Communicative Action Notation with Shared Storage

Martin Musicante[*]

UFPE - Departamento de Informatica
50.732-970 Cidade Universitária
Recife - PE - Brazil

mam@di.ufpe.br

Peter D. Mosses[†]

Computer Science Department
Aarhus University
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C - Denmark

pdmosses@daimi.aau.dk

August 1993

Abstract

A variation of the Communicative Action Notation is presented, in order to allow the sharing of storage among agents. Examples of use of the new notation are given by means of the description of semaphores, monitors and RPC mechanisms. The operational semantics of the new notation is also presented.

Introduction

Action Semantics [2] is a formal style of semantics definitions developed to provide "tractable" descriptions of real-life languages. Action Semantics descriptions resemble those written in denotational semantics [1], in the sense that equations defining semantic functions are given to state the meaning of each phrase of a language. Instead of the functional notation used in denotational semantics, a special notation was developed for use in action

[*]Partly funded by the Brazilian Science Research Council, CNPq, Grant 201.201/91-7.
[†]Partly funded by the Danish Science Research Council project DART (5.21.08.03).

semantics. This notation is called *Action Notation*, and it is used in action semantics descriptions very much in the way in which the λ-notation is used in denotational semantics.

In Action Semantics, the meaning of each phrase of a language is represented in terms of special entities called *actions*. Actions can be performed, with various possible outcomes: normal termination (*complete*), exceptional termination (*escape*), unsuccessful termination (*fail*) or non-termination (*diverge*). Action Notation provides some basic actions, and actions can be combined in order to obtain complex actions. In [2], combinators corresponding to many control patterns present in programming languages are provided, and further data may be specified *ad hoc*.

Together with action notation, a *Data Notation* is used to describe semantic entities in action semantics descriptions. A collection of algebraically defined data constructors is provided within the data notation, including numbers, characters, strings, sets, tuples, maps, etc.

There exists a third class of entities within the action semantics framework. These entities are called *Yielders*, and represent data, whose value depends on the current information managed by the enclosing action. Yielders can be *evaluated* to yield data. An example of a pre-defined yielder is current bindings, which yields the mapping from tokens to *bindable data* that represents the bindings received by the enclosing action.

Action Notation possesses five facets in which each primitive action and action combinator can behave.

Basic: This facet deals with pure control flow, without reference to information processing issues.

Functional: This facet deals with transient data, which is given to or by an action. For example, when the basic action give the given natural is given a natural number as transient data, it completes, giving the received natural number as a transient as well. The compound action A_1 then A_2 performs the action A_1 first. All transient data produced by A_1 is supplied to A_2, which is performed after A_1 completes.

Declarative: This facet deals with the manipulation of scoped information, represented by associations of *tokens* to *bindable data*. For example, the basic action bind "max-length" to 256 completes its performance, producing a binding of the token "max-length" to the natural number 256.

Imperative: Storage handling primitives are provided by this facet. A storage, in the action notation context, is a mapping from *cells* to storable data. For example, the action

```
| allocate a cell
then
| store 26 in the given cell .
```

allocates a new cell of the storage, and stores a value in it. This action combines features of both the functional and imperative facets. (Vertical bars are used to enforce the intended grouping of actions, as an alternative to parentheses.)

Communicative: This facet provides a system of *agents*, whose task is to perform actions. Agents can communicate using asynchronous message passing. Each agent has its own *communication buffer*, in which all the messages sent to the agent are placed. Communication is reliable, in the sense that no message can be lost during transmission; however, there is no bound to the amount of time taken for a message to arrive to its destination buffer, after transmission. Moreover, each agent is created with its own storage.

Encapsulation of actions as data is also provided within action notation. This feature gives a simple way to support the description of procedure and function abstractions in programming languages. For example, the performance of the action

```
give abstraction of | | allocate a cell
                    | then
                    | | store 26 in the given cell .
```

completes, giving an *abstraction* as a transient. An *abstraction* is an item of data which encapsulates an action. Abstractions can be *enacted*; this operation results in the performance of the encapsulated action. Both transients and bindings can be supplied to the enacted action.

For a more detailed description of action notation, the reader can refer to [2] or [5] (the former covers all the aspects related with the formal system; the latter does not cover the communicative facet nor the operational semantics of the notation).

A variation on the usual communicative action notation is presented in this work. Our purpose is to introduce storage sharing capabilities to the agents. The modifications proposed here make it possible to write more succinct and readable specifications of concurrent programming concepts than in the original notation, since it is no longer necessary to represent shared storage by auxiliary agents and communication protocols.

Our variation on the communicative action notation does not add any new primitive actions or combinators to the original action notation. It provides the notion of storage shared among several agents merely by introducing a new sort of contracts (the sharing ones), and the corresponding extension of the offer action. A couple of new yielders are introduced as well.

Then some conclusions are stated. An appendix provides the operational semantics of the entire extended action notation, and can be compared to [2, App. C]. Some examples of use of the new notation are given. The last section of this work provides the operational semantics of the new notation.

1 Communicative Action Notation

In what follows, the shared storage variation of the communicative action notation is presented. It includes the same set of actions present in the original communicative action notation. Some new yielders are introduced.

The reader may like to compare this section with [2, Chapter 9]. See also [2] for an explanation of the meta-notation used below for specifying functionalities of operations, etc.

1.1 Actions

The communicative action notation is based on four primitive actions, whose purpose is to provide the way of defining a system of agents performing actions. Agents can communicate which each other by means of a simple communication mechanism based on asynchronous message passing.

- send_ :: yielder → primitive-action .

- remove_ :: yielder → primitive-action .
- offer_ :: yielder → primitive-action .
- patiently_ :: action → primitive-action .

The primitive operation send Y takes a yielder Y that evaluates to a sort of messages (for instance a message[to a:agent][containing d:data]), specializes the sort to an individual message by determining the identification of the sending agent and a serial number, and sends it. The message is assumed to reach the communication buffer of its destination agent in a finite (but unbounded) time.

The primitive action remove Y takes an individual message (yielder) Y and eliminates the message from the current communication buffer, failing if it was not there.

The action combinator patiently A keeps on trying to perform the action A as long as it fails. Each performance of the argument action A is indivisible.

The action offer Y, where Y yields a sort of contract, agrees on the contract with an agent W. The agent will perform the action corresponding to an abstraction A. The usual forms of Y are:

a contract [to W][containing A]

a sharing contract [to W][containing A]

The first case corresponds to the original notation. The second case simply adds storage sharing capabilities to the newly contracted agent, which will adopt the storage of the contracting agent as its own.

1.2 Yielders

To the already existing current buffer, performing-agent and contracting-agent, yielders to get the set of sharing agents of the current storage and to get the next serial number of the agent configuration are added here.

- current buffer: yielder [of a list [of message]] .
- performing-agent, contracting-agent: yielder [of an agent] .

- next serial-number: yielder [of a natural] .
- sharing-agents: yielder [of a set [of agent]] .

The yielder sharing-agents evaluates to the set of those agents which share the storage together with the performing agent.

The yielder next serial-number yields the natural giving the serial number of the next communication of the agent. Actually, the addition of this yielder is orthogonal to our storage sharing modification to the notation, and it is only used to provide a cheap form of validation of a message, at the sender's side, as illustrated in section 2.1.1.

1.3 Data

Our only modification to the data related to the communicative facet in action notation is the inclusion of a new sort of contracts, the sharing ones. All the operations already defined on contracts are valid for sharing contracts.

Notice that the sorts of contracts and sharing contracts need to be disjoint, in order to avoid ambiguous semantics for the offer action.

- agent $\leq$ distinct-datum .
- user-agent : agent .
- buffer = list [message] .
- communication $\leq$ distinct-datum .
- communication = message | contract | sharing contract (*disjoint*) .
- sharing _ :: contract $\rightarrow$ sharing contract (*total*) .
- sendable = abstraction | agent |□ .
- contents _ :: message $\rightarrow$ sendable (*total*),
  contract $\rightarrow$ abstraction (*total*),
  sharing contract $\rightarrow$ abstraction (*total*) .

- sender _ :: communication → agent (*total*) .
- receiver _ :: communication → agent .
- serial _ :: communication → natural (*total*) .
- _ [containing _] :: message, sendable → message (*partial*),
  contract, abstraction → contract (*partial*),
  sharing contract, abstraction → sharing contract (*partial*) .
- _ [from _] :: communication, agent → communication (*partial*) .
- _ [to _] :: message, agent → message (*partial*),
  contract, agent → contract,
  sharing contract, agent → sharing contract
- _ [at _] :: communication, natural → communication (*partial*) .

1.4 Facets/Outcomes

- committing ≥ communicating .

1.5 Facets/Incomes

- income = □ | current buffer | next serial-number | □ .

2 Examples

Some examples of use of the new notation are now presented. First of all, we consider the description of a Sun-style, serial RPC mechanism. This first example does not exploit sharing. It shows that a description written in the original notation does not need any changes to be adapted to the new one. The serial RPC example helps also to understand the next one: the description of a Sun-style, concurrent RPC mechanism. A third example shows how to describe general semaphores within the new notation. A fourth, and last example shows how to represent monitors.

2.1 Serial RPC

A description of a Sun-style Remote Procedure Call (RPC) system using the modified communicative action notation is given in this section.

The RPC concept is a generalization of the procedure call concept found in most imperative programming languages. The main idea is that the called procedure is performed in a different environment than the calling one. The only communication between the caller and callee procedures is done by means of arguments and results.

The Sun RPC model, as described in [4], proposes the existence of *remote programs*, consisting of versions and procedures within these versions. A remote program is also called a *server*, and each procedure constitutes a *service*. Each program version can be *registered* using a *port mapper* server, present in each host, at a pre-defined address (*port*). The port mapper is also an RPC program.

For simplicity reasons, we omit the notion of version in the following description, since it does not add any theoretical difficulty, but imposes some syntactic complications.

In the serial RPC scheme, the invocation of the remote procedures is made in a serial manner, allowing only one procedure to be executed at a time, by a unique caller. The solution here is, in essence, the same as proposed in [3]. Since only one remote procedure within a program can be performed at a time, there is no need to use the storage sharing allowed by the notation.

As it is described in [4], a remote procedure is determined by giving the *port* in which the enclosing program and version can be localized, together with its own, unique identification within its program and version. Program (and version) names, as well as procedure names are represented in our description by means of syntactic tokens. The *port* notion, present in the Sun description, is represented by the *agent* notion in the action notation.

2.1.1 Actions

introduces: program-ID , service-ID , port , establish serial RPC with _ , rpc-call _ .

- program-ID $\leq$ token .

- service-ID $\leq$ token .

- port $\leq$ agent .

- establish serial-RPC with _ :: yielder[of map [service-ID to abstraction]] $\rightarrow$
  action [communicating] [giving a port] .

- rpc-call _ :: (yielder[of port], yielder[of service-ID], yielder[of data]) $\rightarrow$
  action [giving a tuple | diverging | communicating] [using next serial-number] .

privately introduces: rpc-reply _ , stub-action , perform service using _ .

- rpc-reply _ :: (yielder[of message], yielder[of tuple]) $\rightarrow$ action [communicating] .
- stub-action : action .
- perform service using _ ::yielder[of message[containing (service-ID, tuple)]] $\rightarrow$ action .

The performance of the action establish serial-RPC with M establishes an RPC server whose remote procedures are given by the map M. Each abstraction in the range of the map M, when enacted, can use a given value (the parameter to the remote call) and should give a value, the result, or escape. Note that, as it happens with the existing implementation, *ports* are allocated when the server is started. The agent identified with the service will perform the stub-action, explained below.

```
(1)  establish serial RPC with $M$:yielder =
       bind "services" to the map [token to abstraction] yielded by $M$ before
       | subordinate a port then
       | | give it and send a message[to it] [containing closure of abstraction of
       | | stub-action] .
```

The action rpc-call(A, S, V) is used in order to call the remote procedure identified with the service-ID S, at the remote program A. The tuple V represents the arguments to the remote procedure. The caller agent waits until the service is (eventually) done.

```
(2)  rpc-call ($A$:yielder[of agent], $S$:yielder[of service-ID, $V$:yielder[of tuple]) =
       | indivisibly
       | | give next serial-number and then
       | | send a message [to $A$] [containing ($S$, $V$)]
       then receive a message[from $A$] [containing (the given natural, a tuple)]
       then give rest of contents of the given message .

(3)  rpc-reply ($M$:yielder[of message], $V$:yielder[of tuple]) =
       send a message[containing (serial of $M$, $V$)] [to the sender of $M$] .
```

Notice that we are supposing the existence of the next serial-number yielder. This is a selector over the local state, giving the serial number of the next communication from the performing agent. It is a selector very much in the style of contracting-agent, which yields the agent which originated the performing one, or of current-storage, which yields to a map (from cells to values), representing a snapshot of the storage at the moment. The inclusion of this yielder allows a very simple way of message validation at the caller agent, without any complication to the operational semantics of the action notation.

The stub-action receives each remote procedure call, performs a simple validation and then serves it.

```
(4)  stub-action =
       unfolding
       | | receive a message[containing a (service-ID, tuple)] then
       | | | | check it is a valid call and then
       | | | | | give it and perform service using the given message
       | | | | then rpc-reply(the given message#1, the rest of the given tuple)
       | | | or
       | | | | check not (the given message is a valid call) and then
       | | | | rpc-reply (the given message, error "invalid remote call")
       | and then unfold .

(5)  perform service using $M$:yielder[of message] =
       | give the tuple yielded by the contents of $M$ then
       | enact application of the abstraction yielded by
       |         (the map bound to "services" at the given service-ID#1)
       |     to the rest of the given tuple
       trap give error "procedure failure" .
```

2.1.2 Yielders

- _ is a valid call :: yielder [of message] $\rightarrow$ yielder [of truth-value] .

(1) M:yielder is a valid call =
    component#1 of contents of M is in mapped-set
    (the map bound to "services") .

2.1.3 Data

introduces: error _ , error-value , void-value .

- error _ :: string $\rightarrow$ error-value ($total$) .

(1) error-value $\leq$ datum .

- void-value = ()
- token = string .

2.2 Concurrent RPC

An RPC scheme is presented here that permits the concurrent execution of several remote procedure calls. This example is based on the previous one, in the sense that there exist the notions of *program* and *procedure* within a program. The difference is that several remote procedure calls (possibly to the same procedure) can be served at the same time.

All the remote procedures share the same storage. Each procedure is described using a separate agent in the system. In the original version of communicative action notation [2], each agent has a separate storage, and shared storage can be represented by introducing auxiliary agents and communication protocols for allocating, storing in, and inspecting the local storage cells of these agents. A simpler description is obtained here, by using our extended Action Notation.

As in the previous example, we identify the notion of *port*, in the Sun RPC terminology, with the action semantics notion of *agent*. Once more, each remote procedure within a program can be identified using a syntactic token.

Figure 1 shows an example of a concurrent RPC system containing several remote procedures. The arrows exemplify a possible flow of the messages used to represent the RPC protocol.

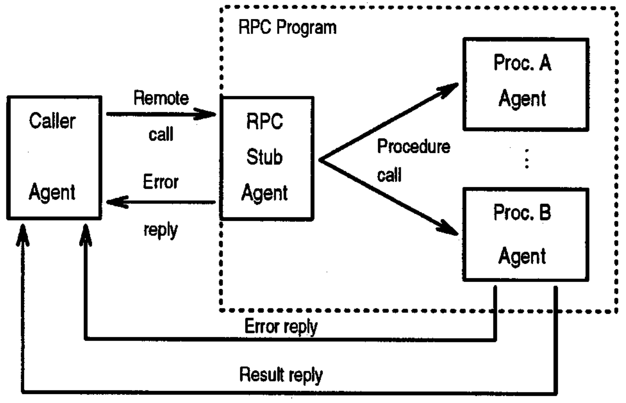

Figure 1: A client and a remote program with two procedures.

There exists an RPC *stub agent* (the one identified with the RPC program *port*). This agent will receive the remote procedure call messages from any caller agent and, after a validation, forward the call message to the corresponding procedure agent. In the case of failure of the validation, an error reply is sent to the caller agent.

Once the agent corresponding to a service receives a call message, it attempts to perform the required service and, then, sends a reply directly to the caller agent (which is blocked, waiting for the answer). The service can possibly yield an error state, in which case an error message is sent to the caller agent.

2.2.1 Actions

introduces: port , program-ID , service-ID , establish RPC service with _ , rpc-call _ .

- program-ID $\leq$ token .

- port $\leq$ agent .

- service-ID $\leq$ token .

- establish concurrent-RPC with :: yielder [of map [service-ID to abstraction] $\rightarrow$

  action [communicating] [giving a port] .

- rpc-call _ :: (yielder[of port], yielder[of service-ID], yielder[of data]) $\rightarrow$
  action [giving a tuple | diverging | communicating]
  [using next serial-number] .

privately introduces: rpc-reply _ , initiate sharing services of _ , server-stub-action, service-stub-action, perform service using _.

- rpc-reply _ :: (yielder[of message], yielder[of tuple]) $\rightarrow$ action [communicating] .

- server-stub-action : action

- initiatesharing services of ::
  map [service-ID to abstraction] $\rightarrow$ action [giving a map [service-ID to agent]] .

- service-stub-action: action[using a given abstraction | current bindings |
  current storage][communicating | binding | storing] .

- perform service using _ :: yielder[of message] $\rightarrow$ action .

Establishing an RPC service is done in the same way as in the serial case of the previous example.

```
(1)  establish concurrent-RPC with M:yielder[of a map] =
       subordinate a port then
       | give it and send a message[to the given port][containing application of
       |           closure abstraction of server-stub-action to M] .
```

An RPC call is performed by sending a call message to the agent representing the remote program and, then, waiting for an answer to the message. Note that, in this case, arbitrary calls are allowed, including recursive remote calls.

The use of the next serial-number yielder here gives an elegant and natural way of call validation at the caller's side. In this way, several calls can be made from the same agent without risk of confusion between the various results.

```
(2)  rpc-call ($A$, $S$, $V$) =
       | indivisibly
       | | give next serial-number and then send a message [to $A$] [containing ($S$, $V$)]
       then receive a message[from $A$ [containing (the given natural, a tuple)]
       then give rest of contents of the given message .
```

```
(3)  rpc-reply ($M$, $V$) =
       send a message[containing (serial of $M$, $V$)] [to the sender of $M$] .
```

The action that defines the remote program stub is a loop. It receives, validates and forwards remote call messages to the procedure agents, and sends an error reply message, in the case the call fails to validation.

As all the remote procedure agents share their storage with the stub agent, there is room in this action to introduce some global initializations. This was not considered here for simplicity reasons.

```
(4)  server-stub-action =
       | initiate sharing services of the given map [service-ID to abstraction] then
       | bind "services" to the given map [service-ID to agent]
       before
       | unfolding
       | | receive a message[containing a (service-ID, tuple)] then
       | | | | check (the given message is a valid call) and then
       | | | | | | give it and perform service using the given message
       | | | | | then rpc-reply(the given message#1, the rest of the given tuple)
       | | | or
       | | | | check not (the given message is a valid call) and then
       | | | | rpc-reply (the given message, error "invalid remote call")
       | | | and unfold .
```

```
(5)  perform service using $M$:yielder[of message] =
       give the map bound to "services" at component#1 of contents of $M$
       then send a message[containing $M$] [to the given agent] .
```

The initiate sharing services M action takes a map from each service-ID in the program to an abstraction, the body of the corresponding remote procedure. The action allocates a sharing agent for each service and gives a map from service-ID to the new contracted agents, as a transient.

```
(6)  initiate sharing services of empty-map = give empty-map .

(7)  initiate sharing services of disjoint-union ($m_1$:map, $m_2$:map) =
       | initiate sharing services of $m_1$ and
       | initiate sharing services of $m_2$
       then give disjoint-union of the given map$^2$ .

(8)  initiate sharing services of map ($T$: service-ID) to ($A$: abstraction) =
       | offer a sharing contract [to an agent][containing abstraction
       | of subordinate-action]
       and then
       | receive a message [containing an agent] then give the contents of it then
       | | give (map $T$ to the given agent) and
       | | send a message[to the given agent]
       | |        [containing application of closure abstraction of service-stub-
       | |        action to $A$] .
```

The following action is used in order to add the call attention and reply features to each abstraction denoting a service body. The service-stub-action receives the service abstraction as a transient. It implements an infinite loop (provided the service does not fail, or perhaps escape). The performance of the service is preceded by the reception of the call message, containing the arguments to the remote procedure. After the service is honored, a reply message is sent to the caller agent, containing the results of the procedure.

```
(9)  service-stub-action =
       bind "service-abstraction" to the given abstraction before
       unfolding
       | receive a message [from the contracting-agentl [containing a message] then
       | | give contents of the given message then
       | | | | regive and enact application of the abstraction bound to "service-
       | | | |    abstraction" to rest of contents of the given message
       | | | then rpc-reply (the given message#1, the rest of the given tuple)
       | | trap rpc-reply (the given message#1, error "procedure failure")
       | and unfold .
```

The following sections are similar to those of the previous examples, and do not deserve further explanation.

2.2.2 Data

introduces: error_ , error-value, void-value .

- error_ :: string → error-value (*total*) .

(1) error-value ≤ datum .

- void-value = () .

2.2.3 Yielders

- _ is a valid call :: message → yielder [of truth-value] .

(1) M is a valid call =
component#1 of contents of M is in mapped-set(the map bound to "services") .

2.3 Semaphores

General semaphore operations are provided in this section.
Operations P(_) and V(_) are supposed to be used by a number of agents, possibly sharing storage.

Though this example does not directly exploit the storage sharing allowed by the notation, it provides a discipline for its use.

2.3.1 Actions

introduces: semaphore, setup _ with value _ , P(_) , V(_) .

- semaphore $\leq$ agent .
- setup _ with value ::
  yielder[of semaphore], yielder[of natural] $\rightarrow$ action [giving a semaphore | communicating] .
- P(_) :: yielder [of semaphore] $\rightarrow$ action [communicating] .
- V(_) :: yielder [of semaphore] $\rightarrow$ action [communicating] [using current buffer] .

privately introduces: semaphore-action .

- semaphore-action: action [using a given natural | current storage | current bindings |
  current buffer] [communicating | storing | binding] .

The action setup S with value N is performed in order to create a semaphore of sort S. The natural number N represents the initial value of the semaphore. The action gives the individual semaphore agent as a transient.

```
(1)  setup S with value N =
       subordinate an (S & semaphore) then
       | give the given semaphore and
       | send a message [to the given semaphore]
       |   [containing application of closure of abstraction
       |   of semaphore-action to N] .
```

The P operation on the semaphore is described here by a simple send-receive pair of actions, while a V operation consists only on sending a V-message to the semaphore agent.

```
(2)  P(S) = give the semaphore yielded by S then
       | send a P-message [to the given semaphore] and then
       | receive a reply-message [from the given semaphore] .
```

(3) V(S) = send a V-message[to the semaphore yielded by S] .

The action describing the semaphore behavior consists of a main loop, within which P and V messages are received. A counter (the value of the semaphore) and a queue (of agents waiting for a positive value) are used, in a standard way.

(4) semaphore-action =

```
| | | give the given value and allocate a cell
| | then
| | | store the given value#1 in the given cell#2 and
| | | bind "value" to the given cell#2
| and
| | allocate a cell then
| | | store empty-queue in it and bind "waiting-queue" to it
before unfolding
| | | receive a P-message then
| | | | | check (the value stored in the cell bound to "value" is greater than 0)
| | | | | and then decrement "value" and then
| | | | | send a reply-message[to sender of the given message]
| | | | or
| | | | | check (the value stored in the cell bound to "value" is 0)
| | | | | and then enqueue the given message in "waiting-queue"
| | or
| | | receive a V-message and then
| | | | | check is-empty ("waiting-queue") and then increment "value"
| | | | or
| | | | | check not is-empty ("waiting-queue") and then
| | | | | | dequeue a message from "waiting-queue" then
| | | | | | send a reply-message[to sender of the given message]
| and then unfold .
```

2.3.2 Data

The data used in this example consists of messages (of three disjoint sorts), stored queues and counters.

Queues are represented by lists, and stored in cells bound to tokens. All operations on queues refer to the data being enqueued or dequeued, and the token bound to the cell where the queue is stored. A similar treatment is done to counters.

introduces: P-message , V-message , reply-message ,
queue , empty-queue , enqueue _ in _ , dequeue _ from _ ,
is-empty _ , increment _ , decrement _ .

(1) message = □ | P-message | V-message | reply-message (*disjoint*) .

- token = string .

2.3.2.1 Queue

- queue ≤ list .

(1) empty-queue = empty-list .

- enqueue _ in _ :: yielder, token → action [using current bindings] [storing] .

- dequeue _ from ::

  datum, token → action [using current bindings] [storing | giving a datum] .

- is-empty :: token → yielder [of a truth-value] .

(2) enqueue D in T =
  | give concatenation(the list stored in the cell bound to T, list of D)
  then store the given list in the cell bound to T .

(3) dequeue D in T =
  give (D & head of the list stored in the cell bound to T) and then
  | give tail of the list stored in the cell bound to T
  | then store it in the cell bound to T .

(4) is-empty T = the queue stored in the cell bound to T is empty-queue .

2.3.2.2 Counters

- increment _ ,
  decrement _ :: token → action [storing] [using current bindings | current storage] .

  (`partial`)

(1) increment T =
    give successor of the natural stored in the cell bound to T
    then store it in the cell bound to T .

(2) decrement T =
    give predecessor of the positive-integer stored in the cell bound to T
    then store it in the cell bound to T .

2.4 Monitors

This section is devoted to the formulation of actions establishing monitor primitives. The example presented here is, in many aspects, similar to the concurrent RPC description of section 2.2.

Figure 2 shows an example of a monitor containing two procedures. All the agents corresponding to the procedures of the monitor share the same storage. Aside from the procedure agents, two other agents are part of our monitor descriptions: an entry queue manager and a condition queues manager. As in the previous cases, the monitor will be accessible by referring to one of its component agents. In this case, the entry queue manager agent will play the role of the entry port.

As in the case of the concurrent RPC scheme, the figure shows a possible flow of messages among agents. The entry manager is similar to the stub agent of section 2.2. The only difference is that it needs the permission of the queues manager to accept each new call to a procedure within the monitor. Also the procedures here are similar to the services of section 2.2. The difference is that they can use the wait and signal operations, and they must send a notification to the queues manager when leaving the monitor. The queues manager is responsible for allowing at most one agent to be active within the monitor at a time.

The identification of the procedures of the monitor, as well as the condition queues is done using syntactic tokens.

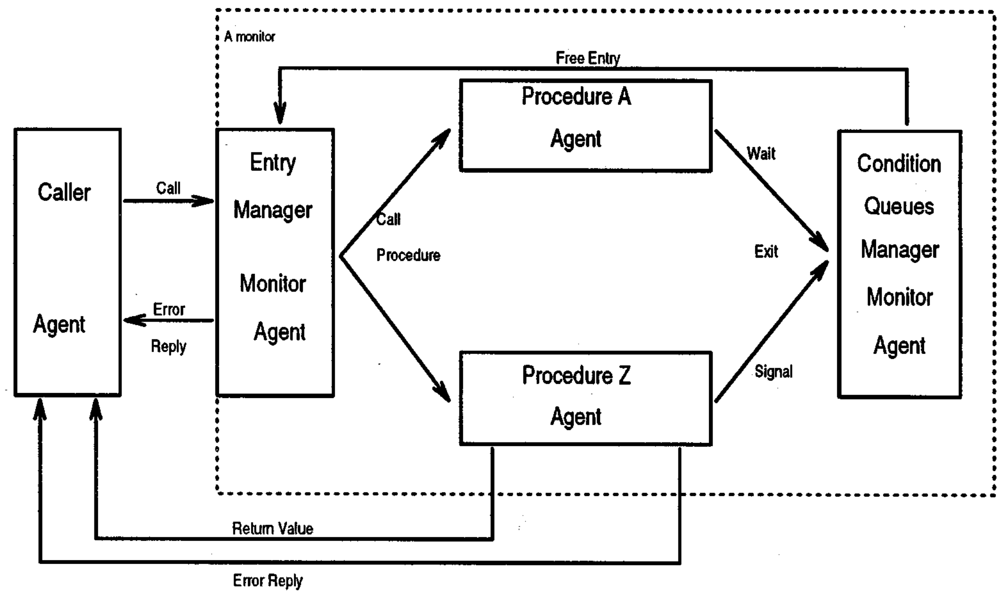

Figure 2: A client agent and a monitor.

introduces: monitor, establish a remote monitor with conditions _ and procedures _ , call , _ .wait , _ .signal .

- monitor < agent .

- establish a remote monitor with conditions _ and procedures _ ::
  yielder lof set[of token]], yielder [of map [token to abstraction]]
  → action [giving a monitor] [communicating] .

- call _ :: (yielder[of monitor], yielder[of token], yielder[of tupel]) →
  action [communicating] .

- _ .wait ,_ .signal ::
  (yielder[of token], → action [communicating] [using current
  buffer | current bindings] .

privately introduces: entry-manager-action , condition-manager-action , procedure-stub-action , initiate sharing procedures of _ , initiate condition queues using _ .

- entry-manager-action .
- condition-manager-action: action .
- procedure-stub-action: action .
- initiate sharing procedures of _ ::
  map Itoken to abstraction] $\rightarrow$ action Igiving a map [token to agent]] .
- initiate condition queues using _ ::
  set [of token] $\rightarrow$ action [storing | binding] [using current storage] .

Establishing a monitor is done in a way that is very similar to that of the previous examples.

```
(1)  establish a remote monitor with conditions (S:yielder[of set])
       and procedures (M:yielderlof map]) =
       subordinate an agent then
       | give the given agent and
       | send a message [to the given agent] [containing application of
       |     closure abstraction of entry-manager-action to
       |    (the set yielded by S, the map yielded by M)] .
```

The call, wait and signal operations are described by a similar pattern of actions as the rpc-call in section 2.2.

```
(2)  (call (M:yielder[of message], T:[of service-ID], V:yielder[of tuple]) =
       | indivisibly
       | | give next serial-number and
       | | send a call-message [to M] [containing (T, V)]
       then receive a reply-message[from M] [containing (the given natural, a tuple)]
       then give rest of contents of the given message .
```

```
(3)   $T$:yielder [of token] $\Rightarrow$
        $T$.wait =
          | indivisibly
          | | give next serial-number and
          | | send a message [to the agent bound to "condition-manager"]
          | | [containing ($T$)]
          then receive a reply-message[from the agent bound to "condition
          -manager"]
                [containing the given natural]

(4)   $T$:yielder [of token] $\Rightarrow$
        $T$.signal =
          | indivisibly
          | | give next serial-number and
          | | send a signal-message [to the agent bound to "condition-manager"]
          | |       [containing ($T$)]
          then receive a reply-message[from the agent bound to "condition
                -manager"][containing the given natural]
```

The entry manager, after the creation of the procedure of the monitor, enters in a loop in which it receives alternatively messages from the clients of the monitor and from the queues manager. No validation of the received messages is made in this case, but it can be added. Some initialization actions can be added to the entry manager, in the same way as discussed in section 2.2.

```
(5)  entry-manager-action =
       | | subordinate an agent then bound "condition-manager" to it
       | and
       | | initiate sharing procedures of the given map#2
       | | then bind "procedures" to the given (map [token to agent])
       before
       | | send a message [to the agent bound to "condition-manager"]
       | |      [containing application of closure abstraction of
       | |        condition-manager-action to the given set #1]
       | and then unfolding
       | | | receive a message[from any agent] [containing (token, data)] then
       | | | | send a call-message[containing the given message] [to the map bound
       | | | |    to "procedures" at component#1 of contents of the given message]
       | | | |
       | | and then
       | | | receive a free-way-message[from the agent bound to "condition-manager"]
       | | and then unfold
```

The condition queues manager action is the most complex action in this example. It behaves as a loop, in which three classes of messages are received. When the agent performing this action receives an exit-message, it means that a call is completed, and a new agent can became active within the monitor.

```
(6)  condition-manager-action =
       | bind "conditions" to the given set and
       | initiate condition queues union(the given set, set of "waiting-in-monitor")
       before unfolding
       | | | | receive an exit-message or
       | | | | | receive a waiting-message[containing a token] then
       | | | | | enqueue it in contents of it
       | | | and then
       | | | | | check (is-empty "waiting-in-monitor") and then
       | | | | | send a free-way-message[containing nothing] [to contracting
       | | | | | -agent]
       | | | | or
       | | | | | check not (is-empty "waiting-in-monitor") and then
       | | | | | dequeue a message from "waiting-in-monitor" then
       | | | | | send a reply-message [to sender of the given message] [containing
       | | | | | serial of it]
       | or
       | | receive a signal-message[containing a token] then
       | | | | check is-empty(contents of the given message) and then
       | | | | send a reply-message [to sender of the given message] [containing
       | | | | serial of it]
       | | | or
       | | | | check not (is-empty(contents of the given message)) and then
       | | | | enqueue the given message in "waiting-in-monitor" and then
       | | | | | dequeue a message from the token yielded by contents of the
       | | | | | given message then send a reply-message
       | | | | | [to the sender of the given message]
       | | | | |      [containing serial of it]
       and then unfold .
```

When a wait-message is received, the call is enqueued, and a new agent can became active in the monitor. When a signal-message is received, the signaling agent is enqueued in a queue of waiting agents within the monitor, and an agent of the signaled queue is permitted to became active in the monitor. If there are no agents pending in the signaled queue, the reception is acknowledged, and the signaling agent continues its performance. The condition queues manager uses the same queues actions as in the semaphores example, in order to implement the waiting queues.

The following actions are similar to those in section 2.2, and do not deserve any explanation here.

```
(7)  initiate sharing procedures of empty-map = give empty-map .

(8)  initiate sharing procedures of disjoint-union ($m_1$:map, $m_2$:map) =
       | initiate sharing procedures of $m_1$ and
       |  initiate sharing procedures of $m_2$
       then give disjoint-union of the given tuple .

(9)  initiate sharing procedures of map ($T$:token) to ($A$:abstraction) =
       | offer a sharing contract [to an agent][containing abstraction of subordinate-action]
       and then
       | receive a message [containing an agent] then
       | | | give map $T$ to contents of the given message
       | | and send a message[to contents of the given message]
       | |      [containing application of closure abstraction of procedure-stub-
       | |      action to $A$] .

(10) procedure-stub-action =
        bind "procedure-abstraction" to the given abstraction before
        unfolding
        | | receive a call-message [from the contracting-agent][containing a message] then
        | | give contents of the given message then
        | | | | | regive and enact application of the abstraction bound to
        | | | | |      "procedure-abstraction" to the rest of contents of the given message
        | | | | then send-reply (the given message#1, the rest of the given tuple)
        | | | trap send-reply (the given message#1, error "procedure failure" )
        | | then send an exit-message [to the agent bound to "condition-manager"
        | and then unfold .

(11) initiate condition queues using empty-set = complete .

(12) initiate condition queues using union ($s_1$:set[token], $s_2$:set[token]) =
        initiate condition queues using $s_1$ and
        initiate condition queues using $s_2$ .

(13) initiate condition queues using set of ($t$:token) =
allocate a cell then
| store empty-queue in it and bind $t$ to it .
```

2.4.1 Data

introduces: wait-message , signal-message , call-message , reply-message, free-way-message , error _ , error-value , void-value , queue , empty-queue , enqueue _ in _ , dequeue _ from _ , is-empty _ .

(1) message = □ | wait-message | signal-message | call-message | reply-message | free-way-message (*disjoint*) .

- token = string .
- error :: string → error-value (*total*) .

(2) error-value ≤ datum .

- void-value = () .

2.4.1.1 Queue

- queue ≤ list .

(1) empty-queue = empty-list .

- enqueue _ in _ :: datum, token → action [using current bindings] [storing].
- dequeue from :: datum, token → action [using current bindings] [storing | giving a datum] .
- is-empty :: token → yielder [of a truth-value] .

```
(2) enqueue $D$ in $T$ =
      | give concatenation(the list stored in the cell bound to $T$, list of $D$)
      then store the given list in the cell bound to $T$ .

(3) dequeue $D$ from $T$ =
      give $D$& head of the list stored in the cell bound to $T$ and then
      | give tail of the list stored in the cell bound to $T$
      | then store it in the cell bound to $T$ .
```

(4) is-empty T = the queue stored in the cell bound to T is empty-queue .

Conclusions

Our work extends the communicative facet of Action Notation, introducing direct sharing storage capabilities among agents. This capability simplifies the description of many concurrent programming concepts, for which the addition of complicated auxiliary actions, agents and data would be necessary, if using the original Action Notation [2].

The examples given in section 2 shows that the new primitives extend in an elegant way the old ones.

In the appendix, the operational semantics of the new notation is given. It is a modification of the operational semantics of the original notation in [2, Appendix C].

Acknowledgements

Are due to Silvio Meira and Giovanni Lucero Palma for reading and commenting on previous versions of this report. Also to the Brazilian Science Research Council (CNPq) and the Danish Science Research Council, project DART, for funding in part the development of this work.

References

[1] P.D.Mosses, *Denotational Semantics.* In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science,* Volume M, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[2] P.D.Mosses, *Action Semantics.* Cambridge University Press, Tracts in Theoretical Computer Science Series No 26, 1992.

[3] M.Musicante, *The Sun RPC Language Semantics*, In Proceedings of: PANEL'92, XVIII Latin-American Conference of Informatics, Las Palmas, Spain, September 1992.

[4] Sun Microsystems, *RPC: Remote Procedure Call Specification.* RFC 1050, 1988.

[5] D.A.Watt, *Programming Language Syntax and Semantics,* Prentice Hall International, 1991.

A Operational Semantics

This section corresponds to the operational semantics of the variation to the action notation we propose. This section is a modified version of [2, Appendix C], and gives the operational semantics of the whole action notation.

A.1 Abstract Syntax

The only modification to the abstract syntax given in [2] consists of adding the terminals sharing-agents and next serial-number to the Yielder rule.

(1) **closed except Data**
grammar:

A.1.1 Actions

(1) Action = Simple-Action | ⟦Action-Prefix Action⟧ |
⟦Action Action-infix Action D⟧ .

(2) Simple-Action = Constant-Action | ⟦Simple-Prefix Yielder⟧ |
⟦To-Prefix Yielder "to" Yielder ⟧ | ⟦"store" Yielder
"in" Yielder⟧ .

(3) Constant-Action = "complete" | "escape" | "fail" |"commit" | "unfold".

(4) Simple-Prefix = "give" | "choose" | "produce" | "unbind" |
"unstore" | "reserve" | "unreserve" |
"enact" | "send" | "remove" | "offer" |
"undirect" | "inderectly produce" .

(5) To-Prefix = "bind" | "indirectly bind" | "redirect" .

(6) Action-Prefix = "unfolding" | "indivisibly" | "patiently" .

(7) Action-infix = "or' | "and" | "and then" | "then" | "trap" |
"moreover" | "and then moreover" |
"then moreover" | "hence" | "thence" |
" before" |"then before" .

A.1.2 Yielders

(1) Yielder = Data-Constant | Abstraction | ⟦Data-Unary "(|" Yielder "|)"⟧ |
⟦Data-Binary a Yielder "(|" Yielder "," Yielder "|) ⟧ |

⟦"if" Yielder "then" Yielder "else" Yielder D⟧ |
⟦"the" Data "yielded by" Yielder⟧ | ⟦Yielder "receiving" Yielder⟧ |
"them" | "current bindings" | "current storage" | "current buffer" |
"current redirections" | "performing agent" | "contracting-agent" |
"sharing-agents" | "next serial-number" .

(2) Abstraction = Yielder | ⟦"abstraction of" Action⟧ | ⟦Abs-prefix Yielder⟧ |
⟦Action-Prefix Abstraction⟧ | ⟦Abstraction Action-infix
Abstraction⟧

A.1.3 Data

(1) Data = Data-Constant | ⟦Data-Unary "(|" Data "|)"⟧ |
⟦Data-Binary "(|" Data "," Data "|)"⟧ |
"if" Data "then" Data "else" Data⟧ | □

(2) Data-Constant = □ .

(3) Data-Unary = □ .

(4) Data-Binary = □ .

A.2 Semantic Entities

Several changes are made to the original semantic entities in [2, appendix C], in order to support shared storage among agents:

- Commitments are now represented as a pair of updates to the storage and list of communications, instead of the original list of communications.
- The storage entity is removed from the local information of an agent, and now appears as shared-info, as a part of the new configuration semantic entity.
- a new sort of semantic entities is introduced, to contain the storage updates performed by an agent.

includes: Data Notation .
includes: Action Notation/*/Data(Abstracting *for* abstraction) .

A.2.1 Acting

As our proposal does not introduce any new actions, the Acting semantic entities are not changed at all.

grammar:

(1) Acting $=$ Terminated | Intermediate .

(2) Terminated $=$ Completed | Escaped | Failed .

(3) Completed $=$ ⟨ "completed" data bindings ⟩ .

(4) Escaped $=$ ⟨ "escaped" data ⟩ .

(5) failed $=$ "failed" .

(6) Intermediate $=$ Simple-Action | ⟦Action-Prefix Acting⟧ |
⟦Acting Action-infix Acting⟧ |⟨Action data⟩|⟨Action bindings⟩|
⟨ Action data bindings⟩|⟨ "redirect" redirections⟩|
⟦Acting ("before" | "then before") Acting bindings ⟧

(7) Sequencing $=$ "and then" | "then" | "trap" |
"and then moreover" | "then moreover" |
"hence" | "thence" | "before" | "then before" .

(8) Interleaving $=$ "and" | "moreover" .

(9) Sequencing $=$ "and" | "and then" | "then" |
"moreover" | "and then moreover" | "then moreover" |
"hence" | "thence" | "before" | "then before" .

(10) Abstracting $=$ ⟦"abstraction of' Acting"⟧ .

A.2.2 Sort Unions

The sort-union of a tuple is the sort of all its components.

introduces: sort-union _ .

(1) sort-union _ :: data $\rightarrow$ datum .

(2) sort-union () $=$ nothing .

(3) sort-union (d:datum, x:data) $= d$ | sort-union x .

A.2.3 States

In order to made the same storage shared by several agents, a new component of the state and info semantic entities is introduced. The shared-info represents information that can be consulted and modified not only by the local agent, but possibly by several others. Each time an agent is being *advanced,* a copy of the current shared storage gets added to its state, as shared information. The storage is no more part of the local information of an agent.

introduces: state , local-info , shared-info , info .

(1) state = (Acting, local-info, shared-info) .
(2) local-info = (redirections, natural, buffer, agent, agent) .
(3) shared-info = (storage, set [of agent]) .
(4) info = (data, bindings, local-info, shared-info) .

A.2.4 Commitments

In the original version of action notation, a commitment is either uncommitted, or a list indicating that the agent performed a committing action. This list contains any communications that the action is emitting. An empty list indicates that the agent performed a committing action that does not require communication, like an storage action, or the commit primitive action.

In our modification, a commitment is uncommitted or a pair, containing a set (of cells) and a list (of communications). Each time an agent performs a committing action, a commitment will be constructed containing the set of cells modified by the agent and the list of the communications the agent performed, if any.

introduces: commitment , committing , committed , commitment of _ , uncommitted, blend _ , changed cells _ , items _ .

(1) commitment = committing | uncommitted (*disjoint*) .
(2) committing = (set[cell], list[of communication]) .
(3) committed = (empty-set, empty-list) .
(4) commitment of c:communication* = (empty-set, list of c) .
(5) commitment of s:set[cell] = (s, empty-list) .
(6) uncommitted: commitment .

(7) blend _ : commitment^2 $\rightarrow$ commitment (*unit is* uncommitted) .
(8) $c_1 = (s_1$:set[cell], l_1:list[of communication])
$c_2 = (s_2$:set[cell], l_2:list[of communication]) $\Rightarrow$

(1) blend(c_1, c_2) = (union(s_1, s_2), concatenation(l_1, l_2)) .

(9) $c = (s$:set[cell], I:list[of communication]) $\Rightarrow$

(1) changed cells $c = s$.
(2) items c = items l .

(10) updates of uncommitted = empty-set .
(11) items of uncommitted = empty-list .

A.2.5 Processing

A configuration is now representing a snapshot of the system of agents. A configuration entity is formed by the shared information at each instant, and a distributed system of agents, possibly using the shared information.

The initial configuration of a system with only one agent (the user-agent,) performing an action A, is also given.

introduces: processing, initial-processing _ , configuration , initial-configuration _ , agents _ , event , time , forever , _ delayed _ , undelayed _ , update of _ .

(1) (1) configuration = (shared-info$^+$, processing) .
(2) initial-configuration (A:Acting) =
(empty-map, set of user-agent, initial-processing A) .
(2) (1) processing = (event delayed time)*.
(2) initial-processing (A:Acting) =
undelayed (A, empty-map, 0, empty-list, user-agent, user-agent) .

In order to know the set of all of the present agents in the system, it is enough to search within the shared info, since every agent has its storage there, possibly shared.

(3) (1) agents _ :: shared-info* $\rightarrow$ set [of agent] .
(2) agents () = empty-set .
(3) initialagents (s:storage, a:set [of agent], i:shared-info*)
= union (a, agents i) .
(4) (1) event = communication | update | (Acting, local-info) .
(2) time = natural | forever .
(3) forever: time .
(4) _ delayed _ :: state, time $\rightarrow$ processing ($total$),
communication*, natural $\rightarrow$ processing,
update, 0 $\rightarrow$ processing ($total$) .
(5) (c:communication) delayed (n:natural): processing .
(6) (c_1:communication*, c_2:communication*) delayed ($t \leq$ natural) =
(c_1 delayed t, c_2 delayed t) .
(7) () delayed ($t \leq$ time) = () .
(8) undelayed e:event = e delayed 0 .

An update encapsulates the information needed to update the shared information of an

(5) update of _ :: (storage, set[cell], agent) $\rightarrow$ update .

A.3 Semantic Functions

Most of the equations of the original operational semantics were changed. Most of these changes are a simple relocation of the (now shared) storage, introducing the set of the sharing agents as well.

A few changes are significant. They deal with the operations related to updating shared storages and to start shared contracts. These modifications are explained in detail below.

A.3.1 Data

No changes here.

introduces: entity _ , unary-operation _ _ , binary-operation _ _ _ .

- entity :: Data $\rightarrow$ Data .

(1) entity $[\![O$:Data-Unary "$(\!|$" D:Data "$|\!)$"$]\!]$ = unary-operation O (entity D) .
(2) entity$[\![O$:Data-Unary "$(\!|$" D_1:Data "," D_2:Data"$|\!)$"$]\!]$ =
binary-operation O (entity D_1) (entity D_2) .
(3) entity $[\![$"if" D_1:Data "then" D_2:Data "else" D_3:Data$]\!]$ =
if (truth-value & entity D_1) then entity D_2 else entity D_3 .

- Unary-operation _ _ :: Data-Unary, Data $\rightarrow$ Data .
- Binary-operation _ _ _ :: Data-Binary, Data, Data $\rightarrow$ Data .

A.3.2 Yielders

The only changes in this section are the addition of equations (7)(8) and (7)(9).

introduces: evaluated

- evaluated _ :: (Yielder, info) $\rightarrow$ data .

(1) evaluated(Y:Data-constant, i :info) = entity Y .
(2) evaluated$[\![O$:Data-Unary, "$(\!|$" Y:Yielder "$|\!)$"$]\!]$, i:info) =
unary-operation O (evaluated (Y, i)) .
(3) evaluated$[\![O$:Data-Binary, "$(\!|$" Y_1:Yielder, Y_2:Yielder "$|\!)$"$]\!]$, i:info) =
binary-operation O (evaluated (Y_1, i)), (evaluated (Y_2, i)) .
(4) evaluated($[\![$"if" Y_1:Yielder "then" Y_2:Yielder "else" Y_3:Yielder$]\!]$, i:info) =
$[\![$"if" (truth-value & evaluated(Y_1, i)) then evaluated(Y_2, i) else evaluated(Y_3, i) .
(5) evaluated($[\![$"the" D:Data "yielded by" Y:Yielder$]\!]$, i:info) =
entity D & evaluated(Y, i)
(6) evaluated($[\![Y_1$ "receiving" Y_2:Yielder$]\!]$, d:data,
b:bindings, x:(local-info, shared-info)) =
evaluated(Y_1, d, bindings & evaluated(Y_2, d, b, x), x) .
(7) i = (d:data, b:bindings, r:redirections, n:natural, q:buffer, a_1:agent, a_2:agent,
s:storage, a:set[agent]) $\Rightarrow$

(1) evaluated("them", i:info) $= d$.
(2) evaluated("current storage", i:info) $= s$.
(3) evaluated("current bindings", i:info) $= b$.
(4) evaluated("current buffer", i:info) $= q$.
(5) evaluated("current redirections", i:info) $= r$.
(6) evaluated("performing-agent", i:info) $= a_1$.
(7) evaluated("contracting-agent", i:info) $= a_2$.
(8) evaluated("sharing-agents", i:info) $= a$.
(9) evaluated("next serial-number", i:info) $= n$.

(8) evaluated($[\![$"abstraction of" A:Action$]\!]$, i:info) $=$ $[\![$"abstraction of" $A]\!]$
(9) evaluated(Y, i) $= d$:data $\Rightarrow$
evaluated($[\![$"provision" Y:Yielder$]\!]$, i:info) $=$
$[\![$"abstraction of" "completed" d empty-map$]\!]$.
(10) evaluated(Y, i) $= b$:bindings $\Rightarrow$
evaluated($[\![$"production" Y:Yielder$]\!]$, i:info) $=$ $[\![$"abstraction of"
"completed" () b $]\!]$.
(11) evaluated(Y, i) $= r$:redirections $\Rightarrow$
evaluated($[\![$"indirect production" Y:Yielder$]\!]$, i:info) $=$ $[\![$"abstraction of"
"redirect" r$]\!]$.
(12) evaluated(Y, i) $=$ $[\![$"abstraction of" A:Acting$]\!]$ $\Rightarrow$
evaluated($[\![O$:Action-prefix Y:Yielder$]\!]$, i:info) $=$ $[\![$"abstraction of" $[\![O\ A]\!]$ $]\!]$.
(13) evaluated(Y_1, i) $=$ $[\![$"abstraction of" A_1:Acting$]\!]$;
evaluated(Y_2, i) $=$ $[\![$"abstraction of" A_2:Acting$]\!]$ $\Rightarrow$
evaluated($[\![Y_1$:Yielder O:Action-prefix Y_2:Yielder$]\!]$, i:info) $=$
$[\![$"abstraction of" $[\![\ A_1 O A_2]\!]$ $]\!]$

A.3.3 Actions

Except for A.3.3.1.4 and A.3.3.1.6, the changes mainly amount to adding the extra component s:shared-info, and removing the old local storage.

introduces: run _ , stepped _ .

(1) run _ :: state $\longrightarrow$ (Terminated, local-info, shared-info, commitment) .
(2) stepped (A, l, s) $\geq$ (A':Intermediate, l':local-info, s':shared-info, c':commitment);
run (A', l', s') $\geq$ (A'':Terminated, l'':local-info, s'':shared-info, c'':commitment)

$\Rightarrow$
run (A:Acting, l:local-info, s:shared-info) $\geq$ (A", l", s", blend(c", c')) .

(3) stepped (A, l, s) $\leq$ (A':Terminated, l':local-info, s':shared-info, c':commitment)
$\Rightarrow$
run (A:Acting, l:local-info, s:shared-info) $\geq$ (A', l', s', c') .

• stepped _ :: state $\rightarrow$ (state, commitment) .

(4) stepped (A:Terminated, l:local-info, s:shared-info) = nothing .

A.3.3.1 Simple

(1) i = (d:data, b:bindings, l:local-info); evaluated (Y, i) = nothing $\Rightarrow$
stepped ([[P:Simple-Prefix Y:Yielder]], i:info) =
stepped ([[P:To-Prefix Y:Yielder "to" Y_2:Yielder]], i:info) =
stepped ([[P:To-Prefix Y_1:Yielder "to" Y:Yielder]], i:info) =
stepped ([["store" Y:Yielder "in" Y_2:Yielder]], i:info) =
stepped ([["store" Y_1:Yielder "in" Y:Yielder]], i:info) =
("failed", l, uncommitted).

A.3.3.1.1 Basic

(1) stepped ("complete", d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), empty-map, l, s, uncommitted) .

(2) stepped ("escape", d:data, b:bindings, l:local-info, s:shared-info) =
("escaped", d, l, s, uncommitted) .

(3) stepped ("fail", d:data, b:bindings, l:local-info, s:shared-info) =
("failed", l, s, uncommitted) .

(4) stepped ("commit", d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), empty-map, l, s, committed) .

(5) stepped ("unfold", d:data, b:bindings, l:local-info, s:shared-info) = nothing .

A.3.3.1.2 Functional

(1) evaluated (Y, d, b, l, s) = d' : data $\Rightarrow$
stepped ([["give" Y:Yielder]], d:data, b:bindings, l:local-info, s:shared-info) =
("completed", d', empty-map, l, s, uncommitted) .

(2) evaluated (Y, d, b, l, s) $\geq$ d': data $\Rightarrow$
stepped ([["choose" Y:Yielder]], d:data, b:bindings, l:local-info, s:shared-info)
$\geq$ ("completed", d', empty-map, l, s, uncommitted) .

A.3.3.1.3 Declarative

```
(1)  evaluated ($Y_1$, $d$, $b$, $l$, $s$) = $t$ : token;
     evaluated ($Y_2$, $d$, $b$, $l$, $s$) = $v$ bindable ⇒
     stepped ([[“bind” $Y_1$:Yielder “to” $Y_2$:Yielder]], $d$:data, $b$:bindings, $x$!) =
              (“completed”, (), map $t$ to $v$, $x$, uncommitted) .
(2)  evaluated ($Y$, $d$, $b$, $l$, $s$) = $b'$: bindings ⇒
     stepped ([[“produce” $Y$:Yielder]], $d$:data, $b$:bindings, $l$:local-info,
     $s$ :shared-info) =
              (“completed”, (), $b'$, $l$, $s$, uncommitted) .
(3)  evaluated ($Y$, $d$, $b$, $l$, $s$) = $t$: token ⇒
     stepped ([[“unbind” $Y$:Yielder]], $d$:data, $b$:bindings, $l$:local-info,
              (“completed”, (), map $t$ to unknown, $l$, $s$, uncommitted) .
```

A.3.3.1.4 Imperative

A change of storage still causes a commitment, but the cells involved are now important, in order to determine the shared storage when several agents make changes at once.

```
(1)  evaluated ($Y_1$, $d$, $b$, $l$, $s$) = $v$ : storable;
     evaluated ($Y_2$, $d$, $b$, $l$, $s$) = $c$ : cell;
     $s$ = ($m$:storage, $a$:set[agent]);
     $s'$ = (overlay(map $c$ to $v$, $m$), $a$) ⇒
     stepped([[“store” $Y_1$:Yielder “in” $Y_2$:Yielder]], $d$:data, $b$:bindings, $l$:local-info, $s$) =
            if $c$ is in mapped-set of $m$
            then (“completed”, (), empty-map, $l$, $s'$, commitment of set($c$))
            else (“failed”, $l$, $s$, uncommitted) .
(2)  evaluated ($Y$, $d$, $b$, $l$, $s$) = $c$ : cell;
     $s$ = ($m$:storage, $a$:set[agent]);
     $s$' = (overlay(map $c$ to uninitialized, $m$), $a$) ⇒
     stepped ([[“unstore” Y:Yielder]], $d$:data, $b$:bindings, $l$:local-info, $s$:shared-info) =
            if $c$ is in mapped-set of $m$
            then (“completed”, (), empty-map, $l$, $s'$, commitment of set($c$))
            else (“failed”, $l$, $s$, uncommitted) .
(3)  evaluated ($Y$, $d$, $b$, $l$, $s$) = $c$ : cell;
     $s$ = ($m$:storage, $a$:set[agent]);
     $s$' = (disjoint-union(map $c$ to uninitialized, $m$), $a$) ⇒
     stepped ([[“reserve” $Y$:Yielder]], $d$:data, $b$:bindings, $l$:local-info, $s$:shared-info) =
            if not ($c$ is in mapped-set of $m$)
```

then (“completed”, (), empty-map, l, s', commitment of set(c))
else (“failed”, l, s, uncommitted) .

(4) evaluated (Y, d, b, l, s) = c : cell;
s = (m:storage, a:set[agent]);
s = ((m omitting set of c), a) $\Rightarrow$
stepped (⟦“unreserve” Y:Yielder⟧, d:data, b:bindings, l:local-info, s:shared-info) =
if c is in mapped-set of m
then (“completed”, (), empty-map, l, s', commitment of set(c))
else (“failed”, l, s, uncommitted) .

A.3.3.1.5 Reflective

(1) evaluated (Y, d, b, l, s) = ⟦“abstraction of” A:Acting⟧ $\Rightarrow$
evaluated (stepped (⟦“enact” Y:Yielder⟧, d:data, b:bindings, I:local-info,
s:shared-info) = (given (received (A,empty-map), ()), l, s, uncommitted) .

A.3.3.1.6 Communicative

The only important modification to this section is the addition of the last equation of the section, dealing with establishing the commitment corresponding to shared contracts. The added equation does not differ in essence from the original one for non-sharing contracts.

(1) evaluated (Y, d, b, l, s) = m: message; m [from a_1] [at n] = m’: message;
l = (r:redirections, n:natural, q:buffer, a_1:agent, a_2:agent);
l’ = (r, successor n, q, a_1, a_2) $\Rightarrow$
stepped(⟦“send” Y:Yielder⟧, d:data, b:bindings, l:local-info, s:shared-info) =
(“completed”, (), empty-map, l’, s, commitment of m’) .

(2) evaluated (Y, d, b, l, s) = m : message;
l = (r:redirections, n:natural, q:buffer, a_1:agent, a_2:agent);
l’ = (r, q omitting set of m, a_1, a_2) $\Rightarrow$
stepped(⟦“remove” Y:Yielder⟧, d:data, b:bindings, I:local-info, s:shared-info) =
if m is in set of items of q
then (“completed”, (), empty-map, l’, s, committed)
else (“failed”, l, s, uncommitted) .

(3) evaluated (Y, d, b, l, s) = $c \leq$ contract; c [from a_1][at n] = c’: contract;

l = (r:redirections, n:natural, q:buffer, a_1:agent, a_2:agent);
l' = (r, successor n, q, a_1, a_2) $\Rightarrow$
stepped($[\![$"offer" Y:Yielder$]\!]$, d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), empty-map, l', s, commitment of c') .

(4) evaluated (Y, d, b, l, s) = $c \leq$ shared contract; c [from a_1][at n] = c': shared contract;
l = (r:redirections, n:natural, q:buffer, a_1:agent, a_2:agent);
I' = (r, successor n, q, a_1, a_2) $\Rightarrow$
stepped($[\![$"offer" Y:Yielder$]\!]$, d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), empty-map, l', s, commitment of c') .

A.3.3.1.7 Directive

(1) evaluated (Y_1:Yielder, d, b, l, s) = t: token;
evaluated (Y_2:Yielder, d, b, l, s) = v: bindable | unknown;
i: indirection [not in mapped-set r];
l = (r:redirections, x!); l' = (overlay(map i to v, r), x) $\Rightarrow$
stepped($[\![$"indirectly bind" Y_1 "to" $Y_2]\!]$, d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), map, t to i, l', s, committed) .

(2) evaluated (Y_1:Yielder, d, b, l, s) = t: token;
evaluated (Y_2:Yielder, d, b, l, s) = v: bindable | unknown;
i = (b at t): indirection; l = (r:redirections, x!); l' = (overlay(map i to v, r), x) $\Rightarrow$
stepped($[\![$"indirectly bind" Y_1 "to" $Y_2]\!]$, d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), map, t to i, l', s, committed) .

(3) evaluated (Y, d, b, l, s) = t: token;
i = (b at t): indirection; l = (r:redirections, x!); l' = (r omitting set of i, x) $\Rightarrow$
stepped($[\![$"indirectly bind" Y:Yielder$]\!]$, d:data, b:bindings, I:local-info, s:shared-info) =
("completed", (), empty-map, l', s, committed) .

(4) evaluated (Y, d, b, l, s) = r': redirections;
l:local-info = (r:redirections, x!); I' = (overlay(r', r), x) $\Rightarrow$
stepped($[\![$"indirectly produce" Y:Yielder$]\!]$, d:data, b:bindings, l:local-info, s:shared-info) =
("completed", (), empty-map, l', s, committed) .

(5) l = (r:redirections, x!); l' = (overlay(r', r), x) ⇒
stepped (“redirect”, r':redirections, d:data, b:bindings, l:local-info,
s:shared-info) =
(“completed”, (), empty-map, l', s, committed) .

A.3.3.2 Compound

In what follows, the modifications to the stepped equations amount only to considering the new location of the storage, and the related information added with the shared-info. The equations corresponding to simplified, unfolded, given and received are not changed at all (except to correct some minor bugs in the original specification; a full list of corrigenda is available from Peter D. Mosses).

introduces: simplified _ , unfolded _ , given _ , received _ .

- simplified _ :: Acting → Acting .
- unfolded _ :: (Action, Action) → Action .
- given _ :: (Acting, data) → Acting .
- received _ :: (Acting, bindings) → Acting .

A.3.3.2.1 Stepping

(1) stepped(⟦“unfolding” A:Action⟧, d:data, b:bindings, l:local-info, s:shared
-info) = (given (received (unfolded (A, ⟦“unfolding” A⟧, b), d), l', s,
uncommitted;
(2) stepped(⟦“indivisibly” A:Acting⟧, I:local-info, s:shared-info) = run (A, l, s) .
(3) run (A, l) $\geq$ (“failed”, l':local-info, s':shared-info, c':commitment) ⇒
stepped (⟦“patiently” A:Acting⟧, l:local-info, s:shared-info) $\geq$
(⟦“patiently” A⟧, l', s', c') .
(4) run (A, l) $\geq$ (A':(Completed | Escaped), l':local-info, s':shared-info,
c':commitment) .
stepped (⟦“patiently” A:Acting⟧, l:local-info, s:shared-info) $\geq$ (A', l', s', c') .

(5) stepped (A_1, l, s) $\geq$ (A_1':Acting, l':local-info, s':shared-info,
c':commitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$Intermediate Sequencing Intermediate$]\!]$ |
$[\![$Intermediate Interleaving (Intermediate | Completed)$]\!]$ $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ simplified($[\![A'_1\ O\ A_2]\!]$,
l', s', c').

(6) stepped (A_2, l, s) $\geq$ (A_2':Acting, l':local-info, s':shared-info,
c':commitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$(Intermediate | Completed) Interleaving (Intermediate)]!] $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ simplified($[\![A_1\ O\ A'_2]\!]$,
l', s', c').

(7) stepped (A_1, l, s) $\geq$ (A_1':Acting, l':local-info, s':shared-info,
uncommitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$Intermediate "or" Interleaving]!] $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ simplified($[\![A'_1\ O\ A_2]\!]$,
l', s',uncommitment); .

(8) stepped (A_2, l, s) $\geq$ (A_2':Acting, l':local-info, s':shared-info,
uncommitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$Intermediate "or" Intermediate]!] $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ simplified($[\![A_1\ O\ A'_2]\!]$,
l', s',uncommitment); .

(9) stepped (A_1, l, s) $\geq$ (A_1':Acting, l':local-info, s':shared-info,
c':uncommitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$Intermediate "or" Intermediate]!] $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ (A'_1, l', s', c') .

(10) stepped (A_2, l, s) $\geq$ (A_2':Acting, l':local-info, s':shared-info,
c':uncommitment);
$[\![A_1\ O\ A_2]\!]$: $[\![$Intermediate "or" Intermediate]!] $\Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info, s:shared-info) $\geq$ (A'_2, l', s', c') .

A.3.3.3 Simplifying

No changes here.

(1) $[\![A'_1\ O\ A_2]\!]$: $[\![$Failed Sequencing Intermediate |
$[\![$(Failed | Escaped) Interleaving (Intermediate | Completed)$]\!]$ |
$[\![$Escaped Normal Intermediate$]\!]$ |
Completed "trap" Intermediate |

$[\![$(Completed | Escaped) "or" Intermediate$]\!] \Rightarrow$
simplified $[\![A'_1\ O\ A_2]\!] = A'_1$.

(2) $[\![A'_1\ O\ A_2]\!]$: $[\![$(Intermediate | Completed) Interleaving (Failed | Escaped)$]\!]$ |
$[\![$Intermediate "or" (Completed | Escaped)$]\!] \Rightarrow$
simplified $[\![A'_1\ O\ A_2]\!] = A'_2$.

(3) $[\![A'_1\ O\ A_2]\!]$: $[\![$(Intermediate | Completed) Interleaving (Failed | Escaped)$]\!]$ |
$[\![$Intermediate Action-infix Intermediate$]\!]$ |
Intermediate "or" (Completed | Escaped)$]\!] \Rightarrow$
simplified $[\![A'_1\ O\ A'_2]\!] = [\![A'_1\ O\ A'_2]\!]$.

(4) simplified $[\![$"failed" "or" A_2:Intermediate$]\!] = A_2$.

(5) simplified $[\![A_1$:Intermediate "or" "failed"$]\!] = A_1$.

(6) $[\![$"completed" d_1:data b_1:bindings "and" "completed" d_2:data b_2:bindings$]\!]$
= $\langle$"completed" $(d_1,\ d_2)$ (disjoint-union $(b_1,\ b_2))\rangle$.

(7) simplified $[\![A_1$:Completed "and then"A_2:Intermediate$]\!] = [\![A_1$ "and" $A_2]\!]$.

(8) simplified $[\![$"completed" d_1:data b_1:bindings "then" A_2:Intermediate$]\!]$ =
$[\![$"completed" () b_1 "and" (given $(A_2,\ d_1))]\!]$.

(9) simplified $[\![A_1$:Completed "and then"A_2:Intermediate$]\!]$ = given $(A_2,\ d_1)$.

(10) simplified $[\![$"completed" d_1:data b_1:bindings "moreover" "completed" d_2:data
b_2:bindings$]\!]$ =
$\langle$"completed" $(d_1,\ d_2)$ (overlay $(b_1,\ b_2))\rangle$.

(11) simplified $[\![A_1$:Completed "and then moreover"A_2:Intermediate$]\!] = [\![A_1$
"moreover" $A_2]\!]$.

(12) simplified $[\![$"completed" d_1:data b_1:bindings "then moreover" A_2:Intermediate$]\!]$
= $[\![$"completed" () b_1 "moreover" (given $(A_2,\ d_1))]\!]$.

(13) simplified $[\![$"completed" d_1:data b_1:bindings "hence" A_2:Intermediate$]\!]$ =
$[\![$"completed" d_1 empty-map "and" (received $(A_2,\ b_1))]\!]$.

(14) simplified $[\![$"completed" d_1:data b_1:bindings "thence" A_2:Intermediate$]\!]$ =
given (received $(A_2,\ b_1)$ d_1) .

(15) simplified $[\![$"completed" d_1:data b_1:bindings "before" A_2:Intermediate b:bindings$]\!]$
= $[\![$"completed" d_1 b_1 "moreover" (received $(A_2$, overlay $(b_1\ ,\ b)))]\!]$.

(16) simplified $[\![$"completed" d_1:data b_1:bindings "then before" A_2:Intermediate
b:bindings$]\!]$ =
$[\![$"completed" () b_1 "moreover" (given (received $(A_2$, overlay $(b_1\ ,\ b))$, $d_1))]\!]$.

A.3.3.4 Unfolding

No changes here.

(1) $(A_1$:Simple-Action, A_0:Action$)$ = if A_1 is "unfold" then A_0 else A_1 .
(2) unfolded (O:Action-Prefix A_1:Action⟧, A_0:action) =
if O is "unfolding" then ⟦O A_1⟧ else ⟦O (unfolded (A_1, A_0))
(3) unfolded (⟦A_1:Action O:Action-Infix A_2:Action⟧, A_0:action) =
⟦(unfolded (A_1, A_0)) O (unfolded (A_2, A_0))⟧ .

A.3.3.5 Giving

No changes here.

(1) given (A:Terminated, d:data) = A .
(2) A: Simple-Action | ⟦"unfolding" Action⟧ | ⟦"redirect" redirections⟧ ⇒
given (A, d:data) = (A, d);
given (A, b:bindings, d:data) = (A, d, b) .
(3) O: "indivisibly" | "patiently" ⇒
given (⟦O A:Acting⟧, d:data) = ⟦O (given (A, d))⟧ .
(4) O: "or" | "and" | "moreover" | "and then moreover" |"hence" | "before"
⇒ given (⟦A_1:Acting O A_2:Acting⟧, d:data) =
⟦(given (A_1, d)) O (given (A_2, d))⟧
(5) O: "then" | "trap" | "then moreover" | "thence" | "then before" ⇒
given (⟦A_1:Acting O A_2:Acting⟧, d:data) = ⟦(given (A_1, d)) O A_2⟧ .

A.3.3.6 Receiving

No changes here.

(1) received (A:Terminated, b:bindings) = A .
(2) A: Simple-Action | ⟦"unfolding" Action⟧ | ⟦"redirect" redirections⟧ ⇒
received (A, b:bindings) = (A, b);
received (A, b:bindings, d:data) = (A, d, b) .
(3) O: "indivisibly" | "patiently" ⇒
received (⟦O A:Acting⟧, b:bindings) = ⟦O (received (A, b))⟧ .
(4) O: "or" | "and" | "and then" | "then" |"trap"
| "moreover" | "and then moreover"
| "then moreover" ⇒
received (⟦A_1:Acting O A_2:Acting⟧, b:bindings) = ⟦(received (A_1, b))
O (received (A_2, b))⟧
(5) O: "hence" | "thence" ⇒

received ($[\![A_1$:Acting O A_2:Acting$]\!]$, b:bindings) = $[\![$(received (A_1, b)) O $A_2]\!]$.

(6) O: "before" | "then before" $\Rightarrow$
received ($[\![A_1$:Acting O A_2:Acting$]\!]$, b:bindings) = $[\![$(received (A_1, b)) O A_2 $b]\!]$.

A.3.4 Processes

The concluded equations were modified, in order to take a configuration as argument, and not simply a processing[1].

introduces: conclusion , concluded _ , advanced _ .

(1) conclusion = "completed" | "escaped" | "failed" .
• concluded _ :: configuration $\rightarrow$ conclusion .

(2) s = (c: conclusion, x!, user-agent, user-agent) $\Rightarrow$
concluded(s:shared-info$^+$, undelayed s:state, p:processing) $\geq c$.

(3) advanced $p \geq p'$:configuration;
concluded $p' \geq c$:conclusion $\Rightarrow$
concluded p:configuration $\geq c$.

The core of our modifications appear in the equations corresponding to the advanced semantic functions. In the original operational semantics of action notation, each processing semantic entity is advanced by performing all the undelayed communications first, followed by the *stepping* of each agent, when no more undelayed communications are present.

Our modification consists mainly of the introduction of an intermediate step, after advancing the undelayed communications and before stepping each agent. The purpose of this step is to update the global shared storages before advancing each agent.

• advanced _ :: configuration $\rightarrow$ configuration .

The order of events is not relevant.

(4) s: shared-info$^+$; p_1, p_2: processing; e_1, e_2: (event delayed time) $\Rightarrow$
advanced(s, p_1, e_1, e_2, p_2) = advanced(s, p_1, e_2, e_1, p_2)

[1]Recall that a configuration is formed by shared information together with a processing entity.

Advancing delayed communications.

(5) p = (c:communication delayed t:positive-integer, p':processing);
s, s': shared-info$^+$;
(s', p'':processing) $\leq$ advanced(s, p') $\Rightarrow$
advanced(s, p) $\geq$ (s', c delayed predecessor t, p'') .

Delivering undelayed messages.

(6) m: message[to a_1];
e = (x!, q:buffer, a_1:agent, a_2:agent); e' = (x, list of(items q, m),
a_1:agent, a_2:agent); $\Rightarrow$
advanced(s:shared-info$^+$, undelayed m, e:state delayed t:time,
p:processing) =
advanced(s, e' delayed t, p) .

(7) m:message [to agent[not in set of agents of s]] $\Rightarrow$
advanced(s:shared-info$^+$, undelayed m, p:processing) = advanced(s, p)

Initiating the execution of contracts.

(8) c:contract [to $a \leq$ agent][from a_2:agent][containing ⟦“abstraction of” A:Acting⟧];
s: shared-info$^+$;
a_1: a[not in agents of s];
s' = (s, empty-map, set of a_1);
e = (given (received(A, empty-map), ()), empty-map, 0, empty-list, a_1, a_2);
advanced(s, undelayed c, p:processing) $\geq$ (advanced(s', p), undelayed e) .

(9) c:sharing contract [to $a \leq$ agent][from a_2:agent]
[containing ⟦“abstraction of” A:Acting⟧];
s = (s_1:shared-info*, (m:storage, a':set[agent]), s_2:shared-info*);
a_2 is in a' = true;
a_1: a[not in agents of s];
e = (given (received(A, empty-map), ()), empty-map, 0, empty-list, a_1, a_2);

$s' = (s_1$, m, union(a', set of a_1), $s_2) \Rightarrow$
advanced(s, undelayed c, p) $\geq$ (advanced(s', p:processing), undelayed e) .

(10) s: shared-info$^+$;
c: contract[to $a \leq$ agent]
a [not in agents of s] = nothing $\Rightarrow$
advanced(s, undelayed c, p:processing) $\geq$ (advanced(s, p), undelayed c) .

(11) s: shared-info$^+$;
c: sharing contract[to $a \leq$ agent]
a [not in agents of s] = nothing $\Rightarrow$
advanced(s, undelayed c, p:processing) $\geq$ (advanced(s, p), undelayed c) .

Updating storages.

(12) sort-union p *&* undelayed communication = nothing;
$s = (s_1$: shared-info*, (m:storage, a:set[agent],) s_2:shared-info*);
a_1 is in a = true;
m_2 = disjoint-union (m omitting u, m_1 restricted to u); $\Rightarrow$
advanced (s:shared-info$^+$, update of (m_1:storage, u:set[cell], a_1:agent),
p:processing) =
    advanced (s_1, m_2, a, s_2, p)

Advancing delayed states.

(13) sort-union p *&* undelayed (communication | update) = nothing;
s: shared-info*; $\Rightarrow$
advanced(s, e:state delayed t:time, p:processing) $\geq$
    advanced (s, p), e delayed predecessor t

(14) sort-union p *&* undelayed (communication | update) = nothing;
s: shared-info$^+$; $\Rightarrow$
advanced(s, e:state delayed forever, p:processing) $\geq$ (advanced(s, p),
e delayed forever) .

Advancing to intermediate states.

```
(15)  sort-union $p$ & undelayed (communication | update) = nothing;
      $s$ = ($s_1$:shared-info$^*$, ($m$:storage, $a$:set[agent]), $s_2$:shared-info$^*$);
      $a_1$ is in $a$ = true;
      $e$ = ($x$!, $a_1$:agent, $a_2$:agent);
      stepped ($e$, $m$, $a$) $\geq$ ($e$':(Intermediate, local-info), $m$':storage,
      $a$':set [of agent], $c$:commitment)
      $\Rightarrow$
      advanced($s$, undelayed $e$:state, $p$:processing) $\geq$
        (advanced($s$, undelayed update of ($m$', changed cells of $c$, $a_1$),
        items of $c$ delayed natural, $p$),
        $e$' delayed natural) .
```

Advancing to terminating states.

```
(16)  sort-union $p$ & undelayed (communication | update) = nothing;
      $s$ = ($s_1$:shared-info$^*$, ($m$:storage, $a$:set[agent]), $s_2$:shared-info$^*$);
      $a_1$ is in $a$ = true;
      $e$ = ($x$!, $a_1$:agent, $a_2$:agent);
      stepped ($e$, $m$, $a$) $\geq$ ($e$':(Tntermediate, local-info), $m$':storage,
          $a$':set [of agent], $c$:commitment)
      $\Rightarrow$
      advanced($s$, undelayed $e$:state, $p$:processing) $\geq$
        (advanced($s$, undelayed update of ($m$', changed cells of $c$, $a_1$),
        items of$c$ delayed natural, $p$),
        $e$' delayed forever) .
```

Contents