Animating $\mathrm{CSP_M}$ Using Action Semantics

by

Leonardo Freitas, Ana Cavalcanti, Hermano Moura

{ljsf,alcc,hermano}@cin.ufpe.br

Informatics Center Federal University of Pernambuco,

Av. Luis Freire s/n CDU, 50.740-540, Recife, Brazil, (81)3271-8430 R.4018

August 30, 2001

Abstract

$\mathrm{CSP_M}$ is a language used to model concurrent and parallel computer systems formally. This paper presents an implementation of a significant part of the operational semantics of $\mathrm{CSP_M}$ using action semantics. This work is a starting point for the development of a formal animator using action semantics engines, compilers, or interpreters like Abaco or Ani, and of a Java library that implements the CSP operators.

Keywords: Action semantics, $\mathrm{CSP_M}$, concurrency, animation.

1 Introduction

The use of abstract and mathematical notation for the specification of concurrent and parallel primitives is a challenging task. Action Semantics [11] covers some very desirable properties for this kind of job: readability, modularity, abstraction, comparability, and concurrency.

Communicating Sequential Processes (CSP) has been used to design concurrent systems based on a formal mathematical theory [17, 9]. Its main objective is to define the dynamic behaviour of concurrent processes. This is modeled using the concept of communication between processes. $\mathrm{CSP_M}$ [18] is the machine readable version of CSP. It was implemented by tools like FDR [3] and PROBE [4], and extends CSP with a subset of a functional language that is used to construct auxiliary expressions and functions.

An Action Semantics description for the implementation of $\mathrm{CSP_M}$ operators can be used as a starting point for the formal development of animators or library implementations of $\mathrm{CSP_M}$. There exist some libraries [16, 7] and an animator [4] for $\mathrm{CSP_M}$. The libraries, however, were not created using formal descriptions. Freitas [6] is building a Java library that implements CSP primitives. The description of $\mathrm{CSP_M}$ presented here will be a guide for that work.

As mentioned in [11], the communicative facet of Action Notation makes available a basic notation to specify concurrency. Some important features like synchronization mechanisms and resource sharing must be implemented on the top of this basic notation. This work also provides some important primitive communicative actions, as an extension library, that can be used to model other languages.

Another contribution of this work is a case study on the use of the communicative facet. As far as we know, only the examples reported in [2, 12, 1] are available. The authors of [2] and [12] give insightful views of the use of Action Semantics to model concurrency, but they do not treat them generically nor they support processing of distributed agents; they rely on centralized agents.

In Section 2 we briefly describe Action Semantics, presenting examples of its notation. Next, in Section 3, we define CSP types, channels, processes, and operators. After that, in Section 4, we present a fragment of the full Action Semantics for CSP [5] discussing only the main contributions. In sequence, we present the abstract syntax of $\mathrm{CSP_M}$, some functional constructs, the processes representation, the *animator protocol actions*, and some primitive concurrent actions. Finally, in Section 5 we present the conclusions and an overview of future and related works.

2 Action Semantics

Action Semantics has many desirable properties that formalisms for specifying languages should have [21]. Action Notation [19, 11], the formal notation used in Action Semantics, provides control flow as well as data flow between actions. This allow us to specify programming language concepts like expressions, commands, declarations, concurrent agents, etc. There are five basic entities that comprise Action Notation: (i) Actions: entities that can be executed processing information, like a piece of a program; (ii) Yielders: expressions that can be evaluated during action execution; (iii) Sorts: define data types with some operational functions (Data Notation); (iv) Agents: entities that encapsulate the execution of actions, like a thread; and (v) Abstractions: a sort of data which encapsulates an action.

An action has facets that deals with particular modes of data flow. There are several facets: basic, functional, declarative, imperative, communicative, reflexive, directive, and hybrid; they are described in [11, 19]. Below we briefly detail those used in this paper.

- Basic facet: processes information independently (just control flow).
- Functional facet: data are called transients and are not available to the subsequent action.
- Declarative facet: an action may produce bindings that are visible to (scoped) sub actions(s) of the action.
- Imperative facet: an action may place data in storage cells, which may be inspected by subsequent actions. Storage is stable (visible to all actions).
- Communicative facet: an action may place a message (any kind of data including action abstractions) in the buffer of an action agent. The buffer is permanent, so it is concerned with the processing of communication between agents. All actions executed by an agent can access the buffer, but it is inaccessible to other agents.

There are, in Action Semantics, primitive actions and action combinators specifically built to deal with concurrency issues. The framework provides a bus for asynchronous message passing, asynchronous remote action invocation between action agents, and also a permanent buffer that can be inspected by any action of the performing agent. An agent acts like a lightweight process (or thread) of execution that behaves as a separate and independent running machine (CPU).

An action can be single-faceted (primitive action) or multi-faceted (composite action). The former only affects one kind of data (either transients, bindings, storage, or buffer);

the latter can compose each facet effect to produce complex and compounded actions. This composition is made by action combinators. There exist action combinators for each kind of the facet. Below we show some examples of actions.

- `bind ''x'' to 10`: produces the binding of token "x" to the value 10.
- `store true in cell1`: stores the value true in the memory location cell1.
- `send a message to agent1 containing the value bound to ''x''`: sends a message to agent1 from the performing-agent containing the value bound to the token "x" (performing-agent is a predefined variable that represent "this" agent).

An Action Semantics description for a programming language is a unified algebraic specification divided in the following modules: (i) Abstract Syntax; (ii) Semantic Functions: describe the mapping from the abstract syntax tree (AST) of programs to their meaning, using Action Notation; and (iii) Semantic Entities: define the data types used by the language, and auxiliary sorts and combinators used by the description in the previous module. For a detailed description of Action Semantics see [11, 19].

3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) can be viewed in two different ways: (i) a notation for describing concurrent systems; (ii) a mathematical theory to study processes which interact with each other and their environment by means of communications [17].

The most fundamental concept of CSP is a communication event. These events are assumed to be drawn from a set Σ (EVENTS in $\mathrm{CSP_M}$) which contains all possible communications for processes in the universe under consideration. A communication can be viewed as an indivisible transaction or synchronization between two or more processes. The fundamental assumptions about communications in CSP [17] are: (i) they are instantaneous; (ii) they only occur when all its participants (the processes and its environment) allow them.

In $\mathrm{CSP_M}$ you can define types and channels. The channels are the wires that allow values to be communicated between processes. For example, considering the dining philosophers example [17], type and channels can be declared as follows:

```
nametype FORKSPHILS = {0..2}.{0..2}
channel a : Int, channel b : FORKSPHILS
```

This allows channel `a` to communicate any integer value and channel `b` to communicate pairs: elements of the cartesian product {`0` ... `2`}`x`{`0` ... `2`} which is represented {`0` ... `2`}.{`0` ... `2`}. The input of a value `x` through channel `a` is written as `a`?`x`. Similarly, output of the fork 0 for the philosopher 1 through channel `b` is written as `b`!`0.1`.

The main unit under consideration in CSP is a process. The alphabet of a process P (αP) is the set of all events this process can communicate. The "STOP" process represents a broken machine (i.e. a machine that was not able to communicate), and the "SKIP" process represents a successfully terminated process. Processes can be defined using the $\mathrm{CSP_M}$ operators. Below we give a brief description of some of the $\mathrm{CSP_M}$ operators.

- Prefix (`->`) - Given an event e in Σ, the process e `->` P is initially willing to communicate e, and then behaves like P.

- External Choice (`[]`) - The process P `[]` Q offers to the environment the opportunity to communicate the initials of either P or Q. By initials we mean the set of events in αP and αQ that can be communicated immediately.
- Internal Choice (`|~|`) - The process P `|~|` Q does not offer to the environment any opportunity to choose any communication. It communicates the initials of either P or Q internally.
- Parallelism (`[|`X`|]`) - The process P `[|`X`|]` Q executes P and Q in parallel, but they must synchronize on the events in the set X, interleaving otherwise.

In the literature we can find many works describing CSP [8, 9, 17], and $\mathrm{CSP_M}$ [18, 3].

4 $\mathrm{CSP_M}$ Action Semantics

In this section we give the action definitions for $\mathrm{CSP_M}$ that are central to the construction of the animator. The complete work can be found in [5].

4.1 Abstract Syntax

Types in $\mathrm{CSP_M}$ can be either single or composite, where the composite types represents the cartesian product of single types. Types representing intervals of a discrete type may be defined as well. Since we are not covering all possible type we leave the definition open (with the symbol □).

- Type = ⟦ Type ("." Type)* ⟧ | ⟦ "{" Constant "..." Constant "}" ⟧ |
  "Int" | "Bool" | ⟦ "{" Identifiers "}" ⟧ | □ .

In $\mathrm{CSP_M}$ we can declare types to be used in channel declarations; channels to be used in processes declarations; and processes to describe the problem domain.

- Declaration = ⟦ "nametype" Identifier "=" Type ⟧ |
  ⟦ "channel" Identifiers (":" Type)? ⟧ |
  Process-Declaration | □ .

A process declaration gives the name of the process and its definition: a process.

- Process = Identifier | ⟦ "(" Expression & Process ")" ⟧ |
  ⟦ Identifier ("?" | "!" | ".") Expression "->" Process ⟧ |
  ⟦ Process ProcOp Process ⟧ | □ .

In the definition of a process, we can refer to other processes, and use guards, prefixing, and the CSP operators presented in Section 3. Since $\mathrm{CSP_M}$ is a functional language we can use functional expressions in guards and communications.

The description of $\mathrm{CSP_M}$ involves well-known functional descriptions [20]. Other important semantic function definitions like channels, types, and expressions are omitted here (but can be found in [5]).

4.2 Process Representation

Here, CSP processes are represented using Action Notation agents [11]. Agents abstract a machine that executes actions, like a thread that runs in a CPU. We extend the default agents (user-agent), calling them process-agents, to have a process status associated with them, and define semantic functions to alter their status and create new agents. This status is used for synchronization purposes.

More specifically, a process is represented by a tuple.

(1) process = (process-agent^2, process^2, cell^2) .

It contains two agents: the executor and the environment. The former executes the CSP operators, and the latter interacts with it.

Respectively, the process tuple contains two other processes: the left and the right operands. The operational execution of the parallelism uses this structure. In a process $R = P \,[|\{a\}|]\, Q$, the left and right of R are P and Q, respectively, and the environment of P and Q is R. In this way we link the processes and can abstract the synchronization mechanism.

More details for sequential execution can be found in Section 4.5. For this, the left and right processes have the special value unknown. For the topmost process, the environment is a special kind of agent presented in Section 4.4. The environment agent field is used to link the process network.

The tuple also has two cells that abstracts the process representation: the first records the process LTS (labeled transition system) [10] and the second its walk history. The main motivation to represent a process as an LTS, instead of an action, is the work in [18]. This is an operational semantics for CSP_M that defines the behaviour of processes as an LTS. We define semantic functions that create a process and access some of its fields.

4.3 Primitive Communicative Actions

Below we describe some primitive communicative actions used in the synchronization protocol. These actions are used in our specification, but they are sufficiently generic to become an extension for the Action Notation communicative facet.

The action wait[for _][on _] is used for synchronization of the current (performing) agent. First it uses the put _ [in _ status] action to set the status of the current agent to WAIT. Next it uses the receive primitive action that blocks until the expected message for the current agent arrives in the buffer. Finally, it uses the patiently check action to wait for the sending agent to set the status of the current agent to ACTIVE, and returns the received message. We use status flags, like WAIT and ACTIVE, to have a detailed control over synchronization.

- wait [for _][on _] :: yielder[of process-agent], yielder[of event$^+$] $\rightarrow$
  action[giving contents of a message | completing | diverging | communicating]
  [using current buffer] (*total*, *restricted*).

(1) wait [for x][on s] =
  | put the performing-agent [in WAIT status]
  and
  | receive a message[from x][containing s]
  then

| | patiently check(the status of the performing-agent is ACTIVE)
| and then
| | give the contents of the given message .

In sequence we have an action that offers a set of events s to be performed by an agent p. This action has the precondition that the receiving agent is not active: its status is WAIT.

- offer _ [to _] :: yielder[of event$^+$], yielder[of process-agent] $\rightarrow$
  action[communicating | diverging][using current buffer] (*total*, *restricted*).

(1) offer s [to p] =
| patiently check(the status of p is WAIT)
then
| send a message [to p][containing s] .

We also have two actions used to synchronize two agents (the performing and the environment agents), with respect to a given synchronization set. The first one is used in the main flow of events and the other to handle the special events $\surd$, yielded by the "SKIP" process, and τ, that represents an internal event to be performed independently of the environment agent (i.e. internal choices). Their definition can be found in [5].

The three actions bellow assume that the process representation is well constructed. They generalize the process representation structure and companion operations. To define concrete descriptions we need to fully specify these actions. Here we give their headers.

The behaviour of alphabet[of process p] is to return the initials of a process: the set of the all events initially communicable by a process, including τ and $\surd$.

- alphabet[of _] :: yielder[of process] $\rightarrow$
  action[giving set of event | completing] (*total*, *restricted*).

The behaviour of fire[event e][of process p] is to adjust the underlying representation of the process returning the next step in the process representation. These can be, for instance, the next node of the LTS graph.

- fire [_] [of _] :: yielder[of event], yielder[of process] $\rightarrow$
  action[giving cell | completing | failing] .

The behaviour of chooses[in s event $^+$][of process p] represents the environment agent of the process p, choosing an event inside the given event set s.

- chooses[in _][[of _] :: yielder[of event$^+$], yielder[of process] $\rightarrow$
  action[giving an event | failing | completing] (*total*).

The actions alphabet[of process p] and fire[event e][of process p] correspond to the **initial** and **after** semantic functions in the denotational semantics presented in [18].

4.4 Initialization Actions

The action presented in this section starts the execution of a process p considering the CSP sequential (prefix, external and internal choices, and recursion) and parallel operators (generalized parallelism and interleaving). It establishes the status precondition for the agents that perform each action.

(1) start-user-environment with p =

```
        | check( both( the environment-agent of p is the performing-agent,  ¹
        |      the process-agent of p is the contracting-agent )
        thence
        | subordinate the process-agent of p and put the performing-agent [in ACTIVE status] ²

        then
        | | send a message [to the given agent][containing function of choose-action of p] ³
        | and then
        | | regive ⁴

        then
        | unfolding⁵
        | | | select-an-event[from p][on the given event⁺]⁶
        | | then
        | | | get-ack-and-wait[from the process-agent of p][on the given event] ⁷
        | | then
        | | | unfold. ⁸
```

To simplify the explanation of the next action we number its important points and refer to these numbers in the text. This action first checks [1] that the environment agent [of p] is running (it is the performing agent) and that its process agent is the one with which it is interacting (the contracting agent). This guarantees that the environment agent controls the process agent. Next, the subordinate action is used to activate the process agent [2]. Afterwards, the ACTIVE status of the environment agent is recorded. In the sequence, it sends a message to the given agent [3] to choose one of the possible CSP operator actions (CSP_basic_protocol [for p] or CSP_parallel_protocol [for p][on e]) to execute according with the structure of the process p (the basic protocol is called if p has the left and right process set to unknown; otherwise parallel is called). This action also sets the agent status to satisfy the protocol actions preconditions. The behaviour of the protocol actions is give events to be selected by the user environment [6], so it regives the received events [4]. In the unfolding part [5], the action select-an-event[from p][on e^+] captures the user environment selecting an element [6]. Next, the action get-ack-and-wait[from p][on e] waits for the set of initially communicable events of p [7], an element of which is selected by the action select-an-event[from p][on e^+] [6]. This goes on [8] until the TERMINATE status flag is set for the process agent [of p], by the actions in the next section. In this case select-an-event[from p][on e^+] terminates. Note that the action start-user-environment executes in its own agent, so it plays the role of the topmost environment for the process network.

4.5 Actions to Animate CSP_M Operators

Here we define an action to animate the prefix, guarded recursion, external and internal choice CSP operators. We have only two actions to represent the CSP_M operators behaviour: one for the sequential (CSP_basic_protocol [for p]), and another for the parallel operators (CSP_parallel_protocol [for p][on s]). These two actions animate a CSP process by controlling the execution flow of the agents. The control is based on the actions alphabet [of p], fire[e][of p], and chooses [in s][of p], which, as explained in the previous section, capture the operational semantics in [18]. There exists a precondition for these actions to function properly: the process-agent and the environment-agent status of the given process must be WAIT and ACTIVE respectively, in order to avoid the live lock.

Firstly, CSP_basic_protocol [for p] synchronously puts the process-agent and the environment agent of the given process in ACTIVE and WAIT(Σ) status, respectively.[1]

- CSP_basic_protocol [for _] :: yielder[of process] $\rightarrow$ action[giving event* | communicating | completing | failing | diverging] [using current bindings | current storage | current buffer] (*total*).

```
(1)  CSP_basic_protocol [for p] =
        | synch_states [from the environment-agent of p][with the process-agent of p] ¹
        |      [containing elements of the set bound to Σ]
        thence
        | unfolding
```

Then it builds the set of events that p can communicate including τ and $\surd$ [2]. After that the action checks if the process is deadlocked (the set ATS of the process alphabet - its initials - is empty) [3] and treats that situation synchronously putting the process and environment agents in TERMINATED and ACTIVE status, respectively [4]. It treats $\surd$ [5] and τ[8] in much the same way. For the former, finishing actions are fired [6] and the processes silently dies [7]. For the latter, invisible actions are fired [9] and the action continues[10] (unfolds). For simplicity, we are prioritazing the selection of τ and $\surd$.

```
| | | bind "ATS" to the alphabet [of p] ²
| | thence
| | | | ...check ³ and treat deadlock ⁴ ...
| | | or
| | | | | ...check ⁵, execute ⁶ and treat termination ⁷ ...
| | | | or
| | | | | ...check ⁸, execute ⁹ and treat invisible actions ¹⁰ ...
| | | or
```

The action checks if the process has any event in the "ATS" set to execute [11]. It offers the possible communication set to the environment and waits for a response to select any given ATS events [12]. The environment must also be waiting for this to happen (barrier granted in [1]).

Then we need to adjust the process representation according to the chosen ATS event and regive the chosen event [13]. This extracts the selected given event from the communication set (initials of the process). After that, we need to send the same given event to the environment and synchronize the status (i.e activate the environment and make the process agent wait on it for any event [14]). Finally, we restart the action after the process and the environment have been resynchronized.[15]

```
| | | | check( not ( either( either( the given event, τ ), √ ) ) ) 11
| | | thence
| | |  | | offer the set bound to "ATS" [to the environment-agent of p] 12
| | |  | then
| | |  | | synch_events[from the process-agent of p][with the environment-agent of p]
| | |  | |       [containing the elements of the set bound to "ATS"]
| | | thence
| | |  | | fire [the given event][of p]  13
| | |  | and
| | |  | | the process-agent of p acks[the given event][to the environment-agent of e]  14
| | |  | |       [waiting on the elements of the set bound to Σ]
| | | then
| | | | unfold .  15
```

The action for the generalized parallelism and interleaving (CSP_parallel_protocol [for p][on s]) is presented below. Since its header is similar to the action above we omit it here. That action has an additional precondition: the left and right processes must be different of unknown.

Firstly, the action needs to ensure that the environment agent of the left process is the process agent of p and the same for the right process [1]; this also checks for unknown processes. This check [1] is needed because the agent performing this action plays the role of the environment for the left and right processes; the process agent of p receives the messages of left or right process agents as their environment agent [4]. Note that the environment agent of the process p can be either other sequential process or the topmost user environment.

As in the previous action, it needs to synchronously put the process-agent and the environment agent of the given process in ACTIVE and WAIT(Σ) status, respectively.[2] Here we want to define a kind of forking inside the process p[3]. Note that there is not a start order of the process [3'], so we use the interleave property of the "and" action combinator to capture this [11].

```
(1)  CSP_parallel_protocol [for p]]on s] =
       | check( both( the environment-agent of the left-process of p is 1
       |       the process-agent of p ) )
       and
       | check( both( the environment-agent of the right-process of p 1
       |       is the process-agent of p ) )

       and then
```

synch_states [from the environment-agent of p][with the process-agent of p] [2]
    [containing elements of the set bound to Σ]
then
  start-user-environment with the left of p [3]
and
  start-user-environment with the right of p [3']
then

The action recursively builds the set "C" of possible communicable events [5]. The auxiliary action build-C-set(A, B, s) applies the step-law for the CSP parallel operator ($C = (A \cap B \cap X) \cup (A \setminus X) \cup (B \setminus X)$, where A, B are the initials of P and Q, of P `[|`X`|]` Q, and X is the given synchronization set s) [17].

Next, the action checks [6] if the process is deadlocked ("C" is empty) and treats that situation synchronously putting the process and environment agents in TERMINATED and ACTIVE status, respectively [7]. Note that it must inform the participants (left and right processes) that this situation has been reached.

unfolding[4]
  bind "C" to build-C-set (the alphabet[of left of p], [5]
      the alphabet[of right of p], s)
thence
  ... check [6] and treat deadlock [7] ...
thence

After that, the action concurrently [9] waits for the acks of its participant processes, inside the elements of "C". The acknowledged events will be the events chosen by either process (left or right) [8, 8']. We have an interleaving because of the "and" property and not an agent based (CPU) concurrency.

respectively get-ack-and-wait[from the process-agent of the right of p][8]
    [on the elements of set bound to "C"]
and[9]
respectively get-ack-and-wait[from the process-agent of the left of p][8']
    [on the elements of set bound to "C"]
thence

In this case, they can interleave [12] if that events are outside the synchronization set s [10], or need to achieve synchronization via barrier [11]. In sequence, the event is selected for execution [13], and the process continues to execute [14].

... the given event $\in s$? [10] barrier (T) [11] or interleave (F) [12]
then
  select-an-event[from p][on the given event$^+$][13]
thence
  unfold. [14]

These actions give an operational view of the definitions of CSP operators and the step law of the parallel operator [17].

5 Conclusions and Related Works

In this work we have used Action Semantics to define an operational semantics of $\mathrm{CSP_M}$ [18] more legibly. We also made extensive use of the communicative facet of action notation extending it with some new primitive actions for synchronization, hand shaking, and communication. Tools like ANI [14] or ABACO [15] will be used in a possible future work to run and check our semantic description in order to have a formal animator implementation of the basic $\mathrm{CSP_M}$ operators. Since action notation has an underlying operational semantics, it can be interesting as a future work to compare this operational view of the action notation against the $\mathrm{CSP_M}$ operational semantics described in [18], with this we can guarantee that the same behaviour were defined in each description (i.e. our action semantics description of $\mathrm{CSP_M}$ against the operational semantics of [18]). Basically, the relationship between raw operational semantics of $\mathrm{CSP_M}$ [18], and our description is the readability, and modularity. It is also a tentative of a more concrete implementation of the $\mathrm{CSP_M}$ execution behaviour.

The work in [2] uses the communicative facet to describe distributed network protocols (SNMPv3). The concurrent primitives of a functional language (ML) is presented in [12]. As far as we know, there is only one work which uses action semantics to describe CSP [1], which is based on informal descriptions of the CSP dialect originally defined in Hoare's seminal paper [8]. Due to the lack of works in action notation that uses communicative facet, it was difficult to make a comparison against other works since there are no wide spread available action notation interpreters that run communicative actions.

The work in [2] inspired us to define an Action Semantics for CSP; the concurrent ML description of [12] inspired us to build a decentralized and generalized version of the communicative actions. This made the construction of a communicative facet extension framework for Action Semantics easier. This is an important part of our future work.

Some concepts were note contemplated in our work, like event hiding and renaming, replicated operators, and data type definition. We also do not consider the whole type expressiveness of $\mathrm{CSP_M}$ as noted in [3]. Due to the lack of space we cannot explain in more detail some of the actions and also the action notation structure, but a comphreensive work in this fied can be found in [11] and [5].

6 Acknowledgments

This work is partially supported by **CNPq**, the Brazilian Research Agency. We would like to acknowledge Peter Mosses for his effort in obtaining out of print references on Action Semantics of CSP and for many valuable discussions. The idea of using an LTS to first give an action semantics to CSP was first pursued by Alexandre Mota [13]; we thank him for discussing his work with us.

References

[1] S. Christensen and M. H. Olsen. Action Semantics of Calculus of Comunicating Systems and Communicating Sequential Processes. Technical report, Aarhus University, Sep 1988.

[2] M. Musicante D. Furlan and E. Duarte. An Action Semantics Description of the SNMPv3 Dispatcher. *IV Brazilian Symphosion of Programming Languages*, 2000.

[3] Formal Methods (Europe) Ltd. *FDR User's Manual version 2.28*, 1997.

[4] Formal Methods (Europe) Ltd. *PROBE Users Manual version 1.25*, 1998.

[5] L.J.S. Freitas. Action Semantics of CSP_M. Technical report, Semantics of Programming Language, available at http://www.cin.ufpe.br/~ljsf/pub/ascsp.ps, Nov 2000.

[6] L.J.S. Freitas. JACK: An approach to process algebra for Java. Master's thesis, UFPE, ago 2001. not yet published.

[7] G.H. Hilderink and E.A.R. Hendriks. Concurrent Threads in Java - CTJ v0.9, r17, Sep 2000. http://www.rt.el.utwente.nl/javapp.

[8] C.A.R. Hoare. Communicating Sequential Process. *Communications of ACM*, 21(8):666–677, Aug 1978. Seminal Paper.

[9] C.A.R. Hoare. *Communicating Sequential Process.* Prentice Hall, 1985.

[10] J. Krammer. *Concurrency: State Models & Java Programs.* Addison Wesley, April 1999. ISBN 0471987107.

[11] P. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts of Theorical Computer Science. Cambridge University Press, 1st edition, 1992.

[12] P. Mosses and M. Musicante. An Action Semantics for ML Concurrent Primitives. Technical Report RS-94-20, Aarhus University/UFPE, July 1994.

[13] A. Mota. Modeling CSP as LTSs using Action Semantics. Technical report, Semantics of Programming Language, available at http://www.cin.ufpe.br/~lmf, Nov 1996.

[14] H. Moura. An Implementation of Action Semantics. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631, pages 477–478, http://www.cin.ufpe.br/~rat, August 1992. Springer-Verlag. Lecture Notes in Computer Science.

[15] H. Moura and L.C. Menezes. The Abaco System - An Algebraic Based Action Compiler. In Armando Martín Haeberer, editor, *17th International Conference on Algebraic Methodology and Software Technology*, volume 1548, pages 527–529, http://www.cin.ufpe.br/~rat, January 1999. Springer-Verlag. Lecture Notes in Computer Science.

[16] P.D.Austin and P.H.Welch. Java Communicating Sequential Process - JCSP, Aug 2000. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

[17] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1st edition, 1997.

[18] B. Scattergood. *The Semantics and Implementation of Machine Readable CSP.* PhD thesis, University of Oxford, The Queen's College, 1998.

[19] D. Watt. *Programming Languages Syntax and Semantics.* Prentice Hall, 1991.

[20] D. Watt. Standard ML Action Semantics version 0.5. Technical report, University of Glasgow, May 1997.

[21] D. Watt and P. Mosses. Action Semantics in Action. Unpublished, 1987.