To appear in *Proceedings of the REX Workshop on Semantics - Foundations and Applications, Beekbergen, The Netherlands, June 1992,* which are to be published as a volume of *Lecture Notes in Computer Science*, Springer-Verlag, 1993. Citations should refer only to the *Proceedings*, not to this preprint.

ON THE ACTION SEMANTICS OF CONCURRENT PROGRAMMING LANGUAGES

Peter D. Mosses

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

ABSTRACT Action semantics is a framework for semantic description of programming languages. In this framework actions are semantic entities, used to represent the potential behaviour of programs — also the contributions that parts of programs make to such behaviour. The notation for expressing actions, called action notation, is combinator-based. It is used in much the same way that lambda-notation is used in denotational semantics. However, the essence of action notation is operational, rather than mathematical, and its meaning is formally defined by a structural operational semantics together with a bisimulation equivalence.

This paper briefly motivates action semantics, and explains the basic concepts. It then illustrates the use of the framework by giving an action semantic description of a small example language. This language includes a simple form of concurrency: tasks that may synchronize by means of rendezvous. The paper also discusses the operational semantics of action notation, focusing on the primitive actions that represent asynchronous message transmission and process initiation.

Keywords semantics, action semantics, action notation, concurrency, synchronization, asynchrony, distributed processing.

CONTENTS

0. INTRODUCTION

Action semantics is a recently-developed framework for formal semantics [12, 14] It combines formality with many good pragmatic features. Regarding comprehensibility and accessibility, for instance, action semantic descriptions compete with informal language descriptions. Action semantic descriptions scale up smoothly from small example languages to full-blown practical languages. The addition of new constructs to a described language does not require reformulation of the already-given description. An action semantic description of one language can make widespread reuse of that of another, related language. All these pragmatic features are highly desirable. Action semantics is, however, so far the ***only*** semantic framework that enjoys them! See [12] for a comprehensive exposition of action semantics, with demonstration of its claimed pragmatic qualities.

Action semantics is ***compositional,*** like denotational semantics [10]. The main difference between action semantics and denotational semantics concerns the universe of semantic entities: action semantics uses entities called ***actions,*** rather than the higher-order functions used with denotational semantics. Actions are inherently more operational than functions: when ***performed,*** actions process information ***gradually.***

Primitive actions, and the various ways of combining actions, correspond to fundamental concepts of information processing. Action semantics provides a particular notation for expressing actions. The symbols of action notation are suggestive words, rather than cryptic signs, which makes it possible to get a broad impression of an action semantic description from a superficial reading, even without previous experience of action semantics. The action ***combinators,*** a notable feature of action notation, obey desirable algebraic laws that can be used for (simple) reasoning about semantic equivalence. We shall consider the basic concepts of action performance in Section 1.

The main aim of this paper is to illustrate the action semantics of concurrent programming languages. In Section 2 we shall describe a simple example language having ***task*** declarations. Tasks may ***synchronize*** by means of rendezvous, arranged by matching entry call and accept statements. The action semantic description of this language shows how the standard primitive actions for ***asynchronous*** message transmission can be used to explicate synchronization The intended interpretation of all the action notation used in the description will be explained (albeit briefly) when we first meet it.

The formal definition of action notation [12, Appendices B and C] consists of a structural operational semantics [13, 5, 1], together with a bisimulation equivalence. In Section 3 we shall consider the configurations that arise in this operational semantics, paying particular attention to aspects supporting message passing and concurrent action performance. We shall also discuss how the asynchrony of message transmission and action performance is related to the straightforward ***distributed*** implementation of concurrent processing.

It is worth pointing out that the structural operational semantics of action notation induces an operational semantics for all languages described using action semantics. However, the induced semantics is not really structural in the usual sense, since configurations involve action terms rather than program syntax. Note that a structural operational semantics for a programming language usually involves repetitious patterns of rules for transitions, for instance determining a sequential order of execution of the components of various phrases; an action semantics for the language uses a single combinator to express the fundamental concept of sequencing, and the structural operational semantics of the combinator specifies the corresponding pattern of transitions, once and for all. Thus action semantics can be regarded as a technique for factorization of a conventional structural operational semantics.

Why isn't action notation defined denotationally? That would have the advantage of inducing denotational models for all languages with action semantic descriptions, as well as making domain theory available for reasoning about actions. The difficulty is that action notation involves concepts, such as concurrency and unbounded nondeterminism, whose available denotational models are not only very intricate but also not fully abstract with respect to the intended operational semantics of actions. Such a denotational 'model' would not satisfy all the desired algebraic laws. However, action notation could be exploited as auxiliary notation in a conventional denotational semantics [11].

On the other hand, although our combination of structural operational semantics and bisimulation does verify the essential algebraic laws, this does not provide a sufficiently strong action theory for reasoning about nontrivial program equivalence. It is currently unclear how to develop a stronger action theory, to avoid the need for direct and tedious reasoning at the operational level. In Section 4 we shall consider some possible directions for future research.

Readers are assumed to be familiar with the general ideas of denotational and structural operational semantics.

1. BASIC CONCEPTS

Just as the lambda-notation is used in denotational semantics for specifying functions, so our action notation is used in action semantics for specifying ***actions.*** Action notation includes also notation for ***data*** and for auxiliary entities called ***yielders.***

Actions are essentially dynamic, ***computational*** entities. The ***performance*** of an action directly represents information processing behaviour and reflects the gradual, step-wise nature of computation. Items of data are, in contrast, essentially static, ***mathematical*** entities, representing pieces of information, e.g.

particular numbers. (Of course actions are 'mathematical' too, in the sense that they are abstract, formally-defined entities, analogous to abstract machines defined in automata theory.) A yielder represents an ***unevaluated*** item of data, whose value depends on the ***current information,*** i.e., the previously-computed and input values that are available to the performance of the enclosing action. For example, a yielder might always evaluate to the datum currently stored in a particular cell, which could change during the performance of an action.

1.1. ACTIONS

A performance of an action, which may be part of an enclosing action, either:

- ***completes,*** corresponding to normal termination (the performance of the enclosing action proceeds normally); or
- ***escapes,*** corresponding to exceptional termination (parts of the enclosing action are skipped until the escape is trapped); or
- ***fails,*** corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- ***diverges,*** corresponding to nontermination (the enclosing action diverges).

Actions can be used to represent the semantics of programs: action performances correspond to possible program behaviours. Furthermore, actions can represent the (perhaps indirect) contribution that ***parts*** of programs, such as statements and expressions, make to the semantics of entire programs.

An action may be nondeterministic, having different possible performances for the same initial information. Nondeterminism represents implementation-dependence, where the behaviour of a program (or the contribution of a part of it) may vary between different implementations-or even between different instants of time on the same implementation. Note that nondeterminism does not imply actual randomness: an implementation of a nondeterministic behaviour may be absolutely deterministic.

The information processed by action performance may be classified according to how far it tends to be propagated, as follows:

- ***transient:*** tuples of data, corresponding to intermediate results;
- ***scoped:*** bindings of tokens to data, corresponding to symbol tables;
- ***stable:*** data stored in cells, corresponding to the values assigned to variables;
- ***permanent:*** data communicated between distributed actions that are performed by separate ***agents.***

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden temporarily. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made ***during*** action performance, and are unaffected by subsequent divergence, failure, or escape.

The different kinds of information give rise to so-called ***facets*** of actions, focusing on the processing of at most one kind of information at a time:

- the ***basic*** facet, processing independently of information (control flows);
- the ***functional*** facet, processing transient information (actions are ***given*** and give data);
- the ***declarative*** facet, processing scoped information (actions ***receive*** and ***produce*** bindings);
- the ***imperative*** facet, processing stable information (actions ***reserve*** and ***unreserve cells*** of storage, and ***change*** the data stored in cells); and
- the ***communicative*** facet, processing permanent information (actions ***send*** messages, ***receive*** messages in buffers, and offer ***contracts*** to ***agents).***

These facets of actions are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect any bindings. There are, however, some ***directive*** actions, which process a mixture of scoped and stable information, so as to provide finite representations of self-referential bindings. There are also some ***hybrid*** primitive actions and combinators, which involve more than one kind of information at once, such as an action that both reserves a cell of storage and gives it as transient data. In this paper, for simplicity, we ignore the directive facet of actions; we also ignore escapes (exceptional termination).

The notation for specifying actions consists of action ***primitives,*** which may involve yielders, and action ***combinators,*** which operate on one or two ***subactions.*** Action notation provides also some notation for specifing ***sorts*** of actions.

1.2. YIELDERS

Yielders are entities that can be ***evaluated*** to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage. In fact action notation provides primitive yielders that evaluate to compound data (tuples, maps, lists) representing entire slices of the current information, such as the current state of storage. Evaluation cannot affect the current information.

Compound yielders can be formed by the application of data operations to yielders. The data yielded by evaluating a compound yielder are the result of applying the operation to the data yielded by evaluating the operands. For instance, one can form the sum of two number yielders. Items of data are a special case of data yielders, and always yield themselves when evaluated.

1.3. DATA

The information processed by actions consists of items of data, organized in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, numbers, characters, strings, lists, sets, and maps. It can also include entities such as tokens and cells, used for accessing other items. Actions themselves are not data, but they can be incorporated in so-called ***abstractions,*** which are data, and subsequently ***enacted*** back into actions. (Abstraction and enaction are a special case of so-called ***reification*** and ***reflection.)*** New kinds of data can be introduced ***adhoc,*** for representing special pieces of information.

2. AN ILLUSTRATIVE EXAMPLE

Now that we have introduced the main concepts underlying action notation, let us take a walk through an illustrative action semantic description of a concurrent programming language, briefly indicating the intended interpretation of the notation that it uses as we go along. For a summary of the entire standard action notatio, see [12, Appendix D].

The language described here is a small-scale, 'ideal' programming language. Syntactically, it is a sublanguage of ADA (and of the language described in [12, Appendix A]), and the specined semantics is quite close to that indicated in the ADA Reference Manual.

The modular structure of our illustrative action semantic description is formally specified as follows.

Abstract Syntax

Semantic Functions

(needs: Abstract Syntax, Semantic Entities.)

Semantic Entities

Sorts	**(needs: Values, Variables, Tasks.)**
Values	**(needs: Numbers.)**
Variables	**(needs: Values, Types.)**
Types	.
Numbers	.
Tasks	.
Required Bindings	**(needs: Types, Numbers.)**

The action semantic description consists of three main modules, concerned with specifying abstract syntax, semantic functions, and semantic entities. Here, let us not worry about the formal details of modularization, and concentrate on the bodies of the modules.

2.1. ABSTRACT SYNTAX

The grammar-like specification given in this subsection consists mainly of a set of (numbered) equations. Ignoring the double brackets ⟦...⟧, the equations have the same form as *productions* in a particular variant of BNF grammar. Terminal symbols are written as quoted strings of characters, such as "(" and "or". Nonterminal symbols are written as unquoted words, such as **Expression,** and we adopt the convention that they generally start with a capital letter, to avoid confusing them with symbols for semantic functions and entities, which we write using lower case letters. There is a precise formal interpretation of a grammar as an algebraic specification of sorts of trees [12, Chapter 3]. Here, it is enough to know that occurrences of ⟦...⟧ indicate the construction of nodes of trees. (In denotational semantics such brackets merely separate abstract syntax from semantic notation, and cannot be nested.)

grammar:

(1) **Identifier** = ⟦ **letter (letter | digit)*** ⟧.

(2) **Literal** = ⟦ **digit+** ⟧.

The standard nonterminals **digit** and **letter** are always implicitly available in our grammars, for convenience when specifying the lexical syntax of identifiers and numerals. The terminal symbols that they generate are single characters, rather than strings of characters. (A string is simply a node whose branches are all characters.)

The equations above involve so-called *regular expressions.* In our notation, a regular expression is either a single symbol, or it consists of a *sequence* $\langle R_1 \ldots R_n \rangle$, a grouped set of *alternatives* $(R_1 | \ldots | R_n)$, an *optional* part $R^?$, an *optional repeatable* part R^*, or an *obligatory repeatable* part R^+.

(3) Expression = Literal | Identifier | ⟦ "(" Expression ")" ⟧ |
⟦ Unary-Operator Expression ⟧ |
⟦ Expression Binary-Operator Expression ⟧ .

Note that literals and identifiers are ***special cases*** (formally, ***subsorts)*** of expressions, rather than merely occurring as components of expressions.

We make no attempt to distinguish syntactically between expressions according to the sort of entity to which they evaluate: truth-values or numbers. Such distinctions between expressions would not simplify the semantic description at all, and they would in any case be context-dependent.

(4) Unary-Operator = "+" | "–" | "not" .
(5) Binary-Operator = "+" | "–" | "*" | "/" | "mod" |
"=" | "<" | "and" | "or" .
(6) Statement = ⟦ "null" ";" ⟧ | ⟦ Identifier ":=" Expression ";" ⟧ |
⟦ "if' Expression "then" Statement$^+$
"else" Statement$^+$ "end" "if' ";" ⟧ |
⟦ "while" Expression "loop" Statement$^+$ "end" "loop" ";" ⟧ |
⟦ "declare" Block ";" ⟧ | ⟦ Identifier "." Identifier ";" ⟧ |
⟦ "accept" Identifier ⟨ "do" Statement$^+$ "end" ⟩$^?$ ";" ⟧.

The statement ⟦ I_1 "." I_2 ⟧ here is a call on the entry I_2 of task I_1, to be matched by an accept statement for I_2 in the body declared for I_1.

(7) Block = ⟦ Declaration* "begin" Statement$^+$ "end" ⟧.

A block is essentially a statement with some local declarations. Following ADA, blocks can occur directly in ordinary statement sequences.

(8) Declaration = ⟦ Identifier ":" "constant" Identifier ":=" Expression ";" ⟧ |
⟦ Identifier ":" Identifier ":=" Expression ";" ⟧ |
⟦ "task" Identifier "is" Entry$^+$ "end" ";" ⟧ |
⟦ "task" "body" Identifier "is" Block ";" ⟧.
(9) Entry = ⟦ "entry" Identifier ";" ⟧.

Task entries are supposed to be declared before the corresponding task bodies, although we cannot insist on this in our context-free grammar. We retain the entries of a task head only for the sake of familiarity, as they are irrelevant to our dynamic semantics of tasks.

(10) Program = ⟦ Block "." ⟧
closed.

That concludes the specification of the abstract syntax of our illustrative language.

2.2. SEMANTIC FUNCTIONS

In action semantics, we specify ***semantic functions*** by ***semantic equations,*** much as in denotational semantics. Each equation defines the semantics of a particular sort of phrase in terms of the semantics of its components, if any, using constants and operations for constructing semantic entities. The required compositionality of semantic functions is generally apparent from the semantic equations.

A semantic function always takes a single, syntactic argument and gives a semantic entity as result. It is usual to specify the ***functionality*** of each semantic function. For instance,

evaluate _ :: Expression → action [giving a value]

asserts that for every abstract syntax tree E for an expression, the semantic entity evaluate E is an action which, when performed, gives a value. The actual definition of evaluate E by the semantic equations is then required to be consistent with this. Formally, action [giving a value] is a term denoting a sort of actions, as specified in [12, Appendix B].

The right hand sides of the semantic equations involve the standard notation for actions and data provided by action semantics, together with any further notation introduced for special semantic entities. It must be emphasized that all the notation is ***absolutely formal***! The fact that it is possible to read it informally-and reasonably fluently-does not preclude reading it formally as well. The grouping of the symbols might not be completely obvious to those who have not seen action notation before, but it is in fact unambiguous. The following hints about the general form of action notation may be helpful.

The standard symbols used in action notation are ordinary English ***words.*** In fact action notation mimics natural language: terms standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g., check it and then escape, whereas terms standing for data and yielders form noun phrases, e.g., the items of the given list. Definite and indefinite articles can be exploited appropriately, e.g., choose a cell then reserve the given cell. (This feature of action notation is reminiscent of Apple's HYPERCARD scripting language HYPERTALK [2], and of COBOL.)

These simple principles for choice of symbols provide a surprisingly grammatical fragment of English, allowing specifications of actions to be made fluently readable-without sacrificing formality at all! To specify grouping unambiguously, we may use parentheses, but for large-scale grouping it is less obtrusive to use indentation, which we emphasize by vertical rules, as illustrated in the semantic equations given later. Moreover, let infix operation symbols always associate to the left, with weaker precedence than prefix symbols (which in turn have weaker precedence than postfix symbols).

Compared to other formalisms, such as the so-called ***λ-notations,*** action notation may appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also take into account that each action combinator usually corresponds to a complex pattern of applications and abstractions in -notation. For instance, (under the simplifying assumption of determinism!) the action term A_1 then A_2 might correspond to something like $\lambda\varepsilon_1.\lambda\rho.\lambda\kappa.A_1\varepsilon_1\rho(\lambda\varepsilon_2.A_2\varepsilon_2\rho\kappa)$. In any case, the increased length of each symbol seems to be far outweighed by its increased perspicuity. It would also be rather misleading to use familiar mathematical signs to express actions, whose essence is unashamedly computational. For some applications, however, such as formal reasoning about program equivalence on the basis of their action semantics, optimal conciseness may be highly desirable, and it would then be appropriate to allow abbreviations for our verbose symbols. Note that the ***essence*** of action notation lies in the standard collection of primitives and combinators with their intended operational interpretation, rather than in the standard verbose symbols themselves.

The informal appearance and suggestive words of action notation should encourage programmers to read it, at first, rather casually, in the same way that they might read reference manuals. Having thus gained a broad impression of the intended actions, they may go on to read the specification more carefully, paying attention to the details. A more cryptic notation might discourage programmers from reading it altogether.

The intended interpretation of the standard notation for actions is specified operationally, once and for all, in [12, Appendix C] . All that one has to do before using action notation is to specify the information that is to be processed by actions, which may vary significantly according to the programming language being described. This may involve ***extending*** data notation with further sorts of data, and ***specializing*** standard sorts, using sort equations. Furthermore, it may be convenient to introduce formal ***abbreviations*** for commonly-occurring, conceptually-significant patterns of notation. Extensions, specializations, and abbreviations are all specified ***algebraically,*** as illustrated in Section 2.3.

Now let us begin to define the semantic functions for our illustrative language. We first declare the symbols used for the semantic functions.

introduces: the value of _, evaluate _,
the unary-operation-result of _, the binary-operation-result of _,
execute _, elaborate _, synchronize _, run _.

The place-holder _ indicates argument positions in operation symbols. For semantic function symbols, we keep to prefix notation, but otherwise we exploit infix and more generally,'mixfix' notation.

For simplicity, let identifiers be their own semantics. They are included in the sort token, which is specified in Section 2.3.1 to be a subsort of strings.

- the value of _ :: Literal → number .

The sort number is specified in Section 2.3.5 .

(1) the value of ⟦ *d*:**digit**$^+$ ⟧ = integer-number of decimal ⟦ *d* ⟧.

The operation decimal _ is a standard data operation on strings. We could define a corresponding semantic function, but it wouldn't be very exciting, so we take this short-cut. The use of ⟦...⟧ in the right hand side of the semantic equation above is atypical; it is needed because decimal _ expects its argument to be a string, not a tuple of characters.

The unbounded natural number returned by decimal ⟦ *d* ⟧ is mapped either to a bounded number, or to nothing (which is included in every sort of data and can be used to represent error values) by the operation integer-number of _, specified in Section 2.3.5 .

- evaluate _ :: Expression → action
  [giving a value]
  [using current bindings | current storage] .

The sort action [giving a value] includes those actions which, whenever performed, complete giving an individual of sort value as transient data; the performance must never give any other sort of transient data, produce any bindings, escape, or diverge. However, failure is ***always*** an implicit possibility (because actions that refer to current information generally fail when performed with no information available).

Similarly, action [using ...] includes actions that refer at most to the indicated kinds of information.

(2) evaluate *L*:**Literal** = give the value of *L* .

The primitive action give *Y* completes, giving the data yielded by evaluating the yielder *Y*.

(3) evaluate *I*:**Identifier** =
```
give the entity bound to I then
| give the given value or
| give the value assigned to the given variable .
```

The functional action combination A_1 then A_2 represents ordinary functional composition of A_1 and A_2: the transients given to the whole action are propagated only to A_1, the transients given by A_1 on completion are given only to A_2, and only the transients given by A_2 are given by the whole action. Regarding control flow, A_1 then A_2 specifies normal left-to-right sequencing.

The primitive action give *Y* fails when *Y* yields nothing. In the above equation, *Y* is the yielder the entity bound to *T*, which refers to the current binding for the particular token T, provided that there is one; otherwise it yields nothing, causing the giving action to fail.

The yielder given Y yields all the data given to its evaluation, provided that this is of the data sort Y. For instance the given value (where 'the' is optional) yields a single individual of sort value, if such is given. Otherwise it yields nothing, and give the given value fails. This causes the alternative currently being performed to be abandoned and, if possible, some other alternative to be performed instead, i.e., *back-tracking.*

The action A_1 or A_2 represents implementation-dependent choice between alternative actions, although here A_1, A_2 are such that one or the other of them is always bound to fail, so the choice is actually deterministic.

The special yielder the value assigned to Y, specified in Section 2.3,3, refers to the current storage for the particular variable yielded by Y, analogously to the entity bound to T. If I is currently bound to an entity that is neither a value nor a variable (e.g., a task) both alternatives fail, causing their combination to fail as well.

The special data sorts entity, value, and variable are specified in Section 2.3.

(4) evaluate $[\![$ "(" E:**Expression** ")" $]\!]$ = evaluate E.

(5) evaluate $[\![$ O:**Unary-Operator** E:**Expression** $]\!]$ =
    evaluate E then give the **unary-operation-result** of O.

(6) evaluate $[\![$ E_1:**Expression** O:**Binary-Operator** E_2: Expression $]\!]$ =
    (evaluate E_1 and evaluate E_2)
    then give the binary-operation-result of O.

The action A_1 and A_2 represents implementation-dependent order of performance of the indivisible subactions of A_1, A_2. When these subactions cannot 'interfere' with each other, as here, it indicates that their order of performance is simply irrelevant. Left-to-right order of evaluation can be specified by using the combinator A_1 and then A_2 instead of A_1 and A_2 above. In both cases, the values given by the subactions get *tupled,* and subsequently passed on by the combinator A_1 then A_2.

The evaluation of an expression may give any individual of sort value.We leave it to the semantics of operators, specified below, to insist on individuals of particular sorts-numbers, for instance. For simplicity, we do not bother with precise error messages in case the given operands are *not* of the right sort for a particular operator: we merely let the application of the corresponding operation yield nothing, so that the action which gives it must fail. In any case, errors arising due to wrong sorts of operands are statically detectable in most languages, and should therefore be the concern of a static semantic description, not of the dynamic semantics that we are developing here.

Note that we would *not* have to modify the above equation at all if we were to extend the example language so that expression evaluation could have 'side-effects', such as changing stored values or communicating. This is in marked contrast to the situation in denotational semantics.

- the unary-operation-result of _ :: **Unary-Operator** → yielder
  [of value] [using given value].

The notation for sorts of yielders is analogous to that for sorts of actions.

(7) the **unary-operation-result** of "+" = the given number.

(8) the **unary-operation-result** of "–" = the negation of the given number.

(9) the **unary-operation-result** of "not" = not the given truth-value.

Numerical operations such as negation _ and absolute _ are specified in Section 2.3.5 . The truth-values are the usual ones from our standard data notation, equipped with various logical operations, such as not _.

- the binary-operation-result of _ :: Binary-Operator → yielder
  [of value] [using given **value**^{2}].

(10) the binary-operation-result of "+" =
the sum of (the given **number#1**, the given **number#2**).

The yielder given $Y\#n$ yields the n'th individual component of a given tuple, for $n > 0$, provided that this component is of sort Y.

(11) the binary-operation-result of "–" =
the difference of (the given **number#1**, the given **number#2**).

(12) the binary-operation-result of "*" =
the product of (the given **number#1**, the given **number#2**).

(13) the binary-operation-result of "/" =
the quotient of (the given **number#1**, the given **number#2**).

(14) the binary-operation-result of "mod" =
the **modulo** of (the given **number#1**, the given **number#2**).

(15) the binary-operation-result of "=" =
the given **value#1** is the given **value#2**.

(16) the **binary-operation-result** of "<" =
the given **number#1** is less than the given **number#2**.

(17) the **binary-operation-result** of "and" =
both of (the given **truth-value#1**, the given **truth-value#2**).

(18) the **binary-operation-result** of "or" =
either of (the given **truth-value#1**, the given **truth-value#2**).

So much for the action semantics of expressions. Now for statements.

- execute_ :: Statement$^+$ → action
  [completing | diverging | storing | communicating]
  [using current bindings | current storage | current buffer] .

(19) execute ⟨ S_1:**Statement** S_2:**Statement**$^+$ ⟩ = execute S_1 and then execute S_2

The basic action combination A_1 and then A_2 combines the actions A_1, A_2 into a compound action that represents their normal, left-to-right sequencing, performing A_2 only when A_1 completes.

(20) execute ⟦ "null" ";" ⟧ = complete .

The primitive action complete is the unit for A_1 and then A_2.

(21) execute ⟦ I:**Identifier** ":=" E:**Expression** ";" ⟧ =
```
        | give the variable bound to I and
        | evaluate E
        then assign the given value#2 to the given variable#1.
```

The special action assign Y_1 to Y_2 is specified in Section 2.3.3.

(22) execute ⟦ "if" E:**Expression** "then" S_1:**Statement**$^+$
"else" S_2:**Statement**$^+$ "end" "if" ";" ⟧ =
```
        evaluate E then
        | | check the given truth-value and then execute S1
        | or
        | | check not the given truth-value and then execute S2.
```

The action check Y requires Y to yield a truth-value; it completes when the value is true, otherwise it fails. It is used for guarding alternatives. Here, the compound action (check Y and then A_1) or (check not Y and then A_2) expresses a deterministic choice between A_1 and A_2, depending on the condition Y. The transients given to the combination A_1 or A_2 are passed on to both its subactions; similarly for the action A_1 and A_2, and for A_1 and then A_2.

(23) execute ⟦ "while" E:**Expression** "loop" S:**Statement**$^+$ "end" "loop" ";" ⟧ =
```
        unfolding
        | evaluate E then
        | | | check the given truth-value and then execute S and then unfold
        | | or
        | | | check not the given truth-value .
```

The action combination unfolding A performs A but whenever it reaches the dummy action unfold, it performs A instead. It is mostly used in the semantics of iterative constructs, with unfold occurring exactly once in A, but it can also be used with several occurrences of unfold.

(24) execute ⟦ "declare" B:**Block** ";" ⟧ = execute B.

(25) execute ⟦I_1: Identifier "." I_2:Identifier ";" ⟧ =
give the agent bound to I_1 then
| send a message [to the given agent] [containing entry of I_2] and then
| receive a message [from the given agent] [containing the done-signal]

Task declarations bind task identifiers to agents, as specified later. They do not bind entry identifiers to anything at all, treating them literally as labels.

The primitive action send Y where Y yields a ***sort*** of message, initiates the transmission of a message. The usual form of Y is message [to Y_1] [containing Y_2], where Y_1 yields an individual ***agent*** and Y_2 yields individual data. The sort yielded by Y is implicitly restricted to messages from the performing agent and this should determine an individual message.

The action receive Y waits indefinitely for a message of the sort specified by Y to arrive, removes it from the buffer, and gives it.

The notation for entries and signals that are contained in the messages is specified in Section 2.3.

(26) execute ⟦ "accept" I:Identifier "end" ";" ⟧ =
receive a message [from any agent] [containing entry of] then
send a message [to the sender of the given message]
[containing the done-signal].

Synchronization is ensured by the entry call statement action waiting for the done-signal before completing. Our action semantics is merely expressing the usual informal explanation of the basic notion of a rendezvous in ADA. Extended rendezvous is just as straightforward:

(27) execute ⟦ "accept" I:Identifier "do" S:Statement$^+$ "end" ";" ⟧ =
receive a message [from any agent] [containing entry of I] then
| execute S and then
| send a message [to the sender of the given message]
| [containing the done-signal].

For simplicity, we do not include selection between alternative accept statements in the language described here. The action semantics of such constructs is given in [12, Chapter 17].

Although Block is not a subsort of Statement, let us overload the semantic function execute _ by extending it to blocks:

- execute_ :: Block → action
[completing | diverging | storing | communicating]
[using current bindings | current storage | current buffer].

(28) execute ⟦ "begin" S:Statement$^+$ "end" ⟧ = execute S.

(29) **execute ⟦ D:Declaration$^+$ "begin" S:Statement$^+$ "end" ⟧** =

```
        furthermore elaborate D hence
        | synchronize D and then
        | execute S.
```

The action furthermore A produces the same bindings as A, together with any received bindings that A doesn't override. In other words, it overlays the received bindings with those produced by A.

The combination A_1 hence A_2 lets the bindings produced by A_1 be received by A_2, which limits their scope—unless they get reproduced by A_2. It is analogous to functional composition. The compound combination furthermore A_1 hence A_2 (recall that prefixes have higher precedence than infixes!) corresponds to ordinary block structure, with A_1 being the block head and A_2 the block body: nonlocal bindings, received by the combination, are also received by A_2 unless they are overridden by the local bindings produced by A_1.

The action synchronize D above is concerned with task initialization, considered later. Now for declarations.

- elaborate_ :: Declaration+ → action
  [binding | diverging | storing | communicating]
  [using current bindings | current storage | current buffer] .

(30) elaborate ⟨ D_1: Declaration D_2: Declaration+ ⟩ =
  elaborate D_1 before elaborate D_2

The action A_1 before A_2 represents sequencing of declarations. Like furthermore A_1 hence A_2, it lets A_2 receive bindings from A_1, together with any bindings received by the whole action that are not thereby overridden. The combination produces all the bindings produced by A_2, *as well as any* produced by A_1 that are not overridden by A_2. Thus A_2 *may* rebind a token that was bound by A_1. Note that the bindings received by the combination are not reproduced at all, unless one of A_1, A_2 explicitly reproduces them.

The use of the combinator A_1 before A_2 in the semantics of declaration sequences allows later declarations to refer to the bindings produced by earlier declarations—but not the other way round. Mutually-recursive task declarations are considered later.

(31) elaborate ⟦ **I_1:Identifier** ":" "constant" I_2 dentifier ":=" E:Expression ";" ⟧ =
  evaluate E then bind I_1 to the given value .

The declarative action bind T to Y produces the binding of the token T to the bindable data yielded by Y. It does *not* reproduce any of the received bindings!

```
(32) elaborate ⟦ I₁: Identifier ":" I₂ :Identifier ":=" E:Expression ";" ⟧ =
        | allocate a variable for the type bound to I₂ and
        | evaluate E
        then
        | bind I₁ to the given variable#1 and
        | assign the given value#2 to the given variable#1.
```

The action allocate *d* for *Y* is special notation, specified in Section 2.3.3. As we only deal with simple variables in this simple example, allocate a variable for *Y* merely chooses, reserves, and gives a single storage cell.

The basic and functional combinators, such as A_1 and A_2, all pass the ***received*** bindings to their subactions without further ado—analogously to the way A_1 and A_2 passes all the given data to both A_1 and A_2. They are similarly unbiased when it comes to combining the bindings produced by their subactions: they produce the ***disjoint union*** of the bindings, providing this is defined, otherwise they simply fail. Here, one or the other of the combined actions never produces any bindings at all, so failure cannot arise.

```
(33) elaborate ⟦ "task" I:Identifier "is" E:Entry+' "end" ";" ⟧ =
        offer a contract [to any agent]
              [containing abstraction of the initial task-action] and  then
        | receive a message [containing an agent] then Y
        | bind I to the task yielded by the contents of the given message .
```

The primitive action offer *Y*, where *Y* yields a sort of contract, initiates the arrangement of a contract with another agent. The usual form of *Y* is a contract [to any agent] [containing abstraction of *A*], where *A* is the action to be performed according to the contract.

The action initial task-action is defined in Section 2.3.6.

```
(34) elaborate ⟦ "task"  "body" I:Identifier "is" B:Block ";" ⟧ =
        send a message [to the agent bound to I]
              [containing task of the closure of abstrraction of execute B].
```

The use of closure above ensures static bindings: the execution of the block ***B*** when the task is initiated receives the same bindings as the declaration. These may include bindings to other tasks: a system of communicating tasks can be set up by first declaring all the task entries, then all the bodies. They may also include bindings to variables; but attempts to assign to these variables, or to inspect their values, always fail, because the cells referred to are not local to the agent performing the action. It is currently a bit complicated to describe the action semantics of distributed tasks that have access to shard varibles the task that declares a variable has to act as a ***server*** for assignments and inspections—so we let our illustrative language deviate from **Ada** in this respect. We shall return to this matter in Section 3.

- synchronize _ :: Declaration+ → action
  [completing | diverging | communicating]
  [using current bindings | current buffer] .

The action synchronize *D* is used to delay the execution of the statements of a block until all the tasks declared locally in the block have been started.

(35) synchronize ⟨ D_1:**Declaration** D_2:**Declaration**$^+$ ⟩ =
synchronize D_1 and synchronize D_2 .

(36) synchronize ⟦ "task" "body" *I*:**Identifier** **"is"** *B*:**Block** **";"** ⟧ =
receive a message [from the agent bound to *I*]
[containing the begin-signal] .

(37) *D*: ⟦ Identifier ":" "constant"' Identifier ":=" Expression ";" ⟧ |
⟦ "task" Identifier "is" Entry+ "end" ";" ⟧ ⇒
synchronize *D* = complete .

The above conditional equation corresponds to several ordinary semantic equations.

Finally, we specify the action semantics of entire programs.

- run _ :: Program → action
  [completing | diverging | storing | communicating]
  [using current storage | current buffer] .

(38) run ⟦ *B*:**Block** **"."** ⟧ =
produce required-bindings hence
| execute *B* and then
| send a message [to the user-agent] [containing the terminated-signal] .

The primitive action produce *Y* produces a binding for each token mapped to a bindable value by the map yielded by *Y*. See Section 2.3.7 for the definition of the bindings of required identifiers in our illustrative language.

The termination message sent above insists that the user should be able to notice when the program has terminated.

Some evidence of the good pragmatic qualities (modinability, extensibility, comprehensibility) of action semantic descriptions may be observed in the semantic equations given above. In particular, notice how the ***polymorphism*** of the action combinators makes the well-formedness of the action terms independent of whether or not subactions might change storage, refer to bindings, communicate, etc.: our semantic equations would not need any significant modifikations when adding, say, function calls with side-effects to expressions.

2.3. SEMANTIC ENTITIES

To complete our semantic description of the illustrative language, we have to specify the notation that is used in the semantic equations for expressing semantic entities. Most of the notation used here has a fairly obvious interpretation, so rather few comments are provided.

includes: [12]/Action Notation.

2.3.1. SORTS

introduces: entity .

- entity = value | variable | type | task ***(disjoint)*** .
- datum = entity | message | entry | □.
- token = string of (letter, (letter | digit)*) .
- bindable = entity .
- storable = value.
- sendable = angent | task | entry | signal | □.

All the sorts specified above have a standard usage in action notation, except for entity. Although our sort equations look a bit like the so-called domain equations used in denotational semantics, their formal interpretation is quite different. We use the same symbol _|_ for ***sort union*** as we used for combining alternatives in grammars. Thinking of sorts of data as ***sets*** we may regard _|_ as ordinary set union; it is associative, commutative, and idempotent. The use of □ above formally expresses an inclusion, leaving open what other sorts might be included in datum and sendable.

2.3.2. VALUES

introduces: value .

includes: [12]/Data Notation/Instant/Distinction (value ***for*** s ,_ is _).

- value = truth-value | number ***(disjoint)***

2.3.3. VARIABLES

introduces: variable , assign _ to _, the _ assigned to _, allocate _ for _.

- assign _ to _ :: yielder [of value], yielder [of variable] → action [storing] .
- the _ assigned to _ :: value, yielder [of variable] → yielder [of value] .
- allocate _ for _ :: variable , yielder [of type] →
  action [giving a variable | [storing] .

(1) **variable = cell .**

(2) **assign (Y_1:yielder [of value]) to (Y_2:yielder [of variable]) =**
store the storable yielded by Y_1 in the cell yielded by Y_2.

(3) the **(v≤value)** assigned to (Y:**yielder** [of variable]) =
the (v & storable) stored in the cell yielded by Y.

(4) allocate **(v≤variable)** for (Y:**yielder** [of type]) =
allocate a cell .

The sort **cell** has a standard usage in action notation, corresponding to 'locations' in denotational semantics. For simplicity here, we do not bother to distinguish between cells for storing different sorts of values so the type entities are quite redundant. In a more realistic example, the specifkation of variable allocation and assignment can become quite complex.

The standard action store Y_1 in Y_2 changes the data stored in the cell yielded by Y_2 to the storable data yielded by Y_1. The cell concerned must have been previously reserved, using reserve Y otherwise the storing action fails. The standard yielder the d stored in Y evaluates to the data of sort d currently stored in the cell yielded by Y.

The standard notation allocate a cell abbreviates the following hybrid action:

```
indivisibly
| choose a cell [not in the mapped-set of the current storage] then
| | reserve the given cell and give it.
```

2.3.4. Types

introduces: type , boolean-type , integer-type .

- type = boolean-type | integer-type *(individual)* .

2.3.5. NUMBERS

introduces: number , min-integer , max-integer , integer-number of _ , negation _ , sum _ , difference _ , product _ , quotient _ , modulo _ .

- min-integer , max-integer : integer .
- integer-number of _ :: integer → number *(partial)*.
- negation _ :: number → number *(partial)*.
- sum _ , difference _ , product _ , quotient _ :: number^2 → number *(partial)*.
- modulo _ :: number^2 → number *(partial)*.
- _ is _ , _ is less than _ :: number, number → truth value *(total)* .

(1) i: integer [min-integer] [max max-integer] ⇒
integer-number of i: number .

(2) i: integer [min successor max-integer] ⇒ integer-number of i = nothing .

(3) i: integer [max predecessor min-integer] ⇒ integer-number of i = nothing .

(4) integer-number of i: number ⇒
negation integer-number of i = integer-number of negation i.

(5) integer-number of i_1: number ; integer-number of i_2: number ⇒

(1) sum (integer-number of i_1, integer-number of i_2) =
integer-number of sum (i_1, i_2);

(2) difference (integer-number of i_1, integer-number of i_2) =
integer-number of difference (i_1, i_2);

(3) product (integer-number of i_1, integer-number of i_2) =
integer-number of product (i_1, i_2);

(4) quotient (integer-number of i_1, integer-number of i_2) =
integer-number of integer-quotient (i_1, i_2);

(5) modulo (integer-number of i_1, integer-number of i_2) =
integer-number of integer-modulo (i_1, i_2).

(6) integer-number of i_1: integer-number ; integer-number of i_2: integer-number ⇒

(1) integer-number of i_1 is integer-number of i_2 = i_1 is i_2;

(2) integer-number of i_1 is less than integer-number of i_2 = i_1 is less than i_2.

The specification of integer arithmetic uses loosely-specified bounds on integers. It extends the standard arithmetic operations from standard integers to the sort number in a uniform way: the result is nothing when it would have been out of bounds.

2.3.6. TASKS

introduces: task, task of _, task-abstraction _, initial task-action , signal, begin-signal , done-signal , terminated-signal , entry , entry of _.

- task of _ :: abstraction → task *(total)*.
- task-abstraction _ :: task → abstraction *(total)* .
- signal = begin-signal | done-signal | terminated-signal *(individual)*.
- initial task-action : action.
- entry of _ :: token → entry *(total)*.

```
(1)  t = task of a ⇒ task-abstraction t:task = a .

(2) initial task-action =
        | send a message [to the contracting-agent]
        |      [containing the performing-agent]
        | and then
        | receive a message [from the contracting-agent] [containing a task]
        then
        | send a message [to the contracting-agent] [containing the begin-signal]
        | and then
        | enact the task-abstraction of the task yielded by
        |      the contents of the given message .

(3)  entry of k1:token is entry of k2:token = k1 is k2 .
```

The action enact Y performs the action incorporated in the abstraction yielded by Y. The use of closure _ on an abstraction ensures that the incorporated action receives whatever bindings were current when the closure was evaluated.

2.3.7. REQUIRED BINDINGS

introduces: required-bindings .

- required-bindings : map [token to value | type].

```
(1) required-bindings =
        disjoint-union of ( map of "TRUE" to true,
                            map of "FALSE" to false,
                            map of "BOOLEAN" to boolean-type,
                            map of "MININT" to integer-number min-integer,
                            map of "MAXINT" to integer-number max-integer,
                            map of "INTEGER" to integer-type ).
```

3. FOUNDATIONS

Now that we have seen the use of action notation in the semantic description of a simple concurrent programming language, let us consider the operational semantics of action notation. We shall pay particular attention to communicative actions, i.e., actions for sending and receiving messages and for offering contracts.

The operational semantics of action notation [12, Appendix C] uses a variant of structural operational semantics [13, 5, 1] to define a transition system. Sequences of transitions correspond to performances of actions, representing program behaviour. The transition system is the basis for defining action equivalence; see [12, Section C.4].

Here we shall consider mainly what ***configurations*** arise in the operational semantics of action notation. Once one has seen that, it should be fairly easy to imagine how particular primitive actions and combinators determine transitions between configurations.

To start with, suppose that a single agent is performing an action in isolation, without any message transmission. The relevant components of the current configuration are just the (abstract) syntax of the rest of the action being performed, together with the current transient data, bindings, and storage. Actually, due to the interleaving of steps in the performance of the combination A_1 **and** A_2, various subactions may have different current transients and bindings at the same time, and it is easiest to keep track of this by inserting transients and bindings directly in the abstract syntax tree of the action. On the other hand, an agent only has one current storage, which is best kept separate from the syntax tree.

Next, let the action being performed involve the sending and receipt of messages. The current configuration should now record what messages have been sent (at least since the previous connguration) and the messages that have been received but not yet removed, i.e., the current buffer. We may imagine the single agent making transitions between such local conngurations. Between transitions, any messages to be sent get dispatched, and fresh messages may be inserted in the buffer. Transitions may inspect and remove messages from the buffer, but not add any new ones.

Finally, consider a (conceptually) ***distributed*** collection of concurrent agents, each performing its own action by making transitions between local configurations as described above. Now the only messages that get inserted in the buffer of a particular agent are supposed to be those messages that have been sent to that agent by other agents in the system; moreover, all messages that get sent are supposed to arrive, sooner or later. A global connguration of the system of agents is essentially a map from the agents to their local configurations; this can be represented as a set of local conngurations, provided the identity of the performing agent is a component of each local connguration.

We are not to make any assumptions at all about the relative processing speeds of different agents—not even that they are stable.[1] Thus it would ***not*** be appropriate for a global transition to consist a local transition for ***each*** agent in the system. Nor would it be satisfactory for each global transition to consist of local transitions for an arbitrary subset of the agents, for then some particular agent might never be included, whereas all action performances are supposed to proceed concurrently.

The simplest way to represent arbitrary relative processing speed while ensuring that all agents eventually make transitions (until there is nothing more

[1]However, we exclude the possibility that one agent can make infinitely many transitions while another agent makes only one!

of their actions to perform, of course) seems to be the following: associate an arbitrary finite ***delay*** with each local transition when it is made; then let each global transition reduce all the delays by one unit, and make new local transitions for all agents whose transition delay has become zero. (It is convenient, but not essential, to let delays be positive integers.)

Similarly we may represent the time-consuming nature of physical message transmission by attaching arbitrary finite delays to messages when they are sent, only inserting them in the buffer of the receiving agent when the delay has been reduced to zero.

This technique is analogous to the way that one can represent ***fairness*** in terms of unbounded nondeterministic choice. The nondeterminism that arises is enormous; in many cases it is also irrelevant, in the sense that the overall message-passing and termination behaviour of a system of agents may be independent of the particular delays chosen for local transitions and message transmissions. The appeal of our operational semantics lies in the directness with which it represents the arbitrary processing speeds of agents.

Notice that each local transition involved in a global transition is determined exclusively by the corresponding local configuration—not by some global property of the entire set of local configurations. This suggests that a distributed implementation of multi-agent action performance could be obtained straightforwardly from implementations of single-agent performances. Models for concurrency based on ***synchronous*** communication, such as CCS [7,8,9] and CSP (with output guards) [3,4] can be surprisingly difficult to implement on a distributed system—without introducing centralistic arbiters, that is.

The above considerations have not addressed the question of how a system of agents gets initialized, with the identities of particular agents known to other agents so as to permit direct communication between them. In fact it is sufficient to start from a single active agent, the so-called user-agent, all other agents being initially inactive. An active agent can offer a ***contract*** that incorporates an action; some inactive agent can accept the contract, whereupon it starts performing the incorporated action. As illustrated in Section 2, one can make the incorporated action report back the identity of its performing agent to its contracting agent (i.e., the sender of its contract) and then wait for a message containing an abstraction to be enacted. By offering several such contracts, binding tokens to the reported agent identities, and forming closures from the abstractions subsequently sent, each agent involved acquires knowledge of the other agents' identities, so that direct communication between them is possible.

Contracts get arbitrary finite delays, just as messages do. In [12] the agent that accepts a contract is chosen from those inactive when the delay on the contract becomes zero. But this involves some global knowledge. To eradicate this remaining trace of synchrony, one could let a contract be offered to each agent in some arbitrary order, with a new delay each time the agent is found to be active, until an inactive agent is found—if ever!

Note that agents correspond more to process activation identities than to processors, and cannot be recontracted: once active, they remain active. This avoids confusion about what to do with messages where the target agent gets recontracted between the sending and receipt of the message.

Message buffers are supposed to be unbounded. If they were bounded, messages that arrived when the target buffer happened to be full would presumably disappear, whereas with the present semantics of action notation, all messages are assumed to arrive safely. Incidentally, to prevent confusion between messages with identical contents sent between the same two agents, each message gets a local serial number, whose latest value is a component of the local configuration of the agent sending the message.

As mentioned in Section 2, each agent has its own local storage, which cannot directly be allocated, updated, or inspected by other agents. Shared storage, which would be needed for the action semantics of tasks in full ADA (and which is also convenient for describing *threads*) can be represented by introducing an auxiliary 'server' agent, contracted to wait for messages from 'client' agents instructing it how to act on its own local storage. Although this representation of shared storage corresponds quite well to conventional computer architecture, it is desirable to provide direct support for the concept of shared storage. It now seems possible to achieve this by a very minor extension to action notation, where the corresponding changes to the operational semantics of action notation do not invalidate the established laws of action equivalence. This extension is joint work with Martín Musicante, and we hope to report on it in detail in the near future.

4. CONCLUSION

We have looked at an action semantic description of a simple programming language that includes constructs for synchronization between concurrent tasks. The presence of concurrency does not affect the description of the other constructs at all—in sharp contrast to the situation with conventional denotational descriptions, where the domains of higher-order functions used to model concurrency and nondeterminism are radically different from those normally used to model sequential computation.

The action semantics of an ordinary rendezvous between tasks is easily expressed by a simple pattern of asynchronous message passing. It is much more complicated to give an action semantics for languages like CCS and CSP, where commitment to one synchronization possibility between two processes can exclude other possibilities—also between other processes. When processes are represented by agents, one is forced to explicate *how* a commitment made by one agent gets communicated to the other agents. But perhaps this difficulty merely reflects the fact that CCS and CSP are abstract specification languages, rather than realistic programming languages for *distributed* processing, where communication delays can be significant.

Further experiments with the action semantic description of concurrent programming languages are needed, to test the adequacy of the communicative part of action notation. We have already discussed the desirability of extending action notation with direct support for shared storage. Other features that are not so easy to represent directly in the current notation include interrupts and time-outs; some preliminary investigations addressing these topics were reported in [6].

Finally, it remains to develop a decent *theory* for reasoning about equivalence between communicative actions. The current theory of action notation [12, Appendix B] is quite weak, and doesn't provide any useful equivalences between systems of communicating agents.

The author welcomes collaboration on all aspects of the development of action semantics. The e-mail address for e-mail is pdmosses@daimi.aau.dk.

Acknowledgments David Watt collaborated on the development of action notation. The work reported here has been partially funded by the Danish science Research Council project DART (5.21.08.03).

References

[1] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts,* IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.

[2] D. Goodman. *The Complete HyperCard Handbook.* Bantam, 1987.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM,* 21:666–677, 1978.

[4] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[5] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science,* number 247 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[6] P. Krishnan and P. D. Mosses. Specifying asynchronous transfer of control. In *RTFT'92, Proc. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Delft,* number 571 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[7] R. Milner. *A* ***Calculus of Communicating Systems.*** Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[8] R. Milner. ***Communication and Concurrency.*** Prentice-Hall, 1989.

[9] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, ***Handbook of Theoretical Computer Science,*** volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[10] 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 B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[11] P. D. Mosses. A practical introduction to denotational semantics. In E. J. Neuhold and M. Paul, editors, ***Formal Description of Programming Concepts,*** IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.

[12] P.D. Mosses. ***Action Semantics.*** Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[13] G.D. Plotkin A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Computer Science Dept., Aarhus University, 1981 Now available only from University of Edinburgh.

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