An Action Semantics for ML Concurrency Primitives

Peter D. Mosses
Martín Musicante

BRICS Report Series RS-94-20

ISSN 0909-0878 July 1994

An Action Semantics for ML Concurrency Primitives[*]

Peter D. Mosses[†]

BRICS,[‡] Dept. of Computer Science, University of Aarhus

Ny Munkegade Bldg. 540, DK-8000 Aarhus C, Denmark

Martín Musicante[§]

Universidade Federal de Pernambuco

Departamento de Informática

Recife – PE – Brazil

Abstract

This paper is about the recently-developed framework of action semantics. The pragmatic qualities of action semantic descriptions are particularly good, which encourages their use in industrial-scale applications where semantic descriptions are needed, e.g., compiler development.

The paper has two main aims: to demonstrate the remarkable extensibility of action semantic descriptions, and to illustrate the action semantics treatment of concurrency. These aims are achieved simultaneously, by first giving the description of a sequential (ML-like) programming language fragment, and then extending the described language with some concurrency primitives (taken from CML). The action semantic description of the sequential part of the language *does not change at all* when the concurrency primitives are added, it merely gets augmented by the description of the new features!

[*]To appear in Proc. FME'94 (Formal Methods Europe, Symposium on Industrial Benefit of Formal Methods, 24–28 October, 1994, Barcelona), Lecture Notes in Computer Science, Springer-Verlag, 1994. References to this work should cite the Proceedings.

[†]E-mail address: pdmosses@daimi.aau.dk. This work was partly supported by the Danish Science Research Council project DART (5.21.08.03).

[‡]Centre for Basic Research in Computer Science, funded by the Danish National Research Foundation

[§]E-mail address: mam@di.ufpe.br. This work was partly supported by CNPq, Brazil.

1 Introduction

Action semantics [Mos92] is a formal framework for semantic description, developed to provide "tractable" descriptions of real-life languages (for example, see [Tof93, NT94, HT94]). Action semantic descriptions, like those written in denotational semantics [Mos90, Sch86], are *compositional*: semantic functions, mapping abstract syntax to semantic entities, are defined inductively using semantic equations. However, in action semantics the semantic entities are *actions* rather than higher-order functions, and the essence of actions is much more *computational* than that of (pure) functions.

A special notation has been developed for use in action semantics. This notation is called *action notation*, and it is used in action semantic descriptions very much in the same way as the λ-notation is used in denotational semantics. The symbols used in action notation are intentionally verbose, so that English-like phrases can be used—completely formally—to express most of the concepts present in programming languages. The operational semantics of the notation is given in [Mos92].

Action semantic descriptions are inherently modular. They are easily extended or modified. Reusing parts of specifications is straightforward. In this paper, we demonstrate these features, by extending the semantic description of a simple, ML-like, sequential language, adding first-order synchronous communication constructors taken from CML [Rep91a].

The next section gives a brief account of the action semantics formalism. Section 3 presents the action semantics of a simple sequential language. Section 4 considers processes and synchronous communication. Section 5 extends the action semantic description of the sequential language to deal with the chosen concurrency primitives.

Both the sequential and the concurrent languages are similar to those presented in [BMT92]. In that work, various changes to the original description of the sequential language were needed to introduce the concurrent constructs and their operational semantics. This is not the case in our description using action semantics: when the first-class synchronous operations are introduced, only *extensions* to the semantic entities of the sequential description are needed, and the rest of the description remains unchanged. Moreover, the use of action semantics has the advantage that the problem of giving a fully *distributed* implementation of CML's concurrency primitives becomes quite apparent. This is ensured by the action semantics treatment of concurrency, which is based on a quite realistic *asynchronous* model; a truly *operational* analysis of synchronization has to be specified. A description of CML's concurrency primitives in terms of CCS- or CSP-like synchronization would not be so revealing. These issues are discussed further in the concluding section.

2 Action 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* to process *information*, with various possible outcomes: normal termination (performance of the action *completes*), exceptional termination (it *escapes*), unsuccessful termination (it *fails*) or non-termination (it *diverges*). Action notation provides some primitive actions, and various *combinators* for forming complex actions, corresponding to the main fundamental concepts of programming languages.

A *data notation* is used to describe the information processed by actions. The standard data notation (included in action notation) provides a collection of algebraically defined abstract data types, including numbers, characters, strings, sets, tuples, maps, etc.; further data may be specified *ad hoc*.

There is also a third class of entities in action notation, called *yielders*. A yielder represents data whose value depends on the current information available to the primitive action in which it occurs. Yielders are *evaluated* to yield data. An example of a standard yielder is the data bound to I, which depends on the current bindings that are received by the enclosing primitive action.

Action notation possesses five so-called 'facets':

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 primitive action give the successor of the given natural is given a natural number n as transient data, it completes, giving $n + 1$ as a transient. The compound action A_1 then A_2 performs the action A_1 first; all transient data given by A_1 is passed on to A_2, which is performed after A_1 completes. The primitive action choose D, where D is a sort of data, makes a non-deterministic choice of an individual of sort D, giving the chosen datum as a transient.

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

Imperative: This facet is concerned with *storage* handling. A storage in action notation is simply a mapping from (the currently allocated) *cells* to storable data. For example, consider the action allocate a cell then store 26 in the given cell, which combines features of the functional and imperative facets.

Communicative: This facet provides a system of *agents*, which can each be 'contracted' to perform particular actions. Initially only a special 'user' agent is active. Agents can communicate using asynchronous message passing: the sending of a message is non-blocking. 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 reach its destination agent. Moreover, each agent is created with its own storage, which cannot be affected (nor inspected) by other agents. Arbitrary data can be contained in messages, including the identities of agents.

Most of the primitive actions have a use in connection with only one facet each, but the action combinators generally involve a mixture of the basic, functional, and declarative facets, determining the flow of control, transient data, and bindings between the subactions.

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. 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 abstractions before their enaction, for use by the encapsulated action. Abstractions can be treated just like any other data, i.e., given as transients, bound to tokens, stored in cells, and sent in messages. They are also used to determine the 'contracts' offered to agents in the communicative facet.

For a more detailed description of action notation, the reader is referred to [Mos92]; for an introduction, see also [Wat91]. An overview of the operational semantics of action notation is given in the Appendix below.

3 A Simple Sequential Example

This section describes the (dynamic) action semantics of a simple sequential functional language, derived from Standard ML [MTH90]. It is essentially the same language as the one described in [BMT92, Sect. 3].

As in denotational semantics, a description in action semantics is divided into three main parts, specifying the abstract syntax of the language being described, the semantic functions that map abstract syntactic phrases to their meaning, and the semantic entities used by the semantic functions.

The modular structure of the specification is itself formally specified, by giving each module a title, and indicating (by writing **needs:** or **includes:**) which other modules it imports, if any. Modules may be nested: a submodule implicitly imports all that is specified (or imported) directly by each enclosing module. The order in which modules are presented is irrelevant, and mutual importation is allowed [Mos89b].

Each module is given below as a numbered, titled section. Interspersed with the formal specification, some informal explanations are given, for the benefit of those readers who are unfamiliar with action notation.

3.1 Abstract Syntax

grammar:

- Expression = Identifier **|** "()" **|**
  ⟦ "(" Expression "," Expression ")" ⟧ **|**
  ⟦ Expression Expression ⟧ **|**
  ⟦ "fn" Identifier "⇒" Expression ⟧ **|**
  ⟦ "rec" Identifier "(" Identifier ")" "=" Expression ⟧ .
- Identifier = ⟦ letter$^+$ ⟧ .
- Program = Expression .

The abstract syntax of the language is defined by a grammar using the standard data notation for trees and strings. The brackets ⟦...⟧ indicate node construction in abstract syntax trees. (In denotational semantics, these brackets are usually regarded as part of the notation for semantic functions; nevertheless, the left-hand sides of action semantic equations are quite similar in form to those of denotational semantic equations.) Strings in the right-hand-sides of the grammar equations correspond to leaves of the trees.

A program in our simple example language is given by a single expression. Notice that the "rec" construct is an expression, rather than a declaration. Also, the sort Identifier encompasses variables, constants, and constructors—we do not rely on some preceding static analysis to distinguish the classes of the different occurrences of identifiers, in contrast to [BMT92] (and to [MTH90], where the dependency between static and dynamic semantics caused some problems).

3.2 Semantic Functions

needs: Abstract Syntax, Semantic Entities.

The action semantics of a programming language is given by means of semantic functions. These functions map abstract syntactic phrases of the language to actions. Each semantic function is introduced at the beginning of the module that defines it, inductively, by semantic equations.

3.2.1 Evaluating Expressions

introduces: evaluate _ .

- evaluate _ :: Expression → action .

The sort action includes all actions. We could be more specific here, using an algebraic notation for subsorts of actions to indicate that evaluate E is always an action which, whenever it completes, gives a value.

```
(1)  evaluate $I$:Identifier = give the value bound to $I$ .
```

When not in the scope of any binding for I, the yielder the value bound to I evaluates to the special entity nothing, whereupon the give action above fails. Note that I may be a constructor, a constant, or a variable, but the distinction is irrelevant for the dynamic semantics.

```
(2)  evaluate "()" = give the unit-value .
```

The constant unit-value is specified in Sect. 3.3.

```
(3)  evaluate ⟦ "(" $E_1$:Expression "," $E_2$:Expression ")" ⟧ =
          ( evaluate $E_1$ and then evaluate $E_2$ )
          then give the pair of the given (value, value) .
```

The action combination A_1 and then A_2 specifies sequential (left-to-right) performance of its subactions, as does A_1 then A_2. The difference is that with the former combination, any transient data given by A_1, A_2 are concatenated and given by the whole action, whereas with the latter combination, the transient data given by A_1 are given only to A_2. There is also a combination A_1 and A_2, used later, that specifies implementation-dependent order of performance, but which is otherwise like A_1 and then A_2.

The operation pair of _ is specified in Sect. 3.3, it serves merely to form a single value from two values. By the way, the articles 'the', 'a', and 'an' are generally insignificant in action notation (formally, they denote the identity function).

```
(4)  evaluate ⟦ $E_1$:Expression $E_2$:Expression ⟧ =
          | evaluate $E_1$ and then evaluate $E_2$
          then
          | give construction of (the given constructor#1, the given value#2) or
          | enact application of body of the given function#1
          |                   to the given value#2 .
```

The combination A_1 or A_2 generally provides a nondeterministic choice between the alternative actions A_1, A_2. However, if the chosen action fails, the other one is performed instead, so the choice may turn out to be deterministic—as above, where A_1 fails unless the first (#1) value is of sort constructor, and A_2 fails unless it is of sort function.

The yielder application of Y_1 to Y_2 evaluates Y_2 and supplies it as a transient to the abstraction yielded by Y_1. So when the action encapsulated in the abstraction representing the body of a function is performed (via enact above), it is given just a single value, representing the argument of the function.

Note that vertical bars are used to enforce the intended grouping of actions, as an alternative to parentheses.

```
(5)  evaluate ⟦ "fn" $I$:Identifier "⇒" $E$:Expression ⟧ =
         give function of closure of abstraction of
             | furthermore bind $I$ to the given value
             | hence evaluate $E$ .
```

The use of closure with abstraction of A ensures that the action A receives the static bindings whenever the abstraction gets enacted. The operation function of _ (specified in Sect. 3.3) merely tags the abstraction so that it can be distinguished from abstractions used for other purposes.

The combination A_1 hence A_2 is similar to A_1 then A_2, but here it is bindings rather than transients that get passed from A_1 to A_2. The use of furthermore at the start of A_1 specifies that the bindings received by the whole action should be received also by A_2, except for those that get hidden by bindings produced by A_1.

```
(6)  evaluate ⟦ "rec" $I_1$:Identifier "(" $I_2$:Identifier ")" "=" $E$:Expression ⟧ =
         recursively bind $I_1$ to function of closure of abstraction of
             | furthermore bind $I_2$ to the given value
             | hence evaluate $E$
         hence give the function bound to $I_1$ .
```

The action recursively bind I to Y allows the closure yielded by Y to refer to itself. (The operational semantics of this action involves so-called indirect bindings, whereby circular bindings can be formed.)

- Identifier $\leq$ token .

Note that $\leq$ indicates sort inclusion.

3.2.2 Running Programs

needs: Evaluating Expressions.

introduces: run _ .

- run _ :: Program $\rightarrow$ action .

(1) run E:Expression = initialize-bindings hence evaluate E .

The initialize-bindings action is defined in Sect. 3.3.

3.3 Semantic Entities

includes: [Mos92]/Action Notation.

The above reference to the official definition of action notation provides all the action primitives and combinators that are needed here; see the Appendix below for a list of the relevant symbols. It remains only to specify what data are to be processed by actions.

3.3.1 Data

needs: Values.

- datum = value .
- bindable = value .
- token = string of letter$^+$.

The sorts datum, bindable, and token are left open by action and data notation, as they depend on the semantics of the language being described.

3.3.2 Values

needs: Pairs, Constructions, Functions.
introduces: value , unit-value .

- value = unit-value | pair | construction | function (*disjoint*) .
- unit-value : value .

The sort value is independent of action notation, and introduced here only for convenience. In practice, it is generally used to correspond to the notion of 'R-values' in denotational semantics.

3.3.3 Pairs

needs: Values.
introduces: pair , pair of _ .

- pair of _ :: value^2 $\rightarrow$ pair (*total, injective*) .
- pair = pair of value^2 .

Note that the notation value^2 is another way of writing the tuple sort (value, value).

3.3.4 Constructions

needs: **Values.**

introduces: constructor , construction , construction of _ .

- constructor = token .
- construction of _ :: (constructor, value) $\rightarrow$ construction (*total, injective*) .
- construction = constructor | construction of (constructor, value) .

The semantics of a constructor identifier is the identifier token itself. In this example language, constructors are untyped, so they can be applied to arbitrary values.

3.3.5 Functions

introduces: function , function of _ , body _ .

- function of _ :: abstraction $\rightarrow$ function (*total*) .

(1) function of a:abstraction = f:function $\Rightarrow$ body f = a .

3.3.6 Initializations

needs: **Functions.**

introduces: initialize-bindings .

```
• initialize-bindings = bind "true" to "true" and bind "false" to "false" and
                        bind "not" to function of abstraction of
                             give ( when there is given "true" then "false" |
                                    when there is given "false" then "true" ) .
```

The initial bindings here provide only the standard constructors for the Booleans, and the negation function. Other constructors and constants could be defined without additional complications.

4 First-Class Synchronous Operations

Reppy [Rep91a] presents CML, a concurrent extension of the Standard ML language. CML has a fork-style primitive for spawning new processes. CML processes communicate values synchronously over typed channels.

In [Rep91b], the operational semantics of the new primitives is given. In [BMT92], a subset of Reppy's primitives is chosen, a new operational semantics is given for the reduced set and several useful properties are proved. Figure 1 is taken from [BMT92]; it shows the signature of the chosen subset of CML operations. These first-class synchronous operations allow not only for sending and

```
signature Concurrency = sig
 type 'a channel
 val channel: unit -> '_a channel
 type 'a com
 val send:    'a channel * 'a -> 'a com
 val receive: 'a channel -> 'a com
 val choose:  'a com * 'a com -> 'a com
 val wrap:    'a com * ('a -> 'b) -> 'b com
 val noevent: 'a com
 val fork:    (unit -> 'a) -> unit
 val sync:    'a com -> 'a end
```

Figure 1: The signature of the concurrency primitives.

receiving values through channels: nondeterministic choice between communications (`choose`) and post-processing of the result of communications (`wrap`) is possible as well.

A value of type `'a com` is called a 'suspended' communication. As it is a first-class value, it can be used as an argument of a function or constructor, just like any other value. The communication can only completed when the `sync` function is applied to it, and then only when a matching communication is given as an argument of `sync` by another process. Communication is synchronous: when a process calls `sync`, it is blocked until it becomes possible to complete the requested communication.

5 Extension to a Simple Concurrent Example

In this section we add processes and synchronous communication primitives to our simple sequential language. These primitives are the same as given in [BMT92], and a subset of those present in the CML language [Rep91a]. We do not foresee any problems in adding the rest of the CML synchronous operations to our definition.

Both the abstract syntax and semantic functions parts of the sequential language specification are *reused without any modification*!

5.1 Abstract Syntax

This module remains unchanged. The concurrency primitives are added as constants and constructors, i.e., identifiers, for which abstract syntax has already been specified.

5.2 Semantic Functions

This module also remains unchanged! This is because of the 'orthogonality' of the facets of action notation: the presence or absence of actions involving the communicative facet in no way affects the usage of the primitive actions and combinators involving the other facets. For example, in the action combination A_1 then A_2, the passing of transient data from A_1 to A_2 is completely independent of whether the subactions send any messages or offer contracts to other agents.

Actually, there is one small part of action notation that *is* sensitive to the presence or absence of the various facets of actions: the algebraic notation for subsorts of actions, which is used just for specifying facets! Had the target sort of evaluate _ been specified not merely as action but more precisely, as action [giving a value | diverging | redirecting], the outcome possibility communicating would now have to be added. But this is a trifling matter, and does not weaken the claim of extensibility of action semantic descriptions.

It should be stressed that this remarkable extensibility is not a peculiarity of the simple examples considered in this paper, it seems to be inherent in the use of action notation. It would be interesting to see whether one could achieve comparable extensibility for this example when using monads in denotational semantics [CM93].

5.3 Semantic Entities

includes: [Mos92]/Action Notation.

5.3.1 Data

needs: Values, Requests.

- datum = value .
- bindable = value .
- sendable = request | response .
- storable = nothing .

The identities of storage cells are used below to distinguish channels, but nothing is ever stored in the cells, hence the above specification of storable.

5.3.2 Values

needs: Pairs, Constructions, Functions, Channels, Requests.

introduces: value , unit-value .

- value = unit-value **|** pair **|** construction **|** function **|**
  channel **|** request (*disjoint*) .
- unit-value : value .

The only change above is the addition of two new subsorts of value.

The modules **Pairs**, **Constructions**, and **Functions** are omitted here, as they are identical to those given in Sect. 3.3.

5.3.3 Initializations

needs: Functions, Forks, Requests.

introduces: initialize-bindings .

```
(1) initialize-bindings =
        bind "true" to "true" and
        bind "false" to "false" and
        bind "not" to function of abstraction of
            give ( when there is given "true" then "false"  |
                   when there is given "false" then "true" )
        and
        bind "send" to "send" and
        bind "receive" to "receive" and
        bind "choose" to "choose" and
        bind "wrap" to "wrap" and
        bind "noevent" to "noevent" and
        bind "channel" to function of abstraction of channel-action and
        bind "fork" to function of abstraction of fork-action and
        bind "sync" to function of abstraction of sync-action and
        initialize-synchronization .
```

The treatment of "send", "receive", "choose", and "wrap" as constructors implies that their real semantics lies in the way that the corresponding constructions influence communication, as specified below in channel-action, fork-action, and sync-action.

5.3.4 Channels

needs: Values.

introduces: channel , channel-action .

- channel $\leq$ cell .
- channel-action = allocate a channel .

Each invocation of the `channel()` function of CML reserves a fresh, new communication channel for use within a program. In our description, channels are represented as storage cells. The primitive action allocate _ reserves a previously unused cell, which is the desired semantics of the `channel()` function.

Note that in the action semantics of other concurrent languages, a channel can often be represented by an agent that (busily) inspects its buffer until it receives a matching pair of requests for reading and writing on the channel. But in CML, a single suspended communication might involve several channels at once, with mutual exclusion between them, and it seems that it would be very complicated to let channels be separate agents in this case.

5.3.5 Forks

needs: Values.

introduces: fork-action .

(1) fork-action =
  offer a contract [to some agent] [containing the abstraction yielded by
    the application of the body of the given function to the unit-value]
  and give the unit-value .

Recall that the fork-action represents the body of the CML `fork` function, which gets applied to an expression to be evaluated by the new process—but the evaluation of the expression has to be delayed, so it is made the body of a function of the unit value (). This is reflected by the explicit application that occurs in the contents of the contract above.

The primitive action offer Y evaluates Y to a *sort* of contract, where the action to be performed has been determined as the contents of the contracts included in the sort (as usual, the action has to be encapsulated in an abstraction). It is also possible to determine a subsort of the agents to which the contract may be offered, but the use of [to some agent] above leaves the sort of agent completely open. As soon as the 'offer' has been made, the performing agent can proceed, without waiting for an agent to accept the contract and start performing the specified action. (One can easily express such waiting: let the specified action start by sending a signal back to the contracting agent, which should then patiently inspect its buffer until the signal arrives.)

5.3.6 Requests

needs: Values, Pairs, Constructions, Functions, Channels.

introduces: request , response , synchronizing-agent ,
initialize-synchronization , sync-action .

- request = construction of (“send”, pair of (channel, value)) **|**
  construction of (“receive”, channel) **|**
  construction of (“choose”, request^2) **|**
  construction of (“wrap”, pair of (request, function)) **|**
  “noevent” .

The request sort corresponds to the `'a com` types in Fig. 1. The polymorphic type information is disregarded here, as it is not relevant for the dynamic semantics of the language.

- response = value **|** abstraction .

The response sort is specified to be the union of the value and abstraction sorts. When no `wrap` operation is involved in the synchronization of two matching requests, the result of such a synchronization will be a value. The need for abstractions arises when a post-synchronization operation is to be performed, as explained below.

- synchronizing-agent : agent [not in set of user-agent] .

In the absence of any assumptions about relative processing speed or message transmission time, it appears to be difficult to distribute the decisions about synchronization between the various processes. The problem arises with symmetrical situations involving mutually-exclusive choices between three or more processes: a tentative choice proposed by one process may be outdated by the time the other processes get to know about it!

As the semantics of the language has to cope with all possible programs, it seems that we are forced to introduce a centralistic arbiter agent to represent the locus of synchronization decisions. Note, however, that the appearance of this artefact in the semantics does not rule out the possibility of a clever implementation of truly distributed synchronization, conforming to the specified semantics (from the point of view of the user-agent).

```
(1) initialize-synchronization =
        offer a contract [to the synchronizing-agent] [containing abstraction of
            unfolding
              | patiently
              |   | choose a synchronized pairing of the items of the current buffer
              | then
              |   | send a message [to the sender of the given message#1]
              |   |       [containing the given response#3] and
              |   | send a message [to the sender of the given message#2]
              |   |       [containing the given response#4] and
              |   | remove the given message#1 and remove the given message#2
              | then unfold ] .
```

The action performed by the synchronizing-agent (given in the contract sort specified above) involves several standard action primitives and combinators that have not yet been explained. The unfolding ... unfold... construct may be regarded as an iteration here, although it is more general. If preferred, it may also be regarded as an abbreviation for the infinite action obtained by actually doing the unfolding syntactically. The action patiently A keeps on performing A while it fails. The primitive action choose Y fails whenever Y evaluates to nothing, otherwise it gives an individual chosen arbitrarily from the sort to which Y evaluates. The primitive action send Y initiates the transmission of the message specified by the sort yielded by Y, where the agent to receive the message is already determined. Finally, remove Y disposes of the message yielded by Y, so that it is no longer in the buffer.

Thus the effect specified above is that of busy-waiting until one or more matching pairs of requests have arrived in the local buffer. The synchronized pairing then yields a sort including not only the matching pairs of request messages, but together with each pair their chosen responses. Having chosen one of these quadruples, it is a straightforward matter to complete the synchronization of the requests by sending back the responses to the agents that made the requests, and removing the chosen messages from the buffer.

```
(2) sync-action =
    send a message [to the synchronizing-agent]
        [containing the given request] then
    receive a message [from the synchronizing-agent]
        [containing a response] then
     | give the value yielded by the contents of the given message or
     | enact the abstraction yielded by the contents of the given message .
```

The action receive Y is actually a standard abbreviation for a compound action that waits busily until a message of the sort specified by Y arrives in the local buffer, then removes it from the buffer and gives it as a transient.

privately introduces: synchronized _ , synchronized-responses _ , first-response-application _ to _ , pairing _ .

The **privately introduces:** directive has the effect of restricting the use of the listed symbols to the current submodule.

- synchronized _ :: message^2 $\rightarrow$ (message^2, response^2) (*linear*, *strict*) .

The synchronized _ operation takes a pair of messages containing requests, and forms a sort of quadruples. The two first components of each quadruple are the original messages, while the third and fourth components are the responses to the requests. These responses are obtained by selecting each possible combination of matching requests, as specified by the synchronized-responses _ operation. The *linear* attribute specifies that the operation distributes over sort union, while *strict* indicates that the operation maps nothing to nothing. Note that operations specified as *total* and *partial* are also *linear* and *strict*.

(Readers who are not used to applying operations to sorts as well as to individual values, as in the definitions below, may find it helpful to regard individuals as singleton sets, and the operation _**|**_ as set union. The constant nothing may be regarded as the empty set. All the operations can then be considered as defined element-wise on sets. See [Mos89a] for the foundations of applying operations to sorts.)

(3) synchronized (m_1:message, m_2:message) =
    (m_1, m_2, synchronized-responses of (contents of m_1, contents of m_2)) .

- synchronized-responses _ :: request^2 $\rightarrow$ response^2 (*linear*, *strict*) .

The synchronized-responses _ operation takes a pair of requests for synchronization and gives the sort of all possible responses from the given requests. If no synchronization is possible, the result of this operation is nothing. The definition below corresponds closely to [BMT92, Figure 5].

(4) synchronized-responses (r_1:request, r_2:request) =
    reverse synchronized-responses (r_2, r_1) .

(5) synchronized-responses
(construction of ("send", pair of (k_1:channel, v:value)),
construction of ("receive", k_2:channel)) =
    when k_1 is k_2 then (v, v) .

(6) synchronized-responses
(construction of ("choose", pair of (r_1:request, r_2:request)), r_3:request) =
    synchronized-responses (r_1, r_3) **|** synchronized-responses (r_2, r_3) .

(7) synchronized-responses
(construction of ("wrap", pair of (r_1:request, f:function)), r_2:request) =
    first-response-application of f to synchronized-responses (r_1, r_2) .

(8) synchronized-responses ("noevent", r:request) = nothing .

- first-response-application _ to _ :: function, response^2 $\rightarrow$ response^2 (*total*) .

When a response to a synchronization is to be post-processed (due to the existence of a `wrap` operation), an abstraction is constructed. This abstraction will be *enacted* as a result of the performance of the sync-action, as explained before.

(9) first-response-application f:function to (v:value, r:response) =
    (application of the body of f to v, r) .

(10) first-response-application f:function to (a:abstraction, r:response) =
    (a then the body of f, r) .

- pairing _ :: message* → message^2 (*linear*, *strict*) .

The pairing _ operation simply forms all possible pairs of elements from a tuple of messages.

(11) pairing ( ) = nothing .

(12) pairing m:message = nothing .

(13) pairing (m_1:message, m_2:message, m:message*) =
(m_1, m_2) | pairing (m_1, m) | pairing (m_2, m) .

6 Discussion

Q: *Why are the action semantic descriptions (a.s.d's) so long?*
A: Well, if one removes all the tutorial comments, the a.s.d. of the sequential language fills about three pages. A transitional semantics for the same language might fill about one page. Some of the extra length of the a.s.d. is, of course, due to the use of verbose, multi-character symbols and the attempt at 'natural' language. The module titles and imports also take up some lines. But try *reading* through the description—at least that doesn't take much longer than it would with a transitional semantics, it seems. In any case, the size of the extension to the concurrent language is hardly excessive, considering the nature of the constructs being described.
Q: *Isn't action notation just another programming language, and an a.s.d. a compiler into it?*
A: Many formal notations, including the λ-notation and action notation, can certainly be implemented and used for programming. The crucial feature that distinguishes them from ordinary programming languages is that they have tractable, well-understood semantics themselves. Action notation has been fine-tuned for use in a.s.d's, and would probably rather tedious to use for programming.

Concerning the second part of the question: an a.s.d. can indeed be regarded as 'compiling' or reducing the described programming language to action notation—just as a denotational semantics reduces it to the λ-notation. It is clearly beneficial to explicate a range of complex phenomena in terms of a fixed set of relatively simple constructs.
Q: *Aren't there really too many primitives in action notation?*
A: No! To get an overview of those we have used, see the Appendix. These account for at least 50% of the full action notation. To eliminate some of the action primitives and combinators would undermine the extensibility of a.s.d's. For instance, to avoid the declarative facet would require passing bindings as transient data, which seems artificial, and which would also be notationally undesirable.
Q: *Why is the communicative facet of action notation based on asynchronous primitives, rather than on the better-studied synchronous ones?*

A: Action notation is intended for use in describing high-level programming languages, where it is often not possible to ignore the time it takes to synchronize distributed processes. The lack of synchrony in action notation also allows a naïve operational semantics that reflects true concurrency. In any case, it is considerably more illuminating to explicate languages like CML in terms of asynchronous message passing, than in terms of (e.g.) CCS. See [Agh86] for further arguments in favour of asynchronous primitives.
Q: *What are the main advantages of action semantics over the popular transitional style of semantics advocated by Plotkin, Milner, et. al.?*
A: In fact action semantics can be regarded as a particular *discipline* for writing transitional semantics: the semantic equations, together with the operational semantics of action notation (which is defined in the transitional style!) induce a transitional semantics for the described language—although it isn't 'structural', at least not in the usual sense. Thus instead of, say, left-to-right evaluation being implicit in the transition rules for various constructs of the described programming language, the concept is *named* by a combinator, which is used where appropriate, and the rules for this combinator are given once and for all.

Another aspect of this discipline is that action notation does *not* allow the instantaneous inspection of the 'current state' of a distributed system of agents; one is forced to analyse synchronization in terms of asynchronous message-passing. When using transitional semantics directly may lead to nondeterministic choices that cannot (easily) be made on the basis of locally-available information.

Finally, there is the extensibility of a.s.d's. This makes it (almost) trivial that the extension of an a.s.d. to describe concurrent constructs indeed preserves the semantics of sequential programs. Such results are not so immediate with transitional semantics, see [BMT92].

Appendix

Action Performance

A formal presentation of the operational semantics of action notation [Mos92] is out of the scope of this paper. The following informal comments indicate its main features, and may be helpful to some readers.

The operational semantics defines what transitions between configurations are possible. Let us first consider the *local* transitions. Each agent has a local configuration consisting of: the (remaining) action to be performed; the current data and bindings; the current storage, determining which cells are reserved and their contents; and the current buffer of messages. When a primitive action is to be performed, any yielder arguments are evaluated with respect to the current (local) information, whereafter a transition to a new configuration is possible. The transitions for compound actions are determined by those of their subactions, as usual in structural operational semantics. Each combinator determines the flow of control (sequential, interleaving, or nondeterministic choice) and how the transient data and bindings for the combined action depend on those for its subactions. Transitions may affect the storage, and remove messages from the buffer. Finally, a local transition may have communicative effects: sending messages to other agents, and offering contracts containing actions to be performed by other agents.

Now for *global* transitions. Initially, only one (user-) agent is active; other agents become active when they accept contracts. A global configuration consists of a local configuration for each active agent, together with all the messages that are being transmitted between agents (and any contracts that are still on offer). The lack of *any* assumptions about the processing speed of agents, or the transmission speed of messages, is modelled by attaching arbitrary but finite positive delays to local transitions and message transmissions. Each global transition counts down all the delays. When a local transition has no more delay, a new transition for the agent is chosen; when a message has no more delay, it is inserted in the buffer of the target agent (assuming that it is already active).

Action equivalence is defined as a testing equivalence, based on the operational semantics. In practice, algebraic laws about action equivalence are verified using a weak bisimulation defined in terms of local transitions. The theory of action notation is still being developed; currently, reasoning about multi-agent action performances has to be based directly on the global transitions.

Action Notation used in Sect. 3

action	yielder	data
A_1 or A_2	the D yielded by Y	(D_1, D_2)
A_1 and A_2	the Y	D^2
A_1 and then A_2	of Y	a D, the D
give Y	the given D	datum
A_1 then A_2	the given $D\#n$	natural
bind D to Y	the D bound to Y	bindable
recursively bind D to Y		token
furthermore A		
A_1 hence A_2		
enact Y	closure Y	abstraction of A
	application Y_1 to Y_2	abstraction
A, A_1, A_2: action	Y, Y_1, Y_2: yielder	D, D_1, $D_2 \leq$ data

Action and Message-Sort Notation used in Sect. 5

action	yielder	data
unfolding A unfold		
choose Y		
send Y remove Y receive Y offer Y patiently A	current buffer	message agent user-agent contract sendable
allocate D		cell , storable
A: action	Y: yielder	$D \leq$ data

message	agent	sendable	contract
m [from a] m [to a] m [containing s]	sender m	contents m	c [to a] c [containing abs]
$m \leq$ message	$a \leq$ agent	$s \leq$ sendable	$c \leq$ contract

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992.

[CM93] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. Draft, May 1993.

[HT94] Bo Stig Hansen and Jens Ulrik Toft. The formal specification of ANDF, an application of action semantics. In Peter D. Mosses, editor, *Proc. First Intl. Workshop on Action Semantics (Edinburgh, April 1994)*, number NS-94-1 in BRICS Notes Series, pages 34–42. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[Mos89a] Peter D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[Mos89b] Peter D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[Mos92] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NT94] Jens P. Nielsen and Jens Ulrik Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1994.

[Rep91a] John H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.

[Rep91b] John H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Computer Science Dept., Cornell Univ., 1991.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[Tof93] Jens Ulrik Toft. Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report 202104/RPT/12, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1993.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.