Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Formal Semantics for Interacting Abstract State Machines

by

Marcelo de Almeida Maia
Roberto da Silva Bigonha

RT 005/98

Caixa Postal, 702

30.161 - Belo Horizonte - MG

September, 1998

Abstract

In this work we propose an extension to the original model of Abstract State Machines. We focus on the modularization support and on the explicit interaction abstraction between the modules (units of specification). We provide the new language syntax and formal semantics, and also some examples showing its use.

Chapter 1

Introduction

Much of the work being done in the software engineering area concerns the development of mechanisms that facilitate the reuse and flexibility of software components. The most powerful resource to achieve these goals is modularity, which is based upon abstraction and information hiding and it is the only effective way to break down the complexity of large systems. Even though Abstract State Machines[6] support abstraction and information hiding, we advocate more powerful abstraction mechanisms. If we consider the inherent methodology of producing ASMs specifications as a methodology that provides a vertical abstraction mechanism, in the sense that the ground model is successively refined until considered adequate, it is reasonable to think that it does lack some kind of horizontal abstraction to support the reuse of existent specifications. An argument to support this view can be found in [9], where are defined some desired characteristics for good modularization mechanisms such as modular composability, modular decomposability, modular understandability, modular continuity and modular protection. Considering these characteristics, a central theme that affects directly each one of them is the specification of how software modules interact with each other. So, our decision is in the direction of a formalism that explicitly enables the software engineer to write down how the interaction occurs between the modules. We adopt a message exchanging style because we believe it provides a natural abstraction of how objects interact in the real world. When we explicitly specify the interaction between modules, we are automatically inclined to think about the concurrency issues involved in the interaction process. In our view, modularization and concurrency concepts are interdependent and should not be addressed separately, and this has influenced our decision of putting them together in a unique framework.

In the context of ASMs, there is already some work in the direction of providing them with some kind of horizontal abstraction. Glavan and

Rosenzweig developed a theory of concurrency [5] that enables the encoding of some traditional calculus as the $\pi-$calculus [10] and the Chemical Abstract Machine [2]. However, we can not see an explicit message passing mechanism and it does not support encapsulation and information hiding mechanisms, issues which will be directly treated in this work. May [8] has developed a work with the same aims as ours, and although it provides some form of encapsulation and information hiding, the usual modularization concepts must be further added to the model. The explicit message passing encoding is not considered too.

Instead of putting on the user the burden of providing the complete specification of the message interchanging between different specifications, our approach provides special constructions to help the explicit specification of how different pieces of specifications interact with each other. This idea can be thought as a better development of the concept of external functions [6], because the approach provides some environment behavior formalization. It is not necessary to know how the environment behaves internally, but it is necessary to know how it interacts with the system being specified. So, when we specify a system, we must have in hands a minimal formalization of the observed environment behavior that affects the system, what is a little diferent from the raw concept of external functions.

Chapter 2

Abstract State Machines

Abstract state machines (ASMs) [6] are transition systems which states are first-order interpretations of functions symbols defined by a signature Υ over a non-empty set U called the *super-universe*. These states are also called static algebras. The transition relation is given by a finite set of transition rules describing the modifications of the interpretation of the function symbols from one state to another. This is the reason why ASMs were formerly called Evolving Algebras. Before introducing the transition rules, let us define the auxiliary notions: locations, updates, update set. A *location* l of a state S is a pair $(f, \bar{x})$, where f is a non-static function symbol, $\bar{x} \in U^n$ and n is the arity of f. An update α over the state S is a pair (l, t), where l is a location and t is a term in the sense of first-order logic. If $v \in U$ is the value for interpreting the term t on S, then firing $\alpha = ((f, \bar{x}), t)$ at state S transforms S into S' such that the result of interpreting $(f, \bar{x})$ is v and all other locations are not affected. An update set $Updates(R, S)$ is a set of updates over the state S, collected from the transition rule R. The update set is consistent if it does not contain any two updates α, α' such that $\alpha = (l, x)$ and $\alpha' = (l, y)$ and $x \neq y$. Otherwise, the update set is inconsistent. To fire an update set over a state S means to fire *simultaneously* all its updates and produce the corresponding state S'. Firing an inconsistent update set means to do nothing, i.e., means to produce a state $S' = S$.

The transition relation of ASMs is defined by the following transition rules:

- $f(t_1, ..., t_n)$ `:=` t
- $R_1 ... R_k$
- `if` e `then` R_1 `else` R_2 `endif`
- `extend` U with v R_1 `endimport`
- `choose` v `in` U `satisfying` e R `endchoose`

- `var` v `ranges over` U R_1 `endvar`

The first three kind of rules are called basic rules, respectively, the *update* instruction, the *block* construction and the *conditional* construction. Their semantics are given by means of an update set $Updates(R, S)$, i.e., to fire R over a state S fire $Updates(R, S)$. This update set is inductively defined on the structure of R:

1. if $R \equiv f(t_1, ..., t_n)$ `:=` t then $Updates(R, S) = \{(l, S(t))\}$, where $l = (f, (S(t_1), ..., S(t_n)))$, and $S(t)$ is the result of interpreting t on S;
2. if $R \equiv R_1 ... R_k$ then $Updates(R, S) = \cup_{i=1}^{k} Updates(R_i, S)$;
3. if $R \equiv$ `if` e `then` R_1 `else` R_2 `endif`, then $Updates(R, S)$ is defined as:
$$\begin{cases} Updates(R_1, S) & \text{if } S(e) \text{ holds} \\ Updates(R_2, S) & \text{otherwise.} \end{cases}$$

The last three kind of rules introduce variables, respectively, the *extend* construction which produces new fresh elements that are added to the extended universe, the *choose* construction which performs non-deterministic choices from a universe, and the *var* construction, which allows a simple form of synchronous parallelism. The variables of these rules must be bound to some value of a universe belonging to the super-universe. So, the definition of the update set is extended with an environment ρ which binds the variables to values, and a choice function ξ which determines the variable bindings for `extend` and `choose` rules. The function ξ maps the bound variables to elements of a special universe called $Reserve$, which is used to produce new elements. The update set $Updates(R, S, \rho, \xi)$ can be defined inductively as well.

Given an initial state S_0, a *run* is a sequence of states S_0, $S_{1,}$... such that the state S_{i+1} is obtained as the result of firing the transition relation at S_i.

Finally, let us define a *distributed ASM,* which contains several computational *agents,* which execute concurrently a number of single-agent programs (called modules). A distributed ASM consists of a finite indexed set of single-agent programs π_v (modules) and finitely many agents a such that, for some module name v, $Mod(a) = v$, where the function name Mod represents the relation between modules names and agents. Each module has a corresponding enumerating universe of agents, which can be extended or retracted as necessary. There is also a nullary function name *Self* that allows the self-identification of agents: *self* is interpreted as a by each agent a.

An agent a makes a *move* from a given state S if the corresponding update set of a is fired at S resulting a new state S'. Thus, a move can

represented as a pair (S, S'). Building upon this basic concept of move, a partially ordered notion of *run* for distributed ASMs can be defined as a triple (M, A, σ), where:

1. M is a partially ordered set where its elements are agent moves.
2. A is a function that, given a move from M, returns the agent performing that move. It is used to impose that the set $\{m : A(m) = a\}$ is linearly ordered.
3. σ is a function that give an initial segment of M (possibly empty) assigns to it the corresponding state S.

In order to make the specifications more readable we will define a concrete syntax which extend the standard notation of ASMs with the notion of types, functions, and pattern-matching as introduced in [4]. We call this specification language ASM-MG[1].

Now we will give an example written is ASM-MG while we show the language constructions. We will specify a very simple programming language which has only output and assignment statements. In ASM-MG we would specify the type of the language statement as:

```
freetype STMT == {
  Assign : STRING * TERM,
  Output : TERM
}
```

The terms of the simple language would be either a constant, or a variable, or an application. We would specify like the following:

```
freetype TERM == {
  Con : INT,
  Var : STRING,
  App : STRING * [TERM]
}
```

The `freetype` definition introduces a new type name into the specification and describes its structure. The above free types are the only one in the example.

An ASM-MG specification is a sequence of definitions. Like the `freetype` definition there are other kinds of definitions. The definitions the ASM-MG language supports are:

- `Freetype` definitions: enables the specifications of user-defined types. It is possible the definition of traditional structured types.

[1] The name MG is due to the joint work of three federal universities of the Minas Gerais state, namely UFMG, UFOP, and UFV

  - tuple types: `()` or `(type_1, ..., type_n)` or `type_1 * ... * type_n`
  - derived form for types:
    * list types: `[type]`
    * set types: {`type`}
    * map types: {`type_1 -> type_2`}

- Function definitions: enables the definition of static, dynamic and derived functions.
  - Static functions are used to bind any term to a function name. It can be parameterized and thus allowing the macro definition for terms.
  - Dynamic functions represent the evolving state of the specification.
  - Derived functions are introduced to allow the specification of computable functions in a purely functional style. They do not affect the ASM philosophy in the sense that they are computed in only one ASM-step.
- Transition definitions: provide a kind of ASM rules abstraction. In every specification there must be a transition named `"main"` which defines the begining of the update set calculation.
- Module definitions: enables the definition of distributed abstract state machines.

Back to our example, we will define the following function:

```
dynamic function env :STRING -> INT
  initially finite map
    { v -> ord (v) - ord ("x") | v in { "x", "y", "z" } }
```

This function `env` , representing the *environment* on which the programs of our simple language will be executed, has a special derived form of type which is called `MAP`.

Besides the derived form `MAP`, there are also derived forms representing set and list types represented by `"{ type }"` and `"[ type ]"`, respectively.

Below we define a function representing the *output* of the programs of our simple language. Note the type of the function *out* is a list of integers.

```
dynamic function out :[INT]
  initially []
```

Next we define a static function defined with two parameters: a string representing a function of the simple language and a list of parameters.

This function introduces the notion of pattern-matching. This notion is implemented by the `case` construction, which gets a pattern and tries to match it sequentially against several other patterns. When a successful matching is found then the corresponding term is returned.

```
static function eval_app (f, args) ==
  case args of
    [ x ] : case f of
      "abs" : abs (x)
    endcase ;
    [ x, y ] : case f of
      "+"   : x + y ;
      "-"   : x - y ;
      "*"   : x * y ;
      "div" : x div y ;
      "mod" : x mod y
    endcase
  endcase
```

Now we define a derived function which is defined in a purely functional style. The function *eval_term* has a parameter t which is matched against a *constructor application* pattern. Note that the constructors are the same as defined in the free type *Term*. Also, note that the function may be recursively called.

```
derived function eval_term (t) ==
  case t of
    Con (x)         : x ;
    App (f, t_list) : eval_app (f, [ eval_term (t) | t in t_list ]) ;
    Var (v)         : env (v)
  end
```

Finally, we have two functions definitions working as a macro for calling the terms constructors.

```
static function Add (x, y) == App ("+", [ x, y ])
static function Mul (x, y) == App ("*", [ x, y ])
```

Now we will define the transition rules that works as an interpreter for our simple language.

The transition which executes an assignment has a parameter which is a variable and a term to be assigned. Note that this transition is executed in only one step, even though the term evaluation is as large as wanted.

The transition which executes an output append the evaluated term to the dynamic function *out*.

```
transition ExecuteAssign (v, t) ==
  env (v) := eval_term (t)
transition ExecuteOutput (t) ==
  out := append (out, [ eval_term (t) ])
```

We define a transition for executing a statement. Note that this transition just matches the parameter *stmt* against the corresponding statement constructor and executes the proper transition. Also note that the previous transitions definitions are working just as macros because the execution of this transition requires just one ASM-step. Indeed, the following transition also works as a macro because it would be expanded into the main transition *Interpreter*.

```
transition ExecuteStmt (stmt) ==
  case stmt of
    Assign (v, t) : ExecuteAssign (v, t) ;
    Output (t)    : ExecuteOutput (t)
  endcase
```

Finally, we define the main transition *Interpreter*[2] which takes the simple language program as a dynamic function and iteratively interprets it. At the end of the simulation the *environment* and the *output* will have the expected values.

```
transition Interpreter ==
  case prog of
    stmt :: rest_of_prog :
      ExecuteStmt (stmt)
      prog := rest_of_prog
  endcase
dynamic function prog
  initially [
    Assign ("x", Con (10)),
    Output (Var ("x")),
    Assign ("y", Con (12)),
    Output (Mul (Var ("x"), Add (Var ("y"), Con (3))))
  ]
```

For the sake of completeness, let us provide additional syntax for terms, patterns and modules.

There are the following possibilities for writting terms:

- Special constants: integers, floats, and strings.
- Variables: an applied occurrence of a variable represented by its identifier.

[2]The definition of which transition is the main one is left to the environment on which the ASM simulator runs

- Ordinary terms: provided by function application. The application may be written in prefixed or infixed style depending on the function definition.
- Tuple terms: are written using the tuple constructor written just like the tuple type definition.
  Ex: `("red", "green", "blue")`
- Conditional terms: are written using an `if-then-else` syntax, regarding that the alternative clauses are terms.
  Ex: `if x > 0 then "black" else "white" endif`
- Let terms: are written using a `let` *patt* `==` *term* ... `in` *term* `endlet` syntax.
  Ex: `let x == 0 in x+1`
- Case terms: are written using a `case` *patt* `of` *patt : term* ... `otherwise:` *term* `endcase` syntax.
  Ex: `case x of [] : 0; x::xs : length(xs) + 1`
- Set, list, and map comprehension terms.
  Ex: {`x - 1 | x in 0,1,2 with x > 0`}
  Ex: { `v -> ord (v) - ord ("x") | v in { "x", "y", "z" } }`
- List terms.
  Ex: `[1, 2, 3]` or [1 :: 2 :: 3 :: nil]
- Set enumeration terms.
- Map enumeration terms.

There are the following possibilities for writting patterns:

- Special constants: such as integers, floats and strings.
- Occurrences of variables.
- Placeholders: written as "`_`".
- Ordinary patterns: defined by constructor application. The constructor may be a nullary one, such as, `nil, true, ...`. It also may be a built-in or user-defined one, or even a infixed or prefixed one.
- Tuple patterns: defined by the constructor "`( ... )`".
- List patterns: defined by the constructors "`[ ... ]`" or " `... :: ... `".

Finally, modules may be written using the following structure:

```
module <module name> [enumerating universe]
   <transitions>
endmodule
```

Chapter 3

The Interactive ASM Language

A specification is defined as a set of unit definitions and unit instances. Units definitions are classified as *system units* and *environment units*. System units are those which will be completely specified, whereas environment units will be partially specified. We use the word *environment* not only referring to the external portion of the system, but also referring to some components of the system that had already been specified and are being reused.

The intention of specifying a system as a set of units is to encapsulate some portion of the state inside small pieces of specification. This leads to an isolation of the internal state of a unit. The information contained in the internal state of a unit only can be communicated to other units by explicitly specifying a pattern of interaction between the involved units. This interaction specification does not only specify the information flow but also the synchronization restrictions within the interaction.

In the sequel we present the abstract syntax of our proposed language.

A system unit definition U_s is composed of several parts and it is defined as:

U_s ::= `unit` *unit_name*
`function names` *function_names*
`interaction` *interaction*
`rules` *rules*

where:

- *function_names* is a subset of the vocabulary that contain the names of the functions. It represents, together with the respective interpretations of the names into the super-universe, a local state alterable only by the local unit rules and interaction. Each function name may be optionally initialized with an arbitrary value. In order to make the ideas clear, we will define an abstract data type (ADT) *Stack*, as we explain

the parts of a unit. For the *Stack* unit we may have the following function names:

```
function names
   max := 100        % Maximum length of Stack
   s                 % The stack itself
   top := 0          % Index of the top elem
   topelem           % The top element
   ack               % Acknowlegdement of pushing
   c                 % The client of the Stack
```

- An interaction i is defined as:

  i ::= *internal_pub_name* -> *u_name*
  | *buffered_var* <-- *u_name*.*pub_name*
  | *var* <- *u_name*.*pub_name*
  | `connect` $u : U.s$ | `connect` $u : U$ | `connect` u
  | `new` $u : U$
  | `destroy` $u : U$
  | $i_1 + i_2$ | $i_1 +? i_2$
  | i_1 `||` i_2 | i_1 `|;` i_2 | i_1 `|` i_2
  | i_1 ; i_2
  | $i : l$
  | `waiting`(*name*)
  | `if` *guard* `then` i
  | `extend` U `with` x i `endextend`
  | `choose` v `in` U `satisfying` e i `endchoose`
  | `var` v `ranges over` U i `endvar`

  The basic operators for interaction are those that provide input and output within a unit. They are the -> and <-, used to send a value to a unit and to receive a value from a unit into a variable, respectively. The operator <-- denotes a buffered input that avoids an inconsistent update if two or more different inputs to same variable occur in the same step. Since we expect to define dynamically the communication topology, we provide the `connect` operator which binds a unit name to some unit instance. The operators *new* and *destroy* are used to create and destroy unit instances. Since units are mapped into agents, these operators update the corresponding enumerating set of agents derived from a module. In order to address complex interaction patterns that may exist between units we provide the well-known composition operators "+, +?" (different kinds of non-deterministic choice), "||, |!, |" (different kinds of parallel composition), and ";" (sequential composition). As we will define soon, one cannot reason about the relative speed of execution of an atomic interaction compared to an internal rule of a unit. Thus in order to synchronize the interaction part with

the computation part of a unit we introduce labeled interactions and the barrier `waiting`*(name)*. The label l uniquely identifies an interaction, and denotes how many times the interaction labeled with l has completely occurred. Its initial state is *zero*. We also inherit from the ASM notation the `if, extend, choose,` and the `var` rules.
Coming back to our example, the ADT *Stack* may be seen as a server and thus it must connect with the *Client* before performing any information exchanging. As we will see, the `connect` operator used below waits until there is an interested unit instance requesting the connection. After the connection, it may receive requests from the *Client* instance. The requests guide the sequel of the interaction, and the unit *Stack* interacts with its *Client* by sending to it the element on the top of the stack (popping it or not) or receiving from it an element to be pushed onto the stack.

```
interaction
   connect c;
   request <- c;
   if request = "top" then
      topelem -> c
   elseif request = "pop" then
      waiting(popped);
      topelem -> c
   elseif top < max then
      elem <- c.elem;
      waiting(pushed);
      ack -> c
   endif
```

- *rules* is defined as an element of ASM_RULES. These rules work by changing the internal state represented by $function\_names$. In the abstract data type *Stack* we may define the rules as:

```
rules
  if waiting(popped) then
      top--;
      waiting(popped) := false;
   endif
   if waiting(pushed) then
      top++;
      s(top+1) := elem;
      waiting(pushed) := false;
   endif
```

Now, let us define an environment unit as a restriction on a system unit. It shows the public portion of a system unit that can be imported by other units and can be defined as:

u_e ::= `environment unit` *unit_name*
`interaction` *interaction*

Each system unit has a corresponding environment unit specifying what will be exported to other units.

Chapter 4

Semantics

In this section we specify the IASM language formal semantics. We provide it in a translational style that maps a syntactic domain corresponding to the IASM constructions into the original ASM language defined by Gurevich[6].

4.1 Unit Definition

Unit definitions are translated by the $\mathcal{D}$ compilation scheme, defined as in Figure 4.1. The idea of this compilation scheme is to put together, inside a module, the rules corresponding to each construction of each unit definition. The target specification of the module will be generated from the unit definition U. This compilation scheme guarantees that each unit instance derived from this unit definition will have its own clock, and thus its execution will be independent from the other instances. We will also make use of the function *Self* which allows an agent to identify itself between other agents. Since a unit definition will correspond to an ASM module, it introduces an enumerating universe of the agents corresponding to unit instances. The elements of this enumerating universe are the instances names, each of them composed of the unit instance declaration name labeled with the unit definition name,

$\mathcal{D}$, $\mathcal{D}_{\text{mod}}$: $IASM\_CONSTRUCTIONS \rightarrow ASM\_RULES$
$\mathcal{D}\ [\![\ U_1 \ldots U_n\ ]\!] = \bigcup_{i=1}^{n} \mathcal{D}_{\text{mod}}[\![U_i]\!]$
$\mathcal{D}_{\text{mod}}\ [\![U]\!] =$
  module U
    $\mathcal{I}\ [\![U.interaction\ ]\!]$
    $\mathcal{R}\ [\![\ U.rules\ ]\!]$
  end module

Figure 4.1: Translation scheme for unit definitions

$\mathcal{U}_{\text{static}}\ [\![u : U]\!] =$
  $\Upsilon(u(\_)) := \text{true}$
  `Mod(u) := U`
  `extend U with x name(x) := "U.u"`

Figure 4.2: Translation scheme for static unit instantiation

and additionally possibly labeled with the instance name from which it has been instantiated.

4.2 Internal State

Since an ASM specification has only a global state, the internal state of each unit instance has to be mapped into the global state.

This can be trivially done by adding an additional element to the tuple that identifies a location. This additional element is a term which value is *self*. So every internal location of a unit instance $f(x_1, ..., x_n)$ has to be replaced by the location $f(self, x_1, ..., x_n)$.

4.3 Unit Instantiation

The instantiation of a unit means extending the universe that enumerates the agents corresponding to the instances of a unit definition. Each element of the enumerating universe is identified by the corresponding unit instance name. There are two ways of declaring units:

1. Static declarations are those done in the startup definition. These declarations actually create the initial unit instances which will live during the whole execution of the specification. The unit instance name is defined by the declared instance name and labeled by the unit definition name. The label is added to the left of the instance declaration name separated by a dot. In Figure 4.2 we define the translation scheme $\mathcal{U}_{\text{static}}$ for static unit declarations (instantiations).

2. Dynamic declarations are those done inside a unit definition, and do not create a unit instance. Instead, it produces a function name that will be dynamically bound to a unit instance name, either a statically created, or a dynamically created one. A dynamic unit may be created and destroyed with the interaction instructions `new` and `destroy`, respectively. Because unit instances are agents, creating and destroying

$\mathcal{U}_{\text{dynamic}}\ [\![u : U]\!] = \quad \Upsilon(u(\_)) := \text{true}$

$\mathcal{I}\ [\![new\ u : U]\!]\ hole\ =$

```
Mod(u) := U;
extend U with x
    name(x) := Self ++ "U.u"
endextend;
```
hole

$\mathcal{I}\ [\![destroy\ u : U]\!]\ hole\ =$

```
Mod(u) := undef;
choose x in U satisfying name(x) = Self ++ "U.u"
    U(x) := false;
endchoose;
```
hole

Figure 4.3: Translation scheme for dynamic unit instantiation

units means to extend or retract the enumerating universe that contains the agent names of the specified module.

In Figure 4.3 we define the translation schemes for dynamic unit declaration and for the interaction instructions *new* and *destroy*. In the translation schemes for *new* and *destroy* we introduce an extra parameter *hole* which means that it is a point where the interaction instruction has been fully performed and it is the point where many useful context information is to be inserted. Such context information are, for instance, updates for transferring the control, updates for internal control of the non-determininistic instruction. We have put a surronding box in *hole* just for readability purposes.

4.4 Input/Output Interaction

There are two possible semantics for receiving a value from another unit. The name responsible to store the received values may be buffered or not. In either case, the special universe MSG is searched to find a message that matches the required input interaction. If this message exists then the corresponding updates are done and the used message is discarded from the universe MSG. For obvious reasons, for each input interaction there must be an output interaction.

In the case where received values are not buffered, the variable is translated into a function name and if there is a message that matches the input interaction then the corresponding updates take place. In Figure 4.4 we define the translation scheme for the single input.

In the other case, the buffered variable will be translated into a universe.

```
I [[var <- u_name.pub_name ]] hole =
  if has_message(‘u_name.pub_name) then
    choose x in MSG satisfying match_msg(x, ‘u_name.pub_name)
      var := cont(x);
      MSG(x) := false;
    endchoose;
  hole
  endif;
```

Figure 4.4: Translation scheme for single input

```
I [[buffered_var <-- u_name.pub_name ]] hole =
  if has_message(‘u_name.pub_name) then
    choose x in MSG satisfying match_msg(x, ‘u_name.pub_name)
      extend buffered_var with y
        cont(y) := cont(x);
      end extend;
      MSG(x) := false;
    endchoose;
  hole
  endif;
```

Figure 4.5: Translation scheme for buffered input

Each time the variable receives a value, the corresponding universe will be extended with that value.

In order to access the buffered values we will assume that a timestamp is assigned to each value received into a buffered variable. This timestamp is incremented with step one, and if there are many incoming values in the same step in the same variable, then the corresponding timestamps are assigned non-deterministically to each value. For example, suppose there is a buffered variable x that is receiving two values, for instance "v1" and "v2", in the same step. If the current timestamp to be assigned to the incoming value is, for example, 8, then the timestamps to be assigned non-deterministically to "v1" and "v2" will be 8 and 9, and the current timestamp will be set to 10. In Figure 4.5 we define the translation scheme for the buffered input.

The output interaction means that an internal value from the current unit is sent to another unit. This sending means that the internal universe MSG is extended with a new message. This universe contains the messages exchanged between the units. A message carries its target, a label denoting the source of this value, and a value. In Figure 4.6 we define the translation scheme for the output interaction.

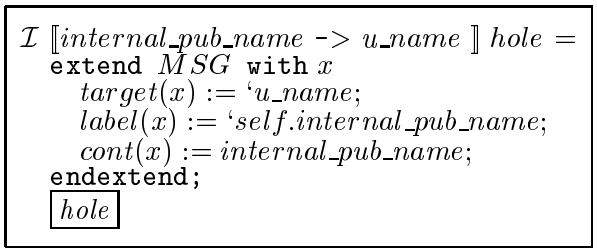

Figure 4.6: Translation scheme for output interaction

4.5 Unit Connections

As stated before, a unit declaration inside a unit definition only produces a function name. Our intention is that this function name should be further bound to another unit instance which also has a function name bound to the former unit instance. This situation indicates an agreement between the two instances, and it is performed with the operator `connect`. The arguments for this operator are: 1) a function name u corresponding to a unit instance, 2) the name U corresponding to the unit definiton from which the instance u was derived, and 3) a function name s declared inside U that we expect to be bound to the current unit instance.

There are some possibilities when using `connect`:

- All arguments are defined. Then it must be checked that if there is another instance that attempted a connection that matches this one. If there is such attempt, then the connection is successfully performed, otherwise it is blocked until such attempt occurs.

- Some arguments are undefined. This possibility is necessary because when establishing a connection we may not know in advance which unit instance will be connected or even from which unit definition the unit to be connected was derived. We may write `"connect u: U"`, where `"u"` is undefined and we are not interested on which function name inside `"U"` will receive the name of the current instance. Alternatively, we may want more flexibility and write `"connect u"`, where `"u"` is undefined. In this case, any unit wanting to connect through the channel `"u"` can match this connection, regarded that it has all arguments defined and matching the channel `"u"`.

The semantics of this operator may be given defining a universe *Connections* that the operator `connect` can update and search in order to establish the connection. Each element of this universe is a pair representing the two connected instances. Each element of the pair is a quadruple (u, U, o, a),

```
C [[connect u:U.f ]] =
  if exists x in Connections.
      wants(x,u,"U","f",self,Mod(self),"u") then
    choose x in Connections satisfying
        wants(x,u,"U","f",self,Mod(self),"u")
      Connections(x) := false;
    endchoose;
    extend Connections with x1
      party1(x1) := (self, Mod(self), "u", true);
      party2(x1) := (u, "U", "f", true);
    endextend;
    [hole]
  else
    extend Connections with x1
      party1(x1) := (self, Mod(self), "u", true);
      party2(x1) := (u, "U", "f", false);
    endextend;
    if exists x in Connections.
        connected(x,u,"U","f",self,Mod(self),"u") then
      choose x in Connections satisfying
          connected(x,u,"U","f",self,Mod(self),"u")
        Connections(x) := false;
      endchoose;
      [hole]
    endif
  endif
```

Figure 4.7: Translation scheme for `connect u:U.f`

where u is the name of the instance involved, U is its corresponding unit definition name, o is the function name whose value is the name of the other instance belonging to the pair, and a is the awareness that each party has of the connection.

Figure 4.7 shows the translation scheme for the fully defined connection. It also can be used for partially defined connection, where the names are provided, but their values are undefined. Figure 4.8 shows the translation scheme for the totally undefined connection.

4.6 Interaction Composition and Runs

Since an IASM specification can be translated into the pure ASM notation, as a set of modules and agents, the notion of run for interactive ASMs is the same as that of pure ASMs. But, compared with the pure ASMs, the composed interaction portion of the specification has a different state transition granularity. So, the reasoning mechanism of pure ASMs should not be used for Interactive ASMs, which have a more elaborated notion of move. Thus, in the sequel, we define a special notion of *interaction cycle* that is independent from the notion of move of the internal rules. The latter obeys

```
C [[connect u ]] hole =
  if exists x in Connections.
      wants(x,undef,undef,undef,self,Mod(self),"u") then
    choose x in Connections satisfying
        wants(x,undef,undef,undef,self,Mod(self),"u")
      Connections(x) := false;
      extend Connections with x1
        party1(x1) := (self, Mod(self), "u", true);
        party2(x1) := other_party(x,self, Mod(self),"u");
      endextend;
    endchoose;
    hole
  endif
```

Figure 4.8: Translation scheme for `connect u`

the partially-ordered semantics of distributed ASMs.

Definition 1 *Interaction tree is the abstract syntax tree derived from the interaction part of a unit definition.*

Definition 2 *Interaction cycle of a node of the interaction tree is the result of executing the moves of the rules corresponding to that node until the end of the cycle, when another cycle begins. The end of a cycle is defined by the* hole *point.*

Definition 3 *Sequence: If there is an enabled interaction tree with the following form:* $(i_1; i_2; \cdots; i_n)$ *then one and only one interaction* i_k $(1 \leq k \leq n)$ *is enabled at each time and all the sequence is executed in the cycle, regarded that each* i_k *finishes its cycle.*

Definition 4 *Cyclic parallelism: If there is an enabled interaction tree with the following form:* $(i_1 || \ i_2 || \ \cdots \ || \ i_n)$ *then all interactions* i_k $(1 \leq k \leq n)$ *are enabled and the execution of each* i_k *is cyclic and do not depend on each other.*

Definition 5 *1-Move-for-slower parallelism: If there is an enabled interaction tree with the following form:* $(i_1 | \ i_2 | \ \cdots \ | \ i_n)$ *then all interactions* i_k $(1 \leq k \leq n)$ *are enabled and the execution of each* i_k *is cyclic until every interaction* i_k *has fully completed at least a cycle.*

Definition 6 *1-Move-for-all Acyclic Parallelism: If there is an enabled interaction tree with the following form:* $(i_1 |; i_2 |; \cdots |; i_n)$ *then all interactions* i_k $(1 \leq k \leq n)$ *are enabled and the execution of each* i_k *is performed only once in each cycle of the whole parallel tree. In other words the end of the cycle is a kind of barrier.*

```
I [[i1;...; in]] hole =
  if seq(Id("i1;...; in")) = 1 then
    I [[i1]] hole ++ "seq(Id("i1;...;in")) := 2"
  elseif ...
  elseif seq(Id("i1;...;in")) = n-1 then
    I [[in-1]] hole ++ "seq(Id("i1;...;in")) := n"
  elseif seq(Id("i1;...;in")) = n then
    I [[in]] hole ++ "seq(Id("i1;...;in")) := 1"
    hole
  endif;
```

Figure 4.9: The translation scheme for the sequential composition

Definition 7 *External Non-determinism: If there is an enabled interaction tree with the following form:* $(i_1 + i_2 + \cdots + i_n)$*, then one and only one interaction* i_k $(1 \leq k \leq n)$ *will be effectively performed on each cycle of the non-deterministic interaction, and the choice is done by an external randomic function.*

Definition 8 *Internal Non-determinism: If there is an enabled interaction tree with the following form:* $(i_1 +? i_2 +? \cdots +? i_n)$*, where* i_k $(1 \leq k \leq n)$ *are atomic interactions, then one and only one interaction* i_k *will be effectively performed on each cycle of the non-deterministic interaction, and the choice is done internally by checking which interaction is ready to be performed. If more than one interaction is actually ready to be performed then the choice is done by an external randomic function, just like the previous definition.*

Definition 9 *Blocking input: Suppose there is an enabled interaction tree with the following form:* $(a <- u.b; i_1)$*, where* a *is a function name, and* $u.b$ *is an incoming value from the function name* b *of the unit* u*. Then, the respective input blocks* i_1*, until the input effectively occurs.*

Proposition 1 *The translation scheme for sequential composition preserves the corresponding definition.*

Proof. The function *Id* by definition assures that each sequence has its own sequence counter, which initial state is alway set to 1. So, in the above definition the first guard is always guaranteed to be true, and then the first interaction always takes place first.

When the first interaction is completed, the sequence counter is set to 2. This is guaranteed by augmenting the hole of the first interaction with the proper update.

```
I [[i1 || ... || in]] hole =
  I [[i1]] hole ++ ""
  ...
  I [[in]] hole ++ ""
```

Figure 4.10: The translation scheme for the cyclic parallel composition

```
I [[i1 | ... | in]] hole =
  if par(Id("i1 | ... | in")) = "init" then
    done(Id("i1")) := false;
    ...
    done(Id("in")) := false;
    par(Id("i1 | ... | in")) := "executing"
  elseif par(Id("i1 | ... | in")) = "executing" then
    if not (done(Id("i1")) and ... and done(Id("in"))) then
      I [[i1]] hole ++ "done(Id("i1")) := true"
      ...
      I [[i1]] hole ++ "done(Id("in")) := true"
    else
      par(Id("i1 | ... | in")) := "init"
      hole
    endif
  endif
```

Figure 4.11: The translation scheme for the 1-move-for-slower parallel composition

By induction, regarded that each interaction i_k $(1 \leq i \leq n)$ eventually finishes, then interaction i_n will be eventually performed and will complete the cycle by updating the sequence counter with 1. ■

Proposition 2 *The translation scheme for cyclic parallel composition preserves the corresponding definition.*

Proof. Since each interaction i_k $(1 \leq k \leq n)$ is mapped into an independent rule in the same level, this implies that if one of them is enabled to execute then all of them is enabled to execute, just like a block of ASM rules, and thus providing the required independence and parallelism stated in the definition. ■

Proposition 3 *The translation scheme for 1-move-for-slower parallel composition preserves the corresponding definition.*

Proof. As the *1-move-for-all*, this kind of parallelism is performed in two phases. Reagrded that the initial state of *par(Id("i_1 | ... | i_n"))* is `"init"`, all the interactions i_k, $(1 \leq k \leq n)$, are labeled with a boolean stating that they were not completely performed.

```
I [[i1 |; ... |; in]] hole =
  if par(Id("i1 |; ...  |; in")) = "init" then
    done(Id("i1")) := false;
    ...
    done(Id("in")) := false;
    par(Id("i1 |; ...  |; in")) := "executing"
  elseif par(Id("i1 |; ...  |; in")) = "executing" then
    if not done(Id("i1")) then
      I [[i1]] hole ++ "done(Id("i1")) := true"
    ...
    if not done(Id("in")) then
      I [[in]] hole ++ "done(Id("in")) := true"
    if done(Id("i1")) and ... and done(Id("in")) then
      par(Id("i1 |; ...  |; in")) := "init"
  endif
```

Figure 4.12: The translation scheme for the 1-move-for-all parallel composition

The main phase has two guards that either enables the parallel execution of all i_k, or finishes the complete parallel interaction cycle. This latter condition occurs if and only if all i_k has been completed. To prove this we can see that when an i_k has been completed the function *done(Id("i_k"))* is set to true. When the *slower* i_k has been completed then all *done(Id("i_k"))*, $(1 \leq k \leq n)$, are set to `true` and thus causing the end of the whole parallel interaction cycle. In the other case, the only way to finish the parallel interaction is to set all *done(Id("i_k"))*,$(1 \leq k \leq n)$, to `true`, since they were all initialized with `false` in the initial phase. These updates are only done when each i_k has been completely performed at least once. ■

Proposition 4 *The translation scheme for 1-move-for-all parallel composition preserves the corresponding definition.*

Proof. The translation scheme imposes two phases for executing this kind of parallelism. Regarded that the initial state of the function *par(Id("i_1 |; ... |; i_n"))* is `"init"`, always only the first guard will be executed, and in the sequence the second guard will be executed. The body of the second guard implies that every i_k $(1 \leq k \leq n)$ will execute, and each of them will be blocked whenever they finish, satisfying the parallelism and the *1-move* execution. ■

Proposition 5 *The translation scheme for external non-deterministic composition preserves the corresponding definition.*

Proof. Let *chooserandom* be a function that chooses randomically a number from a list of integers.

```
I [[i1 + ... + in]] hole =
  if choosing(Id("i1 +?  ...  +?  in")) then
    choose1(Id("i1+?...+?in")) := chooserandom(["1",...,"n"]);
    choosing(Id("i1 +?  ...  +?  in")) := false;
  else
    if choose(Id("i1 +?  ...  +?  in")) = 1 then
      I [[i1]] hole ++ "choosing(Id("i1 +?  ...  +?  in")) := true"
      endif
    elseif ...
    elseif choose(Id("i1 +?  ...  +?  in")) = n then
      I [[i1]] hole ++ "choosing(Id("i1 +?  ...  +?  in")) := true"
    endif
  endif
```

Figure 4.13: The translation scheme for the external non-deterministic composition

This translation scheme is also performed in two phases. Regarded that the function *choosing(Id("i_1 +? ... +? i_n"))* is initialized with `true`, then one interaction i_k, $(1 \leq k \leq n)$ will be selected among the others, and the selection will be placed in the function *choose* which is identified by the label of the external non-deterministic interaction.

In the second phase, only the selected interaction will be performed, and thus, satisfying the definition. ■

Proposition 6 *The translation scheme for internal non-deterministic composition preserves the corresponding definition.*

Proof. This translation is performed in two phases, controled by the function *nondet*. Regarded that the initial state of the function *nondet* is `"checking"`, then in the first phase, all interactions i_k, $(1 \leq k \leq n)$, will be checked to see if they can completely finish a cycle. The checking will be serialized in order to prevent an inconsistent update of the function *possibles* which will contain the number of each interaction that can occur. This serialization does not contradicts the definition 4.14. If there is only one possible interaction then this will be the interaction to be performed. This is assured by the required update on the functions *chosen* and *nondet*. If there is more than one possible interaction to be executed then this *chosen* interaction will be selected by the same function *chooserandom* used in the previous definition, thus assuring the required behavior stated in the definition. When the selection is performed by one of the two previous guard, then mandatorily the execution enters the second phase. In the second phase only the selected interaction will be executed as assured by the mutual exclusive guards on the function *chosen*. The cycle finishes correctly by putting all control functions

```
I [[i1 +? ... +? in]] hole =
  if nondet(Id("i1 +?  ...  +?  in")) = "checking" then
    if checking(Id("i1+?...+?in")) = 1 then
      if can_occur [[i1]] then
        possibles(Id("i1 +?  ...  +?  in")) :=
            cons(1,possibles(Id("i1 +?  ...  +?  in")));
      endif
      checking(Id("i1+?...+?in")) := 2;
    elseif ...
    elseif checking(Id("i1+?...+?in")) = n then
      if can_occur [[in]] then
        possibles(Id("i1 +?  ...  +?  in")) :=
            cons(n,possibles(Id("i1 +?  ...  +?  in")));
      endif
      checking(Id("i1+?...+?in")) := 1;
    elseif length(possibles(Id("i1 +?  ...  +?  in"))) = 1 then
      nondet(Id("i1 +?  ...  +?  in")) := "executing";
      chosen(Id("i1 +?  ...  +?  in") :=
          car(possibles(Id("i1 +?  ...  +?  in")));
    elseif length(possibles(Id("i1 +?  ...  +?  in"))) > 1 then
      nondet(Id("i1 +?  ...  +?  in")) := "executing";
      chosen(Id("i1 +?  ...  +?  in") :=
          chooserandom(possibles(Id("i1 +?  ...  +?  in")));
    endif
  if nondet(Id("i1 +?  ...  +?  in")) = "executing" then
    if chosen(Id("i1 +?  ...  +?  in") = 1 then
      I [[i1]] hole ++
          "possibles(Id("i1 +?  ...  +?  in")) := nil ++"
          "nondet(Id("i1 +?  ...  +?  in")) := "checking";"
    elseif ...
    if chosen(Id("i1 +?  ...  +?  in") = n then
      I [[in]] hole ++
          "possibles(Id("i1 +?  ...  +?  in")) := nil ++"
          "nondet(Id("i1 +?  ...  +?  in")) := "checking";"
    endif
  endif
```

Figure 4.14: The translation scheme for the internal non-deterministic composition

```
I [[ waiting var ]] hole =
  waiting(self, "var") := true;
  if not waiting(self, "var") then
    hole
```

Figure 4.15: The translation scheme for the waiting interaction

(*possible, nondet*) in the initial state. The function *checking* was already set to the initial state in the *checking* phase. ■

4.7 Synchronization of Rules and Interactions

Unit internal rules are just like ASM rules and its semantics is exactly the same. But, there is no direct relation on the synchronism between the interaction rules and internal rules. In order to guarantee appropriated synchronization when executing these rules, the IASM method provides:

1. A waiting rule used in the interaction section. This rule is represented by a boolean function name. When executed the rule updates the function with `true` and freezes the execution of the current node in a interaction cycle until the function is updated with `false`.

   Proposition 7 *The translation scheme preserves the condition that the waiting interaction finishes a cycle if and only if the corresponding function waiting is set to false.*

   Proof. The only way to finish the interaction is to execute the *hole* instructions. But these instructions are only executed if the function waiting is set to `false`. Conversely, if this function is set to `false`, necessarily the *hole* instructions will be executed and thus finishing a cycle. ■

2. All interactions may be labeled, for example:

   msgrec <- S.msgsend : nb_rcvd_msgs

   The corresponding label denotes an integer value which corresponds to how many times an interaction has been completed. This value can be used by the internal rules.

$$\mathcal{I}\,[\![i : \texttt{l}]\!]\, hole = \mathcal{I}\,[\![i : \texttt{l}]\!]\, hole \texttt{ ++ "l(Id(i))+=1"} =$$

Figure 4.16: The translation scheme for the labeled interaction

Proposition 8 *The translation scheme for labeled interaction guarantees the l has the number of times that i was completed, regarded that l is local to the corresponding unit instance and it is properly initialized with zero.*

Proof. By identifying l with the interaction label, we assure that l is local and unique for the corresponding unit instance. The increment of the function l is performed in the *hole* portion of the interaction, which by definiton is where the interaction finishes a cycle. Thus the function l correctly counts the number of performed cycles. ■

Chapter 5

An Example - The Alternating Bit Protocol

In this section we show a more elaborated example: the specification of the *Alternating Bit Protocol* [1]. The problem consists of transmitting messages through an unreliable channel. The channel delivers messages in the same order they were sent, but can occasionally loose some of them.

The specification is composed of three main unit definitions: *Sender*, *Receiver* and *Channel*. In addition, four other units are defined as environment units:

- unit *ClientSender*: Simulates the behaviour of a client that delivers messages; the messages are sent to the unit `Sender`, which sends copies of them to ensure the correct delivery.

- unit *ClientReceiver*: Simulates the behaviour of a client that receives messages from the unit `Receiver`.

- unit *Timer*: Sends periodically a message to the unit `Sender`, at a fixed rate, to indicate that a new copy of the current message must be sent. `Timer` has a behaviour which is similar to that of *external functions* in pure ASM.

- unit *Loose*: Sends messages to the unit `Channel` non-deterministically, indicating when a message will be lost. Like `Timer`, its behaviour is similar to that of external functions.

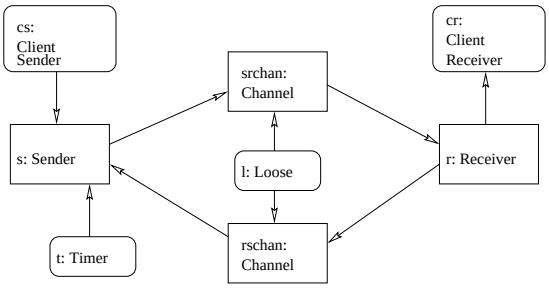

Fig. 1. Units and the flow of messages in the AB Protocol.

Figure 1 shows the relationship between these units. Note that two instances of the unit `Channel` are necessary.

After the unit `Sender` has performed the connections, it receives from the client a message which must be delivered to the receiver through an unreliable channel. Then it sends copies of the message through an output channel (srchan), together with a signal bit. This process is repeated until the unit `Sender` receives a correct acknowledgement message from the input channel. Then it waits for a new request from the client.

The timer is used to indicate that the sender has spent a certain amount of time waiting for the correct acknowledgement signal and must then deliver a new copy of the message. Note that after the message was sent via srchan, either the sender receives non-deterministically an acknowledgement bit or a timeout signal, as we should expect.

```
unit Sender
   function names
      timeout, connected := false;
      msg, recvbit;
      bit := 0;
      next := true;
      srchan := "srchan"; rschan := "rschan"; c := "c1"; t := "t";
   interaction
      if not connected then
         connect srchan: Channel.input, rschan: Channel.output,
                 c: Client, t: Timer;  : Connection
      else
         if next then
            msg <- c.msg;
         msg -> srchan ; bit -> srchan;
         (recvbit <- rschan.bit +
          timeout <- t.timeout);
         if not timeout and recvbit = bit then
            ack -> c;
            waiting(nextmsg);
         else
            waiting(samemsg);
      endif
   rules
      if Connection > 0 then
         connected := true;
      if waiting(nextmsg) then
         bit := toggle(bit); next := true;
         waiting(nextmsg) := false;
      endif
```

```
      if waiting(samemsg) then
         timeout := false; next := false;
         waiting(samemsg) := false;
      endif
end unit
```

After the unit `Receiver` has performed the connections, it receives a message from the input channel (*srchan*) and if it is not a copy of the last value received, it is sent to the client. It also sends an acknowledgement signal via an output channel (*rschan*). Note that *msg* is also sent via *rschan* in order to correctly match the interaction pattern of *Channel*.

```
unit Receiver
   function names
      connected := false;
      msg, bit;
      currbit := 0;
      srchan := "srchan"; rschan := "rschan"; c := "c2";
   interaction
      if not connected then
         connect srchan: Channel.output, rschan: Channel.input,
                 c: Client;  : Connection
      else
         msg <- srchan.msg ; bit <- srchan.bit;
         ((if bit = currbit then
              msg -> c;
              waiting(nextmsg);
           endif)
          |
          (msg -> rschan ; bit -> rschan))
      endif
   rules
      if Connection > 0 then
         connected := true;
      if waiting(nextmsg) then
         currbit := toggle(currbit); waiting(nextmsg) := false;
end unit
```

The unit `Channel` simulates a channel that delivers the messages in the order they are sent, but it may occasionally loose some of them. It connects to the units representing the *input* and *output* of the channel and also to a unit `Loose` that determines non-deterministically when a message will be lost. Note that the connection of the *input* and *output* will be blocked waiting the assignment that will be done in the units *Sender* and *Receiver*.

```
unit Channel
   function names
      input, output, msg, msg2, bit, bit2;
      loose: Loose;
      connected := false;
      queue := nil;
   interaction
      if not connected then
         connect input, output;
         connect loose: Loose;  : Connections
      endif ;
      ((msg2 <- input.msg ; bit2 <- input.bit ;
        waiting(buffering))
```

```
        |
        (if msg <> undef then
           msg -> output ; bit -> output ;
           waiting(cleanmsg);
         endif)
        |
        (loosemsg <- loose.loosemsg ;
         waiting(LoosingInQueue))
      )
   rules
      if Connection > 0 then
         connected := true;
      if waiting(buffering) then
         queue := append( (msg2, bit2), queue);
         waiting(buffering) := false;
      endif
      if waiting(cleanmsg) then
         msg := undef; waiting(cleanmsg) := false;
      if waiting(LoosingInQueue) then
         queue := tail(queue); waiting(LoosingInQueue) := false;
      if msg = undef and queue <> nil then
         msg = first(head(queue)); bit = second(head(queue));
         queue := tail(queue);
      endif;
end unit
```

This protocol has been previously formalized using the ASM method in [7]. In that work, the behaviour of the communication channel was not clearly defined. It was necessary to write identical code for both the communication sender-receiver and receiver-sender.

Now we will specify the environment units *ClientSender*, *ClientReceiver*, *Timer*, and *Loose*.

```
unit ClientSender
   function names
      connected := false;
      msg := 0;
      ack;
      s := "s";
   interaction
      if not connected then
         connect s: Sender;  : Connection
      else
        waiting(preparing_msg);
        msg -> s;
        ack <- s.ack;
      endif
   rules
      if Connection > 0 then
         connected := true;
      if waiting(preparing_msg);
         msg := msg + 1;
         waiting(preparing_msg) := false;
      endif
end unit

unit ClientReceiver
   function names
      connected := false;
      msg := 0;
```

```
      r := "r";
   interaction
      if not connected then
         connect r: Receiver;  : Connection
      else
        msg <- r;
        waiting(processing_msg);
      endif
   rules
      if Connection > 0 then
         connected := true;
      if waiting(processing_msg);
         out := cons(msg, out);
         waiting(processing_msg) := false;
      endif
end unit

unit Timer
   function names
      connected := false; timeout := true;
      TIMEOUTPARAM := 100; firstclock; lastclock;
      mode := "init"; getting_clock := true;
      s := "s"; c := "c"; init_signals := 0; timeouts := 0;
   interaction
      if not connected then
         connect s: Sender;  : Connection
         init_timeout <- s.init_timeout;
         first_clock <- clock.time;
         last_clock <- clock.time;
      else
           if lastclock - firstclock > TIMEOUTPARAM then
             timeout -> s : send_timeout
         | init_timeout <- s.init_timeout : recv_init_signal;
         | ask_clock -> clock; last_clock <- clock.time;
      endif
   rules
      if Connection > 0 then
         connected := true;
      if recv_init_signal > init_signals then
         init_signals := init_signals + 1;
         first_clock := last_clock;
      endif
      if timeouts < send_timeout then
         timeouts := timeouts + 1;
         first_clock := last_clock;
      endif
end unit

environment unit Clock
   interaction
      if not connected then
         connect req;
      else
         ask_clock <- req.ask_clock; time -> req
      endif
end unit

unit Loose
   function names
      connected := false; timeout := true;
      c ; init_signals := 0; timeouts := 0;
   interaction
      if not connected then
         connect c;  : Connection
      else
         waiting(a_loose);
         loosemsg -> c
      endif
   rules
```

```
      if Connection > 0 then
         connected := true;
      if waiting(a_loose) then
         if random() = "loose" then
            waiting(a_loose) := false;
      endif
end unit
```

The startup specification that creates the initial unit instances may be written as:

```
specification ABP
   rschan, srchan: Channel;
   s: Sender; r: Receiver;
   t: Timer; l: Loose;
   c1: Client; c2: Client;
   c: Clock;
end specification
```

Chapter 6

Another Example - Active Mobile Objects

Let us give a problem to be specified. This will be useful in comparing this approach with others and in illustrating its reasoning capability. The following paragraph reproduces verbatim a simplified and eventually modified version of the problem consisting in managing a virtual program committee meeting for a conference. The problem was presented as a challenge in [3].

A conference is announced, and an electronic submission form is publicized. Each author fetches the form and activate it. Each author fills an instance of the form with the required data and attaches a paper. The form checks that none of the required fields are left blank and sent the data and the paper to the program chair. The program chair collects the submission forms and assign the submissions to the committee members, by instructing each submission form to generate a review form for each assigned member. Each assigned member is a reviewer, and may decide to review the paper directly or send it to another reviewer. The review form keeps track of the chain of reviewers. Eventually a review is filled and it finds its way back to the program chair. The program chair collects all review forms. The chair merge all review forms for each paper in a paper report form. Then the chair declares each report form an accepted paper report form, or a rejected paper review form, and finally returns this form to each author. All accepted paper report forms are required to generate final version forms on which the author attaches the final version of the paper and send it back to the program chair.

Now we will give a partial Interactive ASM semantics for this problem. For the sake of conciseness, we will focus on the *interaction section* of the specification and will *omit* the declaration of the internal state and the the internal rules of each unit.

```
unit Author
```

```
...
interaction
   if (wants_to_submit) then
      choose s in Submission
         connect s ;
         wait(submission_ok);
         data_paper -> s;
         disconnect s;
      end choose
   endif
  |
   if (waiting_result) then
      connect the_chair;
      result <- the_chair.result(self);
      disconnect the_chair
   endif
  |
   if (ready_final) then
      connect the_chair;
      final -> the_chair;
      disconnect the_chair
   endif
...
end unit
```

This unit definition models the behavior that the author of an article must have in order to correctly participate of a call for papers.

```
unit Submission
...
interaction
   connect a: Author;
   data_paper <- a.data_paper;
   disconnect a;
   connect the_chair;
   data_paper -> the_chair;
   disconnect the_chair;
...
end unit
```

The unit definition *Submission* models an agent that interacts with the author of a paper, and gets all necessary information with an attached paper. The internal rules should make all the necessary checking.

```
unit Chair
...
interaction
   if (before_deadline) then
      connect s: Submission;
      data_papers <-- s.data_paper;
      disconnect s;
   endif
  |
   if (exist x in data_papers) then
      choose r in Reviewer
         connect r ;
         wait(one_paper);
         a_paper -> r : sent_review;
         disconnect r;
      endchoose
   endif
  |
   if (sent_review > received_review) then
      connect r: Reviewer;
```

```
      review <- r : received_review;
      disconnect r
   endif
   |
   if (result_ok) then
      var r in Result
         connect author(r) ;
         r -> author(r);
         disconnect author(r)
      endvar
   endif
   |
   if (receiving_final_versions) then
      var a in Author
         if (accepted(a) then
            connect a ;
            finals <-- a.final;
            disconnect a
         endif
      endvar
   endif
...
end unit
```

The unit *Chair* is the more interactive one. It has five parallel actions, each one represented by a guarded rule evaluated depending only on its internal state.

```
unit Reviewer
...
interaction
   connect the_chair;
   a_paper <- the_chair.a_paper;
   if (directly_review) then
      wait(the_review);
      review -> the_chair
   else
      choose r in Reviewer
         connect r;
         firsthistory as history -> r | a_paper -> r ;
         disconnect r;
      endchoose
   endif
   |
   connect r;
   history2 <- r.history | a_paper2 <- r ;
   disconnect r;
   if (reviewed(a_paper2) then
      connect head(history2);
      (tail(history2) as history -> head(history2) |
       apaper2 -> head(history2) );
      disconnect head(history2)
   elseif (directly_review) then
      wait(the_review2);
      connect head(history2);
      (tail(history2) as history -> head(history2) |
       review2 -> head(history2) ) ;
      disconnect head(history2)
   else
      choose r in Reviewer
         connect r;
         cons(self, history2) as history -> r | a_paper2 -> r
         disconnect r;
      endchoose
```

```
end unit
```

The unit Reviewer has an elaborated scheme for creating the dynamic communication between the reviewers. When passing a paper forward it adds its identification to the history of the reviewers for that paper. When passing a paper backward in the history, it removes a reviewer from the list being sent backward with the review.

At last, we may specify the specification startup to create some initial unit instances. Other instances must have to be created dynamically by some *Authorization* unit, intentionally not specified.

```
specification startup
   the_chair: unit Chair;
   committee_member1: unit Reviewer;
   ...
   committee_memberN: unit Reviewer;
end specification
```

Now, we are going to present two propositions about the previous example and show their validity.

Proposition 9 *All papers submitted after the deadline date are not received by the chair.*

Proof: The first guard in the unit chair directly guarantees this proposition.

However, as it is, the specification does not treat a notification of this fact to the author.

Proposition 10 *All submissions received by the chair generates a report form to the author if, and only if, there is a reviewer that directly reviews the paper.*

Proof:

1. If there is a reviewer that directly reviews the papers then all submissions generates a report form.

   We have that all paper received by the chair is sent to a reviewer. This is guaranteed by the second guard of the unit definition *Chair*, that checks that if there is a submission that was not sent to any reviewer. Since this guard is parallel with all the other it will be executed without being blocked.

   In all guards of the unit Reviewer, it can be seen that a received paper is either reviewed and sent backward in the history, or sent (backward or

forward) to another reviewer. Since, by assumption, there is a reviewer that directly reviews the paper, the paper will certainly be sent back through its history when reviewed. In the definition the first element of the history is the chair. So, the paper will certainly return to the chair.

Assuming that the internal behavior of the chair produces the final result when all the reviews of the paper had come back, then by the fourth guard of the unit Chair, we can guarantee that all results are sent back to the authors.

2. All submission would have back the result report, only if there is a reviewer that directly reviews each paper.

   Suppose if for a certainly review, there is not a reviewer that directly reviews it. Then, that review will never be sent to the chair, and consequently, will not be sent to the author.

   Depending on the internal behavior of the chair when preparing the result, it may happen that none of the authors receives the review.

Chapter 7

Conclusions

We have presented a proposal to promote the reuse of ASMs specifications while addressing important issues such as communication and concurrency. The idea of explicitly isolating the interaction between computing units with different purposes makes clearer their interdependencies. This also provides a useful mechanism to formalize the environment in which a specification will work.

The approach was successfully used in the specification of the Alternating Bit protocol, where we have reused the specification of the unit *Channel*, which may be connected differently depending on its usage. We have shown that the explicit message passing mechanism composed with well-known operators provides a powerful and natural specification mechanism.

The dynamic configuration of the communication topology presented is an essential feature that may be used to specify mobile systems which are being increasily used but still lacks more suitable formal approaches.

There are some aspects that must be further studied:

- study of static checking mechanism, e.g., type systems;
- study of more powerful reuse mechanisms, e.g., inheritance;
- encoding of procedure and/or function calls and/or method invocations;
- use of the approach in large scale specifications.

Bibliography

[1] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-Duplex transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969.

[2] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[3] L. Cardelli. Abstractions for mobile computation. Position Paper, http://www. luca.demon.co.uk/Papers.html, May 1998.

[4] G. D. Castillo, Y. Gurevich, and K. Stroetmann. Typed Abstract State Machines. Submitted to the Journal of Universal Computer Science, 1998.

[5] P. Glavan and D. Rosenzweig. Communicating Evolving Algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer, 1993.

[6] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[7] J. Huggins and R. Mani. The evolving algebra interpreter version 2.0. Manual of the interpreter (http://www.eecs.umich.edu/gasm).

[8] W. May. Specifying Complex and Structured Systems with Evolving Algebras. In *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, pages 535–549. Springer, 1997.

[9] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

Appendix A

Syntactic Conventions

We have used the following conventions:

- Unit definition is the specification of the code of a unit.
- Unit declaration creates a function name or a unit instance.
- Italic letters: u_i, $u_i.interactions$ denote syntactic elements.
- Typewriter letters: denote reserved words of the original ASM language and of the constructions proposed here.
- ‘a : denotes a label that uniquely identifies the syntactic element a, i.e., is the element of the vocabulary without being interpreted.
- ⟦ ... ⟧ : denotes syntactic elements.
- `a += b;` ≡ `a := a + b;`
- `a -= b;` ≡ `a := a - b;`
- has_message(*origin*) =(∃ `x` ∈ MSG) *target(x)=self* ∧ *label(x)=origin*
- match_msg(x, *origin*) = $target(x) = self \wedge label(x) = origin$
- 1. a_party_is(x, u_1, U_1, f, b) =
     party1(x) = (x, u_1, U_1, f, b) or
     party2(x) = (x, u_1, U_1, f, b)
  2. a_party_is(x, u_1, U_1, b) =
     party1(x) = $(x, u_1, U_1, \_, b)$ or
     party2(x) = $(x, u_1, U_1, \_, b)$

3. a_party_is(x, U_1, b) =
   party1(x) = $(x, \_, U_1, \_, b)$ or
   party2(x) = $(x, \_, U_1, \_, b)$
4. other_party(x, u, U, f) =
   if party1(x) = $(u, U, f, \_)$ then
   party2(x)
   elseif party2(x) = $(u, U, f, \_)$ then
   party1(x)
   else undef
5. wants$(x, u_1, U_1, f1, u_2, U_2, f2)$ = a_party_is$(x, u_1, U_1, f1, true)$ and (a_party_is$(x, undef, undef, undef, false)$ or (a_party_is$(x, u_2, U_2, f_2, false)$ or a_party_is$(x, u_2, U_2, undef, false)$ or a_party_is$(x, undef, U_2, undef, false)$)
6. connected$(x, u_1, U_1, f, u_2, U_2)$ = a_party_is$(x, u_1, U_1, f1, true)$ and a_party_is$(x, u_2, U_2, f2, true)$
7. can_occur$[\![a$`<-` $u.b]\!]$ =
   $has_message('u.b)$
   can_occur$[\![a$`<--` $u.b]\!]$ =
   $has_message('u.b)$
   can_occur$[\![$`connect u:U.f` $]\!]$ =
   `exists x in Connections.`
   `wants(x,u,"U","f",self,Mod(self),"u")`
   can_occur$[\![$`connect u` $]\!]$ =
   `exists x in Connections.`
   `wants(x,undef,undef,undef,self,Mod(self),_)`
   can_occur$[\![i_1; ...; i_n]\!]$ = `false`
   can_occur$[\![i_1 \,||\, ... \,||\, i_n]\!]$ = `false`
   can_occur$[\![i_1 \mid ... \mid i_n]\!]$ = `false`
   can_occur$[\![i_1 \,|; ... \,|; i_n]\!]$ = `false`
   can_occur$[\![i_1 + ... + i_n]\!]$ = `false`
   can_occur$[\![i_1 + ? ... + ? i_n]\!]$ = `false`
   can_occur$[\![_]\!]$ = `true`

Appendix B

Translating the IASM AB Protocol

In this section we present the translation of the AB Protocol into the pure ASM notation.

```
module Sender
  function names
    timeout, connected := false;
    msg, recvbit;
     bit := 0;
     next := true;
     srchan := "srchan"; rschan := "rschan"; c := "c1"; t := "t";
  rules
    if Connection > 0 then
      connected := true;
    if waiting("nextmsg") then
      bit := toggle(bit); next := true;
      waiting("nextmsg") := false;
    endif
    if waiting("samemsg") then
      timeout := false; next := false;
      waiting("samemsg") := false;
    endif
    if not connected then
      if seq1 = 1 then
        if exists x in Connections.
        wants(x,srchan,"Channel","input",self,"Sender","srchan")
          choose x in Connections satisfying
          wants(x, srchan,"Channel","input",self,"Sender","srchan")
            Connections(x) := false;
          endchoose;
          extend Connections with x1
            party1(x) := (self, "Sender", "srchan", true);
            party2(x) := (srchan, "Channel", "input", true);
          endextend
          seq1 := 2;
        else
          extend Connections with x
            party1(x) := (self, "Sender", "srchan", true);
            party2(x) := (srchan, "Channel", "input",false);
          endextend
          if exists x in Connections.
          connected(x,srchan,"Channel","f",self,"Sender","srchan") then
```

```
        choose x in Connections satisfying
        connected(x,srchan,"Channel","f",self,"Sender","srchan")
          Connections(x) := false;
        endchoose;
        seq1 := 2;
      endif
    endif
  elseif seq1 = 2 then
    if exists x in Connections.
    wants(x, rschan,"Channel","output",self,"Sender","rschan")
      choose x in Connections satisfying
      wants(x,rschan,"Channel","output",self,"Sender","rschan")
        Connections(x) := false;
      endchoose;
      extend Connections with x1
        party1(x) := (self, "Sender", "rschan", true);
        party2(x) := (rschan, "Channel", "output", true);
      endextend
      seq1 := 3;
    else
      extend Connections with x
        party1(x) := (self, "Sender", "rschan", true);
        party2(x) := (rschan, "Channel", "output",false);
      endextend
      if exists x in Connections.
      connected(x,rschan,"Channel","output",self,"Sender","rschan") then
        choose x in Connections satisfying
        connected(x,srchan,"Channel","output", self,"Sender","rschan")
          Connections(x) := false;
        endchoose;
        seq1 := 3;
      endif
    endif
  elseif seq1 = 3 then
    if exists x in Connections.
    wants(x, c1, "Client", _, self, "Sender","c1")
      choose x in Connections satisfying
      wants(x, c1, "Client", _, self, "Sender,"c1")
        Connections(x) := false;
      endchoose;
      extend Connections with x1
        party1(x) := (self, "Sender", "c1", true);
        party2(x) := (c1, "Client", _, true);
      endextend
      seq1 := 4;
    else
      extend Connections with x
        party1(x) := (self, "Sender", "c1", true);
        party2(x) := (c1, "Client", _, false);
      endextend
      if exists x in Connections.
      connected(x, c1, "Client", _, self, "Sender","c1") then
        choose x in Connections satisfying
        connected(x, c1, "Client", _, self, "Sender","c1")
          Connections(x) := false;
        endchoose;
        seq1 := 4;
      endif
    endif
  elseif seq1 = 4 then
    if exists x in Connections.
    wants(x, t1, "Timer", _, self, "Sender","t1")
      choose x in Connections satisfying
```

```
      wants(x, t1, "Timer", _, self, "Sender,"t1")
        Connections(x) := false;
      endchoose;
      extend Connections with x1
        party1(x) := (self, "Sender", "t1", true);
        party2(x) := (t1, "Timer", _, true);
      endextend
      seq1 := 4;
    else
      extend Connections with x
         party1(x) := (self, "Sender", "t1", true);
         party2(x) := (t1, "Timer", _, false);
      endextend
      if exists x in Connections.
      connected(x, t1, "Timer", _, self, "Sender","t1") then
        choose x in Connections satisfying
        connected(x, t1, "Timer", _, self, "Sender","t1")
          Connections(x) := false;
        endchoose;
        seq1 := 1;
      endif
    endif
  endif
else
  if seq2 = 1 then
    if next then
      if has_message(c, "msg") then
        choose x in MSG satisfying match_msg(x, c, "msg")
          msg := cont(x);
          MSG(x) := false;
        endchoose;
        seq2 := 2;
      endif
    endif
  elseif seq2 = 2 then
    extend MSG with x
      target(x) := srchan;
      label(x) := self + "msg";
      cont(x) := msg;
    endextend
    seq2 := 3;
  elseif seq2 = 3 then
    extend MSG with x
      target(x) := rschan;
      label(x) := self + "bit";
      cont(x) := bit;
    endextend
    seq2 := 4;
  elseif seq2 = 4 then
    if choosing1 then
      choose1 := chooserandom(["1", "2"]);
      choosing1 := false;
    else
      if will_occur(1, choose1,
           [("input",rschan,"bit"),("input",t,"timeout")] then
        if has_message(rschan, "bit") then
          choose x in MSG satisfying match_msg(x,rschan,"bit")
            bit := cont(x);
            MSG(x) := false;
          endchoose;
          seq2 := 5;
          choosing := true;
```

```
          endif
        elseif will_occur(2, choose1,
             [("input",rschan,"bit"),("input",t,"timeout")] then
          if has_message(t, "timeout") then
            choose x in MSG satisfying match_msg(x, t, "timeout")
              timeout := cont(x);
              MSG(x) := false;
            endchoose;
            seq2 := 5;
            choosing := true;
          endif;
        endif;
      endif
    elseif seq2 = 5 then
      if not timeout and recvbit = bit
        if seq3 = 1 then
          extend MSG with x
            target(x) := c;
            label(x) := self + "ack";
            cont(x) := ack;
          endextend;
          seq3 := 2;
        elseif seq3 = 2 then
          waiting("nextmsg") := true;
          if not waiting("nextmsg") then
            seq3 := 1;
            seq2 := 1;
          endif
        endif
      else
        waiting("samemsg") := true;
        if not waiting("nextmsg") then
          seq2 := 1;
        endif
      endif
    endif
  endif
end unit

unit Receiver
  function names
    connected := false;
    msg, bit;
    currbit := 0;
    srchan := "srchan"; rschan := "rschan"; c := "c2";
  rules
    if Connection > 0 then
      connected := true;
    if waiting("nextmsg") then
      currbit := toggle(currbit); waiting("nextmsg") := false;
    if not connected then
      if seq1 = 1 then
        if exists x in Connections.
        wants(x,srchan,"Channel","output",self,"Sender","srchan")
          choose x in Connections satisfying
          wants(x, srchan,"Channel","output",self,"Sender,"srchan")
            Connections(x) := false;
          endchoose;
          extend Connections with x1
            party1(x) := (self, "Sender", "srchan", true);
            party2(x) := (srchan, "Channel", "output", true);
          endextend
          seq1 := 2;
        else
```

```
    extend Connections with x
      party1(x) := (self, "Sender", "srchan", true);
      party2(x) := (srchan, "Channel", "output",false);
    endextend
    if exists x in Connections.
    connected(x,srchan,"Channel","output",self,"Sender","srchan") then
      choose x in Connections satisfying
      connected(x,srchan,"Channel","output",self,"Sender","srchan")
        Connections(x) := false;
      endchoose;
      seq1 := 2;
    endif
  endif
elseif seq1 = 2 then
  if exists x in Connections.
  wants(x, rschan, "Channel", "input", self, "Sender","rschan")
    choose x in Connections satisfying
    wants(x, rschan, "Channel", "input", self, "Sender,"rschan")
      Connections(x) := false;
    endchoose;
    extend Connections with x1
      party1(x) := (self, "Sender", "srchan", true);
      party2(x) := (rschan, "Channel", "input", true);
    endextend
    seq1 := 3;
  else
    extend Connections with x
      party1(x) := (self, "Sender", "srchan", true);
      party2(x) := (rschan, "Channel", "input", false);
    endextend
    if exists x in Connections.
    connected(x, rschan,"Channel","input",self,"Sender","rschan") then
      choose x in Connections satisfying
      connected(x,rschan,"Channel","input",self,"Sender","rschan")
        Connections(x) := false;
      endchoose;
      seq1 := 3;
    endif
  endif
elseif seq1 = 3 then
  if exists x in Connections.
  wants(x, c2, "Client", _, self, "Sender","c2")
    choose x in Connections satisfying
    wants(x, c2, "Client", _, self, "Sender,"c2")
      Connections(x) := false;
    endchoose;
    extend Connections with x1
      party1(x) := (self, "Sender", "c2", true);
      party2(x) := (c2, "Client", _, true);
    endextend
    seq1 := 4;
  else
    extend Connections with x
      party1(x) := (self, "Sender", "c2", true);
      party2(x) := (c2, "Client", _, false);
    endextend
    if exists x in Connections.
    connected(x, c2, "Client", _, self, "Sender","c2") then
      choose x in Connections satisfying
      connected(x, c2, "Client", _, self, "Sender","c2")
        Connections(x) := false;
      endchoose;
      seq1 := 1;
```

```
      endif
    endif
  endif
else
  if seq2 = 1 then
    if has_message(srchan, "msg") then
      choose x in MSG satisfying match_msg(x, srchan, "msg")
        msg := cont(x);
        MSG(x) := false;
      endchoose;
      seq2 := 2;
    endif
  elseif seq2 = 2 then
    if has_message(srchan, "bit") then
      choose x in MSG satisfying match_msg(x, srchan, "bit")
        bit := cont(x);
        MSG(x) := false;
      endchoose;
      seq2 := 3;
    endif
    elseif seq2 = 3 then
      if not done_par1(1) then
        if bit = currbit then
          if seq3 = 1 then
            extend MSG with x
              target(x) := srchan;
              label(x) := self + "msg";
              cont(x) := msg;
            endextend;
            seq3 := 2;
          elseif seq3 = 2 then
            waiting("nextmsg") := true;
            if not waiting("nextmsg") then
              seq3 := 1
              done_par1 := true;
            endif
          endif
        endif
      endif
      if not done_par1(2) then
        if seq4 = 1 then
          extend MSG with x
            target(x) := rschan;
            label(x) := self + "msg";
            cont(x) := msg;
          endextend;
          seq4 := 2;
        elseif seq4 = 2 then
          extend MSG with x
            target(x) := rschan;
            label(x) := self + "bit";
            cont(x) := bit;
          endextend;
          seq4 := 1;
          done_par1(2) := true;
        endif
      endif
      if done_par1(1) and done_par1(2) then
        done_par1(1) := false;
        done_par1(2) := false;
        seq2 := 1;
```

```
          endif;
        endif
      endif
    endif
end unit

unit Channel
   function names
      input, output, msg, msg2, bit, bit2;
      loose: Loose;
      connected := false;
      queue := nil;
   rules
      if Connection > 0 then
         connected := true;
      if waiting("buffering") then
         queue := append( (msg2, bit2), queue);
         waiting("buffering") := false;
      endif;
      if waiting("cleanmsg") then
         msg := undef; waiting("cleanmsg") := false;
      if waiting("LoosingInQueue") then
         queue := tail(queue); waiting("LoosingInQueue") := false;
      if msg = undef and queue <> nil then
         msg = first(head(queue)); bit = second(head(queue));
         queue := tail(queue);
      endif;
      if not connected then
         if seq1 = 1 then
            if exists x in Connections.
            wants(x, _, _, _, self, "Channel", "input",_)
               choose x in Connections satisfying
               wants(x, _, _, _, self, "Channel", "input",_)
                  Connections(x) := false;
                  extend Connections with x1
                     party1(x1) := (self,"Channel", "input", true);
                     party2(x1) :=
                        other_part(x,self,"Channel","input");
                  endextend
               endchoose;
               seq1 := 2;
            endif
         endif
         elseif seq1 = 2 then
            if exists x in Connections.
            wants(x, _, _, _, self, "Channel", "output",_)
               choose x in Connections satisfying
               wants(x, _, _, _, self, "Channel", "out",_)
                  Connections(x) := false;
                  extend Connections with x1
                     party1(x1) := (self,"Channel","output",true);
                     party2(x1) :=
                        other_part(x, self, "Channel", "output");
                  endextend
               endchoose;
               seq1 := 3;
            endif
         elseif seq1 = 3 then
            if exists x in Connections.
            wants(x, l1, "Loose", _, self, "Channel","l1")
               choose x in Connections satisfying
               wants(x, l1, "Loose", _, self, "Channel","l1")
                  Connections(x) := false;
```

```
            endchoose;
            extend Connections with x1
               party1(x1) := (self, "Channel", "l1", true);
               party2(x1) := (l1, "Loose", _, true);
            endextend
            seq1 := 4;
         else
            extend Connections with x
               party1(x) := (self, "Channel", "l1", true);
               party2(x) := (l1, "Loose", _, false);
            endextend
            if exists x in Connections.
            connected(x, l1, "Loose", _, self, "Channel","l1") then
               choose x in Connections satisfying
               connected(x, l1, "Loose", _,self,"Loose","l1")
                  Connections(x) := false;
               seq1 := 1;
            endif
         endif
      endif
   else
      if seq2 = 1 then
         if has_message(input, "msg") then
            choose x in MSG satisfying match_msg(x,input,"msg")
               msg2 := cont(x);
               MSG(x) := false;
            endchoose;
            seq2 := 2;
         endif
      elseif seq2 = 2 then
         if has_message(input, "bit") then
            choose x in MSG satisfying match_msg(x,input,"bit")
               bit2 := cont(x);
               MSG(x) := false;
            endchoose;
            seq2 := 3;
         endif
      elseif seq2 = 3 then
         waiting("buffering") := true;
         if (not waiting("buffering")) then
            seq1 := 1;
      endif
      if msg <> undef then
         if seq3 = 1 then
            extend MSG with x
               target(x) := output;
               label(x) := self + "msg";
               cont(x) := msg;
            endextend;
            seq3 := 2;
         elseif seq3 = 2 then
            extend MSG with x
               target(x) := output;
               label(x) := self + "bit";
               cont(x) := bit;
            endextend;
            seq3 := 3;
         elseif seq3 = 3 then
            waiting("cleanmsg") := true;
            if (not waiting("cleanmsg")) then
               seq3 := 1;
```

```
            endif
            if seq4 = 1 then
               if has_message(l1, "loosemsg") then
               choose x in MSG satisfying
               match_msg(x,l1,"loosemsg")
                  loosemsg := cont(x);
                  MSG(x) := false;
               endchoose;
               seq4 := 2;
            elseif seq4 = 2 then
               waiting("LoosingInQueue") := true;
               if (not waiting("LoosingInQueue")) then
                  seq4 := 1;
            endif
         endif
      endif
end unit

module ClientSender
   function names
      connected := false;
      msg := 0;
      ack; s := "s";
   rules
      if Connection > 0 then
         connected := true;
      if waiting(preparing_msg);
         msg := msg + 1;
         waiting(preparing_msg) := false;
      endif
      if not connected then
         if exists x in Connections.
               wants(x, s, "Sender", _, self, _,"s")
            choose x in Connections satisfying
                  wants(x, s, "Sender", _, self, _,"s")
               Connections(x) := false;
            endchoose;
            extend Connections with x1
               party1(x) := (self, "ClientSender", "s", true);
               party2(x) := (s, "Sender", _, true);
            endextend
         else
            extend Connections with x
               party1(x) := (self, "ClientSender", "s", true);
               party2(x) := (s, "Sender", _, false);
            endextend
            if exists x in Connections.
            connected(x, s, "Sender", _, self, "ClientSender","s") then
               choose x in Connections satisfying
               connected(x, s, "Sender", _, self, "ClientSender","s")
                  Connections(x) := false;
               seq1 := 2;
            endif
         endif
      else
         if seq1 = 1 then
            waiting("preparing_msg") := true;
            if not waiting("preparing_msg") then
               seq1 := 2;
            endif
         elseif seq1 = 2 then
            extend MSG with x
               target(x) := s;
```

```
               label(x) := self + "msg";
               cont(x) := msg;
            endextend;
            seq := 3;
         elseif seq1 = 3 then
            if has_message(s, "ack") then
               choose x in MSG satisfying match_msg(x, s, "ack")
                  ack := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 1;
            endif
         endif
      endif
end unit

module ClientReceiver
   function names
      connected := false;
      msg := 0;
      r := "r";
   interaction
      if Connection > 0 then
         connected := true;
      if waiting(processing_msg);
         out := cons(msg, out);
         waiting(processing_msg) := false;
      endif
      if not connected then
         if exists x in Connections.
               wants(x, r, "Receiver", _, self, _,"r")
            choose x in Connections satisfying
                  wants(x, r, "Receiver", _, self, _,"r")
               Connections(x) := false;
            endchoose;
            extend Connections with x1
               party1(x) := (self, "ClientReceiver", "r", true);
               party2(x) := (r, "Receiver", _, true);
            endextend
         else
            extend Connections with x
               party1(x) := (self, "ClientReceiver", "r", true);
               party2(x) := (s, "Receiver", _, false);
            endextend
            if exists x in Connections.
            connected(x,r,"Receiver",_, self,"ClientReceiver","r") then
               choose x in Connections satisfying
               connected(x,r,"Receiver",_,self,"ClientReceiver","r")
                  Connections(x) := false;
               seq1 := 2;
            endif
         endif
      else
         if seq1 = 1 then
            if has_message(r, "msg") then
               choose x in MSG satisfying match_msg(x, r,"msg")
                  msg := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 2;
            endif
         elseif seq1 = 2 then
            waiting("processing_msg") := true;
            if not waiting("processing_msg") then
```

```
            seq1 := 1;
         endif
      endif
   endif
end unit

module Timer
   function names
      connected := false; timeout := true;
      TIMEOUTPARAM := 100; firstclock; lastclock;
      mode := "init"; getting_clock := true;
      s := "s"; c := "c"; init_signals := 0; timeouts := 0;
   rules
      if Connection > 0 then
         connected := true;
      if recv_init_signal > init_signals then
         init_signals := init_signals + 1;
         first_clock := last_clock;
      endif
      if timeouts < send_timeout then
         timeouts := timeouts + 1;
         first_clock := last_clock;
      endif
      if not connected then
         if seq1 = 1 then
            if exists x in Connections.
            wants(x, _, _, _, self, "Timer", "s")
               choose x in Connections satisfying
               wants(x, _, _, _, self, "Timer", "s")
                  Connections(x) := false;
                  extend Connections with x1
                     party1(x1) := (self, "Timer", "s", true);
                     party2(x1) := other_part(x,self,"Timer","s");
                  endextend
               endchoose;
               seq1 := 2;
            endif
         elseif seq1 = 2 then
            if has_message(s, "init_timeout") then
               choose x in MSG satisfying
               match_msg(x, s, "init_timeout")
                  init_timeout := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 3;
            endif
         elseif seq1 = 3 then
            if has_message(clock, "time") then
               choose x in MSG satisfying match_msg(x,clock,"time")
                  first_clock := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 4;
            endif
         elseif seq1 = 4 then
            if has_message(clock, "time") then
               choose x in MSG satisfying match_msg(x,clock,"time")
                  last_clock := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 1;
            endif
         endif
      else
         if lastclock - firstclock > TIMEOUTPARAM then
```

```
            extend MSG with x
               target(x) := s;
               label(x) := self + "timeout";
               cont(x) := timeout;
            endextend;
            send_timeout := send_timeout + 1;
         endif
         if has_message(s, "init_timeout") then
            choose x in MSG satisfying
            match_msg(x,s,"init_timeout")
               init_timeout := cont(x);
               MSG(x) := false;
            endchoose;
            recv_init_signal := recv_init_signal + 1;
         endif
         if seq2 = 1 then
            extend MSG with x
               target(x) := clock;
               label(x) := self + "ask_clock";
               cont(x) := ask_clock;
            endextend;
            seq2 := 2;
         elseif seq2 = 1 then
            if has_message(clock, "time") then
               choose x in MSG satisfying match_msg(x,clock,"time")
                  last_clock := cont(x);
                  MSG(x) := false;
               endchoose;
            endif
         endif
      endif
end unit

module Clock
   interaction
      if not connected then
         if exists x in Connections.
               wants(x, _, _, _, self, "Clock", "req")
            choose x in Connections satisfying
                  wants(x, _, _, _, self, "Clock", "req")
               Connections(x) := false;
               extend Connections with x1
                  party1(x1) := (self, "Clock", "req", true);
                  party2(x1) := other_part(x,self,"Clock","req");
               endextend
            endchoose;
            seq1 := 2;
         endif
      else
         if seq1 = 1 then
            if has_message(req, "ask_clock") then
               choose x in MSG satisfying
               match_msg(x,r,"ask_clock")
                  ask_clock := cont(x);
                  MSG(x) := false;
               endchoose;
               seq1 := 2;
            endif
         elseif seq1 = 2 then
            extend MSG with x
               target(x) := req;
               label(x) := self + "time";
```

```
               cont(x) := time;
            endextend;
         endif
      endif
end unit

unit Loose
   function names
      connected := false;
      c ;
   rules
      if Connection > 0 then
         connected := true;
      if waiting(a_loose) then
         if random() = "loose" then
            waiting(a_loose) := false;
      endif
      if not connected then
         if exists x in Connections.
               wants(x, _, _, _, self, "Loose", "c")
            choose x in Connections satisfying
                  wants(x, _, _, _, self, "Loose", "c")
               Connections(x) := false;
               extend Connections with x1
                  party1(x1) := (self, "Loose", "c", true);
                  party2(x1) := other_part(x, self, "Loose", "c");
               endextend
            endchoose;
         endif
      else
         if seq1 = 1 then
            waiting("a_loose") := true;
            if not waiting("a_loose") then
               seq1 := 2;
            endif
         elseif seq1 = 2 then
            extend MSG with x
               target(x) := c;
               label(x) := self + "loosemsg";
               cont(x) := loosemsg;
            endextend;
            seq := 1;
         endif
      endif
end unit
```