1992

E.P.B.M. Rutten, S. Thiébaux

Semantics of Manifold: specification in ASF + SDF and extension

Computer Science/Department of Interactive Systems Report CS-R9269 December

Semantics of MANIFOLD: Specification in ASF+SDF and Extension

E.P.B.M. RUTTEN, S. THIÉBAUX

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

MANIFOLD is a parallel programming language where processes called *manifolds* use broadcast events to coordinate the dynamic data-flow communications among other processes.

This report presents a specification of the *event-driven mechanism* of MANIFOLD in the ASF+SDF formalism, and its implementation in the ASF+SDF system, providing us automatically with a programming environment for MANIFOLD, with syntax-directed editor, parser, static checker, and interpreter.

This subset of the language is augmented with a detailed specification of the unit-exchange mechanism of the language i.e., the *streams*, carrying *units* of data, that compose the data-flow networks in MANIFOLD.

This paper makes extensive references to a preliminary specification, which is only briefly recalled here.

1991 CR Categories: C.1.2, C.1.3, C.2.m, D.1.3, D.3.1, F.1.2, I.1.3.

1991 Mathematics Subject Classification: 68N15 [Software]: Programming Languages; 68Q05, 68U30.

Keywords and Phrases: formal specifications, parallel computing, models of computation, programming language semantics, coordination languages.

Note: The present address of the authors is: IRISA/INRIA, F-35042 RENNES, France; Electronic mail: `rutten@irisa.fr, thiebaux@irisa.fr`.

Report CS-R9269
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 Introduction

We present a formal specification of the MANIFOLD parallel coordination language. MANIFOLD is defined in a detailed informal specification [1], and an implementation is being finalized [2], while there is work in progress on visualization and debugging tools. It is a coordination language, for the orchestration of communication between processes, *not* for the specification of their internal computations: processes are seen as black boxes, known only through the events that they can raise, and the ports through which they can send and receive units of information on a data-flow network. The original motivations for the design of MANIFOLD are practical: specifically, they stemmed from the areas of interactive systems and data-flow hardware, then evolved to encompass the paradigms of concurrent systems and asynchronous communication.

A first part of MANIFOLD, namely its event-driven mechanism, has been provided with an operational semantics [8]. The goal is to provide the language with a complete abstract model, for the clarification of its behavior, the comparison of its characteristics with those of other languages, and the support of analysis techniques for MANIFOLD programs. As a supporting formalism, transition systems were used, defining states of MANIFOLD applications, and rules to describe transitions between them. A prototype interpreter was implemented in PROLOG, and linked to a graphical interface. Parallelly, we study a simplified version of MANIFOLD, that we called MINIFOLD [9], in order to clarify what are the essential basics of the MANIFOLD concept, and to formalize how they behave in the form of automata.

The specification of the event-driven subset of MANIFOLD is reworked here in the ASF+SDF formalism [6], which has also been used for the specification of the static semantics of PASCAL [3] and of an environment for a λ-calculus [4]. It was also used to specify an early version of the concrete syntax of MANIFOLD [5]. It enables us to lay our specification in the framework of algebraic specification, and to make it complete with the addition of a complete syntax including declarations, translation from concrete syntax to abstract syntax, and some static checks. This is performed using the ASF+SDF system, which implements the formalism: it provides us automatically with a programming environment for MANIFOLD, comprising assistance tools such as a syntax-directed editor, a static checker, and an interpreter.

The specification is augmented with a model of the unit exchange mechanism in the data-flow network of MANIFOLD. Representations of the ports and streams are defined, and rules describe how communication along these channels is achieved. This part of the specification benefited from and was inspired by work on MINIFOLD [9]. It could be integrated to the specification in ASF+SDF.

In the remainder of this paper, section 2 describes the semantics of MANIFOLD, by giving an overview of the preliminary version of the specification. Section 3 presents briefly the ASF+SDF formalism, and the ASF+SDF system. The modular organization of the specification of MANIFOLD is described. We also describe the programming environment obtained for MANIFOLD. Section 4 presents a model of streams, ports, and the exchange of units in the data-flow network of an application. Section 5 concludes.

The complete specification of MANIFOLD in ASF+SDF is given in appendix A.

2 Formal specification of MANIFOLD

In this section, we give an overview of the specification already presented in another report [8] which should be consulted for details and for a full understanding of what follows.

2.1 Overview of the formal specification

The formal specification is given in the form of an operational semantics of the kernel MANIFOLD language, which, as said before, is defined in details elsewhere [8]. The choice of an operational representation is motivated by its closeness to the intuitive behavior of the language; this is adequate in the case of MANIFOLD, whose original informal specification has a practical focus [1].

The notation and style used are inspired by the now classical approach of Plotkin [7], involving the definition of states and transition relations between these states. The formal specification of MANIFOLD is structured into four levels, each described by a transition system, hierarchically defined in terms of lower-level ones.

A formal specification methodology. One of the ways to define an operational semantics for a programming language is to use the Structural Operational Semantics (SOS) method. Plotkin proposes a widely-accepted framework and methodology for building transition systems [7]. This formal semantics provides a mathematically well-founded basis for the language, which can be used to validate the correctness of its programs and its implementation.

This approach to formal semantics involves the definition of a number of states (or configurations) noted γ, γ', ..., and of a transition relation between these states. We use the notation: $\gamma \xrightarrow[\beta]{\alpha} \gamma'$ to indicate that *within the context α, a transition can be made from the state γ to the state γ', accompanied by effects β.* A transition may be allowable only under certain conditions or premises (involving α or aspects of γ). Rules for a transition with premises, meaning that the transition from γ to γ' is possible only if the *premises* are true, are noted:

$$\frac{\textit{premises}}{\gamma \xrightarrow[\beta]{\alpha} \gamma'}$$

Special cases of *premises* consist of transitions involving parts of γ: this means that the transition from γ to γ' is possible only if there is a transition from some part of γ to some part of γ'. In the following, e.g., upper-level transitions will be defined with premises involving lower-level transitions in this way.

The four levels of the specification. Our operational semantics consists of four transition systems, each at a different level of abstraction: application, process, handler, and action.

The action level defines the semantics of each action by describing its effect in four aspects: locally raised events, externally raised events, installed pipelines, and (de)activated processes, noted: $\sigma = \langle E_{loc}, E_{ext}, P_{inst}, P_{act} \rangle$. E_{loc} and $Pipe$, represent information local to an executing manifold process; the other two, E_{ext} and $Proc$, represent information meaningful at the application level and are treated there.

An action is within the context of a handling block for the event occurrence e, in a manifold P_{name}. The transition relation is: $\xrightarrow{e, P_{name}}_{act}$. It starts with an action a in an empty four-tuple $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$, and computes σ with all the effects of a.

Examples are the action `DO` that places a local event in E_{loc}, `RAISE` that places an external event in E_{ext}, a pipe-line expression places a set of streams in P_{inst}, and `ACTIVATE` that places a process activation request in P_{act}. A group consists in the execution of each of its members, and the union of all the effects: thus, the group operator is commutative and associative, as the union.

The rules defining the action level semantics are the lowest-level transition system. As an example, here is the rule for `DO`:

$$\boxed{\langle \texttt{DO}\ e_{local},\ \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma \cup \langle \{e_{local}.\texttt{self}\}, \emptyset, \emptyset, \emptyset \rangle}$$

The handler level defines the semantics of an event handling block, and its effects on the process state, using a transition system on top of the action level semantics. It defines, partly in terms of action level transitions, the "normal" case of event handling (going to the next state, executing an action), the reactionary actions (`IGNORE` and `SAVE`), and the manner calls and returns.

The state of a process at the handler level is a triple $\langle P, C, E \rangle$, where:

- P is the state of the program of the process: it records the state of the set of blocks, namely the position of the current block, and the supplementary blocks stacked by manner calls.
- C is the state of the current block of the process, and is one of: `inactive` (before the activation), the state of the current block (of the form *labels* : *action*), `END` (terminated handler), `deactivated` (terminated process).
- E is the local memory of events, containing all the received occurrences of observable events, waiting to be handled.

The transition relation is: $\xrightarrow[E_{ext}, P_{act}]{e, P_p, C_p}_{h}$, where e is the handled event, P_p and C_p are the previous state of the program and the previous state of the current block[1], E_{ext} is the set of externally raised events, P_{act} is the set of (de-)activation of the processes.

[1]This is necessary because reactionary actions like `IGNORE` and `SAVE` "going back" to the previous state of the process.

The rules describe how a manifold process handles a preemptive event by executing the action in the handler (thus involving a transition following the action level rules), adding locally raised events E_{loc} in the events memory, storing P_{inst} in C, and transmitting to upper levels the sets E_{ext} and P_{act}. When the action is a reactionary action, the new state of the process given by P_p and C_p. Manner calls result in a change of the set of blocks in P, by stacking the blocks of the manner in front of those of the caller, and involve handling an internal `start.self` event. A `RETURN` action, quite symmetrically, unstacks the manner blocks from the caller's.

A simplified version of the rule for preemptive event handling, for an action different of `IGNORE`, `SAVE`, a manner call or `RETURN`, is:

$$\boxed{\frac{\langle a_c, \sigma_0\rangle \xrightarrow{e,\, P_{name}}_{action} \langle E_{loc}, E_{ext}, P_{inst}, P_{act}\rangle,}{\langle P, l_c{:}a_c, E\rangle \xrightarrow[E_{ext},\, PP_{act}]{e,\, P_p,\, C_p}_{handler} \langle P, l_c : P_{inst}, E \cup E_{loc}\rangle}}$$

The process level defines the semantics of a process instance. For atomic processes, it describes their starting, raising of events, and terminating. For manifold processes, it describes their reaction to priority events, their reaction to preemptive events, the evolution of their sub-network of streams when a pipe-line breaks, and the termination of a manifold or a manner.

The state of a process is represented as a triple $\langle P, C, E\rangle$ where: P is the state of the program (the name of an atomic, or for a manifold, the same as at the handler level), C is the current state (for an atomic process, it is either `inactive`, `active` or `deactivated`; for a manifold process, it is the same as at the handler level), E is the local memory, as before.

In our model for MANIFOLD, each process has its own local representation of the observed event occurrences that it is interested in. This reflects the notion in MANIFOLD that events are handled asynchronously.

The process level transition relation is $\xrightarrow[E_{ext},\, P_{act}]{}_{p}$ where E_{ext} is the set of events to be raised outside of the process, and P_{act} is the set of names of processes that must be (de-)activated. It is defined in terms of handler and action level transitions for the manifold processes. For each process transition, it gives its external effects and the new state of the process.

This is a one-step transition whose purpose is to select one of the possible alternative transitions for a process. The `deactivated` states of all processes are terminal states, because there are no transitions defined from these states.

Below are given first the rule for atomics raising an event (where *output*(P) gives the set of events declared to be raisable by the atomic process), and then a simplified sketch of the rule for manifolds handling a preemptive event (which involves selecting non-deterministically a preemptive event in E, and another, secondary transition system, performing the search of the handling block in P for the selected event occurrence e [8]).

The rule for the raising of an event by an atomic process is:

$$\frac{e \in output(P)}{\langle P, \texttt{active}, E\rangle \xrightarrow[\{e.P\},\emptyset]{}_p \langle P, \texttt{active}, E\rangle}$$

The rule for event handling is:

$$\frac{\exists e \in E \cap preemptive\text{-}events(P), \qquad P \xrightarrow{e}{}^{*}_{search} \langle P', C'\rangle, \qquad \langle P', C', E \setminus \{e\}\rangle \xrightarrow[E_{ext}, P_{act}]{e, P, C}_h \langle P'', C'', E''\rangle}{\langle P, C, E\rangle \xrightarrow[E_{ext}, P_{act}]{}_p \langle P'', C'', E''\rangle}$$

The application level is the upper-level transition system. It defines the behavior of a set of concurrent processes. The semantics of each of them is defined using the process level transition system. It describes the global behavior of a MANIFOLD application.

The state of an application is the set of states of its member processes. The application level transition relation is noted $\xrightarrow{}_a$. An application changes state when any of its processes causes a change, possibly by raising events in E_{ext}, that must then be propagated to the rest of the application (following the *observability* rules [1]), and requesting (de)activations in P_{act}: these tasks are performed by the function *diffact* [8].

The rule for application level transition is:

$$\frac{\exists p_r \in A,\ p_r \xrightarrow[E_{ext}, P_{act}]{}_p p'_r}{A \xrightarrow{}_a \{\ p'' \ \mid\ \exists p' \in (A \setminus \{p_r\}) \cup \{p'_r\}, \quad p'' = diffact(p'_r, E_{ext}, P_{act})\ \}}$$

The global network connecting all processes can be obtained from the state of a MANIFOLD application by a *multi-union* of the local networks of its individual processes. Each local network is a function of the current state of its manifold.

The network is a *multi-set*, where an element can be redundantly present, in order to represent the property that a stream disappears only when dismantled as many times as it was installed.

A function *net* computes this set, which is composed of couples of port names, the source and the sink: $\langle so, si\rangle$. For a simple stream, it is defined as follows:

$$net(P_1.p_1 \texttt{ -> } P_2.p_2) = \{\ \langle P_1.p_1, P_2.p_2\rangle\ \}$$

```
⟨proc name⟩       ::=  self | system | ⟨ident⟩
⟨port name⟩       ::=  input | output | ⟨ident⟩
⟨port⟩            ::=  ⟨proc name⟩ . ⟨port name⟩
⟨source⟩          ::=  * | ⟨proc name⟩ | ⟨port⟩
⟨event name⟩      ::=  death | returned | abort | terminate
                       | start | break | lambda | ⟨ident⟩
⟨event occ⟩       ::=  *.⟨source⟩ | ⟨event name⟩.⟨source⟩
⟨action⟩          ::=  DO ⟨event name⟩
                       | RAISE ⟨event name⟩
                       | BROADCAST ⟨event name⟩
                       | RETURN
                       | IGNORE
                       | SAVE
                       | ACTIVATE ⟨proc name⟩
                       | DEACTIVATE ⟨proc name⟩
⟨stream⟩          ::=  ⟨port⟩ -> ⟨port⟩
⟨pipeline⟩        ::=  ⟨stream⟩ | [ ⟨stream⟩ [ , ⟨stream⟩ ]* ]
⟨group⟩           ::=  ( ⟨act⟩ [ , ⟨act⟩ ]* ) | ( )
⟨act⟩             ::=  ⟨group⟩
                       | ⟨pipeline⟩
                       | ⟨action⟩
⟨manner call⟩     ::=  ⟨ident⟩
⟨handler⟩         ::=  ⟨act⟩
                       | ⟨manner call⟩
⟨label⟩           ::=  ⟨event occ⟩ [ , ⟨event occ⟩ ]*
⟨block⟩           ::=  ⟨label⟩ : ⟨handler⟩ .
⟨body⟩            ::=  {⟨ block⟩+ }
⟨decl ev⟩         ::=  event ⟨ident⟩ [ , ⟨ident⟩ ]* | ε
⟨decl port in⟩    ::=  in ⟨port name⟩ [ , ⟨port name⟩ ]* | ε
⟨decl port out⟩   ::=  out ⟨port name⟩ [ , ⟨port name⟩ ]* | ε
⟨decl ports⟩      ::=  ⟨decl port in⟩ ⟨decl port out⟩
⟨element⟩         ::=  manifold ⟨ident⟩ ⟨decl port⟩ ⟨body⟩
                       | atomic ⟨ident⟩ ⟨decl port⟩ ⟨decl ev⟩
                       | manner ⟨ident⟩ ⟨body⟩
⟨application⟩     ::=  ⟨element⟩+
```

Figure 1: The grammar of the considered sub-set of the MANIFOLD language.

2.2 Revisions to the preliminary specification

2.2.1 Complete syntax

In the preliminary specification [8], only a restricted grammar was given. The full grammar, including declarations, is given in fig. 1. The non-terminal $\langle ident \rangle$ is an identifier.

Rules are given in the specification to translate this concrete syntax into the abstract syntax used in the semantics.

2.2.2 Revisions to the transition rules

The preliminary specification [8] has been adapted to changes in the informal specification [1], and modified for simplification. However, the rules given in ap›endix A can be found in the preliminary version, where they are commented.

Adaptations are listed below:

- in the preliminary specification, `ACTIVATE` and `DEACTIVATE` are (pseudo-) processes, used in streams, and delivering a boolean unit on their standard output port, giving the result of the operation. In this version, they are treated as actions, terminating immediately.

  The other pseudo-processes, `GETUNIT` nor `GUARD`, are not represented in this specification at all. On the one hand, just as for `ACTIVATE` and `DEACTIVATE`, they would be processes with parameters, and parameter passing is not yet formally supported by the specification. On the other hand, their behavior is related to the unit exchange mechanism, thus their specification is better given in this context, as is done in section 4.

- an internal λ event is used to mark the termination of a handler [1]. The point is that it has to be present when the action, or the group, in a handler terminates. Thus in the rules for `DO`, `RAISE`, `BROADCAST`, `ACTIVATE`, `DEACTIVATE` and stream breaking, an event λ`.self` is added to the set of locally raised events. The rules for groups involve removing λ`.self` if the group is not empty (even if it included members that terminated, hence raising λ), or adding it otherwise. In the concrete syntax of fig. 1, the λ event is noted `lambda`.

- The definitions of preemptive events, observable events and the diffusion of the events are given with more precision than in the preliminary version, e.g., the fact that source `system` is always observable, as well as `death` events even there is no handler for them.

- Rules are given for some aspects of the static semantics of MANIFOLD, stating e.g., that there must be a block featuring `start.self` in its label, that the source processes of event occurrences in labels are declared, ...

3 Specification in ASF+SDF

In this section, we briefly introduce the ASF+SDF specification formalism and the corresponding ASF+SDF system (or meta-environment). We give the basic features of the specification of Manifold in this formalism, and describe the programming environment we obtain within the meta-environment.

3.1 The ASF+SDF formalism and system

ASF+SDF is both a formalism for defining programming languages formally, and a meta-environment that generates automatically programming environments for the languages specified. The combination of these two features is one oı the main reason why we chose to use ASF+SDF. Our first goal was to improve the formal specification given in [8], and the ASF+SDF formalism has been of great assistance 'o debug this specification and to make it more rigorous. Our second goal was to h..ve a programming environment for MANIFOLD at our disposal. Again, the ASF+SDF system has automatically provided us with this environment. The following brief presentation is inspired from [3], which shows the example of the algebraic specification of the static semantics of the ISO standard of PASCAL in ASF+SDF.

3.1.1 Algebraic specifications

The ASF+SDF formalism is used to give an *algebraic specification* of a language. An algebraic specification consists of a signature and a set of conditional equations. The signature declares: the sorts of the sets of values of the algebra (i.e. the names of these sets, e.g. BOOL), the operations on these sorts (e.g. *and*: BOOL × BOOL → BOOL, *true* : → BOOL), and the variables denoting some value of a given sort (e.g. *p*: BOOL). An equation expresses that there exists a relation between two terms over the signature, for example: $and(true, p) = p$. Even if a set of equations specifies implicitly which terms are in relation, the problem of testing whether a relation exists between two terms is undecidable with general equations. However, a mechanical way exists for the restricted forms of equations that rewriting rules are. Rewriting rules must be read from left to right, and denote that their left member can be rewritten into their *smaller* right member term. Note that our equation example can be interpreted as a rewriting rule. A set of such rules is called a rewriting system, and intends to define the *meaning* of a term as its normal form: a term that cannot be reduced further according to the rules. Rewriting systems provide a way to *execute* algebraic specifications, by allowing us to reduce automatically a term (that can be a program) to its normal form (that can be for example some values in the memory of the program at the end of its execution).

3.1.2 Features expected from a specification formalism

If rewriting rules and a signature as defined above constitute a basis for ensuring the executability of an algebraic specification, we expect other additional services from a formal specification formalism.

For example, we would like the specification to be modular, in order to split a large specification into several smaller ones, and to distinguish parts of specifications that are *exported* outside a module from those which are *hidden* in this module.

Another pleasant feature would be to allow conditional equations, i.e. equations specifying that there is a relation between two terms only if there is some other relation between two other smaller terms. As a dummy example, the previously shown equation could be expressed conditionally as:

$$\frac{p = true}{and(p,q) = q}$$

One could also imagine allowing negations in the condition, for example

$$\frac{p \neq true}{imp(p,q) = true}$$

(where imp is the implication).

Also, notational freedom is important to ensure the readability of specifications. For example, one might prefer writing p *and* q instead of $and(p,q)$. This notational freedom enables to view the modified operation declarations as a BNF grammar for the language of terms over the signature, by considering sorts as non-terminals of the grammar.

3.1.3 The ASF, SDF, and ASF+SDF formalisms

ASF+SDF is an algebra-based specification formalism that has all the nice aforementioned additional features.

ASF (Algebraic Specification Formalism) supports modular signatures defining the abstract syntax of operations and conditional equations defining their meaning. It can automatically generate rewriting systems from specifications, and execute them.

SDF (Syntax Definition Formalism) simultaneously enables the definition of lexical and context-free syntax (hence concrete syntax definition), as well as abstract syntax. It also implicitly defines a translation from the concrete syntax to the abstract syntax, thereby allowing the generation of incremental parsers. Thus, the main role of SDF is to allow for free syntax of the specification.

ASF+SDF combines the advantages of both formalisms: the specification can be expressed in SDF concrete syntax, and SDF can translate it into ASF abstract syntax. ASF can then automatically generate a term rewriting system for the specification, and execute it. Hence one can view ASF+SDF as a higher-level formalism for executable algebraic specifications.

We illustrate now the use of this formalism on a small example.

```
module Booleans
exports
      sorts BOOL
      lexical syntax
         [ \t\n]                    -> LAYOUT
      context-free syntax
```

```
         true                      -> BOOL
         false                     -> BOOL
         BOOL or BOOL              -> BOOL {left}
         BOOL and BOOL             -> BOOL {left}
         not (BOOL)                -> BOOL
         "("BOOL")"                -> BOOL {bracket}
      variables
         Bool [0-9']*              -> BOOL
      priorities
        or < and
hiddens
      context-free syntax
        BOOL imp BOOL            -> BOOL {right}
      priorities
        imp < or
equations

      [B1]    not(false) = true
      [B2]    not(true) = false

      [B3]    false imp Bool = true
      [B4]    true imp Bool = Bool

      [B5]    Bool1 or Bool2 = not(Bool1) imp Bool2

      [B6]    Bool1 imp not(Bool2) = Bool',
              not(Bool') = Bool''
              =============================
              Bool1 and Bool2 = Bool''
```

This module defines the syntax and semantics of boolean expressions. It contains the following:

- the lexical syntax for the predefined sort LAYOUT, which defines the sequences of characters that must be skipped during the lexical analysis. Here, blank, tab and return characters are skipped.
- the sort of booleans and the context-free syntax for the boolean expressions. Associativity attributes specify that the disjunction and conjunction operations are left-associative (it could have been right-associative in fact). This is done in order to make the syntax non-ambiguous, without being restricted to infix operators.
- a second disambiguation of the syntax is done by expressing that the conjunction has a higher priority than the disjunction, in a priority declaration.
- note that the bracket function over booleans allows to impose a different priority inside an expression.

- variables over boolean values are defined as Bool, followed by a possibly empty sequence of digits and quotes.
- the hidden context-free syntax defines the implication operation, which must be right-associative. Another priority rule is defined, indicating that the disjunction has a higher priority than the implication (and hence, the conjunction does as well). The implication will not be visible outside the module, but will serve for defining the semantics of the other operations inside the module.
- the equations define first the semantics of the negation and the implication, which is subsequently used to define that of the disjunction and the conjunction. Of course, this has only an illustrative purpose, and a much simpler definition can be directly given.

3.1.4 The ASF+SDF system

The ASF+SDF system is implemented on top of the Centaur System, and is a development environment for both programming languages and programs written in these languages.

ASF+SDF provides the programmer with a syntax-directed editor, a debugger, an interpreter, a pretty printer and some other features that help developing programs. This environment is automatically generated from the language specification.

ASF+SDF provides the language designer with the same tools, allowing to edit, debug and execute specifications. We found that the features of the specification formalism associated with the syntax-directed editor were very useful tool in preventing numerous errors and imprecisions in the development of a specification.

This suffices as a brief presentation of the formalism and the system. Additional material concerning the system implementation can be found in [6], which contains further references.

3.2 Organization of the specification of MANIFOLD

We now describe the organization of the specification of the subset of MANIFOLD in the ASF+SDF formalism. This organization is depicted in figure 2. Groups of modules with the same purpose are gathered into a box. Arrows from a module to another denote that the latter module imports the former.

3.2.1 Syntax

A first set of modules is related to the definition of the syntax of MANIFOLD.

Layout. The module Layout defines the layout in the syntax of MANIFOLD.

Identifiers. The module Identifiers defines the syntax of the identifiers used to denote processes, events and port names.

Programs. The module Manifold-Syntax defines the syntax of programs.

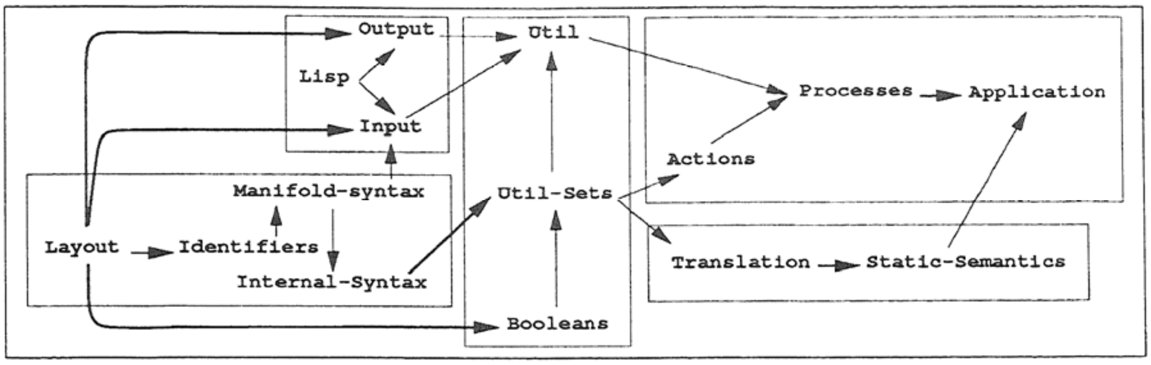


Figure 2: The modular organization of the specification.

Specifications. The module Internal-Syntax defines the internal syntax used to specify MANIFOLD's semantics. It gives, for example, the syntax of the states of the transition systems captured in the semantics equations, as well as that of the environments in which such states are interpreted.

3.2.2 Static semantics

A second set of modules is related to the static semantics of MANIFOLD and to the translation of the syntax of MANIFOLD into the internal syntax.

Translation of MANIFOLD to internal syntax. The module Translation creates a table from a MANIFOLD program, which contains enough information arranged according to a specific syntax, so that the two following operations can be done more easily:

- static checking of the program,
- translation of the program into the initial state at the application level.

Static semantics. The module Static-Semantics defines the static semantics of MANIFOLD. In fact, we haven't specified the complete static semantics, but concentrated on those errors that can prevent the proper interpretation of a program. For example, double declarations of processes, manners, ports and events are forbidden. The use of a port in a stream follows certain rules that exclude using a port declared as an input port, as an output. However, processes can consider their own ports as both input and output ports. Other rules forbid the direct or indirect call of a manner by itself; reasons for that are that our formal specification does not yet support parameters to the manner calls (which restricts the interest of making recursive calls), and that no precise study was made of the risks of endless loops in recursive calls (e.g., a recursive call in the handling block for `start`). Therefore, we made this very restrictive condition, but of course with a closer study it could be partially relaxed and refined, to avoid rejecting programs that could work.

However, nothing prevents that labels of blocks refer to events that can never be emitted, because this does not create an interpretation error. Nevertheless, all the information necessary to easily implement this functionality is available in the table.

3.2.3 Operational semantics

A third set of modules is related to the operational semantics of MANIFOLD. The different modules correspond to the levels of the specification given in [8] and recalled in the first part of this report.

Action level. The module Actions specifies the semantics at the action level

Process level. The module Processes specifies the semantics at the process level, and hides the handler level, as well as the search for an event handling block.

Application level. The module Application specifies the semantics at the global application level.

3.2.4 Input/Output

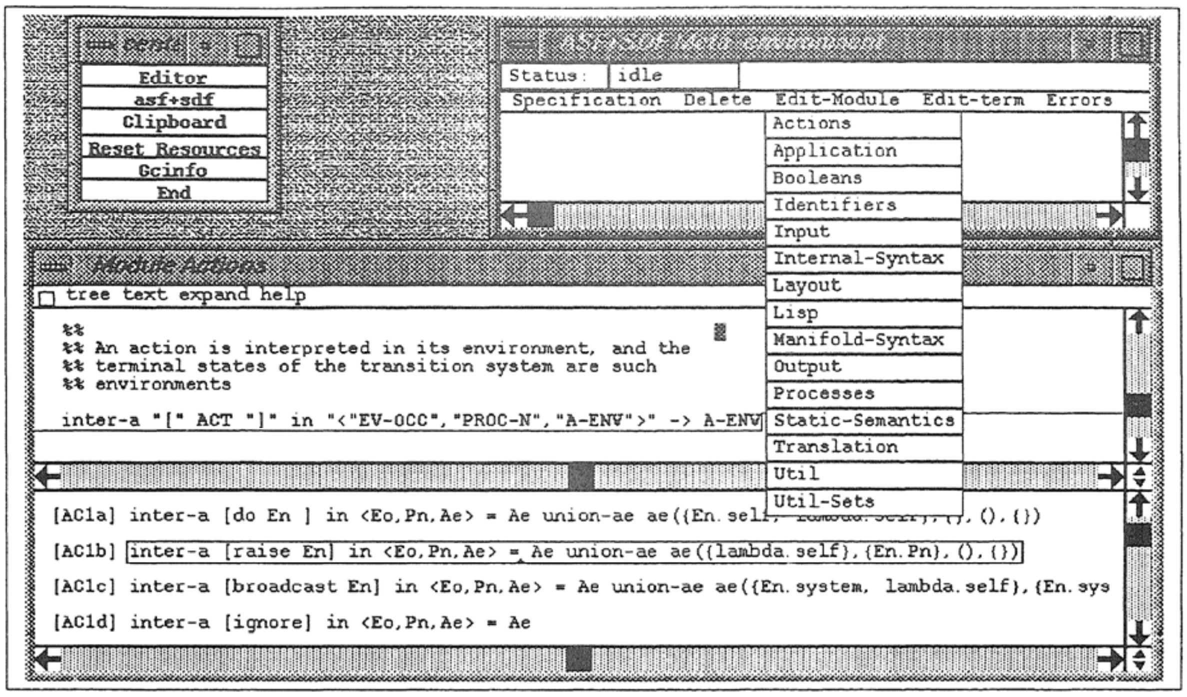
Another set of modules is related to the interaction with the programmer during the interpretation. MANIFOLD is highly asynchronous and non-deterministic, and a multitude of execution traces are possible. In order to debug MANIFOLD programs more easily, we have decided to let the user choose a particular execution trace. We replace all non-determinism concerning the process which is to make the next transition by a choice of the user among the processes that can make a transition. Also, all non-determinism concerning the event that can be handled by a process is replaced by the choice of the user when several events can be handled. In the specification, this is materialized by a specific implementation of the existential quantifier. The actual implementation can be easily modified, in order to randomly select among processes or events. The current interactive implementation requires:

- a module Output that outputs the set of processes or events that the user is allowed to choose among. Also, between each transition at the application level, the current state of the application is displayed, allowing the user to follow the execution trace.
- a module Input that inputs the name of the process or event chosen.

Input-output facilities being not yet directly available from ASF+SDF, both modules as well as the Lisp module necessary to them have been given to us by the team developing ASF+SDF.

In the implementation given in the appendix, output is done on the standard output, while input opens a file whose content is displayed in a window in which the user has to click for the input to be parsed.

We have developed another implementation, in which ASF+SDF interacts with a user interface. The interface enables the graphical visualization of the state of the application, and a more user-friendly selection.



```
%%
%% An action is interpreted in its environment, and the
%% terminal states of the transition system are such
%% environments

inter-a "[" ACT "]" in "<"EV-OCC","PROC-N","A-ENV">" -> A-ENV
```

```
[AC1a] inter-a [do En ] in <Eo,Pn,Ae> = Ae union-ae ae({En.sel                 , (), {})

[AC1b] inter-a [raise En] in <Eo,Pn,Ae> = Ae union-ae ae({lambda.self},{En.Pn}, (), {})

[AC1c] inter-a [broadcast En] in <Eo,Pn,Ae> = Ae union-ae ae({En.system, lambda.self},{En.sys

[AC1d] inter-a [ignore] in <Eo,Pn,Ae> = Ae
```

Figure 3: An editor for a module, with the menu listing the modules.

3.2.5 Utilities

A last set of modules define an additional language that enables us an easier manipulation of the internal syntax.

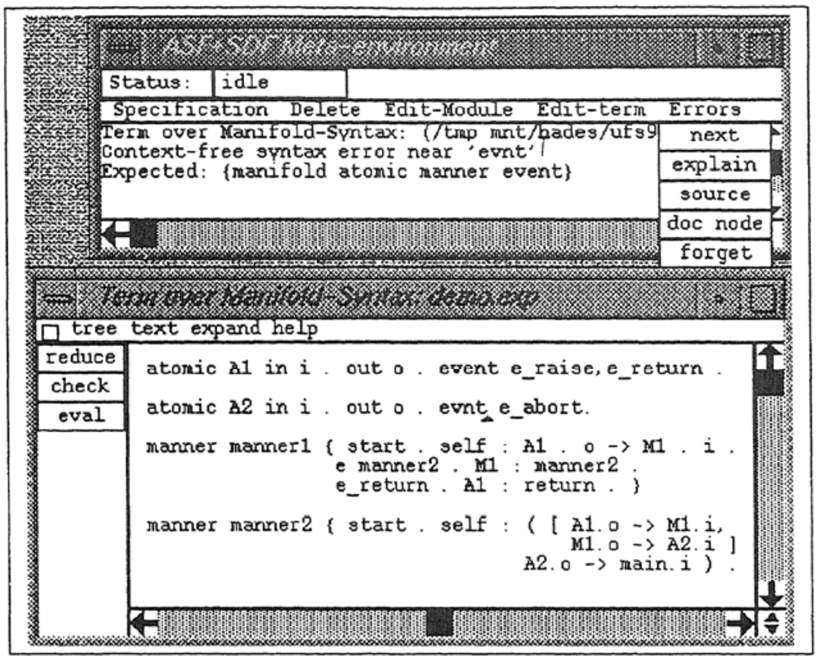
Sets. The module Util-Sets defines a small language for manipulating sets of events, processes, and so on, which are used in the specification of the semantics of MANIFOLD. This language contains basic operations on sets, such as membership test, union, and intersection.

Others. The module Util defines additional constructs in order to manipulate the internal syntax more easily. It contains also the top-level selection and display functions.

3.3 An environment for MANIFOLD

3.3.1 An environment for designing the specification

The ASF+SDF system is a tool for supporting the design of languages themselves. The specification is made in a higher-order language, for which ASF+SDF is the environment. This involves enabling the editing of the specification, in its syntactical and semantical aspects, the checking of the correctness of the syntax of this specification, and the possibility of testing it.



```
Term over Manifold-Syntax: (/tmp mnt/hades/ufs9
Context-free syntax error near 'evnt'
Expected: {manifold atomic manner event}
```

```
atomic A1 in i . out o . event e_raise,e_return .

atomic A2 in i . out o . evnt e_abort.

manner manner1 { start . self : A1 . o -> M1 . i .
                 e manner2 . M1 : manner2 .
                 e_return . A1 : return . }

manner manner2 { start . self : ( [ A1.o -> M1.i,
                                     M1.o -> A2.i ]
                                  A2.o -> main.i ) .
```

Figure 4: A syntax error, located in the source code.

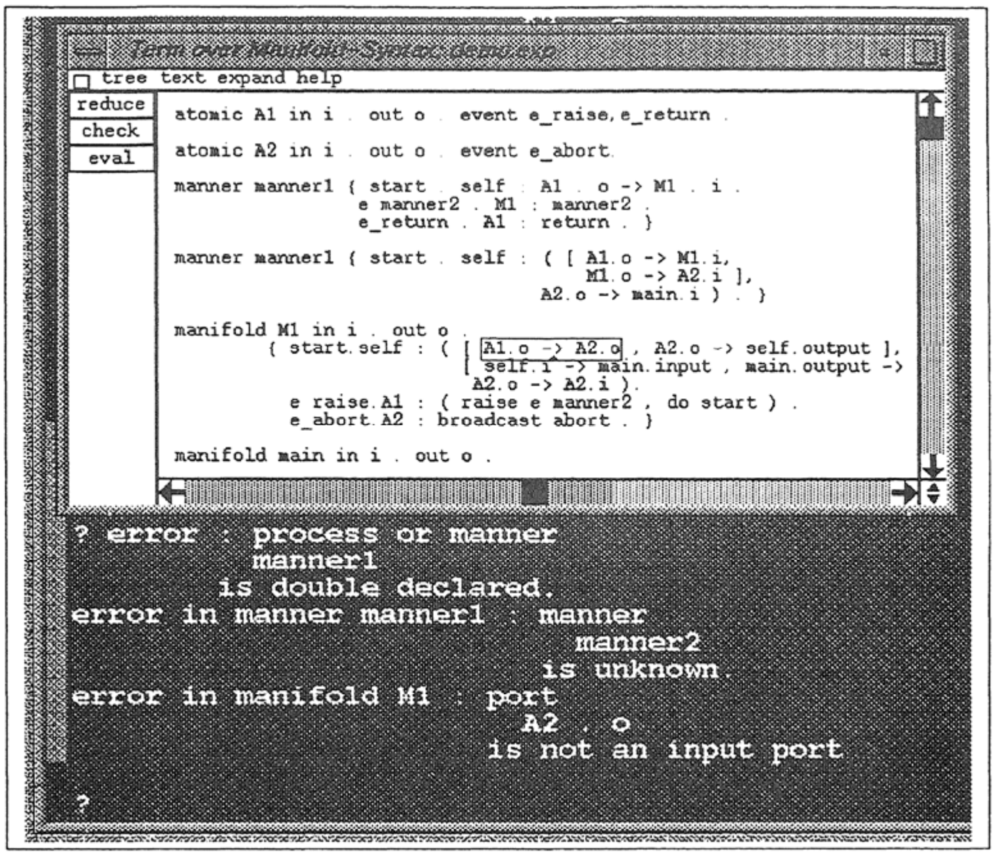
To these ends, the ASF+SDF system provides syntax directed editors on the syntax of specifications: the rules themselves must be given in a precise syntax, and respect typing rules on the elements manipulated, and the operations used. The fig. 3 illustrates how the modules of the specification are accessible from the environment, the pop-up menu giving the names of the available modules. It also shows how they appear in the syntax-directed editor. In the "Module Actions" window, the upper part is an editor of the syntactic part associated to this module, while the lower part contains the equations. In this latter equations part, the area distinguished inside the line has been parsed by ASF+SDF as being an expression following the syntax declared as can be seen in the upper part. When modifying the equations i.e., the semantics of the language,the syntax of the rules is automatically and incrementally checked.

Also shown in fig. 3 is the menu of the CENTAUR system underlying ASF+SDF.

3.3.2 An environment for programming

As a programming environment, ASF+SDF offers the same functionalities, for programs written in the specified language.

When editing a program, the syntax-directed editor enables to compose terms following the syntax specified. When an error is detected, as illustrated in fig, 4, a message is issued in the command window, giving details about the error, telling exactly what was expected at that location of the text. In this example, the error is that, in the editor window, a keyword `event` has been misspelled as `evnt`. It also provides the user with the possibility to find the location of the error in the source code, as shown in the figure.



```
atomic A1 in i . out o . event e_raise, e_return .

atomic A2 in i . out o . event e_abort.

manner manner1 { start . self : A1 . o -> M1 . i .
                 e manner2 . M1 : manner2 .
                 e_return . A1 : return . }

manner manner1 { start . self : ( [ A1.o -> M1.i,
                                    M1.o -> A2.i ],
                                A2.o -> main.i ) . }

manifold M1 in i . out o .
        { start.self : ( [ A1.o -> A2.o , A2.o -> self.output ],
                         [ self.i -> main.input , main.output ->
                          A2.o -> A2.i ).
          e raise.A1 : ( raise e manner2 , do start ) .
          e_abort.A2 : broadcast abort . }

manifold main in i . out o .
```

```
? error : process or manner
          manner1
        is double declared.
error in manner manner1 : manner
                             manner2
                           is unknown.
error in manifold M1 : port
                         A2 . o
                       is not an input port

?
```

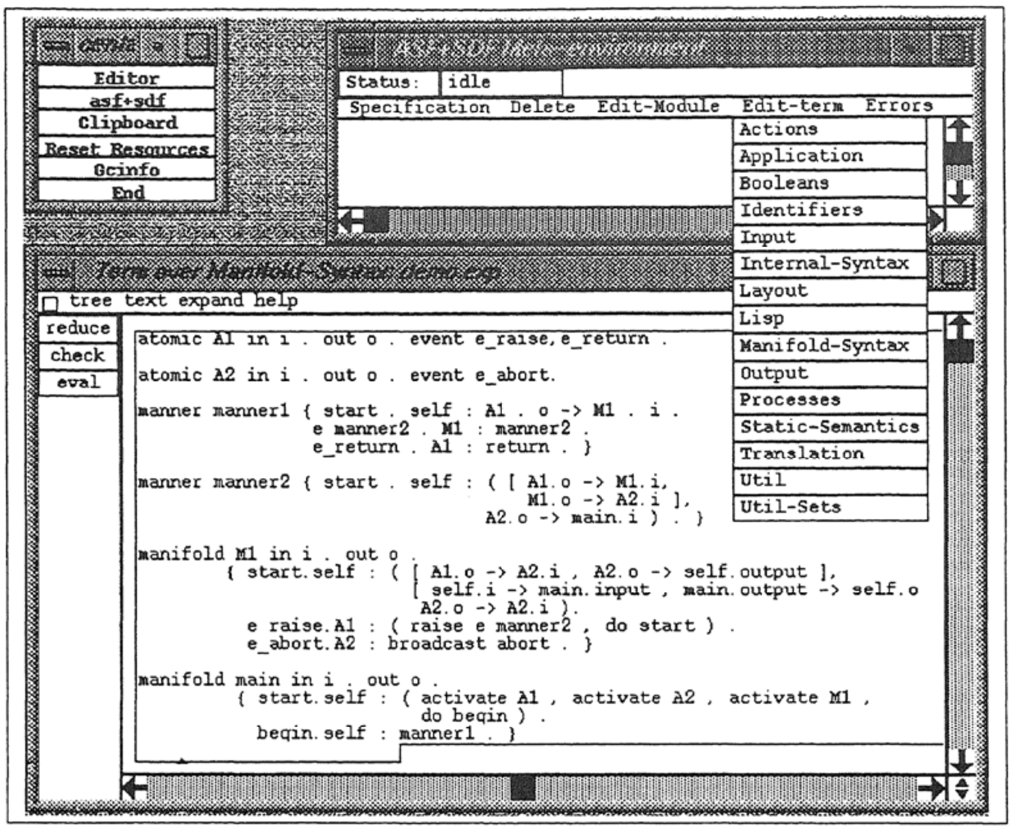
Figure 5: A static error, detected by the functionality "check".

A part of the specification given to ASF+SDF consists of rules for the static checking of programs: identifiers should not be defined more than once, but should be declared, etc ... This functionality is accessible to users through the button "check". ASF+SDF offers the possibility to install user-defined functions in the interface. In fig. 5, an example is given of the detection of the double declaration of `manner1`, causing `manner2` to be undeclared, and the misuse of port `A2.o` in the pipe-line `A1.o -> A2.o` of manifold `M1`: `A2.o` is an output port, and should not be used as the sink of a stream. The message is output in the main window, as a result of the evaluation of the program following the rules of the static checking.

Finally, the ASF+SDF system provides us with an interpreter for the language, by evaluating the result of the application of the rules of the specification on the source program. This is done by calling the evaluation function using the button "eval". This point is further explained in section 3.4.

3.4 Treatment of an example

The example program given in fig. 6 illustrates the various instructions and constructs of the language; it is not meant to have a meaning other than working. In the following,



```
atomic A1 in i . out o . event e_raise, e_return .

atomic A2 in i . out o . event e_abort.

manner manner1 { start . self : A1 . o -> M1 . i .
                 e manner2 . M1 : manner2 .
                 e_return . A1 : return . }

manner manner2 { start . self : ( [ A1.o -> M1.i,
                                    M1.o -> A2.i ],
                                A2.o -> main.i ) . }

manifold M1 in i . out o .
        { start.self : ( [ A1.o -> A2.i , A2.o -> self.output ],
                         [ self.i -> main.input , main.output -> self.o
                          A2.o -> A2.i ).
          e raise.A1 : ( raise e manner2 , do start ) .
          e_abort.A2 : broadcast abort . }

manifold main in i . out o .
         { start.self : ( activate A1 , activate A2 , activate M1 ,
                          do begin ) .
           begin.self : manner1 . }
```

Figure 6: An example in MANIFOLD (with a glance at the ASF+SDF system).

we explain one possible execution of this program, in order to show how the different behavioral patterns can interleave and interact.

Initially, the application starts with a state where `main` is the only process able to make a transition, because it has an event `start.self` in its memory. The activation of `main` corresponds to its handling of the event `start.self`. The corresponding handler `activates` the other processes `A1`, `A2` and `M1`, and raises internally (with a `do` statement) the `begin`.

Starting `A1` and `A2` illustrates the starting of an atomic process.

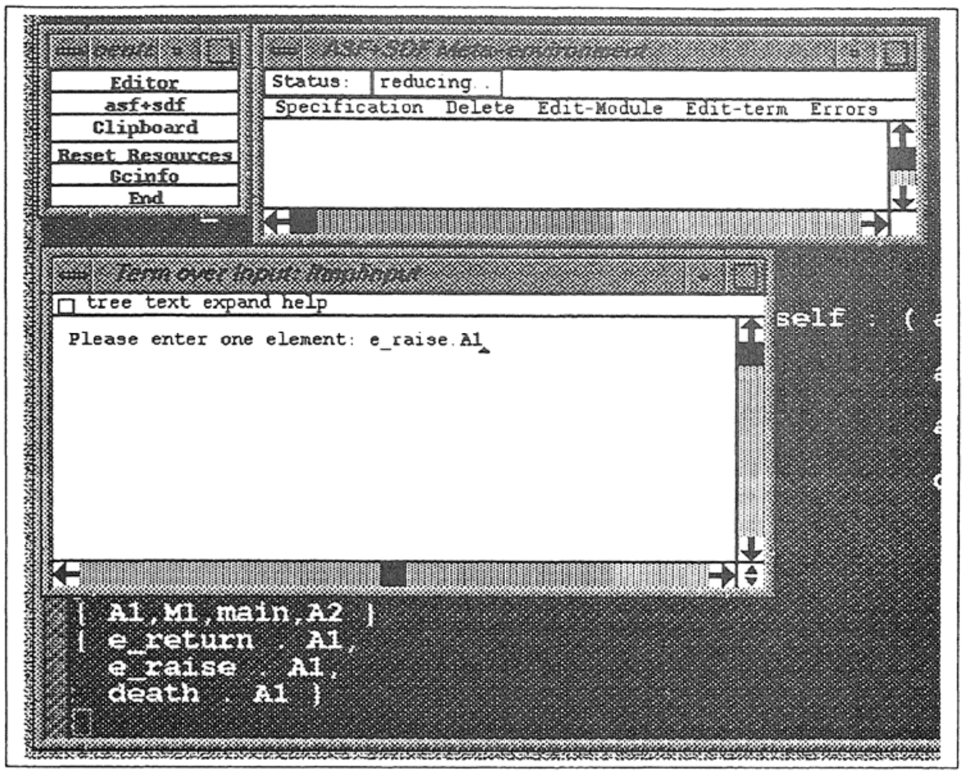
Starting `M1` illustrates the case for a manifold process.

The process `main` can now handle `begin`: it calls the manner `manner1`, which, in turn, handles a `start.self` event, and installs a stream.

Atomic process `A1` raises one of the events that were declared to be raisable: `e_raise` and `e_abort`, and also `death.A1`, that can always be raised. The event occurrence `e_raise.A1` will be received by the manifolds for which it is an observable event i.e., `M1`.

This transition is the one illustrated in fig. 7, where the user selects event `e_raise`.

Manifold `M1` handles `e_raise.A1` by raising externally (by a `raise` instruction) the event `e_manner2`, and internally (by a `do`) the event `start`. This transition leads to



```
Please enter one element: e_raise.A1
```

```
{ A1,M1,main,A2 }
{ e_return . A1,
  e_raise . A1,
  death . A1 }
```

Figure 7: Example of MANIFOLD program: process `A1` raises event `e_raise`.

the application state illustrated in fig. 8: it is displayed in the format defined in the specification.

The manifold **M1** then has to handle **start.self**, which installs the group from which **M1** was preempted by **e_raise**.

The process **main** handles **e_manner2**: from within the manner **manner1**, it makes a nested manner call to the manner **manner2**. This latter handles an internal event occurrence **start.self**, and installs a group.

Atomic process **A1** raises **e_return**.

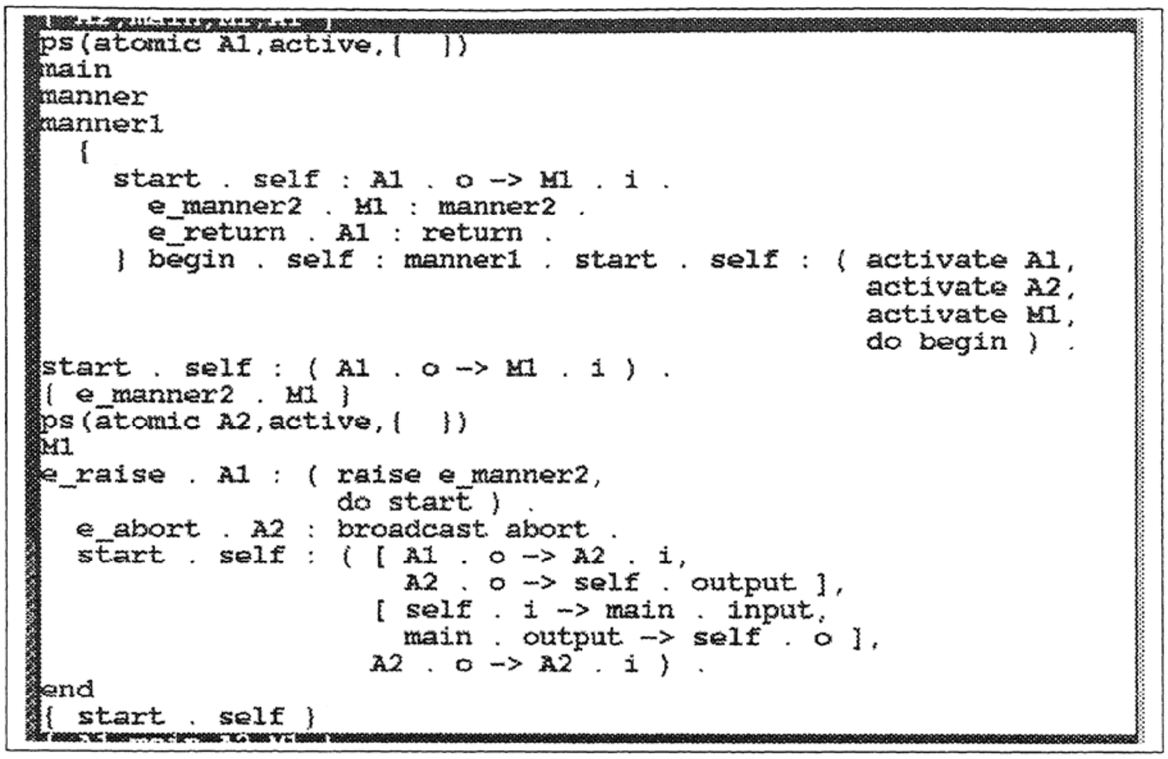
Manifold **main** handles **e_return.A1** in manner1: it exits manner2 (because no handler was found while searching in the blocks of **manner2**), and handles it in **manner1**. There, the handler for **e_return.A1** is the instruction **return**, which returns from the manner, to the caller.

When trying to make a transition, **main** terminates: there is no event received any more in the events memory, and the current block has reached its end.

In manifold **M1**, the group in the current block evolves with **death.main** i.e., one pipe-line breaks because it involved the dead process **main**.

Atomic process **A1** can terminate, raising **death.A1**.

Manifold **M1** reacts, because **A1** is an observable source, and its current group evolves with **death.A1**: another pipe breaks because it involved **A1**, and the group is reduced to the stream involving only **A2**.



```
ps(atomic A1,active,{  })
main
manner
manner1
  {
    start . self : A1 . o -> M1 . i .
      e_manner2 . M1 : manner2 .
      e_return . A1 : return .
    } begin . self : manner1 . start . self : ( activate A1,
                                                activate A2,
                                                activate M1,
                                                do begin ) .
start . self : ( A1 . o -> M1 . i ) .
{ e_manner2 . M1 }
ps(atomic A2,active,{  })
M1
e_raise . A1 : ( raise e_manner2,
                 do start ) .
  e_abort . A2 : broadcast abort .
  start . self : ( [ A1 . o -> A2 . i,
                     A2 . o -> self . output ],
                   [ self . i -> main . input,
                     main . output -> self . o ],
                   A2 . o -> A2 . i ) .
end
{ start . self }
```

Figure 8: Example of MANIFOLD program: displaying the state of the application.

The atomic process A2 raises e_abort. Manifold M1 handles e_abort.A2, as A2 is a preemptive source, and broadcasts abort.

The atomic process A2, having received abort.system, terminates. The remaining manifold, M1, having also received abort.system, terminates too.

All the processes are terminated, thus the application is then terminated.

4 Data-flow communication: the streams

The integration of the streams in the specification extends the application level state representation and transition relation of [8] as described in this section. It is inspired by the semantics of MINIFOLD, a simplification of MANIFOLD [9].

We first describe intuitively the characteristics of ports and streams, before introducing their representation in a formal model, followed by the transition rules defining the behavior of an application including these features.

4.1 Overview of streams and ports

This intuitive presentation of ports and streams, and the way they behave and interact to exchange units, follows the original specification [1], where more details can be found.

Ports. Ports are interfaces between the inside and the outside of a process. They belong to one process, and connect this process with streams. Ports are directed: they transmit units either into or out of a process. They have a buffering ability, storing units produced by the process, when no stream is connected to the port. This buffer has a *first-in first-out* behavior.

Ports can raise events e.g., to signal the arrival of a new unit. These events will be observed only by the owner process, as local events: the source of such an event occurrence, raised by port $\langle p\rangle$ of process $\langle P\rangle$, is $\langle P\rangle.\langle p\rangle$, which is observable by process $\langle P\rangle$ only.

Streams. They are a sequential communication link between ports of processes, carrying units of data.

The units are atomic pieces of information, in the sense that they have no internal structure that is considered at this level: they have no type or value information ·elated to them, and the only assumption that we make concerning them, is that they won't be decomposed into fragments or mixed.

The streams are reliable links between two ports: there is no loss of units, no error in their transmission, or duplication. Streams are directed: all units are carried from the same *source* port to the same *sink* port. They behave as *first-in first-out* queues of unbounded capacity. They are asynchronous, and if empty, suspend the execution of the sink in the awaiting of a unit.

A stream can be redundantly installed by several processes: in this case, the stream ceases to work only when all the installers have broken it down: such a stream is in *dormant* state.

The installation of streams is an atomic operation with regard to the flow of units, in the sense that units flow in the stream only after the completion of all the activations and constructions in the group to which the stream belongs.

On the other side, as said in the original specification [1], a manifold that constructs a stream is in fact oblivious to this flow of information.

4.2 Representation of the states

Processes. As in the previous version of the specification [8], we have processes represented as a triple $\langle P, C, E\rangle$ where:

- P is the state of the program of the process, giving information on whether it is atomic or a manifold and, for manifolds, its set of handling blocks;
- C is its current state i.e.,
  - `inactive` when it has not been activated yet,
  - `deactivated` when it is terminated,
  - and, when it is active,
    - for atomics: `active`
    - for manifolds: the current state of its current block (for more details, see the previous specification [8]);

- E is the events memory, where the received observable event occurrences are stored.

We give ourselves functions accessing the different components of such a state: *current*($\langle P,C,E\rangle$) $= C$, and *name*($\langle P,C,E\rangle$) $=$*name*(P) which is the name of process P. The function *net*(C) gives the set of couples $\langle so,si\rangle$ corresponding to the streams of C, as said earlier.

Ports. The states of ports are represented by tuples[2]: $\langle p,pc\rangle$ where:

- p is the name of the port and its owner process, of the form $\langle process\rangle.\langle port\rangle$, where $\langle process\rangle$ is the owner of the $\langle port\rangle$;
- pc is the contents of the port i.e., a set of units with a *first-in first-out* management policy. It can be represented by lists of the form $[u_1, ..., u_n]$, with the following operations:
  - $empty = [\ ]$,
  - $get([u_1, ..., u_n]) = u_1$,
  - $rest([u_1, ..., u_n]) = [u_2, ..., u_n]$,
  - $append(u, [u_1, ..., u_n]) = [u_1, ..., u_n, u]$.

Streams. We will represent streams as a tuple: $\langle so,si,o,c\rangle$ where:

- so is the name of the source port,
- si is the name of the sink port,
- o is the set of the names of owner processes i.e., the processes that installed this stream; when it is empty, it means that no process installed the stream, which is inactive, and said to be in a *dormant* state,
- sc is the contents of the stream i.e., as for the ports, a set of units with a *first-in first-out* management policy.

Applications. The state of an application is defined by the set of the states of all its components: processes, ports, and streams.

Its is a tuple $\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o\rangle$ where:

- $\mathcal{P}_r$ is the set of process states,
- $\mathcal{S}$ is the (initially empty) set of streams installed by the processes between the ports,
- $\mathcal{P}_o$ is the set of ports (each belonging to a process in $\mathcal{P}_r$).

[2]Another possibility would be to integrate the representation of ports in that of processes, by including them to the process tuple, like in: $\langle P,C,E,Ports\rangle$, where *Ports* is a set of tuples $\langle name, contents\rangle$.

4.3 Transitions between the states

4.3.1 Overview

An application makes a transition when:

a process makes a transition involving handling and raising of events, and modifications in $\mathcal{P}_r$, breaking the streams of the previous state, and installing those of the new one in $\mathcal{S}$, and representing possible effects of a process transition on its ports (i.e., when the process terminates, its ports are not active any more in $\mathcal{P}_o$);

a unit moves from a port to a stream or possibly several ones, which involves taking the first unit of the contents of the port, removing it, and copying it to the contents of *all* the streams of which this port is the source;

a unit moves from a stream to a port which, symmetrically to the previous case, involves taking the first unit of the contents of the stream, and movi ıg it to the contents of its sink port; this also involves the possible firing or reaction of processes in $\mathcal{P}_r$, namely:

- `guard(`$\langle$*port*$\rangle$`,`$\langle$*event*$\rangle$`)` raises the given $\langle$*event*$\rangle$ and terminates;
- `getunit(`$\langle$*port*$\rangle$`)` copies the unit to its own output port, and terminates.

We add rules to the application-level transition relation: $\mathcal{A} \xrightarrow{\qquad}_a \mathcal{A}'$ and modify the one that was defined in the previous formal specification [8].

4.3.2 Process transition rule

Application states are now of the form $\mathcal{A} = \langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o \rangle$ and the case of process transition is defined by rule 1 given below. It is an extension of the application level rule in the former specification [8]. An application $\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o \rangle$ makes a transition to state $\langle \mathcal{P}'_r, \mathcal{S}', \mathcal{P}'_o \rangle$ if:

- some process p_r of $\mathcal{P}_r$ makes a process-level transition to p'_r, with the raising of external events in E_{ext} and (de)activations in P_{act}; these effects are diffused in $\mathcal{P}_r$ where every p is changed into p' by function *diffact*, resulting in the new set of processes $\mathcal{P}'_r$; i.e., formally:

$$\exists p \in \mathcal{P}_r,\ p_r \xrightarrow[E_{ext},\, P_{act}]{}_p p'_r,$$
$$\mathcal{P}'_r = \{p' \mid \ \exists p \in (\mathcal{P}_r \setminus \{p_r\}) \cup \{p'_r\},\quad p' = diffact(p, E_{ext}, P_{act})\ \},$$

  where *diffact*(p, E_{ext}, P_{act}) handles the diffusion of the events in E_{ext} by broadcasting to the process p, and the activation or deactivation of processes in P_{act}; this function is already defined in the previous formal specification [8].

- the ports of p'_r, if p'_r died i.e., p'_r became `deactivated`, are removed from $\mathcal{P}_o$, transforming it into $\mathcal{P}'_o$; i.e., formally:

$$\mathcal{P}'_o = \textit{remove-if-death}(p'_r, \mathcal{P}_o)$$

  where *remove-if-death*$(p'_r, \mathcal{P}_o)$ removes, in the set of ports $\mathcal{P}_o$, the ports of which the owner p'_r is in the state `deactivated`:

$$\textit{remove-if-death}(p'_r, \mathcal{P}_o) = \{\langle P.p,pc\rangle \mid \langle P.p,pc\rangle \in \mathcal{P}_o \wedge P \neq name(p'_r) \wedge current(p'_r) \neq \texttt{deactivated}\ \}$$

- the streams corresponding to the previous state of the process are broken in $\mathcal{S}$, by the function *break-streams*$(p_r, \mathcal{S})$, and on the resulting set of streams, the streams of p'_r are installed, by the function *install-streams*$(p'_r, \mathcal{S}, \mathcal{P'}_o)$; i.e., formally:

  $$\mathcal{S}' = \textit{install-streams}(p'_r,\ \textit{break-streams}(p_r, \mathcal{S}), \mathcal{P}'_o\ )$$

  where:

  - *break-streams*$(p_r, \mathcal{S})$ breaks all streams belonging to p_r in the set of streams $\mathcal{S}$, i.e., the reference to p_r as an owner is removed from the owners names set of all streams:

    $$\textit{break-streams}(p_r, \mathcal{S}) = \{\ \langle si,so,o',sc\rangle \mid \langle si,so,o,sc\rangle \in \mathcal{S}\ \wedge\ o' = o \setminus \{name(p_r)\}\ \}$$

    A "garbage-collecting" could take place here, by discarding dormant streams i.e., $\langle si,so,o',sc\rangle$ for which $o' = \emptyset$.
  - *install-streams*$(p'_r, \mathcal{S}, \mathcal{P}_o)$ verifies the activity state of the processes owning the ports involved in the current network of p'_r: $net(current(p'_r))$. It does not install the streams to or from a dead process: this is done by checking that both *so* and *si* are represented in the set of port states $\mathcal{P}_o$.
    It installs all the streams in $net(current(p'_r))$ (the current state of process p'_r) i.e., adds p'_r's name in the list of owners of those that already exists in $\mathcal{S}$, or adds a new stream to $\mathcal{S}$, with empty contents noted [ ] and a set of owners $\{name(p'_r)\}$.
    It leaves unaffected the streams in $\mathcal{S}$ not concerned by p'_r's state change.

    $$\begin{aligned}\textit{install-streams}(p'_r, \mathcal{S}, \mathcal{P}_o) = \{\ & \langle si,so,o',sc\rangle \mid \langle si,o,pc\rangle \in \mathcal{P}_o, \langle so,o',pc'\rangle \in \mathcal{P}_o,\\ & \langle si,so,o,sc\rangle \in \mathcal{S}\ \wedge (\ (\langle si,so\rangle \in net(current(p'_r))\\ &\qquad \wedge o' = o \cup \{name(p'_r)\})\\ &\qquad \vee\ (\langle si,so\rangle \notin net(current(p'_r)) \wedge o' = o)\ )\\ & \vee\ (\langle si,so,o,sc\rangle \notin \mathcal{S}\ \wedge\ \langle si,so\rangle \in net(current(p'_r))\\ &\qquad \wedge\ o' = \{name(p'_r)\} \wedge sc = [\ ])\ )\ \}\end{aligned}$$

The complete rule is:

Rule 1 *: process transition at application level*

$$\boxed{\dfrac{\begin{array}{c}\exists p_r \in \mathcal{P}_r,\ p_r \xrightarrow[E_{ext},\, P_{act}]{} {}_p\ p'_r,\\ \mathcal{P'}_r = \{p' \mid\ \exists p \in (\mathcal{P}_r \setminus \{p_r\}) \cup \{p'_r\},\quad p' = diffact(p, E_{ext}, P_{act})\ \},\\ \mathcal{P'}_o = \textit{remove-if-death}(p'_r, \mathcal{P}_o),\\ \mathcal{S}' = \textit{install-streams}(p'_r,\ \textit{break-streams}(p_r, \mathcal{S})\ )\end{array}}{\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o\rangle \longrightarrow_a \langle \mathcal{P'}_r, \mathcal{S}', \mathcal{P'}_o\rangle}}$$

4.3.3 Passing a unit between ports and streams

We first consider the rules describing the behavior of each side, port and stream, when receiving or sending a unit. Then the rules that describe actual unit exchange are given, involving transitions on both sides.

Stream receiving a unit. The transition is labeled by ⟨*port name*⟩?⟨*unit*⟩, where ⟨*port name*⟩ is the name of the source port that produces ⟨*unit*⟩. This unit is appended to the contents of a stream connected to this port. The condition for this transition to be made, is that the stream is installed i.e., that its list *sl* of names of installer processes is not empty:

Rule 2 : *stream receiving a unit*

$$\frac{sl \neq \emptyset}{\langle so,si,sl,sc\rangle \xrightarrow{so?u} \langle so,si,sl,append(u,sc)\rangle}$$

Stream sending a unit. The transition is labeled by ⟨*port name*⟩!⟨*unit*⟩, where ⟨*port name*⟩ is the name of the sink port with which the communication is made, namely: to which the unit is sent i.e., the port receiving the unit. This unit is removed from the stream's non empty contents, if the stream is installed:

Rule 3 : *stream sending a unit*

$$\frac{sc \neq [\,] \qquad sl \neq \emptyset}{\langle so,si,sl,sc\rangle \xrightarrow{si!get(sc)} \langle so,si,sl,rest(sc)\rangle}$$

Port receiving a unit. The port p can make a transition when it receives a unit u sent explicitly to him, hence the label $p?u$; it appends it in its contents:

Rule 4 : *port receiving a unit*

$$\langle p,pc\rangle \xrightarrow{p?u} \langle p,append(u,pc)\rangle$$

Port sending a unit. The port p can send the first unit u taken from its non empty contents, and sends it to all the streams connected to it as a source; the label $p!u$ carries the port name:

Rule 5 : *port sending a unit*

$$\frac{pc \neq [\,]}{\langle p,pc\rangle \xrightarrow{p!get(pc)} \langle p,rest(pc)\rangle}$$

Passing a unit from a port to streams. A unit u passes from a port $P = \langle p, pc \rangle$ to streams in $\mathcal{S}$ if the port can make a sending transition and some streams in $\mathcal{S}$ can make a receiving transition for this port. Then, the application is modified in its ports set $\mathcal{P}_o$ and in its streams set $\mathcal{S}$, by the application level transition.

For this latter set of streams, we need to write that all the streams that can make a transition, do it, while the others remain in the same state. Therefore, we introduce a transition relation between sets, defined in terms of the possible transitions of its elements. For a set $\mathcal{E}$, the transition $\mathcal{E} \xrightarrow[\text{each}]{} \mathcal{E}'$ means that $\mathcal{E}'$ is the set of elements, for each element e of $\mathcal{E}$, either e' resulting from the application, when possible, of the transition to an element e of $\mathcal{E}$: $e \xrightarrow{} e'$, or e itself otherwise.

Rule 6 *: passing a unit from a port to streams*

$$\boxed{\dfrac{\exists P \in \mathcal{P}, P \xrightarrow{p!u} P', \quad \mathcal{S} \xrightarrow[\text{each}]{p?u} \mathcal{S}'}{\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o \rangle \xrightarrow{\qquad} \langle \mathcal{P}_r, \mathcal{S}', (\mathcal{P}_o \setminus \{P'\}) \cup \{P\} \rangle}}$$

Passing a unit from a stream to a port. A unit u passes from a stream S to a port $P = \langle p, pc \rangle$ if the stream can make a sending transition for this port and the port can make a receiving transition. Then, the application is modified in its set of ports $\mathcal{P}_o$ and in its streams set $\mathcal{S}$.

Furthermore, interactions between transitions at stream-level and at event-level must be handled here: the two predefined processes **guard(**⟨*port name*⟩**,**⟨*event*⟩**)** and **getunit(**⟨*port name*⟩**)** have an influence at the manifold level, by sending an event, or dying thus breaking the pipe-lines in which they are involved. This is captured by another transition relation: $\xrightarrow{p}$, labeled by the name of the port p receiving the unit (i.e., ⟨*process*⟩.⟨*port*⟩), and giving the new application state after the corresponding process transitions fired have been made. It is applied until no process can make a transition any more, which is noted by the star ($*$).

Rule 7 *: passing a unit from a stream to a port*

$$\boxed{\dfrac{\begin{array}{c}\exists S \in \mathcal{S}, S \xrightarrow{p!u} S', \quad \exists P \in \mathcal{P}, P \xrightarrow{p?u} P' \\ \langle \mathcal{P}_r, (\mathcal{S} \setminus \{S\}) \cup \{S'\}, \mathcal{P}_o \setminus \{P'\}) \cup \{P\} \rangle \xrightarrow{p}{}^{*} \langle \mathcal{P}'_r, \mathcal{S}', \mathcal{P}'_o \rangle\end{array}}{\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o \rangle \xrightarrow{\qquad} \langle \mathcal{P}'_r, \mathcal{S}', (\mathcal{P}'_o \rangle}}$$

The latter transition relation is defined for **guard(**⟨*port name*⟩**,**⟨*e*⟩**)** processes by the diffusion of the event occurrence ⟨*event*⟩.⟨*port name*⟩, and the death of the process (here it is a silent death, as **guard** can not be involved in a stream: hence it is just a removal from $\mathcal{P}_r$):

Rule 8 : *guard firing*

$$\frac{P_r = \langle \texttt{guard}(p,e), \texttt{active}, E\rangle \in \mathcal{P}_r, \quad \mathcal{P}'_r = \{P' \mid \exists P \in \mathcal{P}_r \setminus \{P_r\}, P' = \mathit{diffact}(P, \{e.p\}, \emptyset)\}}{\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o\rangle \xrightarrow{\quad p \quad} \langle \mathcal{P'}_r, \mathcal{S}, \mathcal{P}_o\rangle}$$

For `getunit(`$\langle$*port name*$\rangle$`)`, the transition consists in the removal of the pseudo-process from $\mathcal{P}_r$, and the unit in the port is put through the output port of the **getunit** process to the connected streams. An event **death.getunit(**$\langle$*port name*$\rangle$**)** is raised and diffused in the application: further transitions might lead to the breaking of streams involving the **getunit**. The unit arrived in the port p is the one obtained by the function *get*: it is received by the streams that can make the corresponding transition.

Rule 9 : *getunit firing*

$$\frac{\begin{array}{l} P_r = \langle \texttt{getunit}(p), \texttt{active}, E\rangle \in \mathcal{P}_r, \\ \langle p, pc\rangle \in \mathcal{P}_o, \quad \mathcal{S} \xrightarrow{\texttt{getunit}(p).\texttt{output}!get(pc)} \mathcal{S}', \\ \mathcal{P}'_r = \{P' \mid \exists P \in \mathcal{P}_r \setminus \{P_r\}, P' = \mathit{diffact}(P, \{\texttt{death.getunit}(p)\}, \emptyset)\} \end{array}}{\langle \mathcal{P}_r, \mathcal{S}, \mathcal{P}_o\rangle \xrightarrow{\quad p \quad} \langle \mathcal{P'}_r, \mathcal{S}', \mathcal{P}_o\rangle}$$

It can be noted that these rules are non-deterministic with regard to the order of raising of **guard** events or firing of **getunits**, which corresponds to assumptions of asynchrony taken in MANIFOLD [1].

This completes the specification integrating event-driven transitions and unit exchange through the data-flow network.

5 Conclusion

In this paper, we first recall, only briefly, the main lines of the preliminary formal specification of MANIFOLD [8] on which this work is based. We then present a specification of the event-driven mechanism of MANIFOLD in the ASF+SDF formalism. Formally, this contributes to the clarification of some imprecise aspects of the preliminary specification [8]. Practically, it also provides us with a programming environment comprising a syntax-directed editor, a parser, a checker for some aspects of the static semantics of MANIFOLD, and an interpreter for its dynamic semantics; this is a formal improvement on the previous *ad hoc* implementation of an interpreter in PROLOG.

We then present an extension to the specification, integrating the unit-exchange mechanism of MANIFOLD, that describes the behavior of the data-flow components: streams and ports. The rules are added up the previous specification of the event-driven part, into a coherent whole. This augmented specification could be reformulated and implemented in ASF+SDF as well as the preliminary one: it would require the adaptation of the application level, and the addition of the unit exchange rules on top of the existing ones.

Acknowledgements

Thanks to Paul Klint, Jan Rekers, Frank Tip and Jasper Kamperman for their help and advice with ASF+SDF, and last but not least to Anco Smit for his cooperation on a user interface for the interpreter.

References

[1] F. Arbab. *Specification of* MANIFOLD. CWI Report, Interactive Systems Dept., CS-R 9220, 1992.

[2] F. Arbab, I. Herman, P. Spilling. An overview of MANIFOLD and its implementation. *Concurrency: Practice and Experience*, to appear.

[3] A. van Deursen. *An algebraic specification for the static semantics of Pascal.* CWI Report, Dept. of Software Technology, CS-R 9129, 1991.

[4] A. van Deursen. *Specification and generation of a λ-calculus environment.* CWI Report, Dept. of Software Technology, CS-R 9233, 1992.

[5] J. Kamperman. *An SDF specification of the* MANIFOLD *syntax.* Working paper, 1991.

[6] P. Klint. A meta-environment for generating programming environments. In *Proc. of the METEOR workshop on Methods Based on Formal Specification*, J.A. Bergstra and L.M.G. Feijs *eds.*, Springer-Verlag, LNCS 490, 1991.

[7] G.D. Plotkin. *A structural approach to operational semantics.* Aarhus University, Computer Science Dept., Report DAIMI FN–19, Sept. 1981.

[8] E.P.B.M. Rutten, F. Arbab, I. Herman. *Formal Specification of* MANIFOLD*: a Preliminary Study.* CWI Report, Interactive Systems Dept., CS-R 9215, 1992.

[9] E.P.B.M. Rutten. MINIFOLD*: a kernel for the coordination language* MANIFOLD. CWI Report, Interactive Systems Dept., to appear, 1992.

A The specification of the subset of MANIFOLD in ASF+SDF

A.1 Syntax

A.1.1 Layout

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%          Module Layout
%%
%% This module defines the Layout for
%% Manifold programs.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

exports
  lexical syntax

    "//" ~[\n]* "\n"            -> LAYOUT
    [ \t\n]                     -> LAYOUT
```

A.1.2 Identifiers

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%          Module Identifiers
%%
%% This Module defines the syntax of
%% Manifold Identifiers.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Layout

exports
  sorts IDENT

  lexical syntax
    [a-zA-Z_] [a-zA-Z0-9_]*     -> IDENT

  variables
    Id [0-9']*                  -> IDENT
```

A.1.3 Programs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
```

```
%%        Module Manifold-Syntax
%%
%% This Module defines the syntax of
%% Manifold Programs.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Identifiers

exports
  sorts PROC-N PORT-N EV-N PORT SOURCE EV-OCC

        ACTION STREAM PIPELINE GROUP ACT MANNERCALL

        HANDLER LABEL BLOCK BODY
        DECL-EV DECL-PORT-IN DECL-PORT-OUT DECL-PORT
        ELEMENT APPLICATION


  context-free syntax


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %%      Basic elements of the language
    %%
    %%      processes - ports - events

    IDENT                                 -> PROC-N
    self                                  -> PROC-N
    system                                -> PROC-N


    IDENT                                 -> PORT-N
    input                                 -> PORT-N
    output                                -> PORT-N

    PROC-N"."PORT-N                       -> PORT

    "*"                                   -> SOURCE
    PROC-N                                -> SOURCE
    PORT                                  -> SOURCE

    IDENT                                 -> EV-N
    death                                 -> EV-N
    returned                              -> EV-N
    abort                                 -> EV-N
    terminate                             -> EV-N
    lambda                                -> EV-N
    start                                 -> EV-N
    break                                 -> EV-N
```

```
"*"".""SOURCE                      -> EV-OCC
EV-N"."SOURCE                      -> EV-OCC


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Actions performed by Manifold Processes
%%
%% atomic action - streams installation
%% pipeline installation - group of actions
%% manner call.
%% Note the absence of getunit and guard,
%% which are not covered by this implementation

do EV-N                            -> ACTION
raise EV-N                         -> ACTION
broadcast EV-N                     -> ACTION
return                             -> ACTION
ignore                             -> ACTION
save                               -> ACTION
activate PROC-N                    -> ACTION
deactivate  PROC-N                 -> ACTION

PORT"->"PORT                       -> STREAM

STREAM                             -> PIPELINE
"["{STREAM","}+"]"                 -> PIPELINE

"("{ ACT","}*")"                   -> GROUP

GROUP                              -> ACT
PIPELINE                           -> ACT
ACTION                             -> ACT

IDENT                              -> MANNERCALL


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Structure of the body of a manifold or
%%  of a manner.
%%
%%  handler - label - block - body

ACT                                -> HANDLER
MANNERCALL                         -> HANDLER

{EV-OCC","}+                       -> LABEL

LABEL":"HANDLER"."                 -> BLOCK
```

```
    "{"BLOCK+"}"                        -> BODY


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %%  Declaration of processes and manners
    %%           Global Application
    %%
    %%  events declaration - ports declaration
    %%  process or manner declaration - application

    event {IDENT","}+"."                -> DECL-EV
                                        -> DECL-EV

    in {PORT-N","}+"."                  -> DECL-PORT-IN
                                        -> DECL-PORT-IN
    out {PORT-N","}+"."                 -> DECL-PORT-OUT
                                        -> DECL-PORT-OUT
    DECL-PORT-IN DECL-PORT-OUT          -> DECL-PORT

    manifold IDENT DECL-PORT BODY       -> ELEMENT
    atomic IDENT DECL-PORT DECL-EV      -> ELEMENT
    manner IDENT BODY                   -> ELEMENT

    ELEMENT+                            -> APPLICATION


variables

    Pn  [0-9']*                         -> PROC-N
    Pon [0-9']*                         -> PORT-N
    Po  [0-9']*                         -> PORT
    Sc  [0-9']*                         -> SOURCE
    En  [0-9']*                         -> EV-N
    Eo  [0-9']*                         -> EV-OCC
    EoP [0-9']*                         -> {EV-OCC ","}+
    EoL [0-9']*                         -> {EV-OCC ","}*

    AtAc[0-9']*                         -> ACTION
    St  [0-9']*                         -> STREAM
    StP [0-9']*                         -> {STREAM","}+
    Pi  [0-9']*                         -> PIPELINE
    Gr  [0-9']*                         -> GROUP
    Ac  [0-9']*                         -> ACT
    AcL [0-9']*                         -> {ACT","}*
    Mc  [0-9']*                         -> MANNERCALL

    Hd  [0-9']*                         -> HANDLER
    Lb  [0-9']*                         -> LABEL
    Bl  [0-9']*                         -> BLOCK
```

```
BlP [0-9']*                         -> BLOCK+
Bd  [0-9']*                         -> BODY
DcE [0-9']*                         -> DECL-EV
IdP [0-9']*                         -> { IDENT "," }+
DcP [0-9']*                         -> DECL-PORT
DcPO[0-9']*                         -> DECL-PORT-OUT
PonP[0-9']*                         -> { PORT-N "," }+
El  [0-9']*                         -> ELEMENT
ElP [0-9']*                         -> ELEMENT+
Ap  [0-9']*                         -> APPLICATION
```

A.1.4 Specifications

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                    MODULE  Internal-Syntax
%%
%% This module defines the abstract syntax used for the specification
%% of Manifold.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


imports Manifold-Syntax

exports
  sorts ACTIV ACTIV-SET  EV-OCC-SET  A-ENV

        MANNER-REP MANNER-REP-SET ATOMIC-REP ATOMIC-REP-SET
        CALLED-MAN BLOCKS PROC-ST PROC-ST-AT BLOCK-ST BLOCK-ST-AT P-STATE

        P-STATE-SET  AP-STATE PROC-N-SET

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Action level.
    %% The environment of an action is defined as a 4-tuple:
    %% - set of events raised within the executing manifold,
    %% - set of events raised outside the executing manifold,
    %% - group of pipelines under the control of the executing manifold
    %% - activation set: set of processes that must be (de) activated.

    act(PROC-N)                                          -> ACTIV
    deact(PROC-N)                                        -> ACTIV

    "{" {ACTIV","}* "}"                                  -> ACTIV-SET

    "{" {EV-OCC","}* "}"                                 -> EV-OCC-SET
```

```
ae(EV-OCC-SET, EV-OCC-SET, GROUP, ACTIV-SET)        -> A-ENV

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Process level.  (The handler and search levels are not part of
%%                  the internal syntax, but are hidden in the
%%                  Process module)
%%
%% The state of a manifold process is defined as a triple:
%% - state of the program of the manifold,
%% - state of the current block of the manifold,
%% - set of event occurrences observed by the manifold.
%%
%% The state of an atomic process is defined as a triple:
%% - name of the atomic process,
%% - activation state of the atomic process,
%% - set of event occurrences observed by the atomic process.
%%
%% The internal representation of a manner is given as:
%% - the name of this manner,
%% - its body
%% A set of such internal representations for all manners in the
%% application is used to acces the body of a called manner.
%%
%% The internal representation of an atomic process is given as:
%% - the name of this process,
%% - the events it can raise.
%% A set of such internal representations for all atomic processes
%% in the application is used to know which event can be raised by
%% an atomic process.


MANNERCALL BODY                                     -> MANNER-REP
"{" {MANNER-REP","}* "}"                            -> MANNER-REP-SET

PROC-N  EV-OCC-SET                                  -> ATOMIC-REP
"{" {ATOMIC-REP","}* "}"                            -> ATOMIC-REP-SET


manner MANNER-REP                                   -> CALLED-MAN

BLOCK+                                              -> BLOCKS
CALLED-MAN BLOCKS                                   -> BLOCKS

manifold PROC-N "{" BLOCKS "}"                      -> PROC-ST
atomic PROC-N                                       -> PROC-ST-AT


inactive                                            -> BLOCK-ST
deactivated                                         -> BLOCK-ST
end                                                 -> BLOCK-ST
```

```
  BLOCK                                                 -> BLOCK-ST
  inactive                                              -> BLOCK-ST-AT
  active                                                -> BLOCK-ST-AT
  deactivated                                           -> BLOCK-ST-AT

  ps(PROC-ST, BLOCK-ST, EV-OCC-SET)                     -> P-STATE
  ps(PROC-ST-AT, BLOCK-ST-AT, EV-OCC-SET)               -> P-STATE


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%
  %%                       APPLICATION LEVEL
  %%
  %% The state of an application is defined as a triple:
  %%
  %% - set of states of all processes
  %% - set of internal representations of all manners
  %% - set of internal representation of all atomic processes
  %%
  %% The sort of sets of processes (names) is defined, in order
  %% to have the user select among a set of processes, the
  %% process that must make the next transition.

  "{" {P-STATE","}* "}"                                 -> P-STATE-SET

  aps(P-STATE-SET, MANNER-REP-SET, ATOMIC-REP-SET)      -> AP-STATE

  "{" {PROC-N","}* "}"                                  -> PROC-N-SET

variables
  Av  [0-9']*                                           -> ACTIV
  AvL [0-9']*                                           -> {ACTIV","}*
  AvS [0-9']*                                           -> ACTIV-SET
  EoS [0-9']*                                           -> EV-OCC-SET
  Ae  [0-9']*                                           -> A-ENV

  MnS [0-9']*                                           -> MANNER-REP-SET
  MnL [0-9']*                                           -> {MANNER-REP","}*
  AtS [0-9']*                                           -> ATOMIC-REP-SET
  AtL [0-9']*                                           -> {ATOMIC-REP","}*

  Cm  [0-9']*                                           -> CALLED-MAN
  Bls [0-9']*                                           -> BLOCKS
  PSt [0-9']*                                           -> PROC-ST
  PStAt[0-9']*                                          -> PROC-ST-AT
  BlSt[0-9']*                                           -> BLOCK-ST
  BlStAt[0-9']*                                         -> BLOCK-ST-AT
  BlL [0-9']*                                           -> BLOCK*
  Ps  [0-9']*                                           -> P-STATE

  PsL [0-9']*                                           -> {P-STATE","}*
  PsS [0-9']*                                           -> P-STATE-SET
```

```
    PnL [0-9']*                                         -> {PROC-N ","}*
    PnS [0-9']*                                         -> PROC-N-SET
    Aps [0-9']*                                         -> AP-STATE
```

A.2 Static semantics

A.2.1 Translation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                     Module  Translation
%%
%% This module defines a language for building a table representing
%% a manifold program. This table enables more easily:
%% - the static verification of programs
%% - the translation of programs in the Manifold syntax into
%%   programs of the internal syntax.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Util-Sets

exports
  sorts PORT-SET PORT-INFO EV-INFO CALLS-INFO INFO
        TYPE-PROC TYPE ENTRY TABLE

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Definition of a table as a non empty list of entries, each
    %% of them corresponding to an element of the application.
    %% Each entry contains the type of the element, its name, and
    %% additional information.
    %%
    %% The additional information is:
    %% - for a manifold process:
    %%   - its set of input and output ports
    %%   - the set of events it can raise internally or externally
    %%   - its body
    %% - for an atomic process:
    %%   - its set of input and output ports
    %%   - the set of events it can raise internally or externally
    %%     (the former being the empty set)
    %% - for a manner:
    %%   - its body
    %%   - the set of processes from which the manner can be directly
    %%     or indirectly called, as well as the boolean saying
    %%     whether the manner can be called recursively

    "{" { PORT","} + "}"                                 -> PORT-SET
```

```
   in PORT-SET "," out PORT-SET                              -> PORT-INFO
   loc-ev EV-OCC-SET "," glob-ev EV-OCC-SET                  -> EV-INFO
   called-from PROC-N-SET  and BOOL                          -> CALLS-INFO
   "(" PORT-INFO "," EV-INFO "," BODY ")"                    -> INFO
   "(" PORT-INFO "," EV-INFO ")"                             -> INFO
   "(" BODY "," CALLS-INFO")"                                -> INFO

   manifold                                                  -> TYPE-PROC
   atomic                                                    -> TYPE-PROC
   manner                                                    -> TYPE
   TYPE-PROC                                                 -> TYPE

    "<" TYPE "," PROC-N "," INFO ">"                         -> ENTRY
    "{" ENTRY+ "}"                                           -> TABLE


   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   %%
   %%        Building the table, and translation Manifold-Syntax ->
   %%        Internal-Syntax
   %%
   %% build-table(Ap) is the table corresponding to the manifold
   %% application Ap.
   %% build-appli(Tb) is the initial state of the application whose
   %% table is Tb.

   build-table(APPLICATION)                                  -> TABLE
   build-appli(TABLE)                                        -> AP-STATE


 variables
   PoP  [0-9']*                                              -> { PORT","}+
   PoL  [0-9']*                                              -> { PORT","}*
   IdL  [0-9']*                                              -> { IDENT ","}*
   PoS  [0-9']*                                              -> PORT-SET
   Tp   [0-9']*                                              -> TYPE-PROC
   PoInf[0-9']*                                              -> PORT-INFO
   EvInf[0-9']*                                              -> EV-INFO
   CaInf[0-9']*                                              -> CALLS-INFO
   Inf  [0-9']*                                              -> INFO
   EnT  [0-9']*                                              -> ENTRY
   EnTP [0-9']*                                              -> ENTRY+
   EnTL [0-9']*                                              -> ENTRY*
   Tb   [0-9']*                                              -> TABLE

hiddens

 context-free syntax

   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   %%
   %% Additional constructs allowing to build the table more easily
```

```
%%
%% - build-table(Ap,Ap') is the table corresponding to the
%%   part Ap of the application Ap'.
%% - build-ports(Tp Pn, DcP) is the set of input and output
%%   ports of process with type Tp, name Pn, and ports declaration
%%   DcP.
%% - build-events(manifold Pn, Bd) is the set of internally
%%   and externally raised events of the manifold Pn, according
%%   to its body Bd.
%% - build-events(atomic Pn, DcE) is the set of events that
%%   can be emitted by the atomic process Pn, according to its
%%   event declaration DcE.
%% - build-calls(Id,Ap) is a pair consisting of the set of
%%   processes that can call manner Id in application Ap, and
%%   a boolean saying whether Id can be called recursively in
%%   Ap.

build-table(APPLICATION, APPLICATION)              -> TABLE
build-ports"("TYPE-PROC PROC-N"," DECL-PORT")"     -> PORT-INFO
build-events"("manifold PROC-N"," BODY")"          -> EV-INFO
build-events"("atomic PROC-N"," DECL-EV")"         -> EV-INFO
build-calls(IDENT, APPLICATION)                    -> CALLS-INFO


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% In the table and in the internal syntax, all occurrences of
%% a process name inside the process of same name are replaced
%% by self.
%%
%% - make-self(Pn,Bd) is the body Bd where all occurrences of Pn
%%   are replace by self.
%% - make-self(Pn,Lb) is the label Lb where all occurrences of Pn
%%   are replace by self.
%% - ....
%% - and so on for the other constructs.

make-self(PROC-N, BODY)                            -> BODY
make-self-lab(PROC-N, LABEL)                       -> LABEL
make-self-hand(PROC-N, HANDLER)                    -> HANDLER
make-self-sc(PROC-N, SOURCE)                       -> SOURCE
make-self-port(PROC-N, PORT)                       -> PORT
make-self-proc(PROC-N, PROC-N)                     -> PROC-N


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% look-for-manner Id in Bd is a boolean saying whether manner Id
%% is called in body Bd

look-for-manner IDENT in BODY                      -> BOOL
```

```
 equations

// Since ports input and output are put by default in the set
// of ports of a process, they are removed in case the user
// declared them.

[TR1a] in{PoL1, Pn.input, PoL2, Pn.input, PoL3}, out PoS =
       in{PoL1, Pn.input, PoL2, PoL3}, out PoS

[TR1b] in PoS , out {PoL1, Pn.output, PoL2, Pn.output, PoL3} =
       in PoS, out {PoL1, Pn.output, PoL2, PoL3}


/////////////////////
// building the table
/////////////////////


[TR2a] build-table(Ap) = build-table(Ap,Ap)


// part corresponding to a manifold process

[TR2b] build-table(manifold Id DcP Bd,Ap) =
       { < manifold, Id, ( build-ports(manifold Id, DcP),
                           build-events(manifold Id,Bd),
                           make-self(Id,Bd) ) > }


// part corresponding to an atomic process

[TR2c] build-table(atomic Id DcP DcE,Ap) =
       { < atomic, Id, ( build-ports(atomic Id, DcP),
                         build-events(atomic Id,DcE) ) > }


// part corresponding to a manner

[TR2d] build-table(manner Id Bd,Ap) = { < manner, Id, (Bd,build-calls(Id,Ap)) > }


// putting parts together

[TR2e] build-table(El,Ap) = {EnT},
       build-table(ElP,Ap) = {EnTP}
       ====================================
       build-table(El ElP, Ap) = {EnT EnTP}



//////////////////////////////////////////////////////////
```

```
// building the set of input and output ports of a process
/////////////////////////////////////////////////////////

// default ports: input and output

[TR3a] build-ports(Tp Pn, ) = in {Pn.input} , out {Pn.output}


// building the set of output ports

[TR3b] build-ports(Tp Pn, ) = in PoS , out { PoP}
       ==========================================================
       build-ports(Tp Pn, out Pon.) =  in PoS , out {Pn.Pon, PoP}

[TR3c] build-ports(Tp Pn, out PonP.) = in PoS, out {PoP}
       ================================================================
       build-ports(Tp Pn, out Pon, PonP.) = in PoS , out {Pn.Pon, PoP}


// building the sets of input ports

[TR3d] build-ports(Tp Pn, DcPO) = in {PoP} , out PoS
       ===========================================================
       build-ports(Tp Pn,in Pon. DcPO) = in {Pn.Pon, PoP}, out PoS

[TR3e] build-ports(Tp Pn, in PonP. DcPO) = in {PoP'} , out PoS
       ===================================================================
       build-ports(Tp Pn,in Pon, PonP. DcPO) = in {Pn.Pon, PoP'}, out PoS



/////////////////////////////////////////////////////////////
// building the set of externally raised events by an process
/////////////////////////////////////////////////////////////


// for an atomic process: done with regard to the declarations

[TR4a] build-events(atomic Pn, ) = loc-ev{}, glob-ev{}

[TR4b] build-events(atomic Pn, ) = loc-ev EoS, glob-ev{EoL}
       ====================================================================
       build-events(atomic Pn,event Id.) = loc-ev EoS, glob-ev{Id.Pn, EoL}

[TR4c] build-events(atomic Pn,event IdP.) = loc-ev EoS, glob-ev{EoL}
       =========================================================================
       build-events(atomic Pn,event Id,IdP.) = loc-ev EoS, glob-ev{Id.Pn, EoL}


// for a manifold process: done with regard to its body
// the instructions do, raise, and broadcast are the actions from
// which the sets are built

[TR4d] build-events(manifold Pn, { Lb: do En.}) = loc-ev{En.Pn}, glob-ev{}

[TR4e] build-events(manifold Pn,{ Lb: raise En.}) = loc-ev{} , glob-ev{En.Pn}

[TR4f] build-events(manifold Pn,{ Lb: broadcast En.}) =
                                             loc-ev{En.Pn} , glob-ev{En.Pn}


// the other actions does not change anything

[TR4g] build-events(manifold Pn,{ Lb: return.}) = loc-ev{}, glob-ev{}

[TR4h] build-events(manifold Pn,{ Lb: ignore.}) = loc-ev{}, glob-ev{}

[TR4i] build-events(manifold Pn,{ Lb: save.}) = loc-ev{}, glob-ev{}

[TR4j] build-events(manifold Pn,{ Lb: activate Pn'.}) = loc-ev{}, glob-ev{}

[TR4k] build-events(manifold Pn,{ Lb: deactivate Pn'.}) = loc-ev{}, glob-ev{}

[TR4l] build-events(manifold Pn,{ Lb: Pi.}) = loc-ev{}, glob-ev{}


// putting results together at the group level

[TR4m] build-events(manifold Pn,{ Lb: (). }) = loc-ev{}, glob-ev{}

[TR4n] build-events(manifold Pn,{ Lb: Ac. }) = loc-ev EoS1, glob-ev EoS2,
       build-events(manifold Pn,{ Lb: (AcL). }) = loc-ev EoS1', glob-ev EoS2'
       =====================================================
       build-events(manifold Pn,{ Lb: (Ac,AcL). }) =
       loc-ev EoS1 union-e EoS1',glob-ev EoS2 union-e EoS2'


// a manner call does not change anything

[TR4o] build-events(manifold Pn,{ Lb: Mc. }) = loc-ev{}, glob-ev{}


// putting results together at the block level

[TR4p] build-events(manifold Pn, { Bl }) = loc-ev EoS1, glob-ev EoS2,
       build-events(manifold Pn, { BlP }) = loc-ev EoS1',glob-ev EoS2'
       ============================================================
       build-events(manifold Pn,{ Bl BlP }) =
                   loc-ev EoS1 union-e EoS1',glob-ev EoS2 union-e EoS2'
```

```
///////////////////////////////////////////////
// replacing the occurrences of a process
// name by self inside the body of this process
///////////////////////////////////////////////


// occurrences in a block

[TR5a] make-self(Pn,{Lb:Hd.}) = {make-self-lab(Pn,Lb): make-self-hand(Pn,Hd).}

[TR5b] make-self(Pn,{Bl}) = {Bl'},
       make-self(Pn,{BlP}) = {BlP'}
       ====================================
       make-self(Pn,{Bl BlP}) = {Bl' BlP'}


// occurrences in a label

[TR5c] make-self-lab(Pn,*.Sc) = *.make-self-sc(Pn,Sc)

[TR5d] make-self-lab(Pn,En.Sc) = En.make-self-sc(Pn,Sc)

[TR5e] make-self-lab(Pn,Eo) = Eo',
       make-self-lab(Pn,EoP) =  EoP'
       ====================================
       make-self-lab(Pn,Eo,EoP) = Eo',EoP'


// occurrences in a handler

[TR5f] make-self-hand(Pn,Mc) = Mc

[TR5g] make-self-hand(Pn,AtAc) = AtAc

[TR5h] make-self-hand(Pn,[Po1->Po2]) =
                 [ make-self-port(Pn,Po1) -> make-self-port(Pn,Po2) ]

[TR5i] make-self-hand(Pn,St) = St',
       make-self-hand(Pn,[StP]) = [StP']
       ========================================
       make-self-hand(Pn,[St,StP]) = [St',StP']

[TR5j] make-self-hand(Pn,Po1 -> Po2) =
                           make-self-port(Pn,Po1) -> make-self-port(Pn,Po2)

[TR5k] make-self-hand(Pn,()) = ()

[TR5l] make-self-hand(Pn,Ac) = Ac',
       make-self-hand(Pn,(AcL)) = (AcL')
       ========================================
       make-self-hand(Pn,(Ac,AcL)) = (Ac',AcL')
```

```
// occurrences in a source

[TR5m] make-self-sc(Pn,*) = *

[TR5n] make-self-sc(Pn,Po) = make-self-port(Pn,Po)

[TR5o] make-self-sc(Pn,Pn') = make-self-proc(Pn,Pn')


// occurrences in a port

[TR5p] make-self-port(Pn,Pn1.Pon) = make-self-proc(Pn,Pn1).Pon


// occurrences in a process name

[TR5q] make-self-proc(Pn,Pn) = self

[TR5r] Pn != Pn'
       ============================
       make-self-proc(Pn,Pn') = Pn'



//////////////////////////////////////////
// Building the manner information
//////////////////////////////////////////


// a manner cannot be called by an atomic process

[TR6a] build-calls(Id,atomic Id' DcP DcE) = called-from{} and false


// a manner might be called by a manifold process

[TR6b] look-for-manner Id in Bd = true
       ================================================================
       build-calls(Id,manifold Id' DcP Bd) = called-from{Id'} and false

[TR6c] look-for-manner Id in Bd = false
       ==============================================================
       build-calls(Id, manifold Id' DcP Bd) = called-from{} and false


// a manner might be called by a manner including by itself (in that
// case this is a recursive call)

[TR6d] Id != Id',
       look-for-manner Id in Bd = true
       ==========================================================
       build-calls(Id,manner Id' Bd) = called-from{Id'}  and false

[TR6e]look-for-manner Id in Bd = true
       =====================================================
       build-calls(Id,manner Id Bd) = called-from{}  and true

[TR6f] Id != Id',
       look-for-manner Id in Bd = false
       =======================================================
       build-calls(Id, manner Id' Bd) = called-from{} and false

[TR6g] look-for-manner Id in Bd = false
       ======================================================
       build-calls(Id, manner Id Bd) = called-from{} and false


// putting result together

[TR6h] build-calls(Id,El) = called-from{ PnL1 } and Bool1,
       build-calls(Id,ElP) = called-from{ PnL2 }and Bool2
       ==================================================================
       build-calls(Id,El ElP) = called-from{ PnL1, PnL2 } and Bool1 | Bool2


// transitive closure of the set:

// if a manner is called from another manner, then this manner is replaced
// in the set by the set of processes that call it.

[TR6i]{EnTL1 <manner,Id1,(Bd1,called-from{PnL1,Id2,PnL2} and Bool1)>
        EnTL2 <manner,Id2,(Bd2,called-from{PnL} and Bool2)> EnTL3} =
       {EnTL1 <manner,Id1,(Bd1,called-from{PnL1,PnL2,PnL} and Bool1)>
        EnTL2 <manner,Id2,(Bd2,called-from{PnL} and Bool2)> EnTL3}

[TR6j]{EnTL1 <manner,Id1,(Bd1,called-from{PnL} and Bool1)>
        EnTL2 <manner,Id2,(Bd2,called-from{PnL1,Id1,PnL2} and Bool2)> EnTL3} =
       {EnTL1 <manner,Id1,(Bd1,called-from{PnL} and Bool1)>
        EnTL2 <manner,Id2,(Bd2,called-from{PnL1,PnL2,PnL} and Bool2)> EnTL3}


// if a manner  is called by itself, it is suppressed from its own set, and
// the recursivity boolean is set to true

[TR6k] <manner,Id,(Bd,called-from{PnL1,Id,PnL2} and Bool1 )> =
       <manner,Id,(Bd,called-from{PnL1,PnL2} and true)>


// multiple occurrences from the same process in a set are suppressed

[TR6l] called-from{PnL1,Pn,PnL2,Pn,PnL3} and Bool =
                                 called-from{PnL1,Pn,PnL2,PnL3} and Bool
```

```
////////////////////////////////////////////////////
//Building the initial state of an application from
//its table
////////////////////////////////////////////////////


// initial state of a manifold process other than the main process

[TR7a] Pn != main
       ==================================================
       build-appli({<manifold ,Pn,(PoInf,EvInf,{BlP})>}) =
       aps( {ps(manifold Pn{BlP},inactive,{})},{},{})


// initial state of the main process

[TR7b] build-appli({<manifold ,main,(PoInf,EvInf,{BlP})>}) =
       aps( {ps(manifold main{BlP},inactive,{start.self})},{},{})


// initial state and internal representation of an atomic process

[TR7c] build-appli({<atomic,Id,(PoInf,loc-ev{},glob-ev EoS)>}) =
       aps({ps(atomic Id,inactive,{})},{},{Id EoS})


// internal representation of a manner

[TR7d] build-appli({<manner, Id, (Bd, CaInf)>}) = aps({},{Id Bd},{})


// initial state of the whole application

[TR7e] build-appli({EnT}) = aps({PsL},{MnL},{AtL}),
       build-appli({EnTP}) = aps({PsL'},{MnL'},{AtL'})
       =================================================================
       build-appli({EnT EnTP}) = aps({PsL, PsL'},{MnL, MnL'},{AtL, AtL'})




// is a manner called in a body

[TR8a] look-for-manner Id in {Lb:Id.} = true

[TR8b] Id != Id'
       ========================================
       look-for-manner Id in {Lb:Id'.} = false

[TR8c] look-for-manner Id in {Lb:Ac.} = false

[TR8d] look-for-manner Id in {Bl BlP} =
              look-for-manner Id in {Bl} | look-for-manner Id in {BlP}
```

A.2.2 Static semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%              Module  Static-Semantics
%%
%% This modules checks statically manifold programs from
%% the information contained in their corresponding table.
%%
%% Note that only errors that can prevent the interpretation
%% of programs are checked. As a consequence the existence of
%% an event occurrence of a label is not checked, even if
%% this can be done from the information in the table.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
imports Translation

hiddens
  sorts ERR-LOC ERROR

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Location of errors can be
    %% - in the text of a manifold program
    %% - in the text of an atomic process
    %% - in the text of a manner
    %% - at the level of the global program (e.g. declaration
    %%   of two processes with the same name)
    %% - in the text of a manner when considered in the context
    %%   of a (direct or indirect) call from a manifold (e.g.
    %%   the manner can refer to a port self.Pon, where Pon
    %%   is not a port of the considered manifold).

    "error in manifold" PROC-N ":"                        -> ERR-LOC
    "error in atomic process" PROC-N ":"                  -> ERR-LOC
    "error in manner" IDENT ":"                           -> ERR-LOC
    "error :"                                             -> ERR-LOC
    "error in manner" IDENT "when called from" PROC-N ":"-> ERR-LOC


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Errors detected.
    %%
    %% note that no matter whether a port of a process does not
    %% exist, or exists as an input (resp. output) port but is
    %% employed as an output (resp. input) port, the same error
    %% is produced: port is not an input port (resp. port is
    %% not an output port).

    "port" PORT-N "is double declared"                    -> ERROR
    "event" EV-N "is double declared"                     -> ERROR
    "process or manner" PROC-N "is double declared"       -> ERROR
    "manner" IDENT "is unknown"                           -> ERROR
    "process" PROC-N "is unknown"                         -> ERROR
    "port" PORT "is not an input port"                    -> ERROR
    "port" PORT "is not an output port"                   -> ERROR
    "port output is declared as an input port"            -> ERROR
    "port input is declared as an output port"            -> ERROR
    "no handler for start.self"                           -> ERROR
    "no manifold called main"                             -> ERROR
    "recursivity risks"                                   -> ERROR

  variables

    ErLo [0-9']*                                          -> ERR-LOC


exports
  sorts ERR-MESS ERR-MESS-LIST

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% An error message is composed of a location and of an error
    %% Checking an application can be done from the original syntax
    %% by transiting to the corresponding table, or from the table
    %% directly. Checking an application produces a possibly empty
    %% list of error messages.

    ERR-LOC ERROR                                         -> ERR-MESS
    {ERR-MESS "."}*                                       -> ERR-MESS-LIST

    check(APPLICATION)                                    -> ERR-MESS-LIST
    check-table( TABLE )                                  -> ERR-MESS-LIST


 variables

    ErM  [0-9']*                                          -> ERR-MESS
    ErML [0-9']*                                          -> {ERR-MESS "."}*


hiddens
  sorts CHECK
```

```
context-free syntax

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%
  %% Additional constructs.
  %%
  %% - check-entry(Tb1,Tb2) is the list of error messages detected
  %%   during the analysis of the part Tb1 of the table Tb2.
  %% - check-body(Pn, Bd, Tb) is the list of error messages detected
  %%   during the analysis of the body Bd of process Pn in the application
  %%   whose table is Tb.
  %% - check-manner(Id,Inf,Tb) is the list of error messages detected
  %%   during the analysis of the information Inf concerning manner Id
  %%   in the application whose table is Tb.
  %% - check-man-body1(Id,Bd,Tb) is the list of error messages detected
  %%   during the analysis of the body Bd of manner Id in the application
  %%   whose table is Tb.
  %% - check-man-body2(Id,Pn,Bd,Tb) is the list of error messages detected
  %%   during the analysis of the body Bd of manner Id when called from
  %%   manifold Pn in the application whose table is Tb.

  check-entry(TABLE, TABLE)                             -> ERR-MESS-LIST
  check-body(PROC-N, BODY, TABLE)                       -> ERR-MESS-LIST
  check-manner(IDENT, INFO, TABLE)                      -> ERR-MESS-LIST
  check-mann-body1(IDENT, BODY, TABLE)                  -> ERR-MESS-LIST
  check-mann-body2(IDENT, PROC-N, BODY, TABLE)          -> ERR-MESS-LIST

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%
  %% - in check-double-proc-names(Tb,ErML), ErML is the list of error
  %%   messages related to the declaration of multiple elements with the
  %%   same name in Tb.
  %% - in check-main(Tb,ErML), ErML is the list of error messages related
  %%   to the absence of declaration of a main manifold in Tb.
  %% - in check-double-decl-ports(PoInf,ErLo,ErML), ErML is the list of
  %%   error messages related to the declaration of multiple ports with the
  %%   same name in PoInf, for a process specified in location ErLo.
  %% - in check-double-decl-ports(EvInf,ErLo,ErML), ErML is the list of
  %%   error messages related to the declaration of multiple events with the
  %%   same name in EvInf, for a process specified in location ErLo.
  %% - in check-manner-call(Mc,Tb,ErML), ErML is the list of error messages
  %%   related to the call of a manner Mc that does not exist in table Tb.
  %% - in check-activate (Pn,Tb,ErML), ErML is the list of error messages
  %%   related to the activation of a process Pn that does not exist in
  %%   table Tb.
  %% - in check-start-self(Bd,ErML), ErML is the list of error messages
  %%   related to the absence of an handler for start.self in body Bd.
  %% - in check-port-in(Po,Pn,Tb,ErML), ErML is the list of error messages
  %%   related to the use of port Po as an input port, in the body of
  %%   process Pn for an application with table Tb.
  %% - in check-port-out(Po,Pn,Tb,ErML), ErML is the list of error messages
```

```
    %%   related to the use of port Po as an output port, in the body of
    %%   process Pn for an application with table Tb.
    %% - in check-recursivity(Bool,ErML), ErML is the list of error messages
    %%   related to the recursive call of a manner whose boolean indicating
    %%   the presence of recursivity is Bool.

    check-double-proc-names(TABLE, ERR-MESS-LIST )        -> CHECK
    check-decl-main(TABLE, ERR-MESS-LIST)                 -> CHECK
    check-decl-ports(PORT-INFO, ERR-LOC, ERR-MESS-LIST)   -> CHECK
    check-decl-events(EV-INFO, ERR-LOC, ERR-MESS-LIST)    -> CHECK
    check-manner-call(MANNERCALL, TABLE, ERR-MESS-LIST)   -> CHECK
    check-activate(PROC-N, TABLE, ERR-MESS-LIST)          -> CHECK
    check-start-self(BODY, ERR-MESS-LIST)                 -> CHECK
    check-port-in(PORT, PROC-N, TABLE, ERR-MESS-LIST)     -> CHECK
    check-port-out(PORT, PROC-N, TABLE, ERR-MESS-LIST)    -> CHECK
    check-recursivity(BOOL, ERR-MESS-LIST)                -> CHECK

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% name(EnT) is the name of the element associated with table entry EnT.

    name(ENTRY)                                           -> PROC-N


 equations

// static analysis of an application

[SS1a] check(Ap) = check-table(build-table(Ap))

[SS1b] check-double-proc-names(Tb,)=check-double-proc-names(Tb1,ErML1),
       check-decl-main(Tb,error : no manifold called main) =
                                       check-decl-main(Tb2,ErML2),
       check-entry(Tb,Tb) = ErML3
       ============================================================
       check-table(Tb)= ErML1.ErML2.ErML3



////////////////////////////////////////////
// static analysis at the application level
////////////////////////////////////////////


// all elements must have different names

[SS2a] name(EnT) = name(EnT'),
       check-double-proc-names({EnTL1 EnT EnTL2 EnTL3},)=
       check-double-proc-names(Tb,ErML')
       ==================================================
       check-double-proc-names ({EnTL1 EnT EnTL2 EnT' EnTL3},ErML)  =
             check-double-proc-names(Tb,
                  error : process or manner name(EnT) is double declared .
                                    ErML'.ErML)


// there must exist a manifold called main

[SS2b] check-decl-main({EnTL1 <manifold, main, Inf> EnTL2},
                                     error :no manifold called main.ErML) =
       check-decl-main({EnTL1 <manifold, main, Inf> EnTL2},ErML)



///////////////////////////////////////
// static analysis at the element level
///////////////////////////////////////


// static analysis of a manifold

[SS2c] check-decl-ports(PoInf,error in manifold Pn :,) =
                                             check-decl-ports(PoInf',ErLo',ErML1),
       check-start-self(Bd,
             error in manifold Pn : no handler for start.self) =
                                               check-start-self(Bd1,ErML2),
       check-body(Pn,Bd,Tb) = ErML3
       ==================================================================
       check-entry({<manifold, Pn, (PoInf,EvInf,Bd)>},Tb) = ErML1.ErML2.ErML3


// static analysis of an atomic process

[SS2d] check-decl-ports(PoInf,error in manifold Pn :,) =
                                             check-decl-ports(PoInf',ErLo1,ErML1),
       check-decl-events(EvInf,error in atomic process Pn :,) =
                                             check-decl-events(EvInf',ErLo2,ErML2)
       ================================================================
       check-entry({<atomic, Pn, (PoInf,EvInf)>},Tb)= ErML1.ErML2


// static analysis of a manner

[SS2e] check-start-self(Bd,error in manner Id : no handler for start.self) =
       check-start-self(Bd1,ErML1),
       check-recursivity(Bool,error in manner Id: recursivity risks) =
       check-recursivity(Bool1,ErML2),
       check-manner(Id,(Bd,called-from{PnL} and Bool),Tb) =  ErML3
       ============================================================
       check-entry({<manner,Id,(Bd,called-from{PnL} and Bool)>}, Tb) =
                                                          ErML1.ErML2.ErML3


// putting results together
```

```
[SS2f] check-entry({EnT},Tb) = ErML1,
       check-entry({EnTP},Tb) = ErML2
       ========================================
       check-entry({EnT EnTP},Tb) = ErML1.ErML2


//////////////////////////////////////////////////////////////////////////
// static analysis of the port and event declarations of a given process
//////////////////////////////////////////////////////////////////////////

// the port output must not be declared as an input port

[SS3a] check-decl-ports(in {PoL1, PoL2},out PoS, ErLo,) =
       check-decl-ports(PoInf,ErLo',ErML')
       =====================================================================
       check-decl-ports(in {PoL1,  Pn.output, PoL2}, out PoS, ErLo,ErML) =
       check-decl-ports(PoInf,ErLo,ErLo
                          port output is declared as an input port . ErML'.ErML)


// the port input must not be declared as an output port

[SS3b] check-decl-ports(in PoS,out {PoL1, PoL2} , ErLo,) =
       check-decl-ports(PoInf,ErLo',ErML')
       ================================================================
       check-decl-ports(in PoS , out {PoL1, Pn.input, PoL2} , ErLo,ErML) =
       check-decl-ports(PoInf,ErLo,ErLo
                 port input is declared as an output port . ErML'.ErML)


// there must not be two input ports with the same name

[SS3c] check-decl-ports(in {PoL1,Pn.Pon, PoL2, PoL3},out PoS, ErLo,) =
       check-decl-ports(PoInf,ErLo',ErML')
       ============================================================
       check-decl-ports(in {PoL1, Pn.Pon, PoL2,Pn.Pon,PoL3},
                                               out PoS , ErLo,ErML) =
       check-decl-ports(PoInf,ErLo,ErLo
                            port Pon is double declared . ErML'.ErML)


// there must not be two output ports with the same name

[SS3d] check-decl-ports(in PoS,out {PoL1,Pn.Pon, PoL2, PoL3} , ErLo,) =
                                 check-decl-ports(PoInf,ErLo',ErML')
       ================================================================
       check-decl-ports(in PoS ,
                     out {PoL1, Pn.Pon, PoL2,Pn.Pon,PoL3} , ErLo,ErML) =
       check-decl-ports(PoInf,ErLo,ErLo
                     port Pon is double declared . ErML'.ErML)
```

```
// there must not be an input and an output port with the same name

[SS3e] check-decl-ports(in {PoL1,Pn.Pon,PoL2},out {PoL1',PoL2'} , ErLo,) =
       check-decl-ports(PoInf,ErLo',ErML')
       ==============================================================
       check-decl-ports( in {PoL1,Pn.Pon,PoL2},
                                    out {PoL1',Pn.Pon, PoL2'} , ErLo,ErML) =
       check-decl-ports(PoInf,ErLo,ErLo port Pon is double declared . ErML'.ErML)


// there must not be two events with the same name (for an atomic process)

[SS3f] check-decl-events(loc-ev{}, glob-ev{EoL1, En.Sc, EoL2, EoL3},ErLo,) =
       check-decl-events(EvInf,ErLo',ErML')
       ==================================================================
       check-decl-events(loc-ev{},
                          glob-ev{EoL1, En.Sc, EoL2, En.Sc, EoL3},ErLo,ErML)=
       check-decl-events(EvInf,ErLo,ErLo event En is double declared.ErML'.ErML)



/////////////////////////////////////////////
// static analysis of the body of a manifold
/////////////////////////////////////////////



// there must be a handler for start.self (for manners also)

[SS3g] match(Eo,start.self) = true
       =========================================================
       check-start-self({BlL1 EoL1, Eo, EoL2:Hd. BlL2}, ErLo
                                             no handler for start.self) =
       check-start-self({BlL1 EoL1, Eo, EoL2:Hd. BlL2},)


// putting results for the different blocks together

[SS3h] check-body(Pn,{Bl},Tb) = ErML1,
       check-body(Pn,{BlP},Tb)= ErML2
       ===================================
       check-body(Pn,{Bl BlP},Tb) = ErML1.ErML2


// a called manner must exist

[SS3i] check-manner-call(Id,Tb,error in manifold Pn : manner Id is unknown)=
       check-manner-call(Id',Tb',ErML)
       ==================================================================
       check-body(Pn,{Lb:Id.},Tb)=ErML


// putting results for the different actions in a group together

[SS3j] check-body(Pn,{Lb:().},Tb)=

[SS3k] check-body(Pn,{Lb:Ac.},Tb) = ErML1,
       check-body(Pn,{Lb:(AcL).},Tb) = ErML2
       ================================================
       check-body(Pn,{Lb:(Ac,AcL).},Tb) = ErML1.ErML2


// putting results for the different streams in a pipe together

[SS3l] check-body(Pn,{Lb:[St].},Tb) = check-body(Pn,{Lb:St.} ,Tb)

[SS3m] check-body(Pn,{Lb:St.},Tb) = ErML1,
       check-body(Pn,{Lb:[StP].},Tb) = ErML2
       ===============================================
       check-body(Pn,{Lb:[St,StP].},Tb)= ErML1.ErML2


// checking the ports of a stream

[SS3n] check-port-out(Pn1.Pon1,Pn,Tb,
                 error in manifold Pn : port Pn1.Pon1 is not an output port)=
       check-port-out(Pn1'.Pon1',Pn1'',Tb1',ErML1),
       check-port-in(Pn2.Pon2,Pn,Tb,
                 error in manifold Pn : port Pn2.Pon2 is not an input port)=
       check-port-in(Pn2'.Pon2',Pn2'',Tb2',ErML2)
       ====================================================================
       check-body(Pn,{Lb:Pn1.Pon1 -> Pn2.Pon2.},Tb)= ErML1.ErML2


// checking atomic actions

[SS3o] check-body(Pn,{Lb:do En.},Tb) =

[SS3p] check-body(Pn,{Lb:raise En.},Tb) =

[SS3q] check-body(Pn,{Lb:broadcast En.},Tb) =

[SS3r] check-body(Pn,{Lb:return.},Tb) =

[SS3s] check-body(Pn,{Lb:ignore.},Tb) =

[SS3t] check-body(Pn,{Lb:save.},Tb) =

[SS3u] check-activate(Pn',Tb,error in manifold Pn : process Pn' is unknown) =
       check-activate(Pn'',Tb',ErML)
       =====================================================================
       check-body(Pn,{Lb:activate Pn'.},Tb) = ErML

[SS3v] check-activate(Pn',Tb,error in manifold Pn : process Pn' is unknown) =
       check-activate(Pn'',Tb',ErML)
       ====================================================================
       check-body(Pn,{Lb:deactivate Pn'.},Tb) = ErML



//////////////////////////////
// static analysis of a manner
//////////////////////////////


// call to a manner must not be recursive

[SS4a] check-recursivity(false,ErLo recursivity risks) =
       check-recursivity(false,)


// static analysis of the body of a manner
// A first analysis is performed independently of the process calling this
// manner. This analysis is similar to that of a body of a manifold, except
// that the ports whose process is self are not checked.
// A second analysis performed for all the processes that may call a manner.
// This analysis is dependent of the calling process, and check whether ports
// whose process self are well employed in the context of the calling process.

[SS4b] check-manner(Id,(Bd,called-from{} and Bool),Tb) =
                                               check-mann-body1(Id,Bd,Tb)

[SS4c] check-mann-body2(Id,Pn,Bd,Tb)= ErML1,
       check-manner(Id,(Bd,called-from{PnL} and Bool),Tb) = ErML2
       ================================================================
       check-manner(Id,(Bd,called-from{Pn,PnL} and Bool),Tb) = ErML1.ErML2


// first type of body analysis: similar to that of a manifold body

[SS4d] check-mann-body1(Id,{Bl},Tb) = ErML1,
       check-mann-body1(Id,{BlP},Tb)= ErML2
       ==============================================
       check-mann-body1(Id,{Bl BlP},Tb) = ErML1.ErML2

[SS4e] check-manner-call(Id1,Tb,error in manner Id : manner Id1 is unknown)=
       check-manner-call(Id',Tb',ErML)
       ===================================================================
       check-mann-body1(Id,{Lb:Id1.},Tb)=ErML

[SS4f] check-mann-body1(Id,{Lb:().},Tb)=
[SS4g] check-mann-body1(Id,{Lb:Ac.},Tb) = ErML1,
       check-mann-body1(Id,{Lb:(AcL).},Tb) = ErML2
       ====================================================
       check-mann-body1(Id,{Lb:(Ac,AcL).},Tb) = ErML1.ErML2

[SS4h] check-mann-body1(Id,{Lb:[St].},Tb) = check-mann-body1(Id,{Lb:St.} ,Tb)

[SS4i] check-mann-body1(Id,{Lb:St.},Tb) = ErML1,
       check-mann-body1(Id,{Lb:[StP].},Tb) = ErML2
       ===================================================
       check-mann-body1(Id,{Lb:[St,StP].},Tb)= ErML1.ErML2


// note that  ports of self are not checked

[SS4j] Pn1 != self,
       Pn2 != self,
       check-port-out(Pn1.Pon1,Id,Tb,
                 error in manner Id : port Pn1.Pon1 is not an output port)=
       check-port-out(Pn1'.Pon1',Pn1'',Tb1',ErML1),
       check-port-in(Pn2.Pon2,Id,Tb,
                 error in manner Id : port Pn2.Pon2 is not an input port)=
       check-port-in(Pn2'.Pon2',Pn2'',Tb2',ErML2)
       =================================================================
       check-mann-body1(Id,{Lb:Pn1.Pon1 -> Pn2.Pon2.},Tb)= ErML1.ErML2

[SS4k] Pn2 != self,
       check-port-in(Pn2.Pon2,Id,Tb,
                 error in manner Id : port Pn2.Pon2 is not an input port)=
       check-port-in(Pn2'.Pon2',Pn2'',Tb2',ErML2)
       =================================================================
       check-mann-body1(Id,{Lb:self.Pon1 -> Pn2.Pon2.},Tb)= ErML2

[SS4l] Pn1 != self,
       check-port-out(Pn1.Pon1,Id,Tb,
                 error in manner Id : port Pn1.Pon1 is not an output port)=
       check-port-out(Pn1'.Pon1',Pn1'',Tb1',ErML1)
       ==================================================================
       check-mann-body1(Id,{Lb:Pn1.Pon1 -> self.Pon2.},Tb)= ErML1

[SS4m] check-mann-body1(Id,{Lb:self.Pon1 -> self.Pon2.},Tb) =

[SS4n] check-mann-body1(Id,{Lb:do En.},Tb) =

[SS4o] check-mann-body1(Id,{Lb:raise En.},Tb) =

[SS4p] check-mann-body1(Id,{Lb:broadcast En.},Tb) =

[SS4q] check-mann-body1(Id,{Lb:return.},Tb) =

[SS4r] check-mann-body1(Id,{Lb:ignore.},Tb) =
[SS4s] check-mann-body1(Id,{Lb:save.},Tb) =

[SS4t] check-activate(Pn,Tb,error in manner Id : process Pn is unknown) =
       check-activate(Pn',Tb',ErML)
       =================================================================
       check-mann-body1(Id,{Lb:activate Pn.},Tb) = ErML

[SS4u] check-activate(Pn,Tb,error in manner Id : process Pn is unknown) =
       check-activate(Pn',Tb',ErML)
       =================================================================
       check-mann-body1(Id,{Lb:deactivate Pn.},Tb) = ErML


// second type of analysis: only ports of self checked in the context of a
// calling process

[SS4v] check-mann-body2(Id,Pn,{Bl},Tb) = ErML1,
       check-mann-body2(Id,Pn,{BlP},Tb)= ErML2
       ================================================
       check-mann-body2(Id,Pn,{Bl BlP},Tb) = ErML1.ErML2

[SS4w] check-mann-body2(Id,Pn,{Lb:Mc.},Tb)=

[SS4x] check-mann-body2(Id,Pn,{Lb:().},Tb)=

[SS4y] check-mann-body2(Id,Pn,{Lb:Ac.},Tb)=ErML1,
       check-mann-body2(Id,Pn,{Lb:(AcL).},Tb)=ErML2
       =====================================================
       check-mann-body2(Id,Pn,{Lb:(Ac,AcL).},Tb)= ErML1.ErML2

[SS4z] check-mann-body2(Id,Pn,{Lb:AtAc.},Tb)=

[SS40] check-mann-body2(Id,Pn,{Lb:[St].},Tb) =
                                  check-mann-body2(Id,Pn,{Lb:St.},Tb)

[SS41] check-mann-body2(Id,Pn,{Lb:St.},Tb)=ErML1,
       check-mann-body2(Id,Pn,{Lb:[StP].},Tb)=ErML2
       =====================================================
       check-mann-body2(Id,Pn,{Lb:[St,StP].},Tb)= ErML1.ErML2

[SS42] check-port-out(self.Pon1,Pn,Tb,error in manner Id when called from Pn :
                                     port self.Pon1 is not an output port)=
       check-port-out(Pn1'.Pon1',Pn1'',Tb1',ErML1),
       check-port-in(self.Pon2,Pn,Tb,error in manner Id when called from Pn :
                                   port self.Pon2 is not an input port)=
       check-port-in(Pn2'.Pon2',Pn2'',Tb2',ErML2)
       ======================================================================
       check-mann-body2(Id,Pn,{Lb:self.Pon1 -> self.Pon2.},Tb)= ErML1.ErML2

[SS43] Pn1 != self,
       check-port-in(self.Pon2,Pn,Tb,error in manner Id when called from Pn :
```

```
                                   port self.Pon2 is not an input port)=
       check-port-in(Pn2'.Pon2',Pn2'',Tb2',ErML2)
       ===================================================================
       check-mann-body2(Id,Pn,{Lb:Pn1.Pon1 -> self.Pon2.},Tb)= ErML2

[SS44] Pn2 != self,
       check-port-out(self.Pon1,Pn,Tb,error in manner Id when called from Pn :
                                    port self.Pon1 is not an output port)=
       check-port-out(Pn1'.Pon1',Pn1'',Tb1',ErML1)
       ====================================================================
       check-mann-body2(Id,Pn,{Lb:self.Pon1 -> Pn2.Pon2.},Tb)= ErML1

[SS45] Pn1 != self,
       Pn2 != self
       ====================================================
       check-mann-body2(Id,Pn,{Lb:Pn1.Pon1->Pn2.Pon2.},Tb) =


////////////////////
// Constructs used
////////////////////


// checking the existence of a called manner

[SS5a] check-manner-call(Id, {EnTL1 <manner, Id, Inf> EnTL2},
                                              ErLo manner Id is unknown)=
       check-manner-call(Id,{EnTL1 <manner, Id, Inf> EnTL2},)


// checking the existence of a (de)activated process

[SS5b] check-activate(Pn, {EnTL1 <Tp, Pn, Inf> EnTL2},
                                 ErLo process Pn is unknown)=
       check-activate(Pn,{EnTL1 <Tp, Pn, Inf> EnTL2},)


// checking the good use of ports

// input ports for a process different than self

[SS5c] Pn != self
       ==================================================================
       check-port-in(Pn.Pon,Pn',{EnTL1
            <manifold,Pn,(in {PoL1, Pn.Pon, PoL2}, out PoS ,EvInf,Bd)>
            EnTL2},
         ErLo port Pn.Pon is not an input port) =
       check-port-in(Pn.Pon,Pn',
         {EnTL1 <manifold,Pn,(in {PoL1, Pn.Pon, PoL2}, out PoS ,EvInf,Bd)>
                                                                  EnTL2},)


[SS5d] Pn != self
```

```
       ===================================================================
       check-port-in(Pn.Pon,Pn',
         {EnTL1 <atomic,Pn,(in {PoL1, Pn.Pon, PoL2}, out PoS ,EvInf)> EnTL2},
         ErLo port Pn.Pon is not an input port) =
       check-port-in(Pn.Pon,Pn',
         {EnTL1 <atomic,Pn,(in {PoL1, Pn.Pon, PoL2}, out PoS ,EvInf)> EnTL2},)


// input ports for self.
// Note that self can consider also its output ports as input ports

[SS5e] check-port-in(self.Pon, Pn',{EnTL1
           <manifold ,Pn',(in {PoL1, Pn'.Pon, PoL2}, out PoS,EvInf,Bd)>
           EnTL2},
        ErLo port self.Pon is not an input port) =
       check-port-in(self.Pon, Pn',{EnTL1
           <manifold, Pn',(in {PoL1, Pn'.Pon, PoL2}, out PoS,EvInf,Bd)>
           EnTL2}, )

[SS5f] check-port-in(self.Pon, Pn',{EnTL1
           <manifold, Pn',(in PoS, out {PoL1, Pn'.Pon, PoL2},EvInf,Bd)>
           EnTL2},
        ErLo port self.Pon is not an input port) =
       check-port-in(self.Pon, Pn',{EnTL1
           <manifold, Pn',(in PoS, out {PoL1, Pn'.Pon, PoL2},EvInf,Bd)>
           EnTL2}, )


// output ports for a process different than self

[SS5g] Pn != self
       ===================================================================
       check-port-out(Pn.Pon,Pn',
         {EnTL1 <manifold,Pn,(in PoS, out {PoL1, Pn.Pon, PoL2},EvInf,Bd)> EnTL2},
       ErLo port Pn.Pon is not an output port) =
       check-port-out(Pn.Pon,Pn',
         {EnTL1 <manifold,Pn,(in PoS, out {PoL1, Pn.Pon, PoL2},EvInf,Bd)> EnTL2},)

[SS5h] Pn != self
       ===================================================================
       check-port-out(Pn.Pon,Pn',
         {EnTL1 <atomic,Pn,(in PoS, out {PoL1, Pn.Pon, PoL2},EvInf)> EnTL2},
         ErLo port Pn.Pon is not an output port) =
       check-port-out(Pn.Pon,Pn',
         {EnTL1 <atomic,Pn,(in PoS, out {PoL1, Pn.Pon, PoL2},EvInf)> EnTL2},)


// output ports for self.
// Note that self can consider also its input ports as output ports

[SS5i] check-port-out(self.Pon, Pn',
         {EnTL1 <manifold, Pn',
                 (in {PoL1, Pn'.Pon, PoL2}, out PoS,EvInf,Bd)>
         EnTL2}, ErLo port self.Pon is not an output port) =
      check-port-out(self.Pon, Pn',
        {EnTL1 <manifold, Pn',
                 (in {PoL1, Pn'.Pon, PoL2}, out PoS,EvInf,Bd)>
         EnTL2}, )

[SS5j] check-port-out(self.Pon, Pn',
        {EnTL1 <manifold, Pn',
                 (in PoS, out {PoL1, Pn'.Pon, PoL2},EvInf,Bd)>
         EnTL2}, ErLo port self.Pon is not an output port) =
      check-port-out(self.Pon, Pn',
        {EnTL1 <manifold, Pn',
                 (in PoS, out {PoL1, Pn'.Pon, PoL2},EvInf,Bd)>
         EnTL2}, )


// name of an the element corresponding to an entry

[SS6a] name(<Tp,Pn,Inf>) = Pn

[SS6b] name(<manner,Id,Inf>)=Id
```

A.3 Operational semantics

A.3.1 Action level

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                          Module Actions
%%
%% This module defines the transition system at the action level
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Util-Sets

exports

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% An action is interpreted in its environment, and the
    %% terminal states of the transition system are such
    %% environments

    inter-a "[" ACT "]" in "<"EV-OCC","PROC-N","A-ENV">" -> A-ENV
```

```
equations

//////////////////////////////////////////////////////////////////////////
//
//                              Atomic actions.
//
//////////////////////////////////////////////////////////////////////////

[AC1a] inter-a [do En ] in <Eo,Pn,Ae> =
                         Ae union-ae ae({En.self, lambda.self},{},(),{})

[AC1b] inter-a [raise En] in <Eo,Pn,Ae> =
                         Ae union-ae ae({lambda.self},{En.Pn},(),{})

[AC1c] inter-a [broadcast En] in <Eo,Pn,Ae> =
                  Ae union-ae ae({En.system, lambda.self},{En.system},(),{})

[AC1d] inter-a [ignore] in <Eo,Pn,Ae> = Ae

[AC1e] inter-a [save] in <Eo,Pn,Ae> = Ae union-ae ae({Eo},{},(),{})

[AC1f] inter-a [activate Pn'] in <Eo,Pn,Ae> =
                         Ae union-ae ae({lambda.self},{},(),{act(Pn')})

[AC1g] inter-a [deactivate Pn'] in <Eo,Pn,Ae> =
                         Ae union-ae ae({lambda.self},{},(),{deact(Pn')})


////////////////////////////////////////////////////////////////////////////////
//
//                              Streams and pipelines
//
////////////////////////////////////////////////////////////////////////////////


// Stream breaking

[AC2a] (Eo isin-e {death.Pn1, break.Pn1} | Eo isin-e {death.Pn2, break.Pn2})
                                                                    = true
       ================================================================
       inter-a [Pn1.Pon1 -> Pn2.Pon2] in <Eo,Pn,Ae> =
                                   Ae union-ae ae({lambda.self},{},(),{})


// Stream set-up

[AC2b] (Eo isin-e {death.Pn1, break.Pn1}
        | Eo isin-e {death.Pn2, break.Pn2}) = false
       ==========================================================================
       inter-a [Pn1.Pon1 -> Pn2.Pon2] in <Eo,Pn,Ae> =
               Ae union-ae ae({},{},(Pn1.Pon1 -> Pn2.Pon2),{})
```

```
// Breaking of a pipeline composed of one stream

[AC2c] inter-a [St] in <Eo,Pn,ae({},{},(),{})> =  ae(EoS1,EoS2,(),AvS)
       ==================================================================
       inter-a [[St]] in <Eo,Pn,Ae> =  Ae union-ae ae(EoS1,EoS2,(),AvS)


// Set-up of a pipeline composed of one stream

[AC2d] inter-a [St] in <Eo,Pn,ae({},{},(),{})> =  ae(EoS1,EoS2,(St'),AvS)
       =====================================================================
       inter-a [[St]] in <Eo,Pn,Ae> = Ae union-ae ae(EoS1,EoS2,([St']),AvS)


// Breaking of a pipeline composed of more than one stream
// two cases -> the first stream breaks, or one of the other breaks

[AC2e] inter-a [St] in  <Eo,Pn,ae({},{},(),{})> =  ae(EoS1,EoS1',(),AvS1 ),
       inter-a [[StP]] in <Eo,Pn,ae({},{},(),{})> = ae(EoS2,EoS2',Gr2,AvS2 )
       ========================================================================
       inter-a [[St, StP]] in <Eo,Pn,Ae> = Ae union-ae ae(EoS1, EoS1',(),AvS1)
                                              union-ae ae(EoS2, EoS2',(),AvS2)

[AC2f] inter-a [St] in <Eo, Pn, ae({},{},(),{})> =  ae(EoS1,EoS1',Gr1,AvS1),
       inter-a [[StP]] in <Eo, Pn, ae({},{},(),{})> =  ae(EoS2,EoS2',(),AvS2 ),
       Gr1 != ()
       =========================================================================
       inter-a [[St, StP]] in <Eo,Pn, Ae> = Ae union-ae ae(EoS1,EoS1',(),AvS1)
                                               union-ae ae(EoS2, EoS2', (), AvS2)


// Set-up  of a pipeline composed of more than one stream

[AC2g] inter-a [St] in <Eo,Pn,ae({},{},(),{})> = ae(EoS1,EoS1',(St1),AvS1),
       inter-a [[StP]] in <Eo,Pn,ae({},{},(),{})> = ae(EoS2,EoS2',([StP']),AvS2)
       =========================================================================
       inter-a [[St, StP]] in <Eo,Pn, Ae> =

                                              Ae union-ae ae(EoS1 union-e EoS2,
                                                         EoS1' union-e EoS2',
                                                         ([St1, StP']),
                                                         AvS1 union-a AvS2)



///////////////////////////////////////////////////////////////////////////////
//
//                                 Groups
//
///////////////////////////////////////////////////////////////////////////////
```

```
// Group of one action
// two cases -> this action terminates, or it does not terminate

[AC3a] inter-a [Ac] in <Eo,Pn,Ae> = ae(EoS1,EoS2,(),AvS)
       ==================================================================
       inter-a [(Ac)] in <Eo,Pn,Ae> =
                            ae(EoS1 union-e {lambda.self}, EoS2, (), AvS)

[AC3b] inter-a [Ac] in <Eo,Pn,Ae> = ae(EoS1,EoS2,Gr,AvS),
       Gr != ()
       ==================================================================
       inter-a [(Ac)] in <Eo,Pn,Ae> =
                                      ae(EoS1 minus-e {lambda.self}, EoS2, Gr, AvS)


// Group of several actions -> union of the effects of the different actions

[AC3c] inter-a [Ac] in <Eo,Pn,Ae> = Ae' , (AcL) != ()
       ==============================================================
       inter-a [(Ac, AcL)] in <Eo,Pn,Ae> = inter-a [(AcL)] in <Eo,Pn,Ae'>
```

A.3.2 Process level, handler level and search

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                              Module  Processes
%%
%% This module defines the semantics of Manifold at the process level
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
imports Util Actions

exports
  sorts PROD-P

  context-free syntax


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
    %%
    %% The transition system at the process level evaluates a process state in
    %% an environment consisting of the sets of internal representations for
    %% the manners and the atomic processes. The result is another process
    %% state and a pair consisting of the events raised outside the process
    %% as it makes this transition and the set of processes to be (de)activated.

    P-STATE with "<"EV-OCC-SET","ACTIV-SET">"            -> PROD-P
    inter-p P-STATE in "<"MANNER-REP-SET","ATOMIC-REP-SET">" -> PROD-P

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% When deciding to handle an event, the process makes a distinction between
    %% - prioritary events (whose set is priorities0)
    %% - evolutive events  (whose set is evolutive0)
    %% - preemptive events. Those vary according to the state of the manifold,
    %%   and must
    %%   - have their source referenced in the current handler Hd ( the set
    %%     of such events is current-handler-refs(Hd)) or have a source which
    %%     is a port of the manifold (the set of such events is given by
    %%     all-self-ports).
    %%   - must be referenced in a label of the blocks Bls (the set of such
    %%     events is given by labels-refs(Bls))
    %% The reason why the corresponding syntax is exported, is that it will be
    %% reused at the application level for the calculation of the observable
    %% events.

    priorities0                                              -> EV-OCC-SET
    evolutives0                                              -> EV-OCC-SET

    current-handler-refs(HANDLER)                            -> EV-OCC-SET
    labels-refs(BLOCKS)                                      -> EV-OCC-SET
    all-self-ports(PROC-N, EV-OCC-SET)                       -> EV-OCC-SET


hiddens
  sorts H-STATE PROD-H

        BLOCK-SET CS-STATE S-STATE S-TERM-STATE

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Handler level.
    %%
    %% The state at the handler level is defined as a triple:
    %% - state of the program of the executing manifold,
    %% - state of the current block of the executing manifold,
    %% - set of events observed by the executing manifold, that have
    %%   not yet been handled.
    %%
    %% The transition system at the handler level evaluates a state
    %% in an environment consisting of:
    %% - the handled event,
    %% - the previous state of the program of the executing manifold,
    %% - the previous current block of the executing manifold,
    %% - the set of internal representations of the manners of the application,
    %% The result of this evaluation is a new state and a pair,
%% consisting of the set of externally raised events during the transition,
%% and the set of processes to be (de)activated.
%%
%% Some additional constructs are defined:
%% - pipe2block(Gr,Lb) is the block state for label Lb and group Gr.
%% - is-react-act(act) is a boolean true if and only if act is a
%%   reactionary action.

hs(PROC-ST, BLOCK-ST, EV-OCC-SET)                   -> H-STATE

H-STATE with "<"EV-OCC-SET","ACTIV-SET">"           -> PROD-H
inter-h  H-STATE in "<"EV-OCC","PROC-ST","BLOCK-ST","MANNER-REP-SET">"
                                                          -> PROD-H

pipe2block(GROUP,LABEL)                             -> BLOCK-ST
is-react-act(ACT)                                   -> BOOL


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Search of the handling block.
%%
%% A non-terminal state of the search transition system is a non
%% empty list of blocks (i.e. 0 or more manners blocks stacked on the
%%  top of the blocks of the manifold.
%%
%% Terminal states of the system are
%% - the empty list of blocks if the search fails
%% - the new arrangement of the list of blocks together with the new
%%   current block if the search succeeds.
%%
%% the transition system evaluates a non-terminal state in the
%% environment consisting of the event occurrence to be handled.


se"{"BLOCKS"}"                                      -> S-STATE
"{""}"                                              -> S-TERM-STATE
"(""{"BLOCKS"}"","BLOCK")"                          -> S-TERM-STATE
S-STATE in EV-OCC                                   -> S-TERM-STATE


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Circular search within each block environment.
%%
%% A terminal state of the circular search transition system is
%% a possibly empty list of blocks.
%%
%% A non-terminal state of the system is a pair of possibly
%% empty lists of blocks, the first one containing the blocks
%% to be searched, and the second containing those already searched.
%%
```

```
    %% The transition system evaluates a non-terminal state in the
    %% environment consisting of the event occurrence to be handled.

    "{" BLOCK* "}"                                          -> BLOCK-SET

    cs(BLOCK-SET, BLOCK-SET )                               -> CS-STATE
    CS-STATE in EV-OCC                                      -> BLOCK-SET


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Additional constructs are defined for the process level.
    %%
    %% at-current-state(Eo) is the activation state of an atomic process
    %% that corresponds to the event it raises. A death event corresponds
    %% to a deactivated state, while the other events correspond to an
    %% active state.
    %%
    %% prioritary(EoS1,EoS2), where EoS1 is an arbitrary set of event
    %% occurrences and EoS2 is an ordered set of prioritary event occurrences,
    %% is a pair (Bool,Eo), where Bool is false if there is no prioritary
    %% event of EoS2 in EoS1, and true otherwise. If Bool is true, then
    %% Eo is the most prioritary event in EoS1.
    %%
    %% preemptive(Ps) is the set of preemptive events of the manifold whose
    %% state is P-STATE.


    at-current-state(EV-OCC)                                -> BLOCK-ST-AT
    prioritary(EV-OCC-SET, EV-OCC-SET)                      -> PROD-SELECT-E
    preemptives(P-STATE)                                    -> EV-OCC-SET



  equations

///////////////////////////////////////////////////////////////////////////////
//
//                               Handler Level
//
///////////////////////////////////////////////////////////////////////////////


// block state for a group and a label

[HD1a] pipe2block((),Lb) = end

[HD1b] Gr != ()
       ==========================
       pipe2block(Gr,Lb) = Lb:Gr.
```

```
// is an action a reactionary action

[HD1c] is-react-act(ignore) = true
[HD1d] is-react-act(save) = true
[HD1e] Ac != ignore,
       Ac != save
       ========================
       is-react-act(Ac) = false


// event handling (normal case where actions do not involve a return to a
// previous state)

[HD2a] is-react-act(Ac) = false,
       Ac != return,
       inter-a [Ac] in <Eo,name(PSt),ae({},{},(),{})> = ae(EoS1,EoS2,Gr,AvS)
       ===================================================================
       inter-h hs(PSt,Lb:Ac.,EoS) in <Eo,PSt',BlSt',MnS> =
       hs(PSt,pipe2block(Gr,Lb),EoS union-e EoS1) with <EoS2,AvS>


// reactionary actions (which involve a return to the previous state)

[HD2b] is-react-act(Ac) = true,
       inter-a [Ac] in <Eo,name(PSt),ae({},{},(),{})> = ae(EoS1,EoS2,Gr,AvS)
       ===================================================================
       inter-h hs(PSt,Lb:Ac.,EoS) in <Eo,PSt',BlSt',MnS> =
       hs(PSt',BlSt',EoS union-e EoS1) with <EoS2,AvS>


// manner calls

[HD2c] inter-p ps(manifold name(PSt)
                         {manner Mc man-blocks(Mc,MnS) proc-blocks(PSt)},
       inactive, {start.self}) in < MnS, {}> =
       ps(PSt1,BlSt1,EoS1) with <EoS2,AvS>
       ====================================================
       inter-h hs(PSt,Lb:Mc.,EoS ) in <Eo,PSt',BlSt',MnS> =
       hs(PSt1,BlSt1,EoS union-e EoS1) with <EoS2,AvS>


// manner return

[HD2d] inter-h hs(manifold Pn{Cm Bls},Lb:return.,EoS) in <Eo,PSt',BlSt',MnS> =
       hs(manifold Pn{Bls},end,EoS union-e {returned.self}) with <{},{}>




/////////////////////////////////////////////////////////////////////
//
//                  Search of the handling block for an event
```

```
//
////////////////////////////////////////////////////////////////////////////


// circular search

[SE1a] labels(Bl) gen-inters {Eo} = {}
       ======================================================
       cs({Bl BlL},{BlL'}) in Eo = cs({BlL},{BlL' Bl}) in Eo


// successful circular search

[SE1b] labels(Bl) gen-inters {Eo} != {}
       ==========================================
       cs({Bl BlL},{BlL'}) in Eo = {Bl BlL BlL'}


// unsuccessful circular search

[SE1c] cs({},{BlL})in Eo = {}


// successful search in manner

[SE2a] se{BlP} in Eo = ({BlP'},Bl)
       =========================================================
       se{manner Mc{BlP} Bls} in Eo= ({manner Mc{BlP'} Bls},Bl)


// unsuccessful search in manner

[SE2b] se{BlP} in Eo = {}
       =====================================================
       se{manner Mc{BlP} Bls} in Eo = se{Bls} in Eo


// successful search

[SE2c] cs({Bl BlL},{}) in Eo = {Bl' BlL'}
       =========================================
       se{Bl BlL} in Eo = ({Bl' BlL'}, Bl')


// unsuccessful search

[SE2d] cs({Bl BlL},{}) in Eo = {}
       ===================================
       se{Bl BlL} in Eo = {}
```

```
//////////////////////////////////////////////////////////////////////////
//
//                              Process Level
//
//////////////////////////////////////////////////////////////////////////

// In the following, the existential quantifier on event occurrences is
// replaced by a selection by the user select-ev-occ (in the cases where there are
// several event occurrences verifying a property)

////////////////////
// Atomic processes
////////////////////


// activation state corresponding to an event raised

[PR1a] at-current-state(death.Sc) = deactivated

[PR1b] En != death
       ==============================
       at-current-state(En.Sc) = active


// starting an atomic process

[PR1c] inter-p ps(PStAt,inactive,{start.self}) in <MnS,AtS> =
                                 ps(PStAt,active,{}) with <{},{}>


// rasing an event by an atomic process : if the event raised is not death,
// then the process remains active, otherwise, it is deactivated

[PR1d] abort.* gen-isin EoS = {},
       select-ev-occ(output(PStAt,AtS) union-e {death.name(PStAt)}) = (true,Eo)
       ====================================================================
       inter-p ps(PStAt,active,EoS) in <MnS,AtS> =
                         ps(PStAt,at-current-state(Eo),EoS) with <{Eo},{}>


// aborting an atomic process : the atomic process is deactivated, no matter
// if it was inactive or already active

[PR1d] abort.* gen-isin EoS != {}
       ============================================================
       inter-p ps(PStAt,BlStAt,EoS) in <MnS,AtS> =
                                 ps(PStAt,deactivated,EoS) with <{},{}>



//////////////////////////////////////////////////
```

```
// Manifold processes: handling prioritary events
//////////////////////////////////////////////////


// prioritary events

[PR2a] priorities0 = {abort.*, break.*, returned.*}


// determination of the most prioritary event to handle

[PR2b] prioritary(EoS,{}) = (false,nothing.nothing)

[PR2c] Eo gen-isin EoS = {Eo', EoL'}
       ====================================
       prioritary(EoS,{Eo, EoL}) = (true, Eo')
[PR2d] Eo gen-isin  EoS = {}
       ================================================
       prioritary(EoS,{Eo, EoL}) = prioritary(EoS,{EoL})


// handling abort

[PR2e] abort.* gen-isin EoS != {}
       ==============================================================
       inter-p ps(PSt,BlSt,EoS) in <MnS, AtS> =
                                 ps(PSt,deactivated,EoS) with <{},{}>


// handling break

[PR2f] select-ev-occ( break.* gen-isin EoS ) = (true,Eo),
       abort.* gen-isin EoS = {},
       inter-a [Ac] in <Eo,name(PSt),ae({},{},(),{})> = ae(EoS1,EoS1',Gr,AvS),
       EoS minus-e {Eo} = EoS2
       =====================================================================
       inter-p ps(PSt,Lb:Ac.,EoS) in <MnS, AtS> =
       ps(PSt,pipe2block(Gr,Lb),EoS2 union-e EoS1) with <EoS1',{}>


// handling the highest priority event

[PR2g] prioritary(EoS,priorities0 minus-e {abort.*, break.*}) = (true,Eo),
       se{proc-blocks(PSt)} in Eo = ({Bls'},Bl'),
       inter-h hs(manifold name(PSt){Bls'},Bl',EoS minus-e {Eo})
                                        in <Eo,PSt,BlSt,MnS> =
                                hs(PSt'',BlSt'',EoS'') with <EoS1,AvS1>
       ===============================================================
       inter-p ps(PSt,BlSt,EoS) in <MnS, AtS> =
                                ps(PSt'',BlSt'',EoS'') with <EoS1,AvS1>
```

```
//////////////////////////////////////////////////
// Manifold processes: handling preemptive events
//////////////////////////////////////////////////


// preemptive events definition


// events of which the source process is referenced in the current handler

[PR3a] current-handler-refs(Mc) = {}

[PR3b] current-handler-refs(AtAc) = {}

[PR3c] current-handler-refs(Pn1.Pon1 -> Pn2.Pon2) = {*.Pn1, *.Pn2 }

[PR3d] current-handler-refs([St]) =  current-handler-refs(St)

[PR3e] current-handler-refs([St, StP]) =
                current-handler-refs(St) union-e current-handler-refs([StP])

[PR3f] current-handler-refs(()) = {}

[PR3g] current-handler-refs((Ac, AcL)) =
                current-handler-refs(Ac) union-e current-handler-refs((AcL))


// events for which there is a label

[PR3h] labels-refs(manner Mc {BlP} Bls) =
                             labels-refs(BlP) union-e labels-refs(Bls)

[PR3i] labels-refs(Bl BlP) = labels(Bl) union-e labels-refs(BlP)

[PR3j] labels-refs(Bl) = labels(Bl)


// all ports of the considered manifold

[PR3k] all-self-ports(Pn,{}) = {}

[PR3l] all-self-ports(Pn,{En.Pn.Pon, EoL}) =
                          all-self-ports(Pn,{EoL}) union-e {En.self.Pon}

[PE3m] Pn' != Pn
       ============================================================
       all-self-ports(Pn,{En.Pn'.Pon, EoL}) = all-self-ports(Pn,{EoL})

[PR3n] all-self-ports(Pn,{En.Pn', EoL}) = all-self-ports(Pn,{EoL})
```

```
// preemptive events of a manifold, according to the state of its current block

[PR3o] current-handler-refs(Hd) union-e {*.self} = EoS1,
       EoS1 gen-inters EoS = EoS2,
       all-self-ports(Pn,EoS) = EoS3,
       EoS2 union-e EoS3 = EoS4,
       labels-refs(Bls) gen-inters EoS4 = EoS5,
       EoS5 gen-minus labels-refs(Lb:Hd.) = EoS6
       ===============================================
       preemptives(ps(manifold Pn {Bls},Lb:Hd.,EoS)) = EoS6

[PR3p] { *.self} gen-inters EoS  = EoS1,
       all-self-ports(Pn,EoS) = EoS2,
       EoS1 union-e EoS2 = EoS3,
       labels-refs(Bls) gen-inters EoS3 = EoS4
       ===============================================
       preemptives(ps(manifold Pn {Bls}, end, EoS)) =  EoS4

[PR3q] preemptives(ps(PSt,inactive,EoS)) = { start.self }  gen-inters EoS

[PR3r] preemptives(ps(PSt,deactivated,EoS)) = {}



// handling a preemptive event

[PR3s] priorities0 gen-inters EoS = {},
       select-ev-occ(preemptives(ps(PSt,BlSt,EoS))) = (true,Eo),
       se{proc-blocks(PSt)} in Eo = ({Bls'},Bl'),
       inter-h hs(manifold name(PSt){Bls'}, Bl', EoS minus-e {Eo})
                                            in <Eo,PSt,BlSt,MnS>
                =                 hs(PSt'',BlSt'',EoS'') with <EoS1,AvS1>
       ================================================================
       inter-p ps(PSt,BlSt,EoS) in <MnS, AtS> =
                                  ps(PSt'',BlSt'',EoS'') with <EoS1,AvS1>



////////////////////////////////////////////////////
// Manifold processes: handling an evolutive event
////////////////////////////////////////////////////


// evolutive events

[PR4a] evolutives0 = {death.*, break.*}


// evolution of the network of pipelines

[PR4b] priorities0 gen-inters EoS= {},
       preemptives(ps(PSt,Lb:Ac.,EoS)) = {},
       select-ev-occ( evolutives0 gen-inters EoS ) = (true,Eo),
       inter-a [Ac] in <Eo,name(PSt),ae({},{},(),{})> = ae(EoS1,EoS2,Gr,AvS),
       EoS minus-e {Eo} = EoS'
       ==================================================================
       inter-p ps(PSt,Lb:Ac.,EoS) in <MnS, AtS> =
       ps(PSt,pipe2block(Gr,Lb), EoS' union-e EoS1) with <EoS2,{}>



//////////////////////////////////////////////
// Manifold processes and manners: termination
//////////////////////////////////////////////


// termination of a manifold

[PR5a] PSt = manifold Pn {BlP},
       priorities0 gen-inters EoS = {},
       preemptives(ps(PSt,end,EoS)) = {}
       ==================================================
       inter-p ps(PSt,end,EoS) in <MnS, AtS> =
       ps(PSt,deactivated,EoS) with <{death.name(PSt)},{}>


// termination of a manner

[PR5b] PSt = manifold Pn {manner Mc{BlP} Bls},
       priorities0 gen-inters EoS = {},
       preemptives(ps(PSt,end,EoS)) = {}
       ================================================================
       inter-p ps(PSt,end,EoS) in <MnS, AtS> =
       ps(manifold Pn{Bls},end,EoS union-e {returned.self}) with <{},{}>
```

A.3.3 Application level

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                          Module  Application
%%
%%  This module defines the semantics of Manifold at the application
%%  level.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Static-Semantics Processes

exports
  sorts EVALUATION

  context-free syntax
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% The evaluation of a Manifold application is either a list
    %% of error messages detected statically, or a terminal state
    %% of the application (If any).
    %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    ERR-MESS-LIST                                         -> EVALUATION
    AP-STATE                                              -> EVALUATION

    eval(APPLICATION)                                     -> EVALUATION

hiddens

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% if-errors-then-stop-else-inter(Tb,ErML) is the result of the
    %% evaluation of the manifold application with intermediate table
    %% Tb and list of error messages ErML. If ErML is not empty then
    %% this result is ErML. Otherwise, the result is the terminal
    %% state resulting from the transitions of the initial state
    %% of the application that is obtained with Tb.

    if-errors-then-stop-else-inter(TABLE, ERR-MESS-LIST) -> EVALUATION


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% The transition system at application level maps an application
    %% state to another.

    inter-ap AP-STATE                                     -> AP-STATE

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% if-trans-then-diffact-else-same(Aps,Ps) is the application
    %% state resulting from the transition of the process with
    %% state Ps. If there was a transition for this process then
    %% the new application state is obtained by diffusion of the
    %% result of this transition. Otherwise the application state
    %% is unchanged.

    if-trans-then-diffact-else-same(AP-STATE, P-STATE)    -> AP-STATE

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Additional constructs denoting respectively:
    %% - the set of event occurrences for which there is a handler in
```

```
   %%   a non-empty list of blocks (needed for the definition of
   %%                                 the observable events)
   %% - the set of event occurrences that are observable from a
   %%   process state
   %% - the application state resulting from the diffusion of
   %%   a set of externally raised events and a set of process
   %%  (de)activations.

   handler-refs(BLOCKS)                                  -> EV-OCC-SET
   observables(P-STATE)                                     -> EV-OCC-SET
   diffact "<"EV-OCC-SET","ACTIV-SET">" in AP-STATE      -> AP-STATE




equations

///////////////////////////////////////////////////////////////////////////////
//
//                       Evaluation of an Application
//
///////////////////////////////////////////////////////////////////////////////


// evaluation of an application

[AP1a] build-table(Ap) = Tb,
       check-table(Tb) = ErML
       ================================================
       eval(Ap) = if-errors-then-stop-else-inter(Tb,ErML)


// no error detected statically -> transitions from the initial state
// of the application

[AP1b] if-errors-then-stop-else-inter(Tb,) = inter-ap build-appli(Tb)


// errors detected statically

[AP1c] if-errors-then-stop-else-inter(Tb,ErM.ErML)= ErM.ErML



///////////////////////////////////////////////////////////////////////////////
//
//                  Transition System at the Application level
//
///////////////////////////////////////////////////////////////////////////////
```

```
// non-terminated application: transits by having one of its process transit

[AP1d] select-proc-n(all-proc-non-deactivated Aps) = (true,Pn),
       process-state(Pn,Aps) = Ps,
       if-trans-then-diffact-else-same(Aps,Ps)=aps(PsS,MnS,AtS),
       Dummy = show(PsS)
       ==========================================================
       inter-ap Aps = inter-ap aps(PsS,MnS,AtS)


// terminated application: all processes are deactivated and cannot transit

[AP1e] select-proc-n(all-proc-non-deactivated Aps) = (false,Pn),
       Aps = aps(PsS,MnS,AtS),
       Dummy = show(PsS)
       ============================================================
       inter-ap Aps = Aps


// transition of a process does not lead to a new application state

[AP1f] inter-p Ps in <MnS,AtS> = inter-p Ps' in <MnS',AtS'>
       ====================================================================
       if-trans-then-diffact-else-same(aps(PsS,MnS,AtS),Ps) = aps(PsS,MnS,AtS)


// transition of a process leads to a new application state

[AP1g] inter-p Ps in <MnS,AtS> = Ps' with <EoS,AvS>,
       aps(PsS,MnS,AtS) minus-ap aps({Ps},MnS,AtS) = Aps',
       Aps' union-ap aps({Ps'},MnS,AtS) = Aps''
       ====================================================================
       if-trans-then-diffact-else-same(aps(PsS,MnS,AtS),Ps) =
                                                   diffact <EoS,AvS> in Aps''



///////////////////////////////////////////////////////////////////////////
//
//                 Definition of the new application state by diffusion
//
///////////////////////////////////////////////////////////////////////////


// end of the diffusion

[AP2a] diffact <EoS,AvS> in aps({},MnS,AtS) = aps({},MnS,AtS)


////////////////////////////////////
```

```
// diffusion to inactive processes
//////////////////////////////////

// diffusion to an inactive manifold process that must be activated

[AP2b]  act(name(PSt)) isin-a AvS = true,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps',
        EoS' union-e {start.self} = EoS1,
        {*.system} gen-inters EoS = EoS2
        ==============================================================
        diffact <EoS,AvS> in aps({ps(PSt,inactive,EoS'), PsL},MnS,AtS) =
        aps({ps(PSt,inactive,EoS1 union-e EoS2)},MnS,AtS) union-ap Aps'


// diffusion to an inactive manifold process that must not be activated

[AP2c]  act(name(PSt)) isin-a AvS = false,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps',
        {*.system} gen-inters EoS = EoS1
        ==============================================================
        diffact <EoS,AvS> in aps({ps(PSt,inactive,EoS'), PsL},MnS,AtS) =
        aps({ps(PSt,inactive,EoS' union-e EoS1)},MnS,AtS) union-ap Aps'


// diffusion to an inactive atomic process that must be activated

[AP2d]  act(name(PStAt)) isin-a AvS = true,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps',
        EoS' union-e {start.self} = EoS1,
        {*.system} gen-inters EoS = EoS2
        ================================================================
        diffact <EoS,AvS> in aps({ps(PStAt,inactive,EoS'), PsL},MnS,AtS) =
        aps({ps(PStAt,inactive,EoS1 union-e EoS2)},MnS,AtS) union-ap Aps'


// diffusion to an inactive atomic process that must not be activated

[AP2e]  act(name(PStAt)) isin-a AvS = false,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps',
        {*.system} gen-inters EoS = EoS1
        ================================================================
        diffact <EoS,AvS> in aps({ps(PStAt,inactive,EoS'), PsL},MnS,AtS) =
        aps({ps(PStAt,inactive,EoS' union-e EoS1)},MnS,AtS) union-ap Aps'


/////////////////////////////////////
// diffusion to deactivated processes
/////////////////////////////////////

// diffusion to a deactivated manifold process
```

```
[AP2f]  diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
        ==================================================================
        diffact <EoS,AvS> in aps({ps(PSt,deactivated,EoS'), PsL},MnS,AtS) =
        aps({ps(PSt,deactivated,EoS')},MnS,AtS) union-ap Aps'

// diffusion to a deactivated atomic process

[AP2g]  diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
        ====================================================================
        diffact <EoS,AvS> in aps({ps(PStAt,deactivated,EoS'), PsL},MnS,AtS) =
        aps({ps(PStAt,deactivated,EoS')},MnS,AtS) union-ap Aps'


////////////////////////////////
// diffusion to active processes
////////////////////////////////

// diffusion to an active manifold process that must be deactivated

[AP2h]  BlSt != inactive,
        BlSt != deactivated,
        deact(name(PSt)) isin-a AvS = true,
        EoS' union-e EoS union-e {terminate.self} = EoS1,
        observables(ps(PSt,BlSt,EoS1)) = EoS2,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
        ============================================================
        diffact <EoS,AvS> in aps({ps(PSt,BlSt,EoS'), PsL},MnS,AtS) =
        aps({ps(PSt,BlSt,EoS2)},MnS,AtS) union-ap Aps'


// diffusion to an active manifold process that must not be deactivated

[AP2i]  BlSt != inactive,
        BlSt != deactivated,
        deact(name(PSt)) isin-a AvS = false,
        EoS' union-e EoS = EoS1,
        observables(ps(PSt,BlSt,EoS1)) = EoS2,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
        ============================================================
        diffact <EoS,AvS> in aps({ps(PSt,BlSt,EoS'), PsL},MnS,AtS) =
        aps({ps(PSt,BlSt,EoS2)},MnS,AtS) union-ap Aps'


// diffusion to an active atomic process that must be deactivated

[AP2j]  BlStAt != inactive,
        BlStAt != deactivated,
        deact(name(PStAt)) isin-a AvS = true,
        EoS' union-e EoS union-e {terminate.self} = EoS1,
        {*.self, *.system } gen-inters EoS1 = EoS2,
        diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
        ================================================================
       diffact <EoS,AvS> in aps({ps(PStAt,BlStAt,EoS'), PsL},MnS,AtS) =
       aps({ps(PStAt,BlStAt,EoS2)},MnS,AtS) union-ap Aps'


// diffusion to an active atomic process that must not be deactivated

[AP2k] BlStAt != inactive,
       BlStAt != deactivated,
       deact(name(PStAt)) isin-a AvS = false,
       EoS' union-e EoS = EoS1,
       {*.self, *.system } gen-inters EoS1 = EoS2,
       diffact <EoS,AvS> in aps({PsL},MnS,AtS) = Aps'
       ===============================================================
       diffact <EoS,AvS> in aps({ps(PStAt,BlStAt,EoS'), PsL},MnS,AtS) =
       aps({ps(PStAt,BlStAt,EoS2)},MnS,AtS) union-ap Aps'



//////////////////////////////////////////////////////////////////
//
//                 Definition of the observable events
//
//////////////////////////////////////////////////////////////////


// event occurrences referenced in the handlers of a non-empty list of blocks

[AP3a] handler-refs(manner Mc{BlP} Bls) =
                           handler-refs(BlP) union-e handler-refs(Bls)

[AP3b] handler-refs(Lb:Hd. BlP) =
                           current-handler-refs(Hd) union-e handler-refs(BlP)

[AP3c] handler-refs(Lb:Hd.) = current-handler-refs(Hd)



// observable events

[AP3d] handler-refs(Bls) union-e {*.self } = EoS1,
       EoS1 gen-inters  EoS = EoS2,
       all-self-ports (Pn, EoS) = EoS3,
       EoS2 union-e EoS3 = EoS4,
       labels-refs(Bls) gen-inters EoS4 = EoS5,
       EoS5 gen-minus labels-refs(Lb:Hd.) = EoS6,
       evolutives0 gen-inters EoS4 = EoS7,
       {*.system} gen-inters EoS = EoS8,
       EoS7 union-e EoS8 = EoS9
       ============================================================
       observables(ps(manifold Pn{Bls}, Lb:Hd.,EoS)) = EoS6 union-e EoS9
```

```
[AP3e] handler-refs(Bls) union-e {*.self } = EoS1,
       EoS1 gen-inters  EoS = EoS2,
       all-self-ports (Pn, EoS) = EoS3,
       EoS2 union-e EoS3 = EoS4,
       labels-refs(Bls) gen-inters EoS4 = EoS5,
       evolutives0 gen-inters EoS4 = EoS6,
       {*.system} gen-inters EoS = EoS7,
       EoS6 union-e EoS7 = EoS8
       =========================================================
       observables(ps(manifold Pn{Bls}, end ,EoS)) = EoS5 union-e EoS8
```

A.4 Input/Output

The modules in this section are provided courtesy of Frank Tip.

A.4.1 Input

```
%% Input

imports Lisp  Layout Manifold-Syntax

exports

  context-free syntax

    read-ev-occ0                                     -> EV-OCC
    read-proc-n0                                     -> PROC-N


hiddens

  sorts INPUT

  context-free syntax
    "Please enter one element:" EV-OCC               -> INPUT
    "Please enter one element:" PROC-N               -> INPUT

    "LISP" LISP                                      -> EV-OCC
    "EV-OCC" LISP                                    -> EV-OCC
    "<<" EV-OCC ">>"                                 -> LISP


    "LISP" LISP                                      -> PROC-N
    "PROC-N" LISP                                    -> PROC-N
    "<<" PROC-N ">>"                                 -> LISP


 equations

     [IN1] read-ev-occ0 =
           EV-OCC  (let
                          ( vtp )
                         (#:SEAL:create-input
                            "/tmp/input"
                            "Input"
                            (list "Please enter one element: <EV-OCC>")
                            "INPUT"
                            (send 'config-table META) )
                          (setq vtp (#:SEAL:select
                                      "/tmp/input"
                                     "EV-OCC"
                                     (send 'config-table META)))
                          (#:SEAL:kill-inputs (send 'config-table META))
                         (bitmap-flush)
                          (#:EQM:tree:vtp2 vtp)
                       )

    [IN2] read-proc-n0 =
           PROC-N  (let
                          ( vtp )
                         (#:SEAL:create-input
                            "/tmp/input"
                            "Input"
                            (list "Please enter one element: <PROC-N>")
                            "INPUT"
                            (send 'config-table META) )
                          (setq vtp (#:SEAL:select
                                      "/tmp/input"
                                     "PROC-N"
                                     (send 'config-table META)))
                          (#:SEAL:kill-inputs (send 'config-table META))
                         (bitmap-flush)
                          (#:EQM:tree:vtp2 vtp)
                       )
```

A.4.2 Output

```
%% Output

imports Layout Lisp

exports
    sorts OUTPUT

    context-free syntax
        emit(OUTPUT)            -> OUTPUT
%%      emit-string(OUTPUT)     -> OUTPUT

        "LISP" LISP             -> OUTPUT
        "OUTPUT" LISP           -> OUTPUT
        "<<" OUTPUT ">>"        -> LISP

variables
```

```
        Dummy[0-9']*              -> OUTPUT


equations

    [OUT1] emit(Dummy) = OUTPUT
       (let*
         ( (theEQM (#:EQM:EQMsel:eqm #:EQM:sel))
           (text (#:EQM:real-pretty #:EQM:sel
             (#:EQM:tree:2vtp (#:EQM:tree:leximplode theEQM << Dummy >>))))
         )
         (mapc 'print text)
         << Dummy >>
       )
```

A.4.3 Lisp

```
%%
%% The module LISP
%%
%% The EQM knows about this module.
%% DON'T EDIT, except when you want to add LispIds
%% to the lexical (not context-free!) syntax.
%%
%%
%% Not yet supported:
%% ` and ,

exports
    sorts
         LISP

   sorts
         LispAtom StringPart PackagePart QuotedLispAtom
         LispId LispString


lexical syntax
        %% All non control or meta characters that have character
        %% class pname.

        [$%&*/\-0-9=?@A-Z\\a-z~]+   -> LispAtom

        ":" LispAtom                    -> PackagePart
        "#" PackagePart+                -> LispId
        "#%"[01]+                       -> LispId
        "#$"[0-9a-fA-F]+                -> LispId
        "#^"[a-zA-Z@]                   -> LispId

        "|" ~ [|]* "|"                  -> QuotedLispAtom
        QuotedLispAtom+                 -> LispAtom
        LispAtom                        -> LispId

        "\"" ~["]* "\""                 -> StringPart
        StringPart+                     -> LispString

context-free syntax

        LispId       -> LISP
        LispString   -> LISP

        "(" LISP * ")" -> LISP
%% And when you prefer them, they aren't preferred.
        "<<" Condition ">>" -> LISP
         "'" LISP -> LISP
```

A.5 Utilities

A.5.1 Booleans

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%               Module Booleans
%%
%% Define the sort boolean, and several operation
%% on it.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


imports Layout

exports
      sorts BOOL

      context-free syntax

         true                     -> BOOL
         false                    -> BOOL
         BOOL "|" BOOL            -> BOOL {left}
         BOOL "&" BOOL            -> BOOL {left}
         not(BOOL)           -> BOOL
         "("BOOL")"               -> BOOL {bracket}
      variables
         Bool [0-9']*             -> BOOL

   priorities
      "|" < "&"


equations
```

```
[B1]    true | Bool  = true
[B2]    false | Bool = Bool

[B3]    true & Bool  = Bool
[B4]    false & Bool = false

[B5]    not(false)   = true
[B6]    not(true)    = false
```

A.5.2 Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Module Util-Sets
%%
%% This module defines a small language for manipulating sets
%% of several types, which will be used for the specification of
%% Manifold's semantics
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

imports Booleans Internal-Syntax

exports


  context-free syntax


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on sets of event occurrences

    EV-OCC isin-e EV-OCC-SET                      -> BOOL
    EV-OCC-SET union-e EV-OCC-SET                 -> EV-OCC-SET {left}
    EV-OCC-SET inters-e EV-OCC-SET                -> EV-OCC-SET
    EV-OCC-SET minus-e EV-OCC-SET                 -> EV-OCC-SET



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on groups considered as sets of actions

    ACT isin-g GROUP                              -> BOOL
    GROUP union-g GROUP                           -> GROUP      {left}



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on sets of processes activations
    ACTIV isin-a ACTIV-SET                                   -> BOOL
    ACTIV-SET union-a ACTIV-SET                              -> ACTIV-SET  {left}


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on environments of the action level, considered
    %% as 4 tuples of sets

    A-ENV union-ae A-ENV                                     -> A-ENV      {left}


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on sets of event occurrences, when * is equal (match) to
    %% any event name or any source. Rightmost terms cannot contain *, while
    %% leftmost can. Hence the membership operation is a filter of the
    %% set, conserving only its elements that match the given element.
    %% The intersection operation conserves from the rightmost set, the
    %% elements that match one of the elements of leftmost set (it is
    %% NOT commutative. The difference operation, remove from the rightmost
    %% set, the elements that match one of the elements of the leftmost set.

    match(EV-OCC,EV-OCC)                                     -> BOOL
    EV-OCC gen-isin EV-OCC-SET                               -> EV-OCC-SET
    EV-OCC-SET gen-inters EV-OCC-SET                         -> EV-OCC-SET
    EV-OCC-SET gen-minus EV-OCC-SET                          -> EV-OCC-SET


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on sets of processes.

    PROC-N isin-p PROC-N-SET                                 -> BOOL
    PROC-N-SET union-p PROC-N-SET                            -> PROC-N-SET


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Operations on states of the application level, considered as
    %% sets of states of the process level (the other fields are
    %% ignored and are assumed to be the same for the two elements
    %% of an operation).

    P-STATE isin-ap AP-STATE                                     -> BOOL
    AP-STATE union-ap AP-STATE                                   -> AP-STATE
    AP-STATE minus-ap AP-STATE                                   -> AP-STATE

hiddens
```

```
  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% For each type of set, hidden operations that help defining the
    %% visible operations (add helps defining union, and rem helps
    %% defining inters).

    add-e EV-OCC to EV-OCC-SET                             -> EV-OCC-SET
    rem-e EV-OCC from EV-OCC-SET                           -> EV-OCC-SET

    add-g ACT to GROUP                                     -> GROUP

    add-a ACTIV to ACTIV-SET                               -> ACTIV-SET

    gen-rem EV-OCC from EV-OCC-SET                         -> EV-OCC-SET

    add-p PROC-N to PROC-N-SET                             -> PROC-N-SET

    add-ap P-STATE to AP-STATE                             -> AP-STATE
    rem-ap P-STATE from AP-STATE                           -> AP-STATE

equations

//////////////////////////////////////////////////////////////////////
//
//      Sets of event occurrences
//
//////////////////////////////////////////////////////////////////////


// Is an element a member of a set?

[UTS1a]  Eo isin-e {} = false

[UTS1b]  Eo isin-e {Eo,EoL} = true

[UTS1c]  Eo != Eo'
         ======================================
         Eo isin-e {Eo',EoL} = Eo isin-e {EoL}


// Addition of an element to a set

[UTS1d]  Eo isin-e EoS = true
         =====================
         add-e Eo to EoS = EoS

[UTS1e]  Eo isin-e {EoL} = false
         ===========================
         add-e Eo to {EoL} = {Eo,EoL}


// Removal of an element from a set

[UTS1f]  rem-e Eo from {} = {}

[UTS1g]  rem-e Eo from {Eo,EoL} = {EoL}

[UTS1h]  Eo != Eo'
         ========================================================
         rem-e Eo from {Eo',EoL} = add-e Eo' to rem-e Eo from {EoL}


// Union of two sets

[UTS1i]  {} union-e EoS = EoS

[UTS1j]  add-e Eo to EoS = EoS'
         ========================================
         {Eo,EoL} union-e EoS = {EoL} union-e EoS'


// Intersection of two sets

[UTS1k]  EoS inters-e {} = {}

[UTS1l]  Eo isin-e EoS = false
         ========================================
         EoS inters-e {Eo,EoL}= EoS inters-e {EoL}

[UTS1m]  Eo isin-e EoS = true,
         EoS inters-e {EoL} = EoS'
         ========================================
         EoS inters-e {Eo,EoL} = add-e Eo to EoS'


// Difference of two sets

[UTS1n]  EoS minus-e {} = EoS

[UTS1o]  rem-e Eo from EoS = EoS'
         ========================================
         EoS minus-e {Eo,EoL} = EoS' minus-e {EoL}



//////////////////////////////////////////////////////////////////////
//
//       Groups, considered as sets of actions
//
```

```
/////////////////////////////////////////////////////////////////////


// Is an element a member of a set?

[UTS2a]  Ac isin-g () = false

[UTS2b]  Ac isin-g (Ac,AcL) = true

[UTS2c]  Ac != Ac'
         =====================================
         Ac isin-g (Ac',AcL) = Ac isin-g (AcL)


// Addition of an element to a set

[UTS2d]  Ac isin-g Gr = true
         ===================
         add-g Ac to Gr = Gr

[UTS2e]  Ac isin-g (AcL) = false
         ============================
         add-g Ac to (AcL) = (Ac,AcL)


// Union of two sets

[UTS2f]  () union-g Gr = Gr


[UTS2g]  add-g Ac to Gr = Gr'
         =======================================
         (Ac,AcL) union-g Gr = (AcL) union-g Gr'



/////////////////////////////////////////////////////////////////////
//
//       Sets of processes (de)activations
//
/////////////////////////////////////////////////////////////////////


// Is an element a member of a set?

[UTS3a]  Av isin-a {} = false

[UTS3b]  Av isin-a {Av,AvL} = true

[UTS3c]  Av != Av'
         =====================================
         Av isin-a {Av',AvL} = Av isin-a {AvL}


// Addition of an element to a set

[UTS3d]  Av isin-a AvS = true
         ======================
         add-a Av to AvS = AvS

[UTS3e]  Av isin-a {AvL} = false
         ===========================
         add-a Av to {AvL} = {Av,AvL}


// Union of two sets

[UTS3f]  {} union-a AvS = AvS

[UTS3g]  add-a Av to AvS = AvS'
         ========================================
         {Av,AvL} union-a AvS = {AvL} union-a AvS'



/////////////////////////////////////////////////////////////////////
//
// Environments of the action level, considered as tuples of sets
//
/////////////////////////////////////////////////////////////////////


// union of two environments

[UTS4a]  EoS1 union-e EoS2 = EoS3,
         EoS1' union-e EoS2' = EoS3',
         Gr1 union-g Gr2 = Gr3,
         AvS1 union-a AvS2 = AvS3
         ============================================================
         ae(EoS1,EoS1',Gr1,AvS1) union-ae  ae(EoS2,EoS2',Gr2,AvS2) =
         ae(EoS3,EoS3',Gr3,AvS3)



/////////////////////////////////////////////////////////////////////
//
//                Sets of event occurrences,
//       when * is equal to any source or any  event
//
/////////////////////////////////////////////////////////////////////


// equality of two event occurrences,
// the leftmost parameter might contain *
// the rightmost cannot

[UTS5a]  match(*.*,En.Sc) = true

[UTS5b]  Sc != *
         ========================
         match(*.Sc,En.Sc)= true

[UTS5c]  Sc != *, Sc != Sc'
         ===========================
         match(*.Sc,En.Sc') = false

[UTS5d]  match(En.*,En.Sc) = true

[UTS5e]  En != En'
         ===========================
         match(En.*,En'.Sc) = false

[UTS5f]  Sc != *
         ==========================
         match(En.Sc,En.Sc) = true

[UTS5g]  Sc!= *,
         En != En'
         ============================
         match(En.Sc,En'.Sc) = false

[UTS5h]  Sc != *,
         Sc != Sc'
         ============================
         match(En.Sc,En.Sc') = false

[UTS5i]  Sc != *,
         En != En', Sc != Sc'
         ==============================
         match(En.Sc, En'.Sc') = false


// membership = filtering of a set by an element,
// conserving those elements of the set that match
// the given element

[UTS5j]  Eo gen-isin {} = {}

[UTS5k]  match(Eo,Eo') = true,
         Eo gen-isin {EoL} = EoS
         ==========================================
         Eo gen-isin {Eo',EoL} = EoS union-e {Eo'}

[UTS5l]  match(Eo,Eo') = false
         ==========================================
         Eo gen-isin {Eo',EoL} = Eo gen-isin {EoL}


// removal of elements of a set matching a given element.

[UTS5m]  gen-rem Eo from {} = {}

[UTS5n]  match(Eo,Eo') = false,
         gen-rem Eo from {EoL} = EoS
         ============================================
         gen-rem Eo from {Eo',EoL} = EoS union-e {Eo'}

[UTS5o]  match(Eo,Eo') = true
         ================================================
         gen-rem Eo from {Eo',EoL} = gen-rem Eo from {EoL}


// intersection = filtering of the rightmost set,
// conserving those elements that match some element
// of the leftmost set. The leftmost set's elements might
// contain *, while those of the rightmost set cannot.

[UTS5p]  {} gen-inters EoS = {}

[UTS5q]  Eo gen-isin EoS = EoS',
         {EoL} gen-inters  EoS = EoS''
         ============================================
         {Eo,EoL} gen-inters EoS = EoS' union-e EoS''


// difference = filtering of the rightmost set,
// conserving those elements that DO NOT match some
// element of the leftmost set. The leftmost set's
// elements might contain *, while those of the
// rightmost set cannot.

[UTS5r]  EoS gen-minus {} = EoS

[UTS5s]  gen-rem Eo from EoS = EoS'
         ============================================
         EoS gen-minus {Eo,EoL} = EoS' gen-minus {EoL}



/////////////////////////////////////////////////////////////////////
//
//                 Sets of processes (names)
//
/////////////////////////////////////////////////////////////////////
```

```
// Is an element a member of a set?

[UTS6a]  Pn isin-p {} = false

[UTS6b]  Pn  isin-p {Pn,PnL} = true

[UTS6c]  Pn != Pn'
         ====================================
         Pn isin-p {Pn',PnL} = Pn isin-p {PnL}


// Addition of an element to a set

[UTS6d]  Pn isin-p PnS = true
         =====================
         add-p Pn to PnS = PnS

[UTS6e]  Pn isin-p {PnL} = false
         ============================
         add-p Pn to {PnL} = {Pn,PnL}

// Union of two sets

[UTS6f]  {} union-p PnS = PnS

[UTS6g]  add-p Pn to PnS = PnS'
         ========================================
         {Pn,PnL} union-p PnS = {PnL} union-p PnS'


//////////////////////////////////////////////////////////////////////
//
// States of the application level, considered as sets of states
// of the process level (the other fields are ignored and are
// assumed to be the same for the two elements of an operation).
//
//////////////////////////////////////////////////////////////////////


// Is a process state a member of an application state?

[UTS7a]  Ps isin-ap aps({},MnS,AtS) = false

[UTS7b]  Ps isin-ap aps({Ps,PsL},MnS,AtS) = true

[UTS7c]  Ps != Ps'
         ==================================================================
         Ps isin-ap aps({Ps',PsL},MnS,AtS) = Ps isin-ap aps({PsL},MnS,AtS)


// Addition of a process state to an application state
```

```
[UTS7d]  Ps isin-ap Aps = true
         ======================
         add-ap Ps to Aps = Aps

[UTS7e]  Ps isin-ap aps({PsL},MnS,AtS) = false
         ========================================================
         add-ap Ps to aps({PsL},MnS,AtS) = aps({Ps,PsL} ,MnS,AtS)


// Removal of a process state from an application state

[UTS7f]  rem-ap Ps from aps({},MnS,AtS) = aps({},MnS,AtS)

[UTS7g]  rem-ap Ps from aps({Ps,PsL},MnS, AtS) = aps({PsL},MnS,AtS)

[UTS7h]  Ps != Ps'
         ================================================================
         rem-ap Ps from aps({Ps',PsL},MnS,AtS) =
                          add-ap Ps' to rem-ap Ps from aps({PsL},MnS,AtS)


// Union of two application states

[UTS7i]  aps({},MnS,AtS) union-ap Aps = Aps

[UTS7j]  add-ap Ps to Aps = Aps'
         ======================================================================
         aps({Ps,PsL},MnS,AtS) union-ap Aps = aps({PsL},MnS,AtS) union-ap Aps'


// Difference of two application states

[UTS7k]  Aps minus-ap aps({},MnS,AtS) = Aps

[UTS7l]  rem-ap Ps from Aps = Aps'
         ======================================================================
         Aps minus-ap aps({Ps,PsL},MnS,AtS) = Aps' minus-ap aps({PsL},MnS,AtS)
```

A.5.3 Others

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%                 Module  Util
%%
%% This module defines additional constructs, in order
%% to manipulate the internal syntax more easily.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
imports Util-Sets Input Output

exports
  sorts PROD-SELECT-E PROD-SELECT-P

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Constructs used at the process level
    %%
    %% -labels(Bl) is the set of event occurrences contained
    %%  in the label of block Bl.
    %% -name(PSt) and name(PStAt) are the name of
    %%  respectively the manifold and atomic process with
    %%  respective program states PSt and PStAt.
    %% -proc-blocks(PSt) denotes the blocks of the manifold
    %%  process with program state PSt.
    %% -man-blocks(Mc,MnS) denotes the body Bd of the manner
    %%  Mc such that Mc Bd is in the set of internal
    %%  representation of manners MnS.
    %% -output(PStAt,AtS) denotes the set of event occurrences
    %%  EoS that can be raised by an atomic process with
    %%  program state PStAt. AtS is the set of internal
    %% representation for atomic processes, which allows
    %% one to determine EoS from PStAt.

    labels(BLOCK)                             -> EV-OCC-SET
    name(PROC-ST)                             -> PROC-N
    name(PROC-ST-AT)                          -> PROC-N
    proc-blocks(PROC-ST)                      -> BLOCKS
    man-blocks(MANNERCALL,MANNER-REP-SET)     -> BODY
    output(PROC-ST-AT, ATOMIC-REP-SET)        -> EV-OCC-SET

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Used at the application level.
    %%
    %% -all-proc-non-deactivated(Aps) is the set of processes
    %%  (names) which are  not deactivated in application
    %%  state Aps
    %% -process-state(Pn,Aps) is the state of the process
    %%  with name Pn in application state Aps.

    all-proc-non-deactivated AP-STATE         -> PROC-N-SET
    process-state(PROC-N,AP-STATE)            -> P-STATE


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Input/output at process and application levels.
    %%
    %% -show outputs a set of process states
    %% -select-ev-occ(EoS) is a pair:
    %%  - a boolean saying whether an element of EoS has been
    %%    selected.
    %%  - the element selected which is given by the user
    %%    except if the set was a singleton, or the empty set
    %%    (in this last case, no selection is possible).
    %% -select-pn(PnS): same as select-ev-occ, but for a set
    %%  of process (names).
    %% note that the selection by the user intends to replace
    %% the existential quantifier which leads to an
    %% undeterministic selection.

    show(P-STATE-SET)                        -> OUTPUT

    "("BOOL","EV-OCC")"                      -> PROD-SELECT-E
    select-ev-occ(EV-OCC-SET)                -> PROD-SELECT-E
    "("BOOL","PROC-N")"                      -> PROD-SELECT-P
    select-proc-n(PROC-N-SET)                -> PROD-SELECT-P

hiddens

  context-free syntax

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% Definition of the terms that are output when a process
    %% state is output.
    %% show(Ps) is the output of the process state Ps

    EV-OCC-SET                               -> OUTPUT
    PROC-N-SET                               -> OUTPUT
    PROC-N                                   -> OUTPUT
    P-STATE                                  -> OUTPUT
    BLOCKS                                   -> OUTPUT
    BLOCK-ST                                 -> OUTPUT
    show(P-STATE)                            -> OUTPUT

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%
    %% cont-select-or-stop Eo EoS, where EoS is a set of at
    %% least two event occurrences, is Eo if Eo is in EoS,
    %% and the result of another selection otherwise.
    %% This is used to ensure that the user selects an
    %% element of EoS.
    %% cont-select-or-stop Pn PnS has the same meaning for
    %% process names.

    cont-select-or-stop EV-OCC EV-OCC-SET  -> PROD-SELECT-E
    cont-select-or-stop PROC-N PROC-N-SET  -> PROD-SELECT-P
```

```
 equations

// set of event occurrences in a label

[UT1a]  labels(Eo:Hd.) = {Eo}

[UT1b]  labels(Eo,EoP:Hd.) = labels(EoP:Hd.) union-e {Eo}


// name of a process

[UT2a]  name(manifold Pn {Bls}) = Pn

[UT2b]  name(atomic Pn) = Pn


// blocks of a manifold process

[UT3a]  proc-blocks(manifold Pn {Bls}) = Bls


// blocks of a manner

[UT4a]  man-blocks(Mc,{ MnL1, Mc Bd, MnL2}) =  Bd


// set of events that can be raised by an atomic process

[UT5a]  name(PStAt) = Pn
        =====================================
        output(PStAt,{AtL1, Pn EoS, AtL2}) = EoS



// set of process (names) that are not deactivated in an application state.


// application with no process

[UT6a]  all-proc-non-deactivated aps({},MnS,AtS) = {}


// non-deactivated manifold process

[UT6b]  BlSt != deactivated,
        all-proc-non-deactivated aps({PsL},MnS,AtS) = PnS
        ================================================================
        all-proc-non-deactivated aps({ps(PSt,BlSt,EoS),PsL},MnS,AtS) =
                                                  {name(PSt)} union-p  PnS
```

```
// deactivated manifold process

[UT6c]  all-proc-non-deactivated aps({ps(PSt,deactivated,EoS),PsL},MnS,AtS)=
        all-proc-non-deactivated aps({PsL},MnS,AtS)


// non-deactivated atomic process

[UT6d]  BlStAt != deactivated,
        all-proc-non-deactivated aps({PsL},MnS,AtS) = PnS
        ==================================================================
        all-proc-non-deactivated aps({ps(PStAt,BlStAt,EoS),PsL},MnS,AtS)
                                               = {name(PStAt)} union-p PnS


// deactivated atomic process

[UT6e]  all-proc-non-deactivated aps({ps(PStAt,deactivated,EoS),PsL},MnS,AtS) =
        all-proc-non-deactivated aps({PsL},MnS,AtS)



// state of the process with a given name in a given application

[UT7a]  process-state(Pn,aps({PsL1,ps(manifold Pn{Bls},BlSt,EoS),PsL2},MnS,AtS))
    = ps(manifold Pn{Bls},BlSt,EoS)

[UT7b]  process-state(Pn,aps({PsL1,ps(atomic Pn,BlStAt,EoS),PsL2},MnS,AtS))
    = ps(atomic Pn,BlStAt,EoS)


// output of a set of process states

[UT8a]  Dummy = show(Ps1)
        =====================================
        show({Ps1,Ps2,PsL}) = show({Ps2,PsL})

[UT8b]  Dummy = show(Ps)
        ==================
        show({Ps}) = Dummy


// output of a process state

[UT8c]  Dummy1 = emit(Pn),
        Dummy2 = emit(Bls),
        Dummy3 = emit(BlSt),
        Dummy4 = emit(EoS)
        ============================================
        show(ps(manifold Pn{Bls},BlSt,EoS))  = Dummy4

[UT8d]  Dummy = emit(ps(PStAt,BlStAt,EoS))
        ==================================
        show(ps(PStAt,BlStAt,EoS)) = Dummy


// selection of an event occurrence

[UT9a]  Dummy = emit({Eo1',Eo2',EoL}),
        Eo = read-ev-occ0
        ========================================================================
        select-ev-occ({Eo1',Eo2',EoL}) = cont-select-or-stop Eo {Eo1',Eo2',EoL}

[UT9b]  select-ev-occ({}) = (false,nothing.nothing)

[UT9c]  select-ev-occ({Eo}) = (true,Eo)


[UT9d]  Eo isin-e EoS = true
        ======================================
        cont-select-or-stop Eo EoS = (true,Eo)

[UT9e]  Eo isin-e EoS = false
        ======================================
        cont-select-or-stop Eo EoS = select-ev-occ(EoS)


// selection of a process name

[UT9f]  Dummy = emit({Pn1',Pn2',PnL}),
        Pn = read-proc-n0
        ========================================================================
        select-proc-n({Pn1',Pn2',PnL}) = cont-select-or-stop Pn {Pn1',Pn2',PnL}

[UT9g]  select-proc-n({}) = (false,nothing)

[UT9h]  select-proc-n({Pn}) = (true,Pn)


[UT9i]  Pn isin-p PnS = true
        ======================================
        cont-select-or-stop Pn PnS = (true,Pn)

[UT9j]  Pn isin-p PnS = false
        ======================================
        cont-select-or-stop Pn PnS = select-proc-n(PnS)
```

Contents

List of Figures