BRICS RS-99-56 P. D. Mosses: A Modular SOS for Action Notation

BRICS

Basic Research in Computer Science

A Modular SOS for Action Notation

Peter D. Mosses

BRICS Report Series RS-99-56

ISSN 0909-0878 December 1999

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/99/56/

A Modular SOS for Action Notation *

Peter D. Mosses[1]

BRICS and Department of Computer Science,
University of Aarhus, Denmark

Abstract. Modularity is an important pragmatic aspect of semantic descriptions: good modularity is needed to allow the reuse of existing descriptions when extending or changing the described language. In denotational semantics, the issue of modularity has received much attention, and appropriate abstractions have been introduced, so that definitions of semantic functions may be independent of the details of how computations are modelled. In structural operational semantics (SOS), however, this issue has largely been neglected, and SOS descriptions of programming languages typically exhibit rather poor modularity; the original SOS given for Action Notation (the notation for the semantic entities used in action semantics) suffered from the same problem.

This paper recalls a recent proposal, called MSOS, for obtaining a high degree of modularity in SOS, and presents an MSOS description of Action Notation. Due to its modularity, the MSOS description pin-points some complications in the design of Action Notation, and should facilitate the design of an improved version of the notation. It also provides a major example of the applicability of the MSOS framework.

The reader is assumed to be familiar with conventional SOS and with the basic concepts and constructs of Action Notation. The description of Action Notation is formulated almost entirely in CASL, the common algebraic specification language.

1 Background

This section recalls the main features of MSOS [11], CASL [3, 9], and Action Notation [7]. Subsequent sections introduce and discuss the MSOS of Action Notation, which is provided in the appendices.

1.1 Modular SOS

Conventional SOS [1, 15] involves abstract syntax, computed values, configurations (some of which may be distinguished as terminal), and inference rules for (labelled) transitions. An SOS specifies a labelled transition system $(\Gamma, T, \mathbb{A}, \rightarrow)$, where Γ is the set of *configurations*, $T \subseteq \Gamma$ is the set of *terminal configurations*,

* Full version of [13], reporting research carried out while visiting SRI International and Stanford University, USA

$\mathbb{A}$ is the set of *labels*, and $\rightarrow \subseteq \Gamma \times \mathbb{A} \times \Gamma$ is the *transition relation*. For configurations $\gamma, \gamma' \in \Gamma$ and labels $\alpha \in \mathbb{A}$, the assertion that $(\gamma, \alpha, \gamma')$ is in the transition relation is written $\gamma \xrightarrow{\alpha} \gamma'$.

Modular SOS, abbreviated MSOS [11], is a particularly simple and uniform discipline of SOS with the following features:

- Configurations $\gamma \in \Gamma$ are restricted to abstract syntax trees (where nodes may be replaced by the values that they have computed, as in conventional SOS).
- Initial configurations are pure syntax, and terminal configurations are simply computed values.
- All the usual semantic components of configurations (such as environments and stores) are incorporated in the labels $\alpha \in \mathbb{A}$ on transitions.
- The labels on transitions are equipped with a partial composition operation, written $\alpha\,\boldsymbol{;}\,\alpha'$ (associative whenever the composition is defined), and each label can always be composed on the left and right with identity labels $\iota \in \mathbb{I}[\mathbb{A}]$. The labels $\alpha \in \mathbb{A}$ are considered to be the arrows of a category, also written $\mathbb{A}$. The objects $o \in \mathbb{O}[\mathbb{A}]$ of the category correspond to the usual semantic components of configurations; let us refer to them as *states*.
- Transitions $\gamma_{1} \xrightarrow{\alpha_{1}} \gamma'_{1}$ and $\gamma_{2} \xrightarrow{\alpha_{2}} \gamma'_{2}$ may be adjacent in a computation only when $\gamma'_{1} = \gamma_{2}$ and moreover the composition $\alpha_{1}; \alpha_{2}$ of their labels is defined.
- The actual representation of the labels α is abstracted from the rules that define the transition relations, allowing the former to be changed without invalidating the latter.

1.2 Label Categories

Label categories are defined succinctly using three standard label transformers, which correspond to some simple monad transformers. The following three label transformers, enriching label categories with further labels and states, are fundamental:

- Context_Info adds an extra component of a particular sort both to labels and to states, and its value is preserved by the *pre* and *post* operations. The composition $\alpha; \alpha'$ is defined only when the new component has the same value in both α and α', and the composition preserves that value. This transformer is typically used for dealing with environments.
- Mutable_Info adds an extra component to states, and a *pair* of extra components (of the same sort) to labels, corresponding to the components of their *pre* and *post* states. The composition $\alpha; \alpha'$ is defined only when this component has the same value in both $post(\alpha)$ and $pre(\alpha')$. This transformer is typically used for dealing with stores.
- Emitted_Info adds an extra component only to labels. The composition $\alpha; \alpha'$ combines the values of this component in α and α' using the operations of a given monoid. This transformer is typically used for dealing with output, the given monoid then being sequences with their concatenation.

The notation associated with the above label transformers is specified generically in CASL in Appendix B. It includes the operations *set*, for initializing or overwriting a particular component of a label or state, and *get*, for returning the value of a particular component (or a default value, if that component has not been set). Also the operations *get_pre* and *set_post* are provided in the case of MUTABLE_INFO, to avoid having to deal with pairs explicitly.

1.3 CASL Specifications

For defining abstract syntax, values, configurations, the notation used for labels, and transition relations, it is convenient to use CASL, the Common Algebraic Specification Language [3, 9]. CASL is quite expressive, providing direct support for specifying sort inclusions, partial operations, predicates, definedness assertions, and first-order axioms. CASL also provides datatype declarations (resembling grammars in BNF) that allow sorts equipped with constructors and selectors to be specified concisely. For structuring specifications, CASL provides union, extension, free extension (with initiality as a special case) and generic specifications. CASL does not allow the specification of inference rules for transitions, but we may write SOS transition rules as implications in CASL; the least relation satisfying the implications is obtained by letting the specification of transitions be a free extension.

Action Notation incorporates Data Notation [7, App. E], which provides various familiar datatypes: truth-values, numbers, characters, strings, lists, trees, sets, and finite maps, as well as some that are more closely connected with actions: data tuples, bindings, tokens, stores, cells, and agents. Data Notation is specified algebraically in the framework of Unified Algebras [5, 6]. Action Notation does not depend on the way that data is specified, except that a few primitive actions and yielders do require *sorts* of data as arguments (e.g., the action written 'choose natural' gives an arbitrary element of the sort natural), which is not allowed by CASL . To specify Data Notation in CASL, sorts that are to be used as arguments have to be represented by ordinary constants (or terms).

In fact the unified algebra treatment of sorts as values in a universe *Univ* can easily be simulated in CASL by distinguishing a subsort of 'individual' values $Indiv < Univ$, and declaring suitable operations and relations on *Univ*. The constant $nothing : Univ$ corresponds to an empty subsort of *Univ*. The unified algebra operations of sort union $\_\_|\_\_$ and intersection $\_\_\&\_\_$ are provided as ordinary operations on *Univ*, whereas the unified algebra subsort inclusion $\_\_ \leq \_\_$ and individual inclusion $\_\_ :< \_\_$[1] are simply binary predicates in CASL. The predicate $u :< s$ holds iff the value u is both in *Indiv* and in the subsort represented by the value s. For instance, the unified algebra sort data is represented in CASL by declaring the subsorts $Data < Indiv$ and $DataSort < Univ$, and the constant $data : DataSort$, with $d : Data \iff d :< data$. The full properties of the general unified algebra notation are specified in CASL in Appendix C.

[1] The unified algebra notation '$\_\_ : \_\_$' cannot be declared as a symbol in CASL.

Furthermore, CASL specifications of various basic abstract datatypes have recently been proposed [16], subsuming much of the standard Data Notation.

Therefore we may employ CASL for specifying both Action Notation (operationally, in the MSOS style) and Data Notation (algebraically), and avoid any direct involvement of the Unified Algebras framework in the foundations of Action Semantics.

The only feature of Action Notation that cannot be specified directly in (first-order) CASL is that all data operations are supposed to be implicitly extended to yielder arguments. Here, we give a schematic specification of this lifting; a fully formal treatment would involve the use of higher-order CASL [4].

1.4 Action Notation

Action Notation is a rich algebraic notation for expressing actions, which are used (along with data, and 'yielders' of data) to represent the semantics of constructs of conventional programming languages. 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: each step of an action performance may access and/or change the current information. Yielders occurring in actions may access, but not change, the current information. The evaluation of a yielder always results in a data entity (including a special entity used to represent undefinedness). 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, and become undefined when the cell is freed.

A performance of an action either: *completes*, corresponding to normal termination; or *escapes*, corresponding to exceptional termination; or *fails*, corresponding to abandoning an alternative; or *diverges*.

Action notation consists of several rather independent parts, corresponding to the following so-called 'facets' of information processing:

Basic: for specifying the flow of control in actions;
Functional: for specifying the flow of the data that are given to and by actions;
Declarative: for specifying the scopes of the bindings that are received and produced by actions;
Reflective: for specifying procedural abstraction and application;
Imperative: for specifying the allocation of storage for the values of variables; and
Communicative: for specifying (asynchronous) message passing.

Compound actions are formed from *primitive actions* and action *combinators*. Each primitive action is single-faceted, affecting information in only one facet—although any yielders that it contains may refer to all kinds of information. An action combinator determines control and information flow for each facet of the combined actions, allowing the expression of multi-faceted actions, such as an action that both (imperatively) reserves a cell of storage and then (functionally) gives the identity of the reserved cell. For instance, one combinator determines left-to-right sequencing together with left-to-right transient data flow, but letting the received bindings flow to its sub-actions; another combinator differs from that only regarding data flow: it concatenates any transients that the sub-actions give when completing, not passing transients between the actions at all. Some selections of control and information flow are disallowed, e.g., interleaving together with transient data flow between the interleaved sub-actions. In particular, imperative and communicative information processing always follows the flow of control.

Further informal explanation of the design of Action Notation may be found in the main sources for action semantics [7, 8, 17].

2 Introduction to the MSOS of Action Notation

The intended interpretation of Action Notation was originally defined [7, App. C] using a rather unorthodox style of SOS, exploiting the novel algebraic specification framework of Unified Algebras [5, 6]. The main features of unified algebras are that operations can be applied to, and return, entire sorts, and that individual values are regarded as singleton sorts. Transition relations can thus be represented as functions that map individual configurations to entire *sorts* of configurations (representing the sets of alternative transitions).

Unfortunately, the unorthodox style of the original SOS of Action Notation, combined with the unfamiliarity of Unified Algebras, made the specification somewhat inaccessible. Its lack of modularity also meant that even minor changes to Action Notation (or extensions of it, such as the proposal to allow agents to share storage [14]) might require a major reformulation of the given SOS. Moreover, to decrease the size of the description, the full Action Notation was reduced to a substantially-smaller kernel notation (by means of algebraic equations), and only the latter was given a direct operational semantics.

Appendix A of this paper gives an MSOS for all of Action Notation. It is structured in much the same way as [7, Apps. B and D], describing the various facets of Action Notation in turn; however, the semantics of each construct is here specified directly, without resort to an intermediate kernel notation.

Each section of the MSOS specifies the data notation, abstract syntax, computed values, configurations, label notation, and transition rules for the action notation in the facet concerned. The following explanatory comments apply to all the sections.

2.1 Data

Data notation is specified by reusing abstract datatypes that are already available, perhaps with renaming or instantiation of generic specifications and adding declarations and axioms for new notation. For instance:

spec BASIC_DATA =
  TRUTH_VALUES
    with $\mathit{Truth\_Value}, \mathit{true\_value}, \mathit{false\_value}, \mathit{either}$
  and
    sorts $\mathit{Data} < \mathit{Indiv}; \quad \mathit{DataSort} < \mathit{Univ}$

The symbols listed above after '**with**' are assumed to be declared by the CASL specification of TRUTH_VALUES (which uses slightly different identifiers than those in [7, App. E], to avoid confusion with the reserved CASL predicate symbols *true* and *false*). Many of the symbols of Data Notation are not valid CASL symbols, but generally become so once internal spaces and hyphens have been replaced by underscores.

As mentioned earlier, it is envisaged that the standard Data Notation used in Action Semantics may be replaced by a library of CASL specifications, perhaps incorporating the basic CASL datatype specifications that have recently been proposed [16].

By the way, only the data notation actually needed for the MSOS of Action Notation is specified in Appendix C. In particular, the declarations of constants such as $\mathit{data} : \mathit{DataSort}$, representing proper sorts in unified algebras, are omitted, since assertions such as $d :< \mathit{data}$ can be expressed equivalently as $d \in \mathit{Data}$, and $\mathit{ds} \leq \mathit{data}$ as $\mathit{ds} \in \mathit{DataSort}$.

2.2 Syntax

Abstract syntax is specified in CASL using a datatype declaration, which resembles a BNF-like grammar. Mixfix notation is allowed—for instance, the following fragment specifies ***and*** as an infix operation:

spec BASIC_SYNTAX =
  BASIC_DATA **then**
    types $\mathit{Action} ::= \ldots \mid \_\_\boldsymbol{and}\_\_(\mathit{Action}; \mathit{Action}) \mid \ldots \;;$
    $\qquad\quad \mathit{Yielder} ::= \ldots \mid \mathit{sort}\ \mathit{DataSort} \mid \ldots$

The abstract syntax for actions and yielders extends the associated data notation, and data components are regarded as already evaluated.

It is possible to specify a syntactic congruence by adding axioms to the given datatype declarations, for instance asserting that A_1 ***and*** A_2 = A_2 ***and*** A_1, thereby reducing the need for various symmetric pairs of inference rules when specifying the transition relation.

By the way, several of the words used in Action Notation, such as '*and*', are reserved keywords in CASL, and cannot be complete tokens in CASL input symbols. So-called display annotations (not shown here) allow them to be produced in the formatted specification (using a distinct font, as in '***and***', to avoid confusion between symbols and keywords).

One might expect the types for the abstract syntax of actions and yielders for each facet of Action Notation to be specified as '**free**', to ensure that there can be no syntactic 'junk' (i.e., all syntactic values can be expressed by the declared constructors) nor 'confusion' (i.e., different terms denote different syntactic values, up to syntactic congruence) in models of the specification. However, that would prevent the subsequent combination of facets (as well as the extension of abstract syntax to configurations, see below). Instead, a free extension is specified *after* the facets have been combined.

2.3 Outcomes

The values that may be computed by action performance (and yielder evaluation) are specified algebraically in CASL, by declaring sorts, operations, and predicates, and asserting their essential properties. The specifications often use datatype declarations for conciseness. For instance:

spec FUNCTIONAL_OUTCOMES =
  BASIC_OUTCOMES **and**
  FUNCTIONAL_DATA
  then
    types *Terminated* ::= *sort Completed* | ... ;
        Completed ::= *completed* | *gave*(*Data*)
    axioms
    %[1]   *gave*(*none*) = *completed*;
    ...

2.4 Configurations

The 'value-added' syntax used for configurations is specified simply by adding further alternatives for the datatype declarations which specified abstract syntax: for each sort of the abstract syntax, the sort of value computed by elements of that sort is included as a subsort. Auxiliary syntactic constructs for use in configurations may be added here too.

In fact the configurations for non-distributed action performance are always the same, as specified by:

spec BASIC_CONFIGURATIONS =
  BASIC_SYNTAX **and**
  BASIC_OUTCOMES
  then
    type *Action* ::= *sort Terminated* | __ @ __(*Action*; *Action*)

The sort *Terminated* (of values computed by actions) depends on the facet. (The auxiliary construct A_1 @ A_2 is used only in the basic facet, in connection with unfolding.)

The distributed performance of communicative actions by separate agents is described by embedding *Action* in an auxiliary sort of configurations, *Processing*, which allows collections of agents (with their actions), pending messages, and contracts all to be composed in parallel.

The datatype declaration for *Action* above augments the constructors for this sort, which is left loosely specified in BASIC_SYNTAX.

2.5 Labels

Each facet of Action Notation generally requires the transformation of the category of labels $\mathbb{A}$ to include one or more further components. This is specified concisely in CASL by instantiating one of the generic specifications corresponding to the three fundamental kinds of enrichment described in Section 1.2. For example, the functional facet specifies:

> **spec** FUNCTIONAL_LABELS =
> BASIC_LABELS **and**
> FUNCTIONAL_DATA
> **then**
> CONTEXT_INFO
> [**sort** $\mathbb{A}$] [**op** *data* : *Index*]
> [**sort** *Data* < *ContextInfo* **op** *none* : *Data*]

which defines the operation $set(\alpha, data, d)$ to return a label α' with *data* component d, and the operation $get(\alpha, data)$ to return the *data* component of d, if defined (otherwise *none*).[2] The values of sort *Index* (such as *data*) may be thought of as selection indices; their only property is that different constants denote distinct values.

The fitting morphisms from the parameter specifications of CONTEXT_INFO to the argument specifications above are uniquely determined, and may therefore be left implicit.

2.6 Transitions

Transition rules are of three main kinds:

- Rules that allow performance of a compound construct to start (or continue) with a particular sub-construct: a transition for the sub-construct gives rise to a transition for the enclosing construct, often with the same unrestricted label α. For instance, the following rules allow interleaved performance of A_1 ***and*** A_2:

$$\%\% \frac{A_1 \xrightarrow{\alpha} A'_1}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A'_1 \textbf{\textit{ and }} A_2;} \%\% \Rightarrow$$

$$\%\% \frac{A_2 \xrightarrow{\alpha} A'_2}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A_1 \textbf{\textit{ and }} A'_2} \%\% \Rightarrow$$

  (The line between the conditions and the conclusion is not part of CASL notation, and has to be enclosed in comment signs '%%'.)
- Rules that specify the computation of a value by an atomic construct: the label on the transition is generally well-determined by the current state. For instance, the following rule lets the value computed by ***regive*** depend on the current state, which is not changed by the identity ι:

[2] $set(\alpha, data, d)$ might be written even more suggestively as $\alpha[data := d]$, and $get(\alpha, data)$ as $\alpha.data$.

$$\%\% \frac{d = get(\iota, data)}{\textbf{\textit{regive}} \xrightarrow{\iota} gave(d)} \%\% \Rightarrow$$

- Rules that reduce a compound configuration: once one or more components of a compound construct have computed values, the construct may be 'silently' reduced to a single computed value or syntactic component, the label on the transition being an identity ι. For instance, the following rule combines the values computed by performing the sub-actions of A_1 ***and*** A_2:

$$gave(d_1)\ \textbf{\textit{and}}\ gave(d_2) \xrightarrow{\iota}$$
$$gave(concatentation(d_1, d_2))$$

An action is regarded as 'incorrect' when its performance can get stuck, i.e., lead to a configuration (other than a computed value) from which there is no further transition. For example, the action '***check abstraction_of*** *A*' is incorrect, since transitions are possible for '***check*** *tv*' only when $tv \in Truth\_Value$. The question of whether or not an arbitrary action is 'correct' is undecidable; a static semantics using type inference for action notation could however provide a useful decidable safe approximation to this notion.

The mathematical nature of the evaluation of yielders to data (sorts or individuals) is reflected by the labels on the transitions always being identities ι:

$$Y \xrightarrow{\iota} ds.$$

In general, the evaluation of yielders in a primitive action may be done in any order, and the result is independent of the chosen order. (Primitive actions are supposed to be indivisible, so a small-step gradual evaluation of yielder arguments would be incorrect.)

The ordinary transitive closure $\xrightarrow{\alpha}^{+}$ of $\xrightarrow{\alpha}$ is used in the rule for indivisible actions; its inductive definition is standard:

$$\%\% \frac{A \xrightarrow{\alpha} A'}{A \xrightarrow{\alpha}^{+} A'} \%\% \Rightarrow$$

$$\%\% \frac{A \xrightarrow{\alpha'} A' \wedge A' \xrightarrow{\alpha''}^{+} A'' \wedge \alpha = \alpha'; \alpha''}{A \xrightarrow{\alpha}^{+} A''} \%\% \Rightarrow$$

It is occasionally convenient to abbreviate two rules with the same conclusion by use of a single rule that has a disjunction of conditions. (Casl requires the intended grouping of a mixture of conjunctions and disjunctions to be made explicit, so there can be no doubt about the expansion of such an abbreviated rule.)

3 Discussion

The full MSOS of Action Notation is about 25 pages long, which is roughly twice as long as the original SOS for the kernel of Action Notation. The main reason for this expansion is not so much the difference in size between the kernel and full Action Notation, but more that the author went to great pains to achieve brevity in the original SOS. For instance, various subsorts that corresponded to restrictions of the original grammar were used—such subsorts are easy to express with the sort union operation of unified algebras. Auxiliary operations, effecting internal simplifications of the configuration, were introduced. Each combinator was classified into subsorts, e.g., according to whether it was sequential or interleaving; this allowed transitions to be specified for many combinators at once, rather concisely. Although such techniques might also be applicable in the MSOS of Action Notation, they would tend to undermine its modularity, and make it more difficult to cut down the description when removing entire facets.

The main hope for reducing the size of the MSOS of Action Notation is by means of a substantial simplification of Action Notation during the current reconsideration of its design. For instance, it appears that there is not much use for actions that simultaneously give some transient data and produce some bindings; eliminating them would allow all the hybrid combinators to be removed, and reduce the size of the MSOS of Action Notation by about 10%. The high degree of modularity of MSOS facilitates pin-pointing just which Action Notation constructs are excessively complicated.

It is hoped that the MSOS of Action Notation is much easier to follow than the original SOS—once one has grasped how dependencies between labels determine the flow of processed information, that is. (Readers who have difficulty with this aspect of MSOS might like to contemplate the reduction of MSOS to SOS [11] by moving the *pre* and *post* components of the labels to the configurations.)

Given the good modularity properties of MSOS, one might ask which is better: to describe the operational semantics of a programming language directly, using MSOS, or indirectly, using Action Semantics? In the author's opinion, it is generally better to use Action Semantics, for the following reasons.

The main advantage of the Action Semantics approach over MSOS is that the combinators of Action Notation provide concise abbreviations for particular *patterns* of MSOS (or SOS) transition rules. For instance, the combinator for sequential action performance without data-flow (written A_1 ***and_then*** A_2) abbreviates the pattern of transitions that occurs in many (M)SOS rules for left-to-right evaluation. A further advantage would show up in connection with the description of ML-style exceptions: Action Notation provides the ***escape*** primitive for escaping from normal action performance (with a value), and the combinator A_1 ***trap*** A_2 for trapping such escapes; in (M)SOS, the propagation of the exception value through all the syntactic constructs—apart from the exception handler—has to be specified explicitly.

However, MSOS also has some advantages over Action Semantics. Perhaps the main one is that the only unfamiliar notation provided by MSOS is that for the label transformers, whereas the full standard Action Notation is quite rich, and becoming familiar with it requires a significant initial investment of effort. Another stems from the very generality of the full Action Notation: its equational theory is too weak to be of much practical use. With MSOS, one may be able to prove stronger properties, exploiting awareness of the exact patterns of transitions and configurations that can arise.

Finally, for practical large-scale use of semantic descriptions, tool support is just as crucial as good modularity. Various tools have already been developed for Action Semantics (see other papers in this volume), whereas implementation of tools for MSOS is only just starting.

Those who have grown attached to the expressiveness provided by the framework of Unified Algebras may regret the switch to the more orthodox algebraic specification language CASL; indeed, the author himself has somewhat mixed feelings about abandoning this major application of the Unified Algebras framework, despite the ease with which it can be simulated in CASL . However, the adoption of CASL should not only increase the accessibility of Action Notation (by removing the need to learn first about Unified Algebras), but also it should pave the way for future exploitation of CASL libraries of standard abstract datatypes, and of CASL-based interfaces to existing tools (such as theorem-provers), in connection with action-semantic descriptions. The author was in any case happy to discover that CASL, itself originally designed for algebraic specification and development of software, appears to be quite well-suited also as a meta-notation for MSOS and for Action Semantics.

Acknowledgements Thanks to Christiano Braga, Søren B. Lassen and Carolyn Talcott for comments on earlier versions of the MSOS of Action Notation. The author is supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

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. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from http://www.brics.dk/Projects/CoFI.
3. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [2], Oct. 1998.
4. A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Subsorted partial higher-order logic as an extension of CASL. Note L-10, in [2], Oct. 1998.
5. P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.
6. P. D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.
7. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
8. P. D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.
9. P. D. Mosses. CASL: A guided tour of its design. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Proceedings*, volume 1589 of *LNCS*. Springer-Verlag, 1999.
10. P. D. Mosses. Foundations of modular SOS. Research Series BRICS-RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`. Full version of [11].
11. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available [10].
12. P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [13].
13. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In P. D. Mosses and D. A. Watt, editors, *AS'99, Proc. Second International Workshop on Action Semantics, Amsterdam, The Netherlands*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, May 1999. Full version available [12].
14. P. D. Mosses and M. A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *LNCS*, pages 461–479. Springer-Verlag, 1994.
15. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
16. M. Roggenbach and T. Mossakowski. Basic datatypes in CASL. Note L-12, in [2], Nov. 1999.
17. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

%[**Appendix A**] **library** ACTION_NOTATION

%% The specification BASIC_SYNTAX below is formulated schematically,
%% which is not allowed in CASL.

from LABEL_CATEGORIES %% Appendix B
get CONTEXT_INFO, MUTABLE_INFO, EMITTED_INFO

from DATA_NOTATION %% Appendix C
get TUPLES, TRUTH_VALUES, NUMBERS,
LISTS, SETS, MAPS, DATA_NOTATION

%% **A.1 The Basic Facet**

spec BASIC_DATA =
TRUTH_VALUES
with *Truth_Value*, *true_value*, *false_value*, *either*
and
sorts *Data* < *Indiv*; *DataSort* < *Univ*

spec BASIC_SYNTAX =
BASIC_DATA **then**
types *Action* ::= __ ***or*** __(*Action*; *Action*) | ***fail*** | ***commit*** |
__ ***and*** __(*Action*; *Action*) | ***complete*** |
indivisibly __(*Action*) |
__ ***and_then*** __(*Action*; *Action*) |
__ ***trap*** __(*Action*; *Action*) | ***escape*** |
unfolding __(*Action*) | ***unfold*** | ***diverge***
Yielder ::= ***the*** __ ***yielded_by*** __(*DataSort*; *Yielder*) |
sort DataSort | *data_op*(*Yielder*; ... ; *Yielder*)
%% The schematic alternative *data_op*(*Yielder*; ... ; *Yielder*)
%% stands for a set of alternatives, one for every declared
%% operation *data_op* on *DataSort*.

spec BASIC_OUTCOMES =
BASIC_DATA **and**
type *Terminated* ::= *completed* | *escaped* | *failed*

spec BASIC_CONFIGURATIONS =
BASIC_SYNTAX **and**
BASIC_OUTCOMES
then
type *Action* ::= *sort Terminated* | __ @ __(*Action*; *Action*)

spec BASIC_LABELS =
  BASIC_SYNTAX **then**
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** *unfolding* : *Index*]
    [**sort** *Action* < *ContextInfo* **op** ***fail*** : *Action*]
  then
  EMITTED_INFO
    [**sort** $\mathbb{A}$] [**op** *commitment* : *Index*]
    [TRUTH_VALUES **fit** *default* $\mapsto$ *false_value*, *combine* $\mapsto$ *either*]

spec BASIC_TRANSITIONS =
  BASIC_CONFIGURATIONS **and**
  BASIC_LABELS
  then
    pred $\_\_ \xrightarrow{\_\_} \_\_ : \mathit{Action} \times \mathbb{A} \times \mathit{Action}$
    vars $\alpha, \alpha' : \mathbb{A}; \quad \iota : \mathbb{I}[\mathbb{A}];$
      $A, A_0, A_1, A_2, A', A'_1, A'_2 : \mathit{Action}; \quad t : \mathit{Terminated}$
    axioms
    %[1] $$\%\% \frac{A_1 \xrightarrow{\iota} A'_1}{A_1 \text{ \textbf{or} } A_2 \xrightarrow{\iota} A'_1 \text{ \textbf{or} } A_2;} \%\% \Rightarrow$$
    %[2] $$\%\% \frac{A_2 \xrightarrow{\iota} A'_2}{A_1 \text{ \textbf{or} } A_2 \xrightarrow{\iota} A_1 \text{ \textbf{or} } A'_2;} \%\% \Rightarrow$$
    %[3] $$\textbf{\textit{fail}} \xrightarrow{\iota} \mathit{failed};$$
    %[4] $$\mathit{completed} \text{ \textbf{or} } A_2 \xrightarrow{\iota} \mathit{completed};$$
    %[5] $$A_1 \text{ \textbf{or} } \mathit{completed} \xrightarrow{\iota} \mathit{completed};$$
    %[6] $$\mathit{escaped} \text{ \textbf{or} } A_2 \xrightarrow{\iota} \mathit{escaped};$$
    %[7] $$A_1 \text{ \textbf{or} } \mathit{escaped} \xrightarrow{\iota} \mathit{escaped};$$
    %[8] $$\mathit{failed} \text{ \textbf{or} } A_2 \xrightarrow{\iota} A_2;$$
    %[9] $$A_1 \text{ \textbf{or} } \mathit{failed} \xrightarrow{\iota} A_1;$$

    %[10] $$\%\% \frac{A_1 \xrightarrow{\alpha} A'_1 \wedge \mathit{get}(\alpha, \mathit{commitment}) = \mathit{true\_value}}{A_1 \text{ \textbf{or} } A_2 \xrightarrow{\alpha} A'_1;} \%\% \Rightarrow$$
    %[11] $$\%\% \frac{A_2 \xrightarrow{\alpha} A'_2 \wedge \mathit{get}(\alpha, \mathit{commitment}) = \mathit{true\_value}}{A_1 \text{ \textbf{or} } A_2 \xrightarrow{\alpha} A'_2;} \%\% \Rightarrow$$
    %[12] $$\%\% \frac{\alpha = \mathit{set}(\iota, \mathit{commitment}, \mathit{true\_value})}{\textbf{\textit{commit}} \xrightarrow{\alpha} \mathit{completed};} \%\% \Rightarrow$$

%[13] $$\%\% \frac{A_1 \xrightarrow{\alpha} A'_1}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A'_1 \textbf{\textit{ and }} A_2;} \%\% \Rightarrow$$

%[14] $$\%\% \frac{A_2 \xrightarrow{\alpha} A'_2}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A_1 \textbf{\textit{ and }} A'_2;} \%\% \Rightarrow$$

%[15] $$\textbf{\textit{complete}} \xrightarrow{\iota} \textit{completed};$$

%[16] $$\textit{completed} \textbf{\textit{ and }} \textit{completed} \xrightarrow{\iota} \textit{completed};$$

%[17] $$\textit{escaped} \textbf{\textit{ and }} A_2 \xrightarrow{\iota} \textit{escaped};$$

%[18] $$A_1 \textbf{\textit{ and }} \textit{escaped} \xrightarrow{\iota} \textit{escaped};$$

%[19] $$\textit{failed} \textbf{\textit{ and }} A_2 \xrightarrow{\iota} \textit{failed};$$

%[20] $$A_1 \textbf{\textit{ and }} \textit{failed} \xrightarrow{\iota} \textit{failed};$$

%[21] $$\%\% \frac{A \xrightarrow{\alpha}^{+} t}{\textbf{\textit{indivisibly}}\ A \xrightarrow{\alpha} t;} \%\% \Rightarrow$$

%[22] $$\%\% \frac{A_1 \xrightarrow{\alpha} A'_1}{A_1 \textbf{\textit{ and\_then }} A_2 \xrightarrow{\alpha} A'_1 \textbf{\textit{ and\_then }} A_2;} \%\% \Rightarrow$$

%[23] $$\%\% \frac{A_2 \xrightarrow{\alpha} A'_2}{\textit{completed} \textbf{\textit{ and\_then }} A_2 \xrightarrow{\alpha} \textit{completed} \textbf{\textit{ and\_then }} A'_2;} \%\% \Rightarrow$$

%[24] $$\textit{completed} \textbf{\textit{ and\_then }} \textit{completed} \xrightarrow{\iota} \textit{completed};$$

%[25] $$\textit{completed} \textbf{\textit{ and\_then }} \textit{escaped} \xrightarrow{\iota} \textit{escaped};$$

%[26] $$\textit{completed} \textbf{\textit{ and\_then }} \textit{failed} \xrightarrow{\iota} \textit{failed};$$

%[27] $$\textit{escaped} \textbf{\textit{ and\_then }} A_2 \xrightarrow{\iota} \textit{escaped};$$

%[28] $$\textit{failed} \textbf{\textit{ and\_then }} A_2 \xrightarrow{\iota} \textit{failed};$$

%[29] $$\%\% \frac{A_1 \xrightarrow{\alpha} A'_1}{A_1 \textbf{\textit{ trap }} A_2 \xrightarrow{\alpha} A'_1 \textbf{\textit{ trap }} A_2;} \%\% \Rightarrow$$

%[30] $$\textbf{\textit{escape}} \xrightarrow{\iota} \textit{escaped};$$

%[31] $$\textit{escaped} \textbf{\textit{ trap }} A \xrightarrow{\iota} A;$$

%[32] $$\textit{completed} \textbf{\textit{ trap }} A_2 \xrightarrow{\iota} \textit{completed};$$

%[33] $$\textit{failed} \textbf{\textit{ trap }} A_2 \xrightarrow{\iota} \textit{failed};$$

%[34] $$\textbf{\textit{unfolding}}\ A \xrightarrow{\iota} A \ @\ A;$$

%[35] $$t \ @\ A_0 \xrightarrow{\iota} t;$$

%[36] $$\%\% \frac{\alpha' = \textit{set}(\alpha, \textit{unfolding}, A_0) \wedge A \xrightarrow{\alpha'} A'}{A \ @\ A_0 \xrightarrow{\alpha} A' \ @\ A_0;} \%\% \Rightarrow$$

%[37] $$\%\% \frac{\textit{get}(\iota, \textit{unfolding}) = A_0}{\textbf{\textit{unfold}} \xrightarrow{\iota} A_0;} \%\% \Rightarrow$$

%[38] $$\textbf{\textit{diverge}} \xrightarrow{\iota} \textbf{\textit{diverge}};$$

pred $\_\_ \xrightarrow{\_\_} \_\_ : \mathit{Yielder} \times \mathbb{I}[\mathbb{A}] \times \mathit{DataSort}$
vars $Y : \mathit{Yielder}; \quad d : \mathit{Data}; \quad ds : \mathit{DataSort}$
axioms

%[39] $$\%\% \frac{Y \xrightarrow{\iota} d \wedge d :< ds}{\textbf{\textit{the}}\ ds\ \textbf{\textit{yielded\_by}}\ Y \xrightarrow{\iota} d;} \%\% \Rightarrow$$

%[40] $$\%\% \frac{Y \xrightarrow{\iota} d \wedge \neg(d :< ds)}{\textbf{\textit{the}}\ ds\ \textbf{\textit{yielded\_by}}\ Y \xrightarrow{\iota} \mathit{nothing};} \%\% \Rightarrow$$

%[41] $$\%\% \frac{Y \xrightarrow{\iota} \mathit{nothing}}{\textbf{\textit{the}}\ ds\ \textbf{\textit{yielded\_by}}\ Y \xrightarrow{\iota} \mathit{nothing};} \%\% \Rightarrow$$

%[42] %% For each n-ary $\mathit{data\_op}$, the rule:
$$\%\% \frac{Y_1 \xrightarrow{\iota} ds_1 \wedge \cdots \wedge Y_n \xrightarrow{\iota} ds_n \wedge \\ ds = \mathit{data\_op}(ds_1, \ldots, ds_n)}{\mathit{data\_op}(Y_1, \ldots, Y_n) \xrightarrow{\iota} ds} \%\% \Rightarrow$$

%% A.2 The Functional Facet

spec FUNCTIONAL_DATA
  TUPLES [**sort** *Datum*]
    with $\mathit{Tuple}[\mathit{Datum}] \mapsto \mathit{Data}, \mathit{none}, \mathit{concatenation}, \mathit{nth}$
  and
  NUMBERS **with** *Positive*

spec FUNCTIONAL_SYNTAX =
  BASIC_SYNTAX **and**
  FUNCTIONAL_DATA
  then
    types $\mathit{Action} ::= \textbf{\textit{give}}\ \_\_(\mathit{Yielder}) \mid \textbf{\textit{regive}} \mid$
    $\textbf{\textit{choose}}\ \_\_(\mathit{Yielder}) \mid \textbf{\textit{check}}\ \_\_(\mathit{Yielder}) \mid$
    $\_\_\ \textbf{\textit{then}}\ \_\_(\mathit{Action}; \mathit{Action}) \mid$
    $\textbf{\textit{escape\_with}}\ \_\_(\mathit{Yielder});$
    $\mathit{Yielder} ::= \textbf{\textit{it}} \mid \textbf{\textit{them}} \mid \textbf{\textit{given}}\ \_\_(\mathit{DataSort}) \mid$
    $\textbf{\textit{given}}\ \_\_\#\_\_(\mathit{DataSort}; \mathit{Positive})$

spec FUNCTIONAL_OUTCOMES =
  BASIC_OUTCOMES **and**
  FUNCTIONAL_DATA
  then
    types $\mathit{Terminated} ::= \mathit{sort}\ \ \mathit{Completed} \mid \mathit{sort}\ \ \mathit{Escaped};$
    $\mathit{Completed} ::= \mathit{completed} \mid \mathit{gave}(\mathit{Data});$
    $\mathit{Escaped} ::= \mathit{escaped} \mid \mathit{escape\_gave}(\mathit{Data})$
    axioms
    %[1] $\mathit{gave}(\mathit{none}) = \mathit{completed};$
    %[2] $\neg(\mathit{escape\_gave}(\mathit{none}) = \mathit{escaped})$

spec FUNCTIONAL_LABELS =
  BASIC_LABELS **and**
  FUNCTIONAL_DATA
  then
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** *data* : *Index*]
    [**sort** *Data* < *ContextInfo* **op** *none* : *Data*]

spec FUNCTIONAL_TRANSITIONS =
  BASIC_TRANSITIONS **and**
  FUNCTIONAL_SYNTAX **and**
  FUNCTIONAL_OUTCOMES **and**
  FUNCTIONAL_LABELS
  then
    vars $\alpha, \alpha' : \mathbb{A}$; $\iota : \mathbb{I}[\mathbb{A}]$;
      $A, A_1, A_2, A', A'_1, A'_2 : Action$; $Y : Yielder$;
      $t_1, t_2 : Terminated$; $c_1, c_2 : Completed$; $e_1, e_2 : Escaped$;
      $d, d_1, d_2 : Data$; $ds : DataSort$; $p : Positive$
    axioms
    %[1]
$$\%\% \frac{Y \xrightarrow{\iota} d}{\textbf{\textit{give}}\ Y \xrightarrow{\iota} gave(d);} \%\% \Rightarrow$$
    %[2]
$$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{give}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$
    %[3]
$$\%\% \frac{d = get(\iota, data)}{\textbf{\textit{regive}} \xrightarrow{\iota} gave(d);} \%\% \Rightarrow$$
    %[4]
$$\%\% \frac{Y \xrightarrow{\iota} ds \wedge d :< ds}{\textbf{\textit{choose}}\ Y \xrightarrow{\iota} gave(d);} \%\% \Rightarrow$$
    %[5]
$$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{choose}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$
    %[6]
$$\%\% \frac{Y \xrightarrow{\iota} true\_value}{\textbf{\textit{check}}\ Y \xrightarrow{\iota} gave(none);} \%\% \Rightarrow$$
    %[7]
$$\%\% \frac{Y \xrightarrow{\iota} false\_value \vee Y \xrightarrow{\iota} nothing}{\textbf{\textit{check}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[8] $$c_1 \;\textbf{\textit{or}}\; A_2 \xrightarrow{\iota} c_1;$$
%[9] $$A_1 \;\textbf{\textit{or}}\; c_2 \xrightarrow{\iota} c_2;$$
%[10] $$e_1 \;\textbf{\textit{or}}\; A_2 \xrightarrow{\iota} e_1;$$
%[11] $$A_1 \;\textbf{\textit{or}}\; e_2 \xrightarrow{\iota} e_2;$$
%[12] $$e_1 \;\textbf{\textit{and}}\; A_2 \xrightarrow{\iota} e_1;$$
%[13] $$A_1 \;\textbf{\textit{and}}\; e_2 \xrightarrow{\iota} e_2;$$
%[14] $$e_1 \;\textbf{\textit{and\_then}}\; A_2 \xrightarrow{\iota} e_1;$$
%[15] $$c_1 \;\textbf{\textit{and\_then}}\; e_2 \xrightarrow{\iota} e_2;$$
%[16] $$gave(d_1) \;\textbf{\textit{and}}\; gave(d_2) \xrightarrow{\iota} gave(concatentation(d_1, d_2));$$
%[17] $$gave(d_1) \;\textbf{\textit{and\_then}}\; gave(d_2) \xrightarrow{\iota} gave(concatentation(d_1, d_2));$$
%[18] $$\%\% \frac{A_1 \xrightarrow{\alpha} A_1'}{A_1 \;\textbf{\textit{then}}\; A_2 \xrightarrow{\alpha} A_1' \;\textbf{\textit{then}}\; A_2;} \%\% \Rightarrow$$
%[19] $$\%\% \frac{\alpha' = set(\alpha, data, d_1) \wedge A_2 \xrightarrow{\alpha'} A_2'}{gave(d_1) \;\textbf{\textit{then}}\; A_2 \xrightarrow{\alpha} A_2';} \%\% \Rightarrow$$
%[20] $$gave(d_1) \;\textbf{\textit{then}}\; c_2 \xrightarrow{\iota} c_2;$$
%[21] $$c_1 \;\textbf{\textit{then}}\; e_2 \xrightarrow{\iota} e_2;$$
%[22] $$e_1 \;\textbf{\textit{then}}\; A_2 \xrightarrow{\iota} e_1;$$
%[23] $$c_1 \;\textbf{\textit{then}}\; failed \xrightarrow{\iota} failed;$$
%[24] $$failed \;\textbf{\textit{then}}\; A_2 \xrightarrow{\iota} failed;$$
%[25] $$\%\% \frac{Y \xrightarrow{\iota} d}{\textbf{\textit{escape\_with}}\; Y \xrightarrow{\iota} escape\_gave(d);} \%\% \Rightarrow$$
%[26] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{escape\_with}}\; Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$
%[27] $$\%\% \frac{\alpha' = set(\alpha, data, d_1) \wedge A_2 \xrightarrow{\alpha'} A_2'}{escape\_gave(d_1) \;\textbf{\textit{trap}}\; A_2 \xrightarrow{\alpha} A_2';} \%\% \Rightarrow$$
%[28] $$e_1 \;\textbf{\textit{trap}}\; t_2 \xrightarrow{\iota} t_2;$$
%[29] $$c_1 \;\textbf{\textit{trap}}\; A_2 \xrightarrow{\iota} c_1;$$
%[30] $$failed \;\textbf{\textit{trap}}\; A_2 \xrightarrow{\iota} failed;$$
%[31] $$\%\% \frac{d = get(\iota, data) \wedge d :< datum}{\textbf{\textit{it}} \xrightarrow{\iota} d;} \%\% \Rightarrow$$
%[32] $$\%\% \frac{d = get(\iota, data)}{\textbf{\textit{them}} \xrightarrow{\iota} d;} \%\% \Rightarrow$$
%[33] $$\%\% \frac{d = get(\iota, data) \wedge d :< ds}{\textbf{\textit{given}}\; ds \xrightarrow{\iota} d;} \%\% \Rightarrow$$
%[34] $$\%\% \frac{d = nth(get(\iota, data), p) \wedge d :< ds}{\textbf{\textit{given}}\; ds\#p \xrightarrow{\iota} d;} \%\% \Rightarrow$$

%% A.3 The Declarative Facet

spec DECLARATIVE_DATA =
  MAPS [**sort** *Token*] [**free type** *Range* ::= *sort Bindable* | *unknown*]
    with $Map[Token, Range] \mapsto Bindings$,
      empty_map, *map_of __to__*, *__at__*,
      overlay, *disjoint_union*, *mapped_set*
  and
  SETS [**sort** *Token*]
    with *Set[Token]*, *empty_set*, *set_of*, *__is_in__*

spec DECLARATIVE_SYNTAX =
  BASIC_SYNTAX **and**
  DECLARATIVE_DATA
  then
    types *Action* ::= ***bind __ to __***(*Yielder*; *Yielder*) | ***rebind*** |
      unbind __(*Yielder*) | ***produce __***(*Yielder*) |
      furthermore __(*Action*) |
      __ moreover __(*Action*; *Action*) |
      __ hence __(*Action*; *Action*) |
      __ before __(*Action*; *Action*);
    Yielder ::= ***current_bindings*** |
      the __ bound_to __(*DataSort*; *Yielder*) |
      __ receiving __(*Yielder*; *Yielder*)

spec DECLARATIVE_OUTCOMES =
  BASIC_OUTCOMES **and**
  DECLARATIVE_DATA
  then
    type *Completed* ::= *produced*(*Bindings*)
    axiom
    %[1] *produced*(*empty_map*) = *completed*

spec DECLARATIVE_LABELS =
  BASIC_LABELS **and**
  DECLARATIVE_DATA
  then
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** *bindings* : *Index*]
    [**sort** *Bindings* < *ContextInfo* **op** *empty_map* : *Bindings*]

spec DECLARATIVE_TRANSITIONS =
BASIC_CONFIGURATIONS **and**
DECLARATIVE_SYNTAX **and**
DECLARATIVE_OUTCOMES **and**
DECLARATIVE_LABELS
then
vars $\alpha, \alpha' : \mathbb{A}; \quad \iota, \iota' : \mathbb{I}[\mathbb{A}];$
$A, A_1, A_2, A', A'_1, A'_2 : Action; \quad Y, Y_1, Y_2 : Yielder;$
$t_2 : Terminated; \quad c_1, c_2 : Completed; \quad e, e_1, e_2 : Escaped;$
$d, d' : Data; \quad ds : DataSort;$
$b, b', b_1, b_2 : Bindings; \quad k : Token; \quad bv : Bindable$
axioms

%[1] $$\%\% \frac{Y_1 \xrightarrow{\iota} k \wedge Y_2 \xrightarrow{\iota} bv}{\textbf{\textit{bind}}\ Y_1\ \textbf{\textit{to}}\ Y_2 \xrightarrow{\iota} produced(map\_of\ \ k\ \ to\ \ bv);} \%\% \Rightarrow$$

%[2] $$\%\% \frac{Y_1 \xrightarrow{\iota} nothing \vee Y_2 \xrightarrow{\iota} nothing}{\textbf{\textit{bind}}\ Y_1\ \textbf{\textit{to}}\ Y_2 \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[3] $$\%\% \frac{b = get(\iota, bindings)}{\textbf{\textit{rebind}} \xrightarrow{\iota} produced(b);} \%\% \Rightarrow$$

%[4] $$\%\% \frac{Y \xrightarrow{\iota} k \wedge b = map\_of\ \ k\ \ to\ \ unknown}{\textbf{\textit{unbind}}\ Y \xrightarrow{\iota} produced(b);} \%\% \Rightarrow$$

%[5] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{unbind}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[6] $$\%\% \frac{Y \xrightarrow{\iota} b}{\textbf{\textit{produce}}\ Y \xrightarrow{\iota} produced(b);} \%\% \Rightarrow$$

%[7] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{produce}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[8] $$\%\% \frac{A \xrightarrow{\alpha} A'}{\textbf{\textit{furthermore}}\ A \xrightarrow{\alpha} \textbf{\textit{furthermore}}\ A';} \%\% \Rightarrow$$

%[9] $$\%\% \frac{b = get(\iota, bindings)}{\textbf{\textit{furthermore}}\ produced(b') \xrightarrow{\iota} produced(overlay(b', b));} \%\% \Rightarrow$$

%[10] $$\textbf{\textit{furthermore}}\ e \xrightarrow{\iota} e;$$

%[11] $$\textbf{\textit{furthermore}}\ failed \xrightarrow{\iota} failed;$$

%[12] $$\%\% \frac{A_1 \xrightarrow{\alpha} A_1'}{A_1 \textbf{\textit{moreover}}\ A_2 \xrightarrow{\alpha} A_1' \textbf{\textit{moreover}}\ A_2;} \%\% \Rightarrow$$

%[13] $$\%\% \frac{A_2 \xrightarrow{\alpha} A_2'}{A_1 \textbf{\textit{moreover}}\ A_2 \xrightarrow{\alpha} A_1 \textbf{\textit{moreover}}\ A_2';} \%\% \Rightarrow$$

%[14] $$\mathit{produced}(b_1)\ \textbf{\textit{moreover}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{produced}(\mathit{overlay}(b_2, b_1));$$

%[15] $$e_1\ \textbf{\textit{moreover}}\ A_2 \xrightarrow{\iota} e_1;$$

%[16] $$A_1\ \textbf{\textit{moreover}}\ e_2 \xrightarrow{\iota} e_2;$$

%[17] $$\mathit{failed}\ \textbf{\textit{moreover}}\ A_2 \xrightarrow{\iota} \mathit{failed};$$

%[18] $$A_1\ \textbf{\textit{moreover}}\ \mathit{failed} \xrightarrow{\iota} \mathit{failed};$$

%[19] $$\%\% \frac{b = \mathit{disjoint\_union}(b_1, b_2)}{\mathit{produced}(b_1)\ \textbf{\textit{and}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{produced}(b);} \%\% \Rightarrow$$

%[20] $$\%\% \frac{\neg\ \mathit{def}\ \mathit{disjoint\_union}(b_1, b_2)}{\mathit{produced}(b_1)\ \textbf{\textit{and}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[21] $$\%\% \frac{b = \mathit{disjoint\_union}(b_1, b_2)}{\mathit{produced}(b_1)\ \textbf{\textit{and\_then}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{produced}(b);} \%\% \Rightarrow$$

%[22] $$\%\% \frac{\neg\ \mathit{def}\ \mathit{disjoint\_union}(b_1, b_2)}{\mathit{produced}(b_1)\ \textbf{\textit{and\_then}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[23] $$\%\% \frac{A_1 \xrightarrow{\iota} A_1'}{A_1\ \textbf{\textit{hence}}\ A_2 \xrightarrow{\iota} A_1'\ \textbf{\textit{hence}}\ A_2;} \%\% \Rightarrow$$

%[24] $$\%\% \frac{\alpha' = \mathit{set}(\alpha, \mathit{bindings}, b_1) \wedge A_2 \xrightarrow{\alpha'} A_2'}{\mathit{produced}(b_1)\ \textbf{\textit{hence}}\ A_2 \xrightarrow{\alpha} A_2';} \%\% \Rightarrow$$

%[25] $$\mathit{produced}(b)\ \textbf{\textit{hence}}\ t_2 \xrightarrow{\iota} t_2;$$

%[26] $$e_1\ \textbf{\textit{hence}}\ A_2 \xrightarrow{\iota} e_1;$$

%[27] $$\mathit{failed}\ \textbf{\textit{hence}}\ A_2 \xrightarrow{\iota} \mathit{failed};$$

%[28] $$\%\% \frac{A_1 \xrightarrow{\iota} A_1'}{A_1\ \textbf{\textit{before}}\ A_2 \xrightarrow{\iota} A_1'\ \textbf{\textit{before}}\ A_2;} \%\% \Rightarrow$$

%[29] $$\%\% \frac{\alpha' = \mathit{set}(\alpha, \mathit{bindings}, \mathit{overlay}(b_1, \mathit{get}(\alpha, \mathit{bindings})) \wedge A_2 \xrightarrow{\alpha'} A_2'}{\mathit{produced}(b_1)\ \textbf{\textit{before}}\ A_2 \xrightarrow{\alpha} A_2';} \%\% \Rightarrow$$

%[30] $$\mathit{produced}(b_1)\ \textbf{\textit{before}}\ \mathit{produced}(b_2) \xrightarrow{\iota} \mathit{produced}(\mathit{overlay}(b_2, b_1));$$

%[31] $$e_1\ \textbf{\textit{before}}\ A_2 \xrightarrow{\iota} e_1;$$

%[32] $$c_1\ \textbf{\textit{before}}\ e_2 \xrightarrow{\iota} e_2;$$

%[33] $$\mathit{failed}\ \textbf{\textit{before}}\ A_2 \xrightarrow{\iota} \mathit{failed};$$

%[34] $$c_1\ \textbf{\textit{before}}\ \mathit{failed} \xrightarrow{\iota} \mathit{failed};$$

%[35] $$\%\% \frac{b = get(\iota, bindings)}{\textbf{\textit{current\_bindings}} \xrightarrow{\iota} b;} \%\% \Rightarrow$$

%[36] $$\%\% \frac{Y \xrightarrow{\iota} k \wedge bv = get(\iota, bindings)\ \ at\ \ k \wedge bv :< ds}{\textbf{\textit{the}}\ ds\ \textbf{\textit{bound\_to}}\ Y \xrightarrow{\iota} bv;} \%\% \Rightarrow$$

%[37] $$\%\% \frac{Y \xrightarrow{\iota} k \wedge bv = get(\iota, bindings)\ \ at\ \ k \wedge \neg(bv :< ds)}{\textbf{\textit{the}}\ ds\ \textbf{\textit{bound\_to}}\ Y \xrightarrow{\iota} nothing;} \%\% \Rightarrow$$

%[38] $$\%\% \frac{Y \xrightarrow{\iota} k \wedge \neg(k\ \ is\_in\ \ mapped\_set(get(\iota, bindings))}{\textbf{\textit{the}}\ ds\ \textbf{\textit{bound\_to}}\ Y \xrightarrow{\iota} nothing;} \%\% \Rightarrow$$

%[39] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{the}}\ ds\ \textbf{\textit{bound\_to}}\ Y \xrightarrow{\iota} nothing;} \%\% \Rightarrow$$

%[40] $$\%\% \frac{Y_2 \xrightarrow{\iota} b \wedge \iota' = set(\iota, bindings, b) \wedge Y_1 \xrightarrow{\iota'} ds}{Y_1\ \textbf{\textit{receiving}}\ Y_2 \xrightarrow{\iota} ds;} \%\% \Rightarrow$$

%[41] $$\%\% \frac{Y_2 \xrightarrow{\iota} nothing}{Y_1\ \textbf{\textit{receiving}}\ Y_2 \xrightarrow{\iota} nothing} \%\% \Rightarrow$$

%% A.4 The Imperative Facet

spec IMPERATIVE_DATA =
  MAPS [**sort** *Cell*] [**free type** *Range* ::= *sort Storable* | *uninitialized*]
    with *Map*[*Cell*, *Range*] ↦ *Storage*, *empty_map*, *map_of* __*to*__, __*at*__,
      overlay, *omitting*, *mapped_set*
  and
  SETS [**sort** *Cell*]
    with *Set*[*Cell*], *empty_set*, *set_of*, __[*not_in*__], __*is_in*__

spec IMPERATIVE_SYNTAX =
  BASIC_SYNTAX **and**
  IMPERATIVE_DATA
  then
    types *Action* ::= ***store*** __ ***in*** __(*Yielder*; *Yielder*) |
        unstore __(*Yielder*) |
        reserve __(*Yielder*) | ***unreserve*** __(*Yielder*);
      Yielder ::= ***current_storage*** |
        the __ ***stored_in*** __(*DataSort*; *Yielder*)

spec IMPERATIVE_LABELS =
  BASIC_LABELS **and**
  IMPERATIVE_DATA
  then
  MUTABLE_INFO
    [**sort** $\mathbb{A}$] [**op** *storage* : *Index*]
    [**sort** *Storage* < *MutableInfo* **op** *empty_map* : *Storage*]

spec IMPERATIVE_TRANSITIONS =
  BASIC_TRANSITIONS **and**
  IMPERATIVE_SYNTAX **and**
  IMPERATIVE_LABELS
  then
    vars $\alpha : \mathbb{A};\quad \iota : \mathbb{I}[\mathbb{A}];\quad Y, Y_1, Y_2 : \mathit{Yielder};$
    $d, d' : \mathit{Data};\quad ds : \mathit{DataSort};$
    $s, s' : \mathit{Storage};\quad c : \mathit{Cell};\quad sv : \mathit{Storable}$
    axioms

%[1]
$$\%\% \frac{\begin{array}{c} Y_1 \xrightarrow{\iota} sv \wedge Y_2 \xrightarrow{\iota} c \wedge \\ s = \mathit{get}(\iota, \mathit{storage}) \wedge c\ \mathit{is\_in}\ \mathit{mapped\_set}(s) \wedge \\ s' = \mathit{overlay}(\mathit{map\_of}\ \ c\ \ \mathit{to}\ \ sv, s) \wedge \\ \alpha = \mathit{set}(\mathit{set\_post}(\iota, \mathit{storage}, s'), \\ \qquad \mathit{commitment}, \mathit{true\_value}) \end{array}}{\textbf{\textit{store}}\ Y_1\ \textbf{\textit{in}}\ Y_2 \xrightarrow{\alpha} \mathit{completed};} \%\% \Rightarrow$$

%[2]
$$\%\% \frac{\begin{array}{c} Y_1 \xrightarrow{\iota} sv \wedge Y_2 \xrightarrow{\iota} c \wedge \\ s = \mathit{get}(\iota, \mathit{storage}) \wedge \neg(c\ \mathit{is\_in}\ \mathit{mapped\_set}(s)) \end{array}}{\textbf{\textit{store}}\ Y_1\ \textbf{\textit{in}}\ Y_2 \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[3]
$$\%\% \frac{Y_1 \xrightarrow{\iota} \mathit{nothing} \vee Y_2 \xrightarrow{\iota} \mathit{nothing}}{\textbf{\textit{store}}\ Y_1\ \textbf{\textit{in}}\ Y_2 \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[4]
$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} c \wedge s = \mathit{get}(\iota, \mathit{storage}) \wedge c\ \mathit{is\_in}\ \mathit{mapped\_set}(s) \wedge \\ s' = \mathit{overlay}(\mathit{map\_of}\ \ c\ \ \mathit{to}\ \ \mathit{uninitialized}, s) \wedge \\ \alpha = \mathit{set}(\mathit{set\_post}(\iota, \mathit{storage}, s'), \\ \qquad \mathit{commitment}, \mathit{true\_value}) \end{array}}{\textbf{\textit{unstore}}\ Y \xrightarrow{\alpha} \mathit{completed};} \%\% \Rightarrow$$

%[5]
$$\%\% \frac{Y \xrightarrow{\iota} c \wedge s = \mathit{get}(\iota, \mathit{storage}) \wedge \neg(c\ \mathit{is\_in}\ \mathit{mapped\_set}(s))}{\textbf{\textit{unstore}}\ Y \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[6]
$$\%\% \frac{Y \xrightarrow{\iota} \mathit{nothing}}{\textbf{\textit{unstore}}\ Y \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[7]
$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge s = \mathit{get}(\iota, \mathit{storage}) \wedge \\ \neg(c\ \mathit{is\_in}\ \mathit{mapped\_set}(s)) \wedge c :< ds \wedge \\ s' = \mathit{overlay}(\mathit{map\_of}\ \ c\ \ \mathit{to}\ \ \mathit{uninitialized}, s) \wedge \\ \alpha = \mathit{set}(\mathit{set\_post}(\iota, \mathit{storage}, s'), \\ \qquad \mathit{commitment}, \mathit{true\_value}) \end{array}}{\textbf{\textit{reserve}}\ Y \xrightarrow{\alpha} \mathit{completed};} \%\% \Rightarrow$$

%[8]
$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge s = \mathit{get}(\iota, \mathit{storage}) \wedge \\ (ds\ \&\ (\mathit{cell}[\mathit{not\_in}\ \mathit{mapped\_set}(s)])) = \mathit{nothing} \end{array}}{\textbf{\textit{reserve}}\ Y \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[9]
$$\%\% \frac{Y \xrightarrow{\iota} \mathit{nothing}}{\textbf{\textit{reserve}}\ Y \xrightarrow{\iota} \mathit{failed};} \%\% \Rightarrow$$

%[10]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} c \wedge s = get(\iota, storage) \wedge c\ is\_in\ mapped\_set(s) \wedge \\ s' = omitting(c, s) \wedge \\ \alpha = set(set\_post(\iota, storage, s'), \\ commitment, true\_value)}{\boldsymbol{unreserve}\ Y \xrightarrow{\alpha} completed;}\ \text{\%\%} \Rightarrow$$

%[11]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} c \wedge s = get(\iota, storage) \wedge \neg(c\ is\_in\ mapped\_set(s))}{\boldsymbol{unreserve}\ Y \xrightarrow{\iota} failed;}\ \text{\%\%} \Rightarrow$$

%[12]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} nothing}{\boldsymbol{unreserve}\ Y \xrightarrow{\iota} failed;}\ \text{\%\%} \Rightarrow$$

%[13]
$$\text{\%\%}\ \frac{s = get(\iota, storage)}{\boldsymbol{current\_storage} \xrightarrow{\iota} s;}\ \text{\%\%} \Rightarrow$$

%[14]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} c \wedge sv = get(\iota, storage)\ at\ c \wedge sv :< ds}{\boldsymbol{the}\ ds\ \boldsymbol{stored\_in}\ Y \xrightarrow{\iota} sv;}\ \text{\%\%} \Rightarrow$$

%[15]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} c \wedge sv = get(\iota, storage)\ at\ c \wedge \neg(sv :< ds)}{\boldsymbol{the}\ ds\ \boldsymbol{stored\_in}\ Y \xrightarrow{\iota} nothing;}\ \text{\%\%} \Rightarrow$$

%[16]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} c \wedge \neg(c\ is\_in\ mapped\_set(get(\iota, storage))}{\boldsymbol{the}\ ds\ \boldsymbol{stored\_in}\ Y \xrightarrow{\iota} nothing;}\ \text{\%\%} \Rightarrow$$

%[17]
$$\text{\%\%}\ \frac{Y \xrightarrow{\iota} nothing}{\boldsymbol{the}\ ds\ \boldsymbol{stored\_in}\ Y \xrightarrow{\iota} nothing}\ \text{\%\%} \Rightarrow$$

%% A.5 The Reflective Facet

spec REFLECTIVE_DATA =
  BASIC_SYNTAX
  then
    type *Abstraction* ::= *abstraction_of*(*Action*)
    sort *Abstraction* < *Data*

spec REFLECTIVE_SYNTAX =
  BASIC_SYNTAX **and**
  FUNCTIONAL_SYNTAX **and**
  DECLARATIVE_SYNTAX **and**
  REFLECTIVE_DATA
  then
    types *Action* ::= ***enact*** __(*Yielder*);
      Yielder ::= ***application*** __ ***to*** __(*Yielder*; *Yielder*) |
        closure __(*Yielder*)

spec REFLECTIVE_TRANSITIONS =
BASIC_TRANSITIONS **and**
FUNCTIONAL_TRANSITIONS **and**
DECLARATIVE_TRANSITIONS **and**
REFLECTIVE_SYNTAX
then
vars $\alpha : \mathbb{A}$; $\iota : \mathbb{I}[\mathbb{A}]$;
$A : Action$; $Y, Y_1, Y_2 : Yielder$; $d : Data$; $b : Bindings$
axioms

%[1] $$\%\% \frac{Y \xrightarrow{\iota} abstraction\_of(A)}{\textbf{\textit{enact}}\ Y \xrightarrow{\iota}} \%\% \Rightarrow$$
$$gave(none)\ \textbf{\textit{then}}\ (produced(empty\_map)\ \textbf{\textit{hence}}\ A);$$

%[2] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{enact}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[3] $$\%\% \frac{Y_1 \xrightarrow{\iota} abstraction\_of(A) \wedge Y_2 \xrightarrow{\iota} d}{\textbf{\textit{application}}\ Y_1\ \textbf{\textit{to}}\ Y_2 \xrightarrow{\iota}} \%\% \Rightarrow$$
$$abstraction\_of(gave(d)\ \textbf{\textit{then}}\ A);$$

%[4] $$\%\% \frac{Y_1 \xrightarrow{\iota} nothing \vee Y_2 \xrightarrow{\iota} nothing}{\textbf{\textit{application}}\ Y_1\ \textbf{\textit{to}}\ Y_2 \xrightarrow{\iota} nothing;} \%\% \Rightarrow$$

%[5] $$\%\% \frac{Y \xrightarrow{\iota} abstraction\_of(A) \wedge b = get(\iota, bindings)}{\textbf{\textit{closure}}\ Y \xrightarrow{\iota}} \%\% \Rightarrow$$
$$abstraction\_of(produced(b)\ \textbf{\textit{hence}}\ A);$$

%[6] $$\%\% \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{closure}}\ Y \xrightarrow{\iota} nothing} \%\% \Rightarrow$$

%% A.6 The Communicative Facet

```
spec COMMUNICATIVE_DATA
  LISTS [ sort Message ]
    with List[Message] ↦ Buffer,
         empty_list, concatenation, __is_in__, __omitting__
  and
  NUMBERS with Natural, 0, successor
  then
    sorts  Agent, Communication, Sendable < Data;
           Message, Contract < Communication;
           AgentSort, MessageSort < DataSort
    ops    user_agent          : Agent;
           contents            : Message → Sendable;
           sender, receiver    : Message → Agent;
           serial              : Message → Natural;
           __[containing__]    : MessageSort × DataSort → MessageSort;
           __[from__], __[to__] : MessageSort × AgentSort → MessageSort;
           __[at__]            : MessageSort × DataSort → MessageSort;
    axioms  %% See [7, App. B.6.3].
```

```
spec COMMUNICATIVE_SYNTAX =
  BASIC_SYNTAX then
  COMMUNICATIVE_DATA then
    types  Action ::= send __(Yielder) | remove __(Yielder) |
                      offer __(Yielder) | patiently __(Action);
           Yielder ::= current_buffer |
                       performing_agent | contracting_agent
```

```
spec COMMUNICATIONS =
  COMMUNICATIVE_DATA
  then
    type  Communicating ::= nil | sort Communication |
                            __ || __(Communicating; Communicating)
    op    __ || __ : Communicating × Communicating → Communicating,
                     assoc, comm, unit nil
```

spec COMMUNICATIVE_CONFIGURATIONS =
  BASIC_CONFIGURATIONS **and**
  COMMUNICATIVE_SYNTAX **and**
  COMMUNICATIONS
  then
    types $Processing ::= \_\_\ \textbf{\textit{performing}}\ \_\_(Agent; Action) \mid$
      $sort\ Action \mid sort\ Communicating \mid$
      $\_\_ \parallel \_\_(Processing; Processing)$
    op $\_\_ \parallel \_\_ : Processing \times Processing \to Processing,$
      $assoc, comm, unit\ nil$
    var $A : Action$
    axiom
    %[1] $A : Processing = user\_agent\ \textbf{\textit{performing}}\ A;$

spec COMMUNICATIVE_LABELS =
  BASIC_LABELS **and**
  COMMUNICATIVE_DATA
  then
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** $performer : Index$]
    [**sort** $Agent < ContextInfo$ **op** $user\_agent : Agent$] **and**
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** $contractor : Index$]
    [**sort** $Agent < ContextInfo$ **op** $user\_agent : Agent$] **and**
  MUTABLE_INFO
    [**sort** $\mathbb{A}$] [**op** $buffer : Index$]
    [**sort** $Buffer < MutableInfo$ **op** $empty\_list : Buffer$] **and**
  MUTABLE_INFO
    [**sort** $\mathbb{A}$] [**op** $serial : Index$]
    [**sort** $Natural < MutableInfo$ **op** $0 : Natural$]
  then
  EMITTED_INFO
    [**sort** $\mathbb{A}$] [**op** $communicating : Index$]
    [COMMUNICATIONS **fit** $default \mapsto nil, combine \mapsto \_\_ \parallel \_\_$]
  then
  SET [**sort** $\mathbb{O}[\mathbb{A}]$]
  then
  MUTABLE_INFO
    [**sort** $\mathbb{S}$] [**op** $states : Index$]
    [**sort** $Set[\mathbb{O}[\mathbb{A}]] < MutableInfo$ **op** $empty\_set : Set[\mathbb{O}[\mathbb{A}]]$] **and**
  EMITTED_INFO
    [**sort** $\mathbb{S}$] [**op** $acting : Index$]
    [SET [**sort** $Agent$] **fit** $default \mapsto empty\_set, combine \mapsto union$]

spec COMMUNICATIVE_TRANSITIONS =
BASIC_TRANSITIONS **and**
COMMUNICATIVE_CONFIGURATIONS **and**
COMMUNICATIVE_LABELS
then
vars $\alpha : \mathbb{A}$; $\iota : \mathbb{I}[\mathbb{A}]$; $t : \mathit{Terminated}$; $Y : \mathit{Yielder}$;
$ds : \mathit{DataSort}$; $n : \mathit{Natural}$; $m : \mathit{Message}$;
$a : \mathit{Agent}$; $c : \mathit{Contract}$; $l : \mathit{Buffer}$
axioms

%[1]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge n = get(\iota, serial) \wedge \\ m = ds[from\ get(\iota, performer)][at\ n] \wedge \\ \alpha = set(set(set\_post(\iota, serial, successor(n)), \\ communicating, set\_of(m)), \\ commitment, true\_value) \end{array}}{\boldsymbol{send}\ Y \xrightarrow{\alpha} completed;} \%\% \Rightarrow$$

%[2]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge n = get(\iota, serial) \wedge \\ ds[from\ get(\iota, performer)][at\ n] = nothing \end{array}}{\boldsymbol{send}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[3]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} m \wedge l = get\_pre(\iota, buffer) \wedge m\ is\_in\ items(l) \wedge \\ \alpha = set(set\_post(\iota, buffer, l\ omitting\ m), \\ commitment, true\_value) \end{array}}{\boldsymbol{remove}\ Y \xrightarrow{\alpha} completed;} \%\% \Rightarrow$$

%[4]

$$\%\% \frac{Y \xrightarrow{\iota} m \wedge l = get\_pre(\iota, buffer) \wedge \neg(m\ is\_in\ items(l))}{\boldsymbol{remove}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[5]

$$\%\% \frac{Y \xrightarrow{\iota} nothing}{\boldsymbol{remove}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[6]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge n = get(\iota, serial) \wedge \\ c = ds[from\ get(\iota, performer)][at\ n] \wedge \\ \alpha = set(set(set\_post(\iota, serial, successor(n)), \\ communicating, set\_of(c)), \\ commitment, true\_value) \end{array}}{\boldsymbol{offer}\ Y \xrightarrow{\alpha} completed;} \%\% \Rightarrow$$

%[7]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge n = get(\iota, serial) \wedge \\ ds[from\ get(\iota, performer)][at\ n] = nothing \end{array}}{\boldsymbol{offer}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[8]

$$\%\% \frac{A \xrightarrow{\alpha}{}^{+}\ t \wedge \neg(t = failed)}{\boldsymbol{patiently}\ A \xrightarrow{\alpha} t;} \%\% \Rightarrow$$

%[9]

$$\%\% \frac{A \xrightarrow{\alpha}{}^{+}\ failed}{\boldsymbol{patiently}\ A \xrightarrow{\alpha} \boldsymbol{patiently}\ A;} \%\% \Rightarrow$$

%[10]
$$\%\% \; \frac{l = get(\iota, buffer)}{\boldsymbol{current\_buffer} \xrightarrow{\iota} l;} \; \%\% \Rightarrow$$

%[11]
$$\%\% \; \frac{a = get(\iota, performer)}{\boldsymbol{performing\_agent} \xrightarrow{\iota} a;} \; \%\% \Rightarrow$$

%[12]
$$\%\% \; \frac{a = get(\iota, contractor)}{\boldsymbol{contracting\_agent} \xrightarrow{\iota} a} \; \%\% \Rightarrow$$

pred $\_\_ \xrightarrow{\_\_} \_\_ : Processing \times \$ \times Processing$
vars $\alpha : \mathbb{A}; \quad o, o' : \mathbb{O}[\mathbb{A}]; \quad \sigma, \sigma_1, \sigma_2 : \$; \quad \epsilon : \mathbb{I}[\$];$
$P_1, P_2, P'_1, P'_2 : Processing; \quad C : Communicating;$
$s, s' : Set[\mathbb{O}[\mathbb{A}]]$
axioms
%[13]
$$\%\% \; \frac{P_1 \xrightarrow{\sigma} P'_1}{P_1 \parallel P_2 \xrightarrow{\sigma} P'_1 \parallel P_2;} \; \%\% \Rightarrow$$

%[14]
$$\%\% \; \frac{\begin{array}{c} P_1 \xrightarrow{\sigma_1} P'_1 \wedge P_2 \xrightarrow{\sigma_2} P'_2 \wedge \\ intersection(get(\sigma_1, acting), get(\sigma_2, acting)) = empty\_set \wedge \\ \sigma = \sigma_1 \,\boldsymbol{;}\, \sigma_2 \end{array}}{P_1 \parallel P_2 \xrightarrow{\sigma} P'_1 \parallel P'_2;} \; \%\% \Rightarrow$$

%[15]
$$\%\% \; \frac{\begin{array}{c} s = get\_pre(\sigma, states) \wedge o \ is\_in \ s \wedge \\ a = get(o, performer) \wedge \\ o = pre(\alpha) \wedge A \xrightarrow{\alpha} A' \wedge \\ C = get(\alpha, communicating) \wedge o' = post(\alpha) \wedge \\ s' = union(difference(s, set\_of(o)), set\_of(o')) \wedge \\ \sigma = set\_post(set(\epsilon, acting, set\_of(a)), states, s') \end{array}}{a \ \boldsymbol{performing} \ A \xrightarrow{\sigma} a \ \boldsymbol{performing} \ A' \parallel C;} \; \%\% \Rightarrow$$

%[16]
$$\%\% \; \frac{\begin{array}{c} s = get\_pre(\sigma, states) \wedge o \ is\_in \ s \wedge \\ receiver(m) = get(o, performer) \wedge l = get(o, buffer) \wedge \\ o' = set(o, buffer, concatenation(l, list\_of(m))) \wedge \\ s' = union(difference(s, set\_of(o)), set\_of(o')) \wedge \\ \sigma = set\_post(\epsilon, states, s') \end{array}}{m \xrightarrow{\sigma} nil;} \; \%\% \Rightarrow$$

%[17]
$$\%\% \; \frac{\begin{array}{c} s = get\_pre(\sigma, states) \wedge a :< receiver(c) \wedge \\ \neg(\exists o \bullet o \ is\_in \ s \wedge get(o, performer) = a) \wedge \\ o' = set(set(void, performer, a), contractor, sender(c)) \wedge \\ s' = union(s, set\_of(o')) \wedge \\ \sigma = set\_post(set(\epsilon, acting, set\_of(a)), states, s') \wedge \\ contents(c) = \boldsymbol{abstraction\_of} \ A \end{array}}{c \xrightarrow{\sigma} a \ \boldsymbol{performing} \ A;} \; \%\% \Rightarrow$$

%% **A.7 The Directive Facet**

spec DIRECTIVE_DATA =
  MAPS [**sort** *Token*] [**free type** *Range* ::= *sort Bindable* | *unknown*]
    with *Map*[*Indirection*, *Range*] ↦ *Redirections*,
      empty_map, *map_of __to__*, *__at__*,
      overlay, *disjoint_union*, *mapped_set*
  and
  SETS [**sort** *Indirection*]
    with *Set*[*Indirection*], *empty_set*, *set_of*, *__is_in__*

spec DIRECTIVE_SYNTAX =
  DECLARATIVE_SYNTAX **and**
  REFLECTIVE_SYNTAX **then**
  types *Action* ::= ***indirectly_bind* __ *to*** __(*Yielder*; *Yielder*) |
      redirect* __ *to __(*Yielder*; *Yielder*) |
      recursively_bind* __ *to __(*Yielder*; *Yielder*) |
      undirect __(*Yielder*) |
      indirectly_produce __(*Yielder*);
    Yielder ::= ***current_redirections*** |
      indirect_closure __(*Yielder*)

spec DIRECTIVE_LABELS =
  DIRECTIVE_DATA **then**
  MUTABLE_INFO
    [**sort** $\mathbb{A}$] [**op** *redirections* : *Index*]
    [**sort** *Redirections* < *MutableInfo* **op** *empty_map* : *Redirections*]

spec DIRECTIVE_TRANSITIONS =
  BASIC_CONFIGURATIONS **and**
  DECLARATIVE_OUTCOMES **and**
  DIRECTIVE_LABELS
  then
    vars $\alpha, \alpha', \alpha'' : \mathbb{A}$; $\iota, \iota', \iota'' : \mathbb{I}[\mathbb{A}]$;
      Y, Y_1, Y_2 : *Yielder*;
      r, r' : *Redirections*; i : *Indirection*;
      b, b', b_1, b_2 : *Bindings*; k : *Token*; bv : *Bindable*
    axioms
    %[1]
$$\text{\%\%}\ \frac{\begin{array}{c} Y_1 \xrightarrow{\iota} k \wedge Y_2 \xrightarrow{\iota} bv \wedge \\ r = get(\iota, redirections) \wedge \neg(i \ is\_in \ mapped\_set(r)) \wedge \\ r' = overlay(map\_of \ i \ to \ bv, r) \wedge \\ \alpha = set(set\_post(\iota, redirections, r'), \\ commitment, true\_value) \end{array}}{\boldsymbol{indirectly\_bind}\ Y_1\ \boldsymbol{to}\ Y_2 \xrightarrow{\alpha} produced(map\_of \ k \ to \ i);}\ \text{\%\%} \Rightarrow$$
    %[2]
$$\text{\%\%}\ \frac{Y_1 \xrightarrow{\iota} nothing \vee Y_2 \xrightarrow{\iota} nothing}{\boldsymbol{indirectly\_bind}\ Y_1\ \boldsymbol{to}\ Y_2 \xrightarrow{\iota} failed;}\ \text{\%\%} \Rightarrow$$

%[3]
$$\%\%\ \frac{\begin{array}{c} \mathit{Y_1} \xrightarrow{\iota} k \wedge \mathit{Y_2} \xrightarrow{\iota} bv \wedge \\ i = get(\iota, bindings)\ \ at\ \ k \wedge \\ r = get(\iota, redirections) \wedge i\ \ is\_in\ \ mapped\_set(r) \wedge \\ r' = overlay(map\_of\ \ i\ \ to\ \ bv, r) \wedge \\ \alpha = set(set\_post(\iota, redirections, r'), \\ \quad commitment, true\_value) \end{array}}{\textbf{\textit{redirect}}\ \mathit{Y_1}\ \textbf{\textit{to}}\ \mathit{Y_2} \xrightarrow{\alpha} completed;}\ \%\% \Rightarrow$$

%[4]
$$\%\%\ \frac{\begin{array}{c} \mathit{Y_1} \xrightarrow{\iota} k \wedge \mathit{Y_2} \xrightarrow{\iota} bv \wedge \\ \neg(k\ \ is\_in\ \ mapped\_set(get(\iota, bindings))) \end{array}}{\textbf{\textit{redirect}}\ \mathit{Y_1}\ \textbf{\textit{to}}\ \mathit{Y_2} \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[5]
$$\%\%\ \frac{\begin{array}{c} \mathit{Y_1} \xrightarrow{\iota} k \wedge \mathit{Y_2} \xrightarrow{\iota} bv \wedge \\ i = get(\iota, bindings)\ \ at\ \ k \wedge \\ r = get(\iota, redirections) \wedge \neg(i\ \ is\_in\ \ mapped\_set(r)) \end{array}}{\textbf{\textit{redirect}}\ \mathit{Y_1}\ \textbf{\textit{to}}\ \mathit{Y_2} \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[6]
$$\%\%\ \frac{\mathit{Y_1} \xrightarrow{\iota} nothing \vee \mathit{Y_2} \xrightarrow{\iota} nothing}{\textbf{\textit{redirect}}\ \mathit{Y_1}\ \textbf{\textit{to}}\ \mathit{Y_2} \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[7]
$$\%\%\ \frac{\begin{array}{c} Y \xrightarrow{\iota} i \wedge \\ r = get(\iota, redirections) \wedge i\ \ is\_in\ \ mapped\_set(r) \wedge \\ r' = omitting(i, r) \wedge \\ \alpha = set(set\_post(\iota, redirections, r'), \\ \quad commitment, true\_value) \end{array}}{\textbf{\textit{undirect}}\ Y \xrightarrow{\alpha} completed;}\ \%\% \Rightarrow$$

%[8]
$$\%\%\ \frac{\begin{array}{c} Y \xrightarrow{\iota} i \wedge \\ r = get(\iota, redirections) \wedge \neg(i\ \ is\_in\ \ mapped\_set(r)) \end{array}}{\textbf{\textit{undirect}}\ Y \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[9]
$$\%\%\ \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{undirect}}\ Y \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[10]
$$\%\%\ \frac{\begin{array}{c} Y \xrightarrow{\iota} r \wedge \\ \alpha = set(set\_post(\iota, redirections, r), \\ \quad commitment, true\_value) \end{array}}{\textbf{\textit{indirectly\_produce}}\ Y \xrightarrow{\alpha} completed;}\ \%\% \Rightarrow$$

%[11]
$$\%\%\ \frac{Y \xrightarrow{\iota} nothing}{\textbf{\textit{indirectly\_produce}}\ Y \xrightarrow{\iota} failed;}\ \%\% \Rightarrow$$

%[12]

$$\text{\%\%}\ \frac{\begin{array}{c} Y_1 \xrightarrow{\iota} k \wedge \\ r = get(\iota, redirections) \wedge \neg(i\ is\_in\ mapped\_set(r)) \wedge \\ r' = overlay(map\_of\ \ i\ \ to\ \ unknown, r) \wedge \\ \alpha' = set\_post(\iota, redirections, r') \wedge \\ b = overlay(map\_of\ \ k\ \ to\ \ i, get(\iota, bindings)) \wedge \\ \iota'' = set(\iota', bindings, b) \wedge Y_2 \xrightarrow{\iota''} bv \wedge \\ r'' = overlay(map\_of\ \ i\ \ to\ \ bv, r) \wedge \\ \alpha'' = set\_post(\iota', redirections, r'') \wedge \\ \alpha = set(\alpha'; \alpha'', commitment, true\_value) \end{array}}{\begin{array}{c} \boldsymbol{recursively\_bind}\ Y_1\ \boldsymbol{to}\ Y_2 \xrightarrow{\alpha} \\ produced(map\_of\ \ k\ \ to\ \ i); \end{array}}\ \text{\%\%} \Rightarrow$$

%[13]

$$\text{\%\%}\ \frac{\begin{array}{c} Y_1 \xrightarrow{\iota} k \wedge \\ r = get(\iota, redirections) \wedge \neg(i\ is\_in\ mapped\_set(r)) \wedge \\ r' = overlay(map\_of\ \ i\ \ to\ \ unknown, r) \wedge \\ \alpha' = set\_post(\iota, redirections, r') \wedge \\ b = overlay(map\_of\ \ k\ \ to\ \ i, get(\iota, bindings)) \wedge \\ \iota'' = set(\iota', bindings, b) \wedge Y_2 \xrightarrow{\iota''} nothing \end{array}}{\boldsymbol{recursively\_bind}\ Y_1\ \boldsymbol{to}\ Y_2 \xrightarrow{\iota} failed;}\ \text{\%\%} \Rightarrow$$

%[14]

$$\text{\%\%}\ \frac{Y_1 \xrightarrow{\iota} nothing}{\boldsymbol{recursively\_bind}\ Y_1\ \boldsymbol{to}\ Y_2 \xrightarrow{\alpha} failed}\ \text{\%\%} \Rightarrow$$

%[15]

$$\text{\%\%}\ \frac{r = get(\iota, redirections)}{\boldsymbol{current\_redirections} \xrightarrow{\iota} r;}\ \text{\%\%} \Rightarrow$$

%% A.8 Hybrid Facets

spec HYBRID_SYNTAX =
FUNCTIONAL_SYNTAX **and**
DECLARATIVE_SYNTAX **and**
REFLECTIVE_SYNTAX **and**
COMMUNICATIVE_SYNTAX **then**
types *Action* ::= ***allocate*** __(*Yielder*) |
receive __(*Yielder*) |
subordinate __(*Yielder*) |
__ ***and_then_moreover*** __(*Action*; *Action*) |
__ ***then_moreover*** __(*Action*; *Action*) |
__ ***thence*** __(*Action*; *Action*) |
__ ***then_before*** __(*Action*; *Action*)

spec HYBRID_OUTCOMES =
  BASIC_OUTCOMES **and**
  FUNCTIONAL_OUTCOMES **and**
  DECLARATIVE_OUTCOMES **and**
  COMMUNICATIVE_OUTCOMES
  then
    type $Completed ::= gave\_produced(Data; Bindings)$
    axioms
    %[1] $gave\_produced(none, b) = produced(b);$
    %[2] $gave\_produced(d, empty\_map) = gave(d)$

spec HYBRID_LABELS =
  FUNCTIONAL_LABELS **and**
  DECLARATIVE_LABELS **and**
  IMPERATIVE_LABELS **and**
  COMMUNICATIVE_LABELS

spec HYBRID_TRANSITIONS =
  BASIC_CONFIGURATIONS **and**
  HYBRID_OUTCOMES **and**
  HYBRID_LABELS
  then
    vars $\alpha, \alpha' : \mathbb{A}; \quad \iota, \iota' : \mathbb{I}[\mathbb{A}];$
    $A, A_1, A_2, A', A'_1, A'_2 : Action; \quad Y, Y_1, Y_2 : Yielder;$
    $t_2 : Terminated; \quad c_1, c_2 : Completed; \quad e, e_1, e_2 : Escaped;$
    $d, d' : Data; \quad ds : DataSort;$
    $b, b', b_1, b_2 : Bindings; \quad k : Token; \quad bv : Bindable;$
    $m : Message; \quad ms : MessageSort; \quad as : AgentSort$

    axioms
    %[1]
$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge \\ s = get(\iota, storage) \wedge \neg(c\ is\_in\ mapped\_set(s)) \wedge c :< ds \wedge \\ s' = overlay(map\_of\ \ c\ \ to\ \ uninitialized, s) \wedge \\ \alpha = set(set\_post(\iota, storage, s'), commitment, true\_value) \end{array}}{\textbf{\textit{allocate}}\ Y \xrightarrow{\alpha} gave(c);} \%\% \Rightarrow$$
    %[2]
$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ds \wedge s = get(\iota, storage) \wedge \\ (ds\ \&\ (cell[not\_in\ \ mapped\_set(s)])) = nothing \end{array}}{\textbf{\textit{allocate}}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[3]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ms \wedge l = get\_pre(\iota, buffer) \wedge \\ m :< ms \wedge m\ is\_in\ items(l) \wedge \\ \alpha = set(set\_post(\iota, buffer, l\ omitting\ m), \\ commitment, true\_value) \end{array}}{\boldsymbol{receive}\ Y \xrightarrow{\alpha} gave(m);} \%\% \Rightarrow$$

%[4]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} ms \wedge l = get\_pre(\iota, buffer) \wedge \\ \neg(\exists m \bullet m :< ms \wedge m\ is\_in\ items(l)) \end{array}}{\boldsymbol{receive}\ Y \xrightarrow{\iota} \boldsymbol{receive}\ Y;} \%\% \Rightarrow$$

%[5]

$$\%\% \frac{Y \xrightarrow{\iota} nothing}{\boldsymbol{receive}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[6]

$$\%\% \frac{\begin{array}{c} Y \xrightarrow{\iota} as \wedge a = get(\iota, performer) \wedge \\ A = \boldsymbol{send}\ message[to\ a][containing\ \boldsymbol{performing\_agent}] \\ \boldsymbol{then}\ (\boldsymbol{receive}\ message[from\ a][containing\ abstraction] \\ \boldsymbol{then}\ \boldsymbol{enact}\ contents(\boldsymbol{it})) \wedge \\ A' = \boldsymbol{offer}\ contract[to\ as][containing\ abstraction\_of(A)] \\ \boldsymbol{and\_then}\ (\boldsymbol{receive}\ message[from\ as][containing\ as] \\ \boldsymbol{then}\ \boldsymbol{give}\ contents(\boldsymbol{it})) \end{array}}{\boldsymbol{subordinate}\ Y \xrightarrow{\iota} A';} \%\% \Rightarrow$$

%[7]

$$\%\% \frac{Y \xrightarrow{\iota} nothing}{\boldsymbol{subordinate}\ Y \xrightarrow{\iota} failed;} \%\% \Rightarrow$$

%[8]

$$\%\% \frac{b = disjoint\_union(b_1, b_2)}{gave\_produced(d_1, b_1)\ \boldsymbol{and}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}} \%\% \Rightarrow$$
$$gave\_produced(concatentation(d_1, d_2), b);$$

%[9]

$$\%\% \frac{\neg\ def\ disjoint\_union(b_1, b_2)}{gave\_produced(d_1, b_1)\ \boldsymbol{and}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}} \%\% \Rightarrow$$
$$failed;$$

%[10]

$$\%\% \frac{b = disjoint\_union(b_1, b_2)}{gave\_produced((d_1, b_1)\ \boldsymbol{and\_then}} \%\% \Rightarrow$$
$$gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$gave\_produced(concatentation(d_1, d_2), b);$$

%[11]

$$\%\% \frac{\neg\ def\ disjoint\_union(b_1, b_2)}{gave\_produced(d_1, b_1)\ \boldsymbol{and\_then}} \%\% \Rightarrow$$
$$gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$failed;$$

%[12]

$$\%\% \frac{b = disjoint\_union(b_1, b_2)}{gave\_produced(d_1, b_1)\ \boldsymbol{then}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}} \%\% \Rightarrow$$
$$gave\_produced(d_2, b);$$

%[13]

$$\%\% \frac{\neg\ def\ disjoint\_union(b_1, b_2)}{gave\_produced(d_1, b_1)\ \boldsymbol{then}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}} \%\% \Rightarrow$$
$$failed;$$

%[14] $$\text{%%}\frac{b = get(\iota, bindings)}{\textbf{\textit{furthermore}}\ gave\_produced(d, b') \xrightarrow{\iota}}\text{%%} \Rightarrow$$
$$gave\_produced(d, overlay(b', b));$$

%[15] $$gave\_produced(d_1, b_1)\ \textbf{\textit{moreover}}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$gave\_produced(concatentation(d_1, d_2), overlay(b_2, b_1));$$

%[16] $$gave\_produced(d_1, b_1)\ \textbf{\textit{hence}}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$gave\_produced(concatentation(d_1, d_2), b_2);$$

%[17] $$gave\_produced(d_1, b_1)\ \textbf{\textit{before}}\ gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$gave\_produced(concatentation(d_1, d_2), overlay(b_2, b_1));$$

%[18] $$\text{%%}\frac{A_1 \xrightarrow{\alpha} A'_1}{A_1\ \textbf{\textit{and\_then\_moreover}}\ A_2 \xrightarrow{\alpha}}\text{%%} \Rightarrow$$
$$A'_1\ \textbf{\textit{and\_then\_moreover}}\ A_2;$$

%[19] $$t_1\ \textbf{\textit{and\_then\_moreover}}\ A_2 \xrightarrow{\iota} t_1\ \textbf{\textit{moreover}}\ A_2;$$

%[20] $$\text{%%}\frac{A_1 \xrightarrow{\alpha} A'_1}{A_1\ \textbf{\textit{then\_moreover}}\ A_2 \xrightarrow{\alpha} A'_1\ \textbf{\textit{then\_moreover}}\ A_2;}\text{%%} \Rightarrow$$

%[21] $$\text{%%}\frac{\alpha' = set(\alpha, data, d_1) \wedge A_2 \xrightarrow{\alpha'} A'_2}{gave\_produced(d_1, b_1)\ \textbf{\textit{then\_moreover}}\ A_2 \xrightarrow{\alpha}}\text{%%} \Rightarrow$$
$$gave\_produced(d_1, b_1)\ \textbf{\textit{then\_moreover}}\ A'_2;$$

%[22] $$gave\_produced(d_1, b_1)\textbf{\textit{then\_moreover}}$$
$$gave\_produced(d_2, b_2) \xrightarrow{\iota}$$
$$gave\_produced(d_2, overlay(b_2, b_1));$$

%[23] $$e_1\ \textbf{\textit{then\_moreover}}\ A_2 \xrightarrow{\iota} e_1;$$

%[24] $$c_1\ \textbf{\textit{then\_moreover}}\ e_2 \xrightarrow{\iota} e_2;$$

%[25] $$failed\ \textbf{\textit{then\_moreover}}\ A_2 \xrightarrow{\iota} failed;$$

%[26] $$c_1\ \textbf{\textit{then\_moreover}}\ failed \xrightarrow{\iota} failed;$$

%[27] $$\text{%%}\frac{A_1 \xrightarrow{\iota} A'_1}{A_1\ \textbf{\textit{thence}}\ A_2 \xrightarrow{\iota} A'_1\ \textbf{\textit{thence}}\ A_2;}\text{%%} \Rightarrow$$

%[28] $$\text{%%}\frac{\alpha' = set(set(\alpha, data, d_1), bindings, b_1) \wedge A_2 \xrightarrow{\alpha'} A'_2}{gave\_produced(d_1, b_1)\ \textbf{\textit{thence}}\ A_2 \xrightarrow{\alpha}}\text{%%} \Rightarrow$$
$$gave\_produced(d_1, b_1)\ \textbf{\textit{thence}}\ A'_2;$$

%[29] $$gave\_produced(d_1, b_1)\ \textbf{\textit{thence}}\ t_2 \xrightarrow{\iota} t_2;$$

%[30] $$e_1\ \textbf{\textit{thence}}\ A_2 \xrightarrow{\iota} e_1;$$

%[31] $$failed\ \textbf{\textit{thence}}\ A_2 \xrightarrow{\iota} failed;$$

%[32] $$\%\% \frac{A_1 \xrightarrow{\iota} A'_1}{A_1 \textbf{\textit{ then\_before }} A_2 \xrightarrow{\iota} A'_1 \textbf{\textit{ then\_before }} A_2;} \%\% \Rightarrow$$

%[33] $$b = \mathit{overlay}(b_1, \mathit{get}(\alpha, \mathit{bindings})) \wedge$$
$$\alpha' = \mathit{set}(\mathit{set}(\alpha, \mathit{data}, d_1), \mathit{bindings}, b) \wedge$$
$$\%\% \frac{A_2 \xrightarrow{\alpha'} A'_2}{\mathit{gave\_produced}(d_1, b_1) \textbf{\textit{ then\_before }} A_2 \xrightarrow{\alpha} A'_2;} \%\% \Rightarrow$$

%[34] $$\mathit{gave\_produced}(d_1, b_1) \textbf{\textit{ then\_before }} \mathit{gave\_produced}(d_2, b_2) \xrightarrow{\iota} \mathit{gave\_produced}(d_2, \mathit{overlay}(b_2, b_1));$$

%[35] $$e_1 \textbf{\textit{ then\_before }} A_2 \xrightarrow{\iota} e_1;$$

%[36] $$c_1 \textbf{\textit{ then\_before }} e_2 \xrightarrow{\iota} e_2;$$

%[37] $$\mathit{failed} \textbf{\textit{ then\_before }} A_2 \xrightarrow{\iota} \mathit{failed};$$

%[38] $$c_1 \textbf{\textit{ then\_before }} \mathit{failed} \xrightarrow{\iota} \mathit{failed};$$

%% A.9 The Full Action Notation

spec ACTION_NOTATION =
 BASIC_TRANSITIONS **and**
 FUNCTIONAL_TRANSITIONS **and**
 DECLARATIVE_TRANSITIONS **and**
 IMPERATIVE_TRANSITIONS **and**
 REFLECTIVE_TRANSITIONS **and**
 DIRECTIVE_TRANSITIONS **and**
 COMMUNICATIVE_TRANSITIONS **and**
 HYBRID_TRANSITIONS

%[Appendix B] **library** LABEL_CATEGORIES

spec CATEGORIES [**sort** $\mathbb{A}$] =
  sorts $\mathbb{I}[\mathbb{A}] < \mathbb{A}; \mathbb{O}[\mathbb{A}] = \mathbb{I}[\mathbb{A}]$
  ops $\_\_\,;\,\_\_ : \mathbb{A} \times \mathbb{A} \rightarrow? \mathbb{A}, \mathit{assoc};$
    $\mathit{pre}, \mathit{post} : \mathbb{A} \rightarrow \mathbb{O}[\mathbb{A}]$
  vars $\alpha, \alpha' : \mathbb{A}; \quad \iota : \mathbb{I}[\mathbb{A}]$
  axioms
  %[1] $\mathit{def}(\alpha\,;\alpha') \Leftrightarrow \mathit{post}(\alpha) = \mathit{pre}(\alpha');$
  %[2] $\iota\,;\alpha = \alpha \; \mathit{if} \; \mathit{def}(\iota\,;\alpha);$
  %[3] $\alpha\,;\iota = \alpha \; \mathit{if} \; \mathit{def}(\alpha\,;\iota)$

spec INDICES = **sort** *Index*

spec COMPONENTS =
  sorts *ContextInfo*, *MutableInfo*, *EmittedInfo*
  type *LabelComp* ::= *sort ContextInfo* |
    $\mathit{pair}(\pi_1, \pi_2 : \mathit{MutableInfo})$ |
    sort EmittedInfo
  type *StateComp* ::= *sort ContextInfo* |
    sort MutableInfo

spec LABEL_CATEGORIES [**sort** $\mathbb{A}$] =
  CATEGORIES [**sort** $\mathbb{A}$] **and**
  INDICES **and**
  COMPONENTS **then**
    ops $\mathit{void}: \mathbb{O}[\mathbb{A}];$
      $\mathit{get} : \mathbb{A} \times \mathit{Index} \rightarrow? \mathit{LabelComp};$
      $\mathit{set} : \mathbb{A} \times \mathit{Index} \times \mathit{LabelComp} \rightarrow? \mathbb{A}$
    vars $\alpha : \mathbb{A}; \quad i, i' : \mathit{Index}; \quad c, c' : \mathit{LabelComp}$
    axioms
    %[1] $\mathit{get}(\mathit{set}(\alpha, i, c), i') = c \; \mathit{when} \; i = i' \; \mathit{else} \; \mathit{get}(\alpha, i');$
    %[2] $\mathit{set}(\mathit{set}(\alpha, i, c), i', c') =$
      $\mathit{set}(\alpha, i, c') \; \mathit{when} \; i = i' \; \mathit{else} \; \mathit{set}(\mathit{set}(\alpha, i', c'), i, c)$
    ops $\mathit{get\_pre}(\alpha : \mathbb{A}; i : \mathit{Index}) :? \mathit{MutableInfo} = \pi_1(\mathit{get}(\alpha, i));$
      $\mathit{set\_post}(\alpha : \mathbb{A}; i : \mathit{Index}; m : \mathit{MutableInfo}) :? \mathbb{A} =$
        $\mathit{set}(\alpha, i, \mathit{pair}(\pi_1(\mathit{get}(\alpha, i)), m))$
    ops $\mathit{get}: \mathbb{O}[\mathbb{A}] \times \mathit{Index} \rightarrow? \mathit{StateComp};$
      $\mathit{set}: \mathbb{O}[\mathbb{A}] \times \mathit{Index} \times \mathit{StateComp} \rightarrow? \mathbb{O}[\mathbb{A}]$
    vars $o : \mathbb{O}[\mathbb{A}]; \quad i, i' : \mathit{Index}; \quad s, s' : \mathit{StateComp}$
    axioms
    %[3] $\mathit{get}(\mathit{set}(o, i, s), i') = s \; \mathit{when} \; i = i' \; \mathit{else} \; \mathit{get}(o, i');$
    %[4] $\mathit{set}(\mathit{set}(o, i, s), i', s') =$
      $\mathit{set}(o, i, s') \; \mathit{when} \; i = i' \; \mathit{else} \; \mathit{set}(\mathit{set}(o, i', s'), i, s)$

spec CONTEXT_INFO [**sort** $\mathbb{A}$] [**op** $i : Index$]
[**sort** $CI < ContextInfo$ **op** $default : CI$]
given LABEL_CATEGORIES [**sort** $\mathbb{A}$] =
vars $\alpha, \alpha_1, \alpha_2 : \mathbb{A}; \quad c, c' : CI$
axioms
%[1] $get(void, i) = default;$
%[2] $set(void, i, default) = void;$
%[3] $pre(set(\alpha, i, c)) = set(pre(\alpha), i, c);$
%[4] $post(set(\alpha, i, c)) = set(post(\alpha), i, c);$
%[5] $set(\alpha_1, i, c) \,\mathbf{;}\, set(\alpha_2, i, c) = set(\alpha_1 \,\mathbf{;}\, \alpha_2, i, c);$
%[6] $def\ set(\alpha_1, i, c) \,\mathbf{;}\, set(\alpha_2, i, c') \Rightarrow c = c'$

spec MUTABLE_INFO [**sort** $\mathbb{A}$] [**op** $i : Index$]
[**sort** $MI < MutableInfo$ **op** $default : MI$]
given LABEL_CATEGORIES [**sort** $\mathbb{A}$] =
vars $\alpha, \alpha_1, \alpha_2 : \mathbb{A}; \quad m, m', m_1, m_2 : MI$
axioms
%[1] $get(void, i) = default;$
%[2] $set(void, i, default) = void;$
%[3] $pre(set(\alpha, i, pair(m_1, m_2))) = set(pre(\alpha), i, m_1);$
%[4] $post(set(\alpha, i, pair(m_1, m_2))) = set(post(\alpha), i, m_2);$
%[5] $set(\alpha_1, i, pair(m_1, m)) \,\mathbf{;}\, set(\alpha_2, i, pair(m, m_2)) =$
$set(\alpha_1 \,\mathbf{;}\, \alpha_2, i, pair(m_1, m_2));$
%[6] $def\ set(\alpha_1, i, pair(m_1, m) \,\mathbf{;}\, set(\alpha_2, i, pair(m', m_2)) \Rightarrow m = m'$

spec EMITTED_INFO [**sort** $\mathbb{A}$] [**op** $i : Index$]
[**sort** $EI < EmittedInfo$
ops $default : EI; \quad combine : EI \times EI \to EI, assoc, unit\ default$]
given LABEL_CATEGORIES [**sort** $\mathbb{A}$] =
vars $\alpha, \alpha_1, \alpha_2 : \mathbb{A}; \quad e, e' : EI$
axioms
%[1] $get(\iota, i) = default;$
%[2] $set(\iota, i, default) = \iota;$
%[3] $pre(set(\alpha, i, e)) = pre(\alpha);$
%[4] $post(set(\alpha, i, e)) = post(\alpha);$
%[5] $set(\alpha_1, i, e) \,\mathbf{;}\, set(\alpha_2, i, e') = set(\alpha_1 \,\mathbf{;}\, \alpha_2, i, f(e, e'))$

%[**Appendix C**] **library** DATA_NOTATION

spec UNIFIED_ALGEBRAS =
  sorts $\mathit{Indiv} < \mathit{Univ}$
  pred $\_\_ \leq \_\_ : \mathit{Univ} \times \mathit{Univ}$
  vars $u, u', u'' : \mathit{Univ}$
  axioms
  %[1] $u \leq u' \wedge u' \leq u'' \Rightarrow u \leq u''$
  %[2] $u \leq u' \wedge u' \leq u \Rightarrow u = u'$
  %[3] $u \leq u$
  op $\mathit{nothing} : \mathit{Univ}$
  axiom
  %[4] $\mathit{nothing} \leq u$
  pred $\_\_ :< \_\_(u, u') \Leftrightarrow u \in \mathit{Indiv} \wedge u \leq u'$
  axiom
  %[5] $\neg(u :< \mathit{nothing})$
  ops $\_\_ \mid \_\_ : \mathit{Univ} \times \mathit{Univ} \to \mathit{Univ}, \mathit{assoc}, \mathit{comm}, \mathit{idem}, \mathit{unit}\ \mathit{nothing};$
  $\_\_ \,\&\, \_\_ : \mathit{Univ} \times \mathit{Univ} \to \mathit{Univ}, \mathit{assoc}, \mathit{comm}, \mathit{idem};$
  axioms
  %[6] $u \,\&\, \mathit{nothing} = \mathit{nothing};$
  %[7] $u \leq u'' \wedge u' \leq u'' \Rightarrow (u \mid u') \leq u'';$
  %[8] $u \leq (u \mid u');$
  %[9] $u'' \leq u \wedge u'' \leq u' \Rightarrow u'' \leq (u \,\&\, u');$
  %[10] $(u \,\&\, u') \leq u;$
  %[11] $u \,\&\, (u' \mid u'') = (u \,\&\, u') \mid (u \,\&\, u'');$
  %[12] $u \mid (u' \,\&\, u'') = (u \mid u') \,\&\, (u \mid u'')$

from [7, App. E]
  get TUPLES, TRUTH_VALUES, NUMBERS, LISTS, SETS, MAPS
  %% with translation to CASL!

Recent BRICS Report Series Publications

RS-99-56 **Peter D. Mosses.** ***A Modular SOS for Action Notation*****. December 1999. 39 pp. Full version of paper appearing in Mosses and Watt, editors,** ***Second International Workshop on Action Semantics*****, AS '99 Proceedings, BRICS Notes Series NS-99-3, 1999, pages 131–142.**

RS-99-55 **Peter D. Mosses.** ***Logical Specification of Operational Semantics*****. December 1999. 18 pp. Invited paper. Appears in Flum, Rodríguez-Artalejo and Mario, editors,** ***European Association for Computer Science Logic: 13th International Workshop*****, CSL '99 Proceedings, LNCS 1683, 1999, pages 32–49.**

RS-99-54 **Peter D. Mosses.** ***Foundations of Modular SOS*****. December 1999. 17 pp. Full version of paper appearing in Kutyłowski, Pacholski and Wierzbicki, editors,** ***Mathematical Foundations of Computer Science: 24th International Symposium*****, MFCS '99 Proceedings, LNCS 1672, 1999, pages 70–80.**

RS-99-53 **Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen.** ***Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL*****. December 1999. 9 pp.**

RS-99-52 **Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, and P. S. Thiagarajan.** ***Towards a Theory of Regular MSC Languages*****. December 1999.**

RS-99-51 **Olivier Danvy.** ***Formalizing Implementation Strategies for First-Class Continuations*****. December 1999. Extended version of an article to appear in** ***Programming Languages and Systems: Ninth European Symposium on Programming*****, ESOP '00 Proceedings, LNCS, 2000.**

RS-99-50 **Gerth Stølting Brodal and Srinivasan Venkatesh.** ***Improved Bounds for Dictionary Look-up with One Error*****. December 1999. 5 pp.**

RS-99-49 **Alexander A. Ageev and Maxim I. Sviridenko.** ***An Approximation Algorithm for Hypergraph Max k-Cut with Given Sizes of Parts*****. December 1999. 12 pp.**