while ( TRUE ) {  /* wait for data or requests */
    if ( MSG_RECVALL ( &tid, &arg,
        &alen, &res, &rlen, INFINITY) == -1 ) {
        EC_ERROR("PORT_THREAD receive");
        exit(1);
    }
    m.data=arg->data; m.type=arg->type;
    m.tid=tid;

   /* the queue is empty */
   if ( isempty(p) ) {
        enqueue(p, m);
        continue;
    }
    /* queue - messages of the same type */
    current=first(p);
    if ( current.type==m.type )  {
     if ( isfull ( p ) ) {
        EC_ERROR("the port queue is full ");
        exit(1);
      }
      enqueue(p, m);
      continue;
    }

    /* queue - complementary messages */
    dequeue ( p ) ;
    if ( m.type == SEND &&
                current.type == RECV ) {
      bcopy(m.data, current.data, p->data_size);
    }
    else if ( m.type == RECV &&
                current.type == SEND) {
      bcopy(current.data, m.data, p->data_size);
    }
    else {
        EC_ERROR("incorrect message types");
        exit(1);
    }
    /* resume the tasks      */
    msg_reply(m.tid);
    msg_reply(current.tid);
  }
}
```

The thread receives messages from the tasks and checks if there is a complementary message in the queue. If there is a such a message the thread transmits the data from one task to the other and resumes both by sending replies. If there are not complementary messages in the queue the new message is stored in the queue and the sending thread is blocked.

`ECpcreate` creates a thread for a new port. `ECsend` and `ECrecv` send a message to the thread which implements the port. `ECexist-data` and `ECexist-request` check if there are messages in the port queue.

⟨*EC code*⟩+≡

```
int ECpcreate(PORT *p, int data_size)
{
  p->head = p->tail = 0;
  p->data_size = data_size;
  lwp_create(&(p->tid), ECPortCode,
        PORT_PRIO, 0, lwp_newstk(),
        sizeof(PORT*), p);
}

static void ECcomm(PORT *p, void *data,
        int msg_type)
{
  MESSAGE cm;

  cm.data=data; cm.type=msg_type;
  if ( msg_send(p->tid, &cm,
        sizeof(MESSAGE), NULL, 0) == -1) {
        EC_ERROR("Send") ;
        exit(1);
  }
}
```

```
void ECsend(PORT *p, void *data)
{
        ECcomm(p, data, SEND);
}

void ECrecv(PORT *p, void *data)
{
        ECcomm(p, data, RECV);
}
```

```
static bool ECexist_message(PORT *p,
        int m_type)
{
  if ( isempty(p)  )
    return FALSE;
  if ( first(p).type == m_type)
    return TRUE;
  return FALSE;
}
```

```
bool ECexist_data(PORT *p)
{
        return ECexist_message(p, SEND);
}

bool ECexist_request(PORT *p)
{
        return ECexist_message(p, RECV);
}
```

Two functions operates on timers: `ECstart_timer` and `ECtimeout`. To start the timer means to store the current time and the time constant in the timer data structure. To check the timeout means to compare the current time with start time. It returns `TRUE` if the time has expired and `FALSE` otherwise.

⟨*EC code*⟩+≡

```
void ECstart_timer(TIMER* t, long timeconst)
{
  t->start=current_time;
  t->last=timeconst;
}
```

```
bool ECtimeout(TIMER *t)
{
  return (current_time - t->start) > t->last;
}
```

B The implementation of the Positive Acknowledgement with Retransmission Protocol

Based on the examples presented in the Section 4 we have implemented several communication protocols. We present in this section the implementation of the Positive Acknowledgement with Retransmission (PAR) Protocol. For the PSF specification we use a modified version of the specification given in [MV93]. The modifications were necessary because the original version does not satisfy all the requirements presented in Section 5.

The specification and the implementation are given in a "literate programming" layout [Knu84]. The source code is organized in "chunks" which assembled form the complete PSF specification and the EC program. The specification and the implementation are presented in parallel. On the left side is the PSF specification and on the right side is the EC code.

⟨*PSF specification*⟩≡
    ⟨*PSF data modules*⟩
    ⟨*PSF process modules*⟩

⟨*EC program*⟩≡
    ⟨*EC data structures*⟩
    ⟨*EC tasks*⟩

B.1 Data structures

The PSF data modules define the data structures used in the protocol. The PSF functions are implemented as C functions with the same name.

⟨*PSF data modules*⟩≡
    ⟨*PSF data module Data*⟩
    ⟨*PSF data module Bits*⟩
    ⟨*PSF data module Frames*⟩
    ⟨*PSF data module Timer-Signals*⟩

⟨*EC data structures*⟩≡
```
#include <stdio.h>
#include <ec.h>
```

⟨*PSF data module Data*⟩≡
```
data module Data
begin
  exports begin
    sorts
        DATA
    functions
        du : -> DATA -- data unit
    end
end Data
```

⟨*EC data structures*⟩+≡
```
typedef char DATA;
```

⟨*PSF data module Timer-Signals*⟩≡
```
data module Timer-Signals
begin
  exports begin
    sorts
        TIMER-SIGNAL
    functions
        st      :     -> TIMER-SIGNAL
        ti      :     -> TIMER-SIGNAL
        eto     :     -> TIMER-SIGNAL
    end
end Timer-Signals
```

⟨*EC data structures*⟩+≡
```
typedef enum { TIMER_ST, TIMER_ETO, TIMER_TI
        } TIMER_SIGNAL;
```

The specification for `Bits` contains functions to check the equality of bits and to compute the inverse of a bit. From the implementation for `Bits` we used the `int` data type.

⟨*PSF data module Bits*⟩≡
```
data module Bits
begin
  exports  begin
    sorts
        BIT
    functions
        0 :             -> BIT
        1 :             -> BIT
        invert : BIT    -> BIT
        eq: BIT # BIT   -> BOOLEAN
    end
  imports
    Booleans,  Data
  variables
    b: -> BIT  d: -> DATA
  equations
    [bit01] invert(0) = 1
    [bit02] invert(1) = 0
    [bit03] eq ( 0 , 0) = true
    [bit04] eq ( 1, 1) = true
    [bit05] eq ( 0, 1 ) = false
    [bit06] eq ( 1, 0 ) = false
end Bits
```

⟨*EC data structures*⟩+≡
```
typedef int BIT;

BIT invert(BIT b)
{
    return b ? 0 : 1;
}
```

A *frame* contains the user data and a bit used by the protocol. The functions `datum` and `bit` retrieve the components of frames. If the channel damages the information, the frame consists only in an error information. The acknowledgements are messages sent to confirm that the data was received.

⟨*PSF data module Frames*⟩≡

```
data module Frames
begin
  exports begin
    sorts
        FRAME,  ACK
    functions
        frame    : DATA # BIT -> FRAME
        ce       :            -> FRAME
        error    : FRAME      -> BOOLEAN
        ae       :            -> ACK
        ac       :            -> ACK
        error    : ACK        -> BOOLEAN
        bit      : FRAME      -> BIT
        datum    : FRAME      -> DATA
    end
  imports
    Booleans, Bits, Data
  variables
    b: -> BIT
    d: -> DATA
  equations
    [frame01] error(frame(d,b)) = false
    [frame02] error(ce) = true
    [frame03] error ( ac ) = false
    [frame04] error ( ae ) = true
    [frame05] bit ( frame (d,b)) = b
    [frame06] datum ( frame (d,b)) = d
end Frames
```

⟨*EC data structures*⟩+≡

```
typedef struct {
    int error;
    BIT b;
    DATA d;
} FRAME;

DATA datum(FRAME f)
{
    return f.d;
}

BIT bit(FRAME f)
{
    return f.b;
}

int frame_error(FRAME f)
{
    return f.error;
}

typedef enum { AC, AE } ACK;

int ack_error(ACK a)
{
    return a == AE;
}
```

B.2 Parallel composition and communications

The specification is organized in several process modules.

⟨*PSF process modules*⟩≡
⟨*PSF process module PAR*⟩
⟨*PSF process module AB*⟩
⟨*PSF process module Sender*⟩
⟨*PSF process module Channel-K*⟩
⟨*PSF process module Channel-L*⟩
⟨*PSF process module Receiver*⟩
⟨*PSF process module Timer*⟩

⟨*EC tasks*⟩≡
⟨*EC function* `main`⟩
⟨*Code for tasks A and B*⟩
⟨*Code for task Sender*⟩
⟨*Code for task Channel-K*⟩
⟨*Code for task Channel-L*⟩
⟨*Code for task Receiver*⟩
⟨*Code for task Timer*⟩

There is only one process definition which contains parallel composition. It defines the process `PAR` as the parallel composition of the processes `A`, `B`, `S` (Sender), `R` (Receiver), `K` (Channel-K), `L` (Channel-L) and the `T` (Timer). The processes `A` and `B` communicate with each other using the *PAR* protocol.

⟨*PSF process module PAR*⟩≡

```
process module PAR
begin
  exports begin
    atoms
      c1, c2      : DATA
      c3, c4      : FRAME
      c5, c6      : ACK
      c7, c8, c9  : TIMER-SIGNAL
    processes
        PAR
  end
  imports
    Data, Frames, Timer-Signals,
    Sender, Channel-K, Receiver,
    Channel-L, Timer, AB
```

⟨*EC function* `main`⟩≡

```
PORT port1, port2, port3,
        port4, port5, port6,
        port7, port8, port9;

TASK A();
TASK B();
TASK Sender();
TASK Receiver();
TASK Channel_K();
TASK Channel_L();
TASK Timer();
TASK Par();
```

```
  sets of atoms
   H = { r1(d), s1(d), r2(d), s2(d)|
               d in DATA } +
       { r3(f), s3(f), r4(f), s4(f)|
               f in FRAME } +
       { r5(a), s5(a), r6(a), s6(a)|
               a in ACK } +
       { r7(t), s7(t), r8(t), s8(t),
               r9(t) , s9(t)|
               t in TIMER-SIGNAL }
 communications
   s1(d)|r1(d)=c1(d) for d in DATA
   s2(d)|r2(d)=c2(d) for d in DATA
   s3(f)|r3(f)=c3(f) for f in FRAME
   s4(f)|r4(f)=c4(f) for f in FRAME
   s5(a)|r5(a)=c5(a) for a in ACK
   s6(a)|r6(a)=c6(a) for a in ACK
   s7(t)|r7(t)=c7(t)
               for t in TIMER-SIGNAL
   s8(t)|r8(t)=c8(t)
               for t in TIMER-SIGNAL
   s9(t)|r9(t)=c9(t)
               for t in TIMER-SIGNAL
 definitions
   PAR = encaps( H, A||S||K||R||L||T||B )
end PAR
```

```
int main()
{
    ECinit();

    ECpcreate(&port1, sizeof(DATA));
    ECpcreate(&port2, sizeof(DATA));
    ECpcreate(&port3, sizeof(FRAME));
    ECpcreate(&port4, sizeof(FRAME));
    ECpcreate(&port5, sizeof(ACK));
    ECpcreate(&port6, sizeof(ACK));
    ECpcreate(&port7, sizeof(TIMER_SIGNAL));
    ECpcreate(&port8, sizeof(TIMER_SIGNAL));
    ECpcreate(&port9, sizeof(TIMER_SIGNAL));

    ECtcreate(Par, 0);
    return 0;
}
TASK Par()
{
    ECtcreate(A, 0);
    ECtcreate(B, 0);
    ECtcreate(Sender, 0);
    ECtcreate(Channel_K, 0);
    ECtcreate(Channel_L, 0);
    ECtcreate(Receiver, 0);
    ECtcreate(Timer, 0);
}
```

B.3 Processes A and B

The two processes use the protocol to transmit data between them. The process A "produces" data, sends it to the Sender through port 1. Process B gets the data from the Receiver and "consumes" it.

⟨*PSF process module AB*⟩≡

```
process module AB
begin
  exports begin
    atoms
      s1, r2   : DATA
    processes
      A, B
  end

  imports
    Data
  definitions
    A = s1(du). A
    B = sum ( d in DATA, r2(d)). B
end AB
```

⟨*Code for tasks A and B*⟩≡

```
TASK A()
{
    DATA d; int i=0;
    while ( TRUE ) {
        d = 'A' + i; i = (i + 1) % 26;
        printf("produce data --- %c\n", d);
        ECsend(&port1, &d);
    }
}
TASK B()
{
    DATA d;
    while ( TRUE ) {
        ECrecv(&port2, &d);
        printf("consume data --- %c\n", d);
    }
}
```

B.4 Sender

The Sender receives data from process `A` through the port 1. Then it starts the timer by sending a message to the port 7 and sends a frame which contains the data to Channel-K through the port 3. After that, the Sender waits for an acknowledgement or time-out (communication through the ports 6 and 9).

⟨*PSF process module Sender*⟩≡

```
process module Sender
begin
  exports begin
    atoms
      r1     : DATA  s3     : FRAME
      r6     : ACK   s7, r9 : TIMER-SIGNAL
    processes
      S
  end
  imports
      Data, Frames, Timer-Signals
  processes
    RM : BIT
    ST, SF, RA : FRAME
  variables
    b :   -> BIT     f :  -> FRAME
    a :   -> ACK     t :  -> TIMER-SIGNAL
  definitions
    ⟨PSF Sender process definitions⟩
end Sender
```

⟨*Code for task Sender*⟩≡

```
TASK Sender()
{
    enum { S, RM, ST, SF, RA} state;
    DATA d;           FRAME f;
    BIT b;            TIMER_SIGNAL t;
    ACK a;

    state = S;
    while (TRUE) {
       ⟨EC Sender transitions⟩
    }
}
```

⟨*PSF Sender process definitions*⟩≡

```
    S = RM(0)


    RM(b) = sum(d in DATA, r1(d).
        ST(frame(d,b)) )



    ST(f) = s7(st).SF(f)



    SF(f) = s3(f).RA(f)


    RA(f) =
        sum ( t in TIMER-SIGNAL, r9(t)).ST(f)
        +
        sum ( a in ACK, r6(a).(
          [error(a)=true] -> ST(f)
          +
          [error(a)=false] -> RM(invert(bit(f))))
        )
```

⟨*EC Sender transitions*⟩≡

```
        if ( state == S ) {
            b=0; state = RM;
        }
        if (state == RM) {
            ECrecv(&port1, &d);
            f.d = d; f.b = b; f.error = FALSE;
            state = ST;
        }
        if (state == ST) {
            t = TIMER_ST;
            ECsend(&port7, &t); state = SF;
        }
        if (state == SF) {
            ECsend(&port3, &f); state = RA;
        }
        if (state == RA) {
            while (TRUE) {
                if (ECexist_data(&port9)) {
                    ECrecv(&port9, &t);
                    state = ST;
                    break; }
                if (ECexist_data(&port6)) {
                    ECrecv(&port6, &a);
                    if (ack_error(a)) {
                        state = ST; }
                    else {
                        b = invert(bit(f));
                        state = RM;
                    break;}
                ECtresched();}
            }
        }
```

B.5 Channel K and Channel L

The process Channel-K receives the frame from the Sender through port 3. The frame can be lost along the channel, it can be transmitted correctly or it can be damaged. This behaviour is specified using a non-deterministic choice. If the message does not get lost, it is sent to the Receiver through port 4.

⟨*PSF process module Channel-K*⟩≡

```
process module Channel-K
begin
  exports begin
    atoms
      r3, s4   : FRAME
    processes
        K
  end
  imports
    Data, Channel-L
  processes
    K1 : FRAME
    K2
  variables
    f : -> FRAME
  definitions
    ⟨PSF Channel K - process definitions⟩
end Channel-K
```

⟨*Code for task Channel-K*⟩≡

```
TASK Channel_K()
{
    enum {K, K1, K2} state;
    FRAME f;
    TIMER_SIGNAL t;

    state = K;
    while (TRUE) {
       ⟨EC Channel_K transitions⟩
    }
}
```

⟨*PSF Channel K - process definitions*⟩≡

```
K = sum( f in FRAME, r3(f).K1(f) )


K1(f) = skip.K2
        +
        skip.s4(ce).K2
        +
        skip.s4(f).K
```

⟨*EC Channel_K transitions*⟩≡

```
if (state == K) {
    ECrecv(&port3, &f); state = K1;
}
if (state == K1) {
    switch ( random () %3 ) {
    case 0: /* frame lost */
        state = K2;
        break;
    case 1: /* frame damaged */
        f.error = TRUE;
        ECsend(&port4, &f); state = K2;
        break;
    case 2:/* correct */
        ECsend(&port4, &f); state = K;
        break;
    }
}
```

In the specification, the channels "enable" the timers to send *time-out*. This solution was used in [MV93] because the PAR protocol is not reliable if the time-out comes too early. The EC implementation uses real timers and, because of that, we remove the communication between the channels and task for the timer.

⟨*PSF Channel K - process definitions*⟩+≡

```
K2 = s8(eto). K
```

⟨*EC Channel_K transitions*⟩+≡

```
if (state == K2) {
    /* ECsend(&port8, &t) */
    state = K;
}
```

The channel L has a similar specification as the channel-K but it transmits acknowledgement instead of data. The communication is through port 5 with the Receiver and, through port 6, with the Sender.

⟨*PSF process module Channel-L*⟩≡

```
process module Channel-L
begin
  exports begin
    atoms
      r5, s6: ACK     s8: TIMER-SIGNAL
    processes
        L
  end
  imports
    Data, Frames, Timer-Signals
  processes
    L1, L2
  definitions
    ⟨PSF Channel-L process definitions⟩
end Channel-L
```

⟨*Code for task Channel-L*⟩≡

```
TASK Channel_L()
{
    enum { L, L1, L2 } state;
    TIMER_SIGNAL t;
    ACK a;

    state = L;
    while (TRUE) {
      ⟨EC Channel_L transitions⟩
    }
}
```

⟨*PSF Channel-L process definitions*⟩≡

```
L = r5(ac).L1
```

⟨*EC Channel_L transitions*⟩≡

```
if (state == L) {
    ECrecv(&port5, &a); state = L1;
}
```

```
L1 = skip.s6(ae).L
     +
     skip.s6(ac).L
     +
     skip.L2
```

```
if (state == L1) {
    switch (  random() % 3  ) {
    case 0: /* ack error */ a = AE;
        ECsend(&port6, &a); state = L;
        break;
    case 1:  /* ack correct */  a = AC;
        ECsend(&port6, &a); state = L;
        break;
    case 2: /* ack lost */ state = L2;
        break;
    }
}
```

```
L2 = s8(eto).L
```

```
if (state == L2) {
    /* ECsend(&port8, &t) */
    state = L;
}
```

B.6 Receiver

The Receiver communicates with the Channel-K through the port 4. If the message was damaged it is ignored. If the message was not damaged the Receiver sends an acknowledgement to Channel-L through port 5. If the frame contains correct data it is sent to the process B through port 2.

⟨PSF process module Receiver⟩≡

```
process
module Receiver begin
  exports begin
    atoms
      r4 : FRAME s2 : DATA s5 : ACK
    processes
        R
  end
  imports
    Data, Frames
  processes
    RF : BIT  SM : FRAME SA : BIT
  variables
    b : -> BIT  f : -> FRAME
  definitions
    ⟨PSF Receiver process definitions⟩
end Receiver
```

⟨Code for task Receiver⟩≡

```
TASK Receiver()
{
    enum { R, RF, SM, SA } state;
    FRAME f; DATA d; ACK a; BIT b;

    state = R;
    while (TRUE) {
      ⟨EC Receiver transitions⟩
    }
}
```

⟨PSF Receiver process definitions⟩≡

```
R = RF(0)


RF(b) = sum ( f in FRAME, r4(f). (
  [error(f)=false] -> (
    [bit(f)=b]->SM(f)
    +
    [invert(bit(f))=b]->SA(b)
    )
  +
    [error(f)=true]->RF(b)
   ))

SA(b) = s5(ac).RF(b)



SM(f) = s2(datum(f)). SA(invert(bit(f)))
```

⟨EC Receiver transitions⟩≡

```
if ( state == R ) {
        b=0; state = RF;
}
if (state == RF) {
    ECrecv(&port4, &f);
    if (!frame_error(f)) {
        if (bit(f) == b) {
            state = SM;
        }
        else  state = SA;
     }
     else state = RF;
}
if (state == SA) {
    a = AC; ECsend(&port5, &a);
    state = RF;
}
if (state == SM) {
    d = datum(f); ECsend(&port2, &d);
    b = invert(bit(f)); state = SA;
}
```

B.7 Timer

The Timer process receives a `start-timer` message from the Sender through port 7. Then, it waits for `enable-time-out` from the channels or a new `start-timer` from the Sender. In the former case it sends `time-out` to the Sender through port 9.

⟨*PSF process module Timer*⟩≡

```
process module Timer
begin
  exports begin
    atoms
      r7, r8, s9 : TIMER-SIGNAL
    processes
        T
  end
  imports
    Data, Timer-Signals
  processes
    TR, TE
  variables
    t :        -> TIMER-SIGNAL
  definitions
    ⟨PSF Timer process definitions⟩
end Timer
```

⟨*Code for task Timer*⟩≡

```
#define TIMEOUT_USEC 5000

TIMER timer;

TASK Timer()
{
    enum { T, TR, TE} state;
    TIMER_SIGNAL t;

    state = T;
    while (TRUE) {
      ⟨EC Timer transitions⟩
    }
}
```

⟨*PSF Timer process definitions*⟩≡

```
  T = sum ( t in TIMER-SIGNAL,  r7(t).TR)
```

⟨*EC Timer transitions*⟩≡

```
        if (state == T) { ECrecv(&port7, &t);
            ECstart_timer(&timer, TIMEOUT_USEC);
            state = TR;
        }
```

The implementation of the process TR is an example for the implementation of timers. The communication between the channel and the timer was removed (see also the implementation of the channels) and a call to the function `ECtimeout` checks if the time since the start of the timer has expired.

```
TR =
   sum ( t in TIMER-SIGNAL, r7(t).TR)
   +
   sum ( t in TIMER-SIGNAL, r8(t).TE)
```

```
if (state == TR) {
   while (TRUE) {
       if (ECexist_data(&port7)) {
            ECrecv(&port7, &t);
            ECstart_timer(&timer,
                    TIMEOUT_USEC);
            state = TR;
            break;
       }
       if (
            ECtimeout(&timer)
            ) {
            /* ECrecv(&port8, &t); */
           state = TE;
           break;
       }
       ECtresched();
   }
}
```

```
TE =  s9(ti).T
```

```
if (state == TE) { t = TIMER_TI;
    ECsend(&port9, &t);
    state = T;
}
```

The result of the execution of the EC program is a a sequence of messages produced by the task A which prints a message before producing data and by the task B which display a message after consuming the data.

C A formal specification of the implementation of PSF process definitions

This section presents a formal specification of the implementation of PSF process definitions. The formalism used for the specification is ASF+SDF . An ASF+SDF specification consist of a sequence of named modules. Each module may contain:

Imports of other modules.

Sorts declarations defining the sorts of a signature.

Lexical syntax defining the lexical tokens.

Context-free syntax defining the concrete syntax of the functions in the signature.

Variables to be used in equations.

Equations define the meaning of the function defined in the context-free syntax.

The ASF+SDF specification contains the definition of syntax and the equations. It is presented in a "literate programming" layout [Knu84]. Informal description and "chunks" of specification are mixed. The whole ASF+SDF specification can be obtained by "assembling" the chunks.

⟨*PSF2EC*⟩≡
⟨*PSF2EC syntax*⟩
⟨*PSF2EC equations*⟩

⟨*PSF2EC syntax*⟩≡
```
imports Layout Identifiers  Integers
exports
   variables
     Id[0-9']*        -> ID
     Int[0-9']*       -> INT
```
⟨*Data*⟩
⟨*EC syntax*⟩
⟨*Process expressions*⟩
⟨*Atomic actions*⟩
⟨*Simple processes*⟩
⟨*Parallel composition*⟩
⟨*Sequential composition*⟩
⟨*Alternative composition - 1*⟩
⟨*Alternative composition - 2*⟩
⟨*Alternative composition - 3*⟩
⟨*Process definitions*⟩

⟨*PSF2EC equations*⟩≡
```
equations
```
⟨*Equations*⟩

The syntax of data is the same in PSF definitions as in EC programs. This specification of data is very simple. It allows only prefix functions and variables.

⟨*Data*⟩≡
```
sorts DATA
context-free syntax
   ID                              -> DATA
   INT                             -> DATA
   ID "(" {DATA ","}+ ")"          -> DATA
variables
   Data[0-9']*                     -> DATA
```

For the syntax of EC programs we use a simplified form of the syntax of a C-like imperative language.

⟨*EC syntax*⟩≡
```
sorts   EC-CODE EC-DNAME EC-DEF
```

Task definitions:

⟨*EC syntax*⟩+≡
```
context-free syntax
   "TASK" ID "(" {DATA","}* ")"  EC-CODE -> EC-CODE
```

EC statements:

⟨*EC syntax*⟩+≡
```
   while "(" DATA ")"  EC-CODE       -> EC-CODE
   switch "(" DATA ")"  EC-CODE      -> EC-CODE
   case DATA ":" EC-CODE             -> EC-CODE
   break ";"                         -> EC-CODE
   if "(" DATA ")"   EC-CODE         -> EC-CODE
   "{"  EC-CODE*  "}"                -> EC-CODE
   DATA ";"                          -> EC-CODE
```

The code of the EC program is organized in EC definitions. The EC definition has a name and a body. The body is a piece of EC code which can contain references to other EC definitions. The EC program is obtained from the EC definitions by unfolding the references. This approach, called "literate programming" is presented in [Knu84].

⟨*EC syntax*⟩+≡
```
        "<-<" ID* ">->"                 -> EC-DNAME
        EC-DNAME "=" EC-CODE            -> EC-DEF
        EC-DNAME                        -> EC-CODE
        "[" EC-DEF* "]"                 -> EC-DEF
    variables
        EcCodes[0-9']*                  -> EC-CODE*
        EcDefs[0-9']*                   -> EC-DEF*
```

⟨*Equations*⟩≡
```
      [ec-1] {EcCodes{EcCodes'}EcCodes''} = {EcCodes EcCodes' EcCodes''}
      [ec-2] [EcDefs [EcDefs'] EcDefs''] = [EcDefs EcDefs' EcDefs'']
```

The declaration of the sorts for process expressions, bracketed process expressions and the function `map` which transforms a process expression into EC code is:

⟨*Process expressions*⟩≡
```
    sorts PROC-EXPR
    context-free syntax
      "(" PROC-EXPR ")"                 -> PROC-EXPR
      map "(" PROC-EXPR ")"             -> EC-CODE
    variables
      ProcExpr[0-9']*                   -> PROC-EXPR
```

⟨*Equations*⟩+≡
```
     [map-01] map ( ( ProcExpr ) ) = map ( ProcExpr )
```

The specification of atomic actions uses several conventions often used in PSF specifications. For example, a typical *send* atomic action is `s4(d)` where `s` indicates a *send*, 4 is the number of the port and `d` is the sent data. A similar convention holds for the *recv* atomic action. In this case a typical example is `r4(d)`. *Recv* atoms occur only in generalized alternative composition.

⟨*Atomic actions*⟩≡
```
    sorts ATOM
    context-free syntax
      s NAT "(" DATA ")"                                -> ATOM
      sum "(" ID in ID "," r NAT "(" DATA ")"  ")"      -> ATOM
   variables
      Atom[0-9']*                                       -> ATOM
```

We use an internal representation for the atomic actions:

⟨*Atomic actions*⟩+≡
```
    context-free syntax
      atom-send ( NAT , DATA )                          -> ATOM
      atom-recv ( NAT , ID )                            -> ATOM
```

⟨*Equations*⟩+≡
```
      [atom-01] s Nat ( Data ) = atom-send ( Nat, Data )
      [atom-03] sum ( Id in Id', r Nat ( Id ) ) = atom-recv ( Nat , Id )
```

The processes which have a name are called "simple processes":

⟨*Simple processes*⟩≡
```
    sorts SIMPLE-PROC
    context-free syntax
      ID                                        -> SIMPLE-PROC
      ID "(" DATA ")"                           -> SIMPLE-PROC
      SIMPLE-PROC                               -> PROC-EXPR
      name ( SIMPLE-PROC )                      -> ID
    variables
      SProc[0-9']*                              -> SIMPLE-PROC
```

⟨*Equations*⟩+≡
```
      [proc-1] name ( Id )          = Id
      [proc-2] name ( Id ( Data ) ) = Id
      [map-02] map ( Id )           = assign(state, Id );
      [msp-03] map ( Id ( Data ) )  = { attrib ( Id  , Data ); assign(state, Id ); }
```

The specification of the parallel composition is:

⟨*Parallel composition*⟩≡
```
    sorts PAR-EXPR
    context-free syntax
      SIMPLE-PROC "||" { SIMPLE-PROC "||"}+      -> PAR-EXPR
      encaps ( ID , PAR-EXPR )                   -> PAR-EXPR
      PAR-EXPR                                   -> PROC-EXPR
      map-par-tcreate (  { SIMPLE-PROC "||"}+ )  -> EC-CODE
      map-par-tdef (  { SIMPLE-PROC "||"}+)      -> EC-DEF
    variables
      ParExpr                                    -> PAR-EXPR
      SProcs                                     -> { SIMPLE-PROC "||"}+
```
⟨*Equations*⟩+≡
```
    [par-01] encaps ( Id ,  ParExpr )           = ParExpr
    [par-02] map-par-tcreate ( SProc )          = ECtcreate ( name(SProc));
    [par-03] map-par-tcreate ( SProc || SProcs ) =
                 { map-par-tcreate ( SProc ) map-par-tcreate ( SProcs)}
    [par-04] map-par-tdef ( SProc )             =  <-< Code for task name(SProc) >-> =
                 TASK name ( SProc )  ( ) {
                   while ( TRUE ) { assign ( state, name ( SProc));
                     <-< code for transitions name ( SProc) >->
                     }
                 }
    [par-05] map-par-tdef ( SProc || SProcs )     =
                 [ map-par-tdef(SProc) map-par-tdef(SProcs) ]
```

The syntax and the equations for translating the sequential composition:

⟨*Sequential composition*⟩≡
```
    sorts COMM-EXPR
    context-free syntax
      ATOM "." PROC-EXPR                                      -> COMM-EXPR
      sum "(" ID in ID "," r NAT "(" DATA ")" "." PROC-EXPR ")"  -> COMM-EXPR
      COMM-EXPR                                               -> PROC-EXPR
```
⟨*Equations*⟩+≡
```
    [co-01] sum ( Id in Id', r Nat ( Id ) . ProcExpr ) = atom-recv ( Nat , Id ) . ProcExpr
    [co-02] map (   atom-send ( Nat , Data ) . ProcExpr ) =
                    { ECsend ( refport ( Nat ) , refdata (Data) ) ; map ( ProcExpr) }
    [co-03] map ( atom-recv ( Nat , Id ) . ProcExpr ) =
                    { ECrecv ( refport( Nat ) , refvar(Id) ) ; map ( ProcExpr )}
```

The alternative composition when all the alternatives are prefixed by atomic actions:

⟨*Alternative composition - 1*⟩≡
```
    context-free syntax
      COMM-EXPR "+" { COMM-EXPR "+"}+                -> PROC-EXPR
      map-alt1 "(" { COMM-EXPR "+" }+ ")"            -> EC-CODE
      exist-func ( ATOM )                            -> DATA
    variables
      CommExpr                                       -> COMM-EXPR
      CommExprs                                      -> { COMM-EXPR "+"}+
```
⟨*Equations*⟩+≡
```
    [alt1-01] map ( CommExpr + CommExprs) =
                       while ( TRUE )  map-alt1 ( CommExpr + CommExprs)
    [alt1-02] map-alt1 ( Atom. ProcExpr ) =
                       if ( exist-func ( Atom ) ) {map ( Atom.ProcExpr ) break;}
    [alt1-03] map-alt1 ( CommExpr + CommExprs) =
                       { map-alt1 ( CommExpr) map-alt1 (CommExprs) }
    [alt1-04] exist-func ( atom-send ( Nat , Data )  ) =  ECexist-request ( refport( Nat ) )
    [alt1-05] exist-func ( atom-recv ( Nat , Id )  ) = ECexist-data ( refport( Nat ) )
```

The alternative composition when the operands are conditional expressions:

⟨*Alternative composition - 2*⟩≡
```
    sorts BCOND COND-EXPR
    context-free syntax
      DATA "=" BOOL                              -> BCOND
      "[" BCOND "]" "->" PROC-EXPR               -> COND-EXPR
      COND-EXPR "+" { COND-EXPR "+" }+           -> PROC-EXPR
      map-alt2 "(" { COND-EXPR "+" }+ ")"        -> EC-CODE
      map-bcond ( BCOND )                        -> DATA
    variables
      CDExpr                                     -> COND-EXPR
      CDExprs                                    -> { COND-EXPR "+" }+
      BCond[0-9']*                               -> BCOND
```

⟨*Equations*⟩+≡
```
    [alt2-01] map ( CDExpr  + CDExprs ) = { map-alt2 ( CDExpr) map-alt2 ( CDExprs) }
    [alt2-02] map-alt2 (   [ BCond ] -> ProcExpr   ) =
                         if ( map-bcond ( BCond ) ) map ( ProcExpr )
    [alt2-03] map-alt2 ( CDExpr + CDExprs ) =
                         { map-alt2 ( CDExpr ) map-alt2 ( CDExprs ) }
    [alt2-04] map-bcond  ( Data = true ) = Data
    [alt2-05] map-bcond ( Data = false ) = neg ( Data )
```

In the third kind of alternative composition the operands are non-deterministic alternatives:

⟨*Alternative composition - 3*⟩≡
```
    sorts NONDET-EXPR
    context-free syntax
      skip "." PROC-EXPR                                  -> NONDET-EXPR
      NONDET-EXPR "+" { NONDET-EXPR "+"}+                 -> PROC-EXPR
      map-alt3 "(" INT "," { NONDET-EXPR "+"}+ ")"        -> EC-CODE
      no "(" { NONDET-EXPR "+"}+ ")"                      -> INT
    variables
      NDExpr                                              -> NONDET-EXPR
      NDExprs                                             -> {NONDET-EXPR "+"}+
```

⟨*Equations*⟩+≡
```
    [alt3-01] map ( NDExpr + NDExprs ) = switch ( xrand ( no(NDExpr + NDExprs )))
                 map-alt3 ( 0, NDExpr + NDExprs  )
    [alt3-02] map-alt3 ( Int , skip. ProcExpr ) = case Int: {map(ProcExpr) break;}
    [alt3-03] map-alt3 ( Int, NDExpr + NDExprs ) =
                 { map-alt3 ( Int  , NDExpr) map-alt3 ( Int+1, NDExprs) }
    [no-01] no ( NDExpr ) = 1
    [no-02] no ( NDExpr + NDExprs ) = 1 + no ( NDExprs )
```

The translation of process definitions results in a collection of EC definitions:

⟨*Process definitions*⟩≡
```
    sorts PSF-DEF
    context-free syntax
      SIMPLE-PROC "=" PROC-EXPR                 -> PSF-DEF
      map ( PSF-DEF* )                          -> EC-DEF
      map-def ( PSF-DEF )                       -> EC-DEF
    variables
      PsfDefs                                   -> PSF-DEF*
      PsfDef                                    -> PSF-DEF
```

⟨*Equations*⟩+≡
```
    [map-1] map ( PsfDef PsfDefs ) = [ map-def ( PsfDef )  map ( PsfDefs )  ]
    [map-2] map ( )  = [ ]
    [def-01] map-def ( SProc = SProc' || SProcs) =
                 [   <-< Code for task name(SProc) >-> =
                         map-par-tcreate ( SProc' || SProcs )
                     map-par-tdef (SProc' || SProcs )
                 ]
    [def-03] map-def ( SProc = SProc' ) = <-< Code for state name(SProc) >-> =
                         if ( eq ( state, name(SProc))) map ( SProc' )
    [def-03] map-def ( SProc = CommExpr ) = <-< Code for state name(SProc) >-> =
                         if ( eq ( state, name(SProc))) map ( CommExpr )
    [def-03] map-def ( SProc = CommExpr + CommExprs ) = <-< Code for state name(SProc) >-> =
                         if ( eq ( state, name(SProc))) map ( CommExpr + CommExprs )
    [def-03] map-def ( SProc = NDExpr + NDExprs ) = <-< Code for state name(SProc) >-> =
                         if ( eq ( state, name(SProc))) map ( NDExpr + NDExprs )
    [def-03] map-def ( SProc = CDExpr + CDExprs ) = <-< Code for state name(SProc) >-> =
                         if ( eq ( state, name(SProc))) map ( CDExpr + CDExprs )
```

As an example we will consider the process definitions of the Positive Acknowledgement with Retransmission protocol. A complete specification of this protocol is give in Appendix B.

⟨*Input term*⟩≡

```
map(
 PAR     = encaps (H, A||S||K||R||L||T||B)
 S       = RM(0)
 RM(b)   = sum (d in DATA, r1(d) . ST(d) )
 ST(f)   =   s7(st) . SF(f)
 SF(f)   =   s3(f) . RA ( f )
 RA(f)   =   sum (t in TIMER-SIGNAL,
                     r9(t) . ST(f))
             +
             sum (a in ACK,  r6(a ).
               [error(a) = true]->ST(f)
               +
               [error(a) = false]->
                       RM(invert(bit(f))) )
 K       = sum (f in FRAME,  r3(f) . K1(f))
 K1(f)   = skip . K2
           +  skip  .  s4(ce) . K2
           +  skip .  s4(f) . K
 K2      = s8(eto) . K
 L       = sum ( a in ACK , r5(a)) . L1
 L1      = skip .  s6(ae) . L
           +  skip .  s6(ac) . L
           + skip . L2
 L2      =  s8(eto) . L
 R       = RF(0)
 RF(b)   = sum (f in FRAME, r4(f)  .
             [error(f) = false]-> (
               [eq(bit(f), b) = true]->SM(f)
               + [eq(bit(f), b) = false]->SA(b))
               + [error(f) = true]->RF(b))
 SA(b)   = s5(ac) . RF(b)
 SM(f)   = s2(datum(f)) .  SA(invert(bit(f)))
 T       = sum (t in TIMER-SIGNAL,  r7(t) . TR)
 TR      = sum (t in TIMER-SIGNAL,  r7(t) . TR)
           +
           sum (t in TIMER-SIGNAL,  r8(t) . TE)
 TE      = s9(ti) . T
 A       = s1(du) . A
 B       = sum (d in DATA , r2(d).B)
)
```

The result after reducing the term presented above is a collection of EC definitions. It contains the control structures of the implementation and suggests how to implement the data. We use the following conventions in the code: `refport(N)` is a pointer to the port N (`&portN`), `refvar(V)` is a pointer to the variable V (`&V`), `refdata(expr)` is a pointer to a memory location where the expression `expr` is evaluated, `attrib(StateName, Data)` assign the value `Data` to the attribute of state `StateName`, `assign`, `eq`, `neg` are functions for assignment, test of equality, and boolean negation, `xrand(n)` returns a random number in the range $0..n-1$.

⟨*Output term*⟩≡

```
<-<Code for task PAR>->=  {
    ECtcreate( A );
    ECtcreate( S );
    ECtcreate( K );
    ECtcreate( R );
    ECtcreate( L );
    ECtcreate( T );
    ECtcreate( B );
}
<-<Code for task A>->=
TASK A ( ) {
    while( TRUE ) {
        assign( state, A );
        <-<code for transitions A>->
    }
}
<-<Code for task S>->=
TASK S ( ) {
    while( TRUE ) {
        assign( state, S );
        <-<code for transitions S>->
    }
}
<-<Code for task K>->=
TASK K ( ) {
    while( TRUE ) {
        assign( state, K );
        <-<code for transitions K>->
    }
}
<-<Code for task R>->=
TASK R ( ) {
    while( TRUE ) {
        assign( state, R );
        <-<code for transitions R>->
    }
}
<-<Code for task L>->=
TASK L ( ) {
    while( TRUE ) {
        assign( state, L );
        <-<code for transitions L>->
    }
}
<-<Code for task T>->=
TASK T ( ) {
    while( TRUE ) {
        assign( state, T );
        <-<code for transitions T>->
    }
}
<-<Code for task B>->=
TASK B ( ) {
    while( TRUE ) {
        assign( state, B );
        <-<code for transitions B>->
    }
}
<-<Code for state S>->=
if( eq ( state, S )) {
    attrib( RM, 0 );
    assign( state, RM );
}
<-<Code for state RM>->=
if( eq ( state, RM )) {
    ECrecv( refport ( 1 ), refvar ( d ));
    attrib( ST, d );
    assign( state, ST );
}
<-<Code for state ST>->=
if( eq ( state, ST )) {
    ECsend( refport ( 7 ), refdata ( st ));
    attrib( SF, f );
    assign( state, SF );
}
<-<Code for state SF>->=
if( eq ( state, SF )) {
    ECsend( refport ( 3 ), refdata ( f ));
    attrib( RA, f );
    assign( state, RA );
}
```

```
<-<Code for state RA>->=
if( eq ( state, RA ))
while( TRUE ) {
    if( ECexist-data ( refport ( 9 ))) {
        ECrecv( refport ( 9 ), refvar ( t ));
        attrib( ST, f );
        assign( state, ST );
        break;
    }
    if( ECexist-data ( refport ( 6 ))) {
        ECrecv( refport ( 6 ), refvar ( a ));
        if( error ( a )) {
            attrib( ST, f );
            assign( state, ST );
        }
        if( neg ( error ( a ))) {
            attrib( RM, invert ( bit ( f )));
            assign( state, RM );
        }
        break;
    }
}
<-<Code for state K>->=
if( eq ( state, K )) {
    ECrecv( refport ( 3 ), refvar ( f ));
    attrib( K1, f );
    assign( state, K1 );
}
<-<Code for state K1>->=
if( eq ( state, K1 ))
switch( xrand ( 3 )) {
    case 0 :  {
        assign( state, K2 );
        break;
    }
    case 1 :  {
        ECsend( refport ( 4 ), refdata ( ce ));
        assign( state, K2 );
        break;
    }
    case 2 :  {
        ECsend( refport ( 4 ), refdata ( f ));
        assign( state, K );
        break;
    }
}
<-<Code for state K2>->=
if( eq ( state, K2 )) {
    ECsend( refport ( 8 ), refdata ( eto ));
    assign( state, K );
}
<-<Code for state L>->=
if( eq ( state, L )) {
    ECrecv( refport ( 5 ), refvar ( a ));
    assign( state, L1 );
}
<-<Code for state L1>->=
if( eq ( state, L1 ))
switch( xrand ( 3 )) {
    case 0 :  {
        ECsend( refport ( 6 ), refdata ( ae ));
        assign( state, L );
        break;
    }
    case 1 :  {
        ECsend( refport ( 6 ), refdata ( ac ));
        assign( state, L );
        break;
    }
    case 2 :  {
        assign( state, L2 );
        break;
    }
}
<-<Code for state L2>->=
if( eq ( state, L2 )) {
    ECsend( refport ( 8 ), refdata ( eto ));
    assign( state, L );
}
<-<Code for state R>->=
if( eq ( state, R )) {
    attrib( RF, 0 );
    assign( state, RF );
}
<-<Code for state RF>->=
if( eq ( state, RF )) {
    ECrecv( refport ( 4 ), refvar ( f ));
    if( neg ( error ( f ))) {
        if( eq ( bit ( f ), b )) {
            attrib( SM, f );
            assign( state, SM );
        }
        if( neg ( eq ( bit ( f ), b ))) {
            attrib( SA, b );
            assign( state, SA );
        }
    }
    if( error ( f )) {
        attrib( RF, b );
        assign( state, RF );
    }
}
<-<Code for state SA>->=
if( eq ( state, SA )) {
    ECsend( refport ( 5 ), refdata ( ac ));
    attrib( RF, b );
    assign( state, RF );
}
<-<Code for state SM>->=
if( eq ( state, SM )) {
    ECsend( refport ( 2 ), refdata ( datum ( f )));
    attrib( SA, invert ( bit ( f )));
    assign( state, SA );
}
<-<Code for state T>->=
if( eq ( state, T )) {
    ECrecv( refport ( 7 ), refvar ( t ));
    assign( state, TR );
}
<-<Code for state TR>->=
if( eq ( state, TR ))
while( TRUE ) {
    if( ECexist-data ( refport ( 7 ))) {
        ECrecv( refport ( 7 ), refvar ( t ));
        assign( state, TR );
        break;
    }
    if( ECexist-data ( refport ( 8 ))) {
        ECrecv( refport ( 8 ), refvar ( t ));
        assign( state, TE );
        break;
    }
}
<-<Code for state TE>->=
if( eq ( state, TE )) {
    ECsend( refport ( 9 ), refdata ( ti ));
    assign( state, T );
}
<-<Code for state A>->=
if( eq ( state, A )) {
    ECsend( refport ( 1 ), refdata ( du ));
    assign( state, A );
}
<-<Code for state B>->=
if( eq ( state, B )) {
    ECrecv( refport ( 2 ), refvar ( d ));
    assign( state, B );
}
```